WIP: An Engaging Undergraduate Intro to Model Checking in Software Engineering Using TLA$^+$

Konstantin Läufer*, Gunda Mertin†, and George K. Thiruvathukal*
*Software and Systems Laboratory, Department of Computer Science, Loyola University Chicago
{laufer,gkt}@cs.luc.edu
†Institute for Software Engineering and Programming Languages, University of Lübeck
gunda.mertin@student.uni-luebeck.de

***Abstract*—*Background*: In this Innovative Practice Work in Progress, we present our initial efforts to integrate formal methods, with a focus on model-checking specifications written in Temporal Logic of Actions (TLA$^+$), into computer science education, targeting undergraduate juniors/seniors and graduate students. Many safety-critical systems and services crucially depend on correct and reliable behavior. Formal methods can play a key role in ensuring correct and safe system behavior, yet remain underutilized in educational and industry contexts. *Aims*: We aim to (1) qualitatively assess the state of formal methods in computer science programs, (2) construct level-appropriate examples that could be included midway into one's undergraduate studies, (3) demonstrate how to address successive "failures" through progressively stringent safety and liveness requirements, and (4) establish an ongoing framework for assessing interest and relevance among students. *Methods*: We detail our pedagogical strategy for embedding TLA$^+$ into an intermediate course on formal methods at our institution. After starting with a refresher on mathematical logic, students specify the rules of simple puzzles in TLA$^+$ and use its included model checker (known as TLC) to find a solution. We gradually escalate to more complex, dynamic, event-driven systems, such as the control logic of a microwave oven, where students will study safety and liveness requirements. We subsequently discuss explicit concurrency, along with thread safety and deadlock avoidance, by modeling bounded counters and buffers. *Results*: Our initial findings suggest that through careful curricular design and choice of examples and tools, it is possible to inspire and cultivate a new generation of software engineers proficient in formal methods. *Conclusions*: Our initial efforts suggest that 84% of our students had a positive experience in our formal methods course. Our future plans include a longitudinal analysis within our own institution and proposals to partner with other institutions to explore the effectiveness of our open-source and open-access modules.**

***Index Terms*—Computer Science Education, Formal Methods, Model Checking, Software Testing, Safety-Critical Systems**

I. Introduction

In the evolving landscape of computer science education, methodologies that help to enhance the correctness and reliability of safety-critical systems are of vital importance [1], [2]. Recent advances in formal methods, particularly model checking of specifications written in TLA$^+$ [3] have emerged as a powerful tool in bridging the gap between theoretical computer science and practical software engineering. Although formal methods—rooted in automata theory, formal logic and, more specifically, Hoare logic [4]—have a long history and continue to evolve, they have only modestly impacted CS education compared to many topics (e.g. machine learning, cybersecurity, Internet of Things, among others). Although generative AI shows promise to reduce the error-prone nature of human coding, we remain convinced that humans—and non-humans—can employ formal methods to provide critical correctness and reliability checks on solutions, whether crafted by hand or generated by (statistical) AI and large-language ML models.

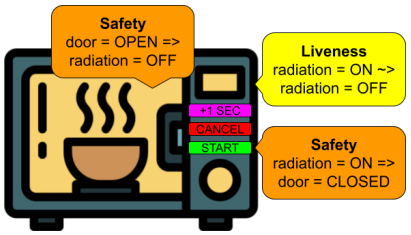

Figure 1. The microwave oven is an engaging and comprehensible, yet non-trivial, example of a safety-critical embedded system worthy of student attention in a Formal Methods course. This figure illustrates how we model two critical requirements: (1) To prevent radiation exposure, the appliance must *not* run with the door open, and (2) To prevent overheating while radiating, it must eventually turn off (per the TLA$^+$ *leads to* operator $\rightsquigarrow$).

In this paper, we describe our initial efforts to introduce students (undergraduate juniors/seniors and graduate students) to formal methods in software engineering with a focus on model checking using TLA$^+$, backed by preliminary results from ongoing anonymous input from our own students. We note that the selection of TLA$^+$ is intentional, as it is one of the most actively-maintained model checking systems (evidence provided in related work). We also incorporate other strategies and systems in our teaching: Students give presentations and demos on a broad landscape of tool- and language-based formal methods, including many of those shown in Table I. By applying TLA$^+$ to practical, familiar, and compelling examples, such as the microwave oven shown in Figure 1 and those listed in Table III, we aim to demystify formal methods and make them more relevant to students.

Our contributions include:

- Performing an initial analysis of academic programs offering formal methods courses, organized by course level, tools and techniques covered, and when last taught.
- Creating a set of curricular materials (i.e., exemplars) to motivate student interest in formal methods. We empha-

size reproducible behavior using modern software engineering principles (i.e., version control and continuous integration) as our own materials and TLA$^+$ both continue to evolve.
- Demonstrating how to address successive "failures" in model checking through a set of progressively stringent safety and liveness requirements.
- Conducting an initial evaluation of the effectiveness of formal methods among our current students. This effort will inform ongoing study on the long-term effectiveness of our curricular approach.

II. Background and Related Work

The private computing industry and high-level government agencies have long recognized the value of formal methods for building safety-critical and other highly reliable systems [5]. A major example is the National Aeronautics and Space Administration (NASA), which "develops formal methods technology for the development of mission-critical and safety-critical digital systems of interest to NASA [6]." Another safety-critical sector is ground transportation; a comprehensive survey of the use of formal methods in the railroad sector found that model checking is the most commonly adopted technique (47%), followed by simulation (27%) and theorem proving (19.5%) [7]. Specifically, *model checking* arose in the early 1980s in response to the challenges of concurrent program verification [8] and is now well established and mature [5], [9]–[11]. Among various specification languages that support model checking, TLA$^+$ has emerged as a practical and effective option [1], [12]–[14].

The need to recruit qualified professionals in these areas became apparent several decades ago and is now receiving renewed attention [2], [15]–[17]. Accordingly, academic educators have increasingly focused on formal methods teaching since the 1990s [9] and have been teaching model checking more broadly since the early 2000s [18]–[25]. Several academic institutions have reported on their use of TLA$^+$ in their courses [21], [26].

Nevertheless, multiple educators have recognized the challenges of motivating the need for formal methods topics and teaching them effectively in the face of some resistance [27], [28]. Key challenges include insufficient mathematical background from high school and university-level prerequisite courses, which may not focus sufficiently on discrete math and proof techniques; a lack of practical, engaging case studies that motivate the use of formal methods; insufficient documentation of some of the tools for students to gain confidence in using them; and learning styles of today's students (millennials and beyond), such as active, discovery-driven learning, an emphasis on solutions over theory, and a desire for immediate feedback [29], [30].

While some educators actually propose starting in the first year of the post-secondary curriculum [23] or even earlier [31], [32], we are concerned that these efforts will reach a relatively small number of students mostly at selective institutions. Therefore, we target juniors and seniors (at undergraduate level) and graduate students, and start with a three-week review of fundamentals studied earlier.

Our work builds on Askarpour and Bersani's study [30], which not only reports on specific experiences at their institution, but also includes a general analysis of the Formal Methods Education Database (FMEDB) [33] and a review of related work on the main challenges students face in formal methods courses. We analyzed FMEDB for the most commonly used tools in 34 courses focused on model checking (out of 88 courses total) and found that 7 use NuSMV/NuXMV, 2 use Dafny, and 2 use TLA$^+$ (one in industry and one in academia). Because these entries are self-submitted and the database doesn't say when a course was last taught, we also conducted our own preliminary survey of relevant courses taught during the last five years, summarized in Table I. As practicing software engineers, there are other reasons to use it: It is mature and well-maintained (by Leslie Lamport, a Turing Award winner) and has a strong community and ecosystem. In sum, the overall related work strongly supports the decision to emphasize TLA$^+$ in our own course.

TABLE I
Commonly used tools in FM courses, ranked by frequency of adoption as the primary tool, based on our preliminary survey of 32 recent courses; not shown are 15 courses without a primary tool or a less widely used one.

Tools	Number of Courses	Level
TLA+	5	Grad
Alloy	4	Both
SPIN	4	Both
Coq	2	Both
Z3	2	Grad

III. Curricular Goals

We target primarily undergraduates majoring in computer science or software engineering who have completed at least five foundational courses over three semesters: Introduction to Programming (CS1), Data Structures I (CS2), Introduction to Computer Systems (CS3), Discrete Structures, and a one-semester, hands-on introduction to the Linux command line. In our semester-long, 15-week, three-credit course, we focus on the most relevant learning outcomes from the ACM/IEEE Computer Society Software Engineering 2014 Curriculum Guidelines [34] in the order listed in Table II.

For many of our students, the most recent logic-based mathematics course is Discrete Structures, which most students take in their first year. To aid in the transition to our course on Formal Methods, we reinforce the essential mathematical foundations. This material, however, is targeted toward what we need here, using unit testing as a vehicle to connect these foundations with the programming practice they have undergone in their CS1 and CS2 courses.

We first build on students' understanding of correctness of a system under test w.r.t. a set of functional requirements and their ability to write automated, non-exhaustive unit tests that

TABLE II
Formal Methods course: Approximate mapping from second-level knowledge areas to contact hours (15 weeks, 3 semester credits).

SE 2014 Knowledge Area (Reference Code)	Hours
Mathematical foundations (FND.mf)	6
Modeling foundations (MAA.md)	3
Testing (VAV.tst)	9
Types of models (MAA.tm)	12
Analysis fundamentals (MAA.af)	6
Requirements specification (REQ.rsd)	3
Developing secure software (SEC.dev)	3
Various professional practice topics (PRF)	3
TOTAL	45

TABLE III
Formal Methods course: Examples with targeted knowledge areas.

Example	Tool	Knowledge Areas
Unit testing: palindrome checker	JUnit	Testing
Property-based testing: palindrome	jqwik	Testing
Stateful testing: circular buffer	jqwik	Testing
Microwave oven (see §IV)	TLA$^+$	Modeling, Requirements
Elevator control logic	TLA$^+$	Modeling, Requirements
Shared counter, explicit threads	TLA$^+$	Modeling, Concurrency
Bounded buffer, explicit threads	TLA$^+$	Modeling, Concurrency

reflect these requirements. We also leverage students' understanding of discrete structures, especially predicate logic, to progress from example-based testing to universally quantified properties that more generally reflect the functional requirements, using libraries that automatically test these properties on a range of pseudo-randomly generated arguments.

We then use TLA$^+$ to progressively introduce students to the formal specification and verification of dynamic and concurrent software systems. In TLA$^+$, a model has a finite set of variables, and each state maps these variables to their current values. TLA$^+$ allows us to model specific system behaviors that start in an initial state and undergo a sequence of states by performing transitions available in the next-state relation. Beginning with intuitive examples, such as puzzles, we specify the rules in TLA$^+$ and use its included model checker (known as TLC) to falsify the absence of a solution, thereby finding a counterexample that constitutes a solution. We gradually escalate to more complex, dynamic systems, such as the control logic of a microwave oven, where students will study safety and liveness requirements; for greater engagement, we emphasize event-driven, interactive systems. We subsequently study explicit concurrency, along with thread safety and deadlock avoidance, by modeling bounded counters and buffers. By proceeding gradually, we scaffold student understanding and application of formal methods (see Table III).

IV. Course Activities

Here we present a series of activities for students based on a simple microwave oven as a running example.

A. Running Example: A Simple Microwave Oven

We represent the state of our microwave as two booleans, *door* (open or closed) and *radiation* (on or off), along with a natural number, *timeRemaining*. The system also supports various actions, mostly corresponding to user controls:

- The user can open or close the door.
- The user can increment the remaining time.
- The user can start or cancel the heating process.
- An internal timer decrements the remaining time to zero.

As seen in Figure 2, radiation is initially off with no time remaining, while the door may be open or closed. The model's next-state relation *Next* is defined as the logical disjunction (choice) of the user-triggered actions *OpenDoor*, *CloseDoor*, *IncTime*, *Start*, *Cancel*, as well as the internal *Tick* action.

$$\begin{aligned} Init \triangleq\ & \land door \in \{OPEN, CLOSED\} \\ & \land radiation = OFF \\ & \land timeRemaining = 0 \end{aligned}$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

Figure 2. Valid initial states and top-level specification for the microwave oven: Initially, the microwave is not radiating, there is no time remaining, and the door is either open or closed. Subsequently, the model can repeatedly perform any available action, as indicated by the temporal operator "always" ($\Box$).

Activity 1 Starting with a skeletal model where each action manually controls the corresponding variable, develop appropriate state-dependent behavior, such as the timer counting down and automatically shutting off radiation when reaching zero. Figure 3 defines the resulting *Tick* action, and Figure 4 shows a step in a normal behavior.

$$\begin{aligned} Tick \triangleq\ & \land radiation = ON \\ & \land timeRemaining' = timeRemaining - 1 \\ & \land timeRemaining' \geq 0 \\ & \land \text{IF } timeRemaining' = 0 \\ & \quad \text{THEN } radiation' = OFF \\ & \quad \text{ELSE UNCHANGED } \langle radiation \rangle \\ & \land \text{UNCHANGED } \langle door \rangle \end{aligned}$$

Figure 3. *Tick* action for decrementing the timer, defined as a conjunction of preconditions, effects, and postconditions, where v' refers to a variable's value in the next state following the action. According to the preconditions, this action is enabled only when the oven is radiating and there is nonzero time remaining. Its effect is to decrement the remaining time; when the remaining time reaches zero, its additional effect is to shut off radiation.

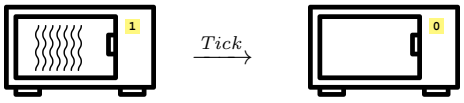

Figure 4. Scenario for normal operation: The microwave is initially radiating with the door closed and one second of time remaining. When the internal timer ticks, the remaining time goes to zero, and the oven stops radiating.

B. Invariants and Safety

To study system safety, we will initially allow the door and radiation to operate independently. The TLC model checker finds nothing wrong with this model because we have not yet defined any invariants it can attempt to falsify.

Exercise 2a Define an invariant *DoorSafety* to guarantee that the microwave will never be radiating with the door open. What happens after you add this invariant?

$$DoorSafety \triangleq door = OPEN \implies radiation = OFF$$

Once we add this invariant to our model, TLC considers *unsafe* any state in which the invariant is false and proves that there is at least one behavior leading to such a state (see Figure 5).

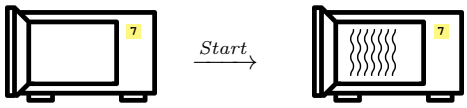

Figure 5. Scenario leading to an unsafe condition: The microwave is initially stopped with the door open and several seconds of time remaining. When the user presses the start button, the microwave emits harmful radiation.

Exercise 2b Refine the microwave model to satisfy the *DoorSafety* invariant, making the smallest possible changes.

This requires introducing more state dependency within the actions constituting the next-state relation. To prevent the scenario from Figure 5, we can require the door to be closed by adding a precondition to the *Start* action:

$$\wedge\ door = CLOSED$$

But TLC quickly finds another unsafe behavior, as shown in Figure 6. To prevent this scenario, we will add to the *OpenDoor* action the effect of immediately shutting off radiation.

$$\wedge\ radiation' = OFF$$

Figure 6. Scenario leading to an unsafe condition: The microwave is initially radiating with the door closed and several seconds of time remaining. When the user opens the door, harmful radiation comes out.

C. Temporal Properties and Liveness

Our microwave model is now safe in terms of preventing any behaviors that allow radiation to occur when the door is open. *But is this notion of safety enough?* How do we know it won't radiate indefinitely, overheat, and catch on fire?

TLA$^+$ includes temporal logic operators for predicates on a sequence of steps, e.g., some false condition eventually becoming true. Temporal logic predicates are often used to express liveness requirements, such as a radiating microwave oven eventually reaching zero remaining time and shutting off.

Exercise 3a Define a temporal property *HeatLiveness* requiring a radiating microwave to eventually turn off. What happens after you add this property?

$$HeatLiveness \triangleq radiation = ON \rightsquigarrow radiation = OFF$$

The "leads to" operator in $p \rightsquigarrow q$ indicates that if p is currently true, then q must become true in a future step.

Once we add this liveness property to our model, TLC will indicate that our current model does allow the scenario from Figure 7. Even though this behavior involves only valid states, it is undesirable because it allows radiation to continue indefinitely and the food inside the microwave to catch on fire.

In general, TLA$^+$ does not require the model to choose an action even if one is available. Instead, the model is allowed to perform a transition where it stays in its current state; this is called a *stuttering* step. This can model scenarios where a system loses power or, as in Figure 7, fails to make progress.

Figure 7. Scenario showing lack of liveness in the form of stuttering: The microwave is radiating but not receiving ticks required to reach zero.

Exercise 3b Refine the model to satisfy the *HeatLiveness* temporal property, making the smallest possible changes.

To break the observed stutter-invariance, we need to use a mechanism called *(weak) fairness*. We can enable this for the *Tick* action by adding $\wedge \mathrm{WF}_{vars}(Tick)$ to our top-level specification. The resulting model satisfies our previously established safety and liveness requirements.

V. Preliminary Evaluation

We offered the course for the first time in the Fall 2022 semester to 15 students, and again in the Spring 2024 semester to 22 students. Our new course-specific 15-question survey, given as a pretest-posttest in Spring 2024, indicates a marked improvement in student receptiveness to formal methods, with enhanced ability to conceptualize and apply these techniques in real-world scenarios; the cohort average of equally weighted composite scores increased from 3.0 to 4.1 ($n = 19$), with 84% reporting a positive experience. Notably, the use of TLA$^+$ has not only facilitated a deeper understanding of software correctness but also stimulated student interest in the subject matter. In addition, preliminary results from standardized teacher-course evaluations are highly encouraging, with average scores significantly above departmental averages and relatively high consensus among respondents.

VI. Conclusion

In summary, this work in progress demonstrates the successful integration of model checking using TLA$^+$ into our intermediate undergraduate computer science curriculum. With just three semesters (five courses) of prerequisites, students can learn about formal methods and model checking as early as their second semester of sophomore year and most certainly by the first semester of junior year, grounded in the ACM/IEEE Computing Curricula, which strongly suggest what students should learn and when they should learn it, including all appropriate foundational preparation. With our open source and open access materials (code and lecture) notes continuously

updated on GitHub and rebuilt and tested using continuous integration, anyone at any institution can use and contribute to our work. Our initial findings strongly suggest that this approach, including the use of effective, well-documented tools such as TLA$^+$, is engaging to students.

Our future plans include reaching out to other universities, starting with our own region, which has dozens of universities, and potentially offering a workshop aimed at workforce development for those who may have missed the opportunity to learn this material while studying computer science or a closely-related discipline.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. DGE-1919004. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors thank Ryan Hasler for helpful suggestions for improving the included TLA$^+$ examples.

References

[1] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015.

[2] M. Huisman, D. Gurov, and A. Malkis, *Formal Methods: From Academia to Industrial Practice. A Travel Guide*, arXiv:2002.07279 [cs], Feb. 2020.

[3] M. A. Kuppe, L. Lamport, and D. Ricketts, "The TLA+ Toolbox," *Electronic Proc. in Theoretical Computer Science*, vol. 310, pp. 50–62, Dec. 2019, arXiv:1912.10633 [cs].

[4] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[5] T. Lecomte, D. Deharbe, E. Prun, and E. Mottin, *Applying a Formal Method in Industry: A 25-Year Trajectory*, arXiv:2005.07190 [cs], May 2020.

[6] J. Maddalon, *NASA Langley Formal Methods Research Program*, Mar. 2024.

[7] A. Ferrari and M. H. T. Beek, "Formal Methods in Railways: A Systematic Mapping Study," *ACM Comput. Surv.*, vol. 55, no. 4, Nov. 2022.

[8] E. M. Clarke, "The Birth of Model Checking," en, in *25 Years of Model Checking: History, Achievements, Perspectives*, O. Grumberg and H. Veith, Eds., Springer, 2008, pp. 1–26.

[9] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," en, *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, Dec. 1996.

[10] O. Grumberg and H. Veith, Eds., *25 Years of Model Checking* (Lecture Notes in Computer Science), en. Springer, 2008, vol. 5000, ISSN: 0302-9743, 1611-3349.

[11] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*, en. Cham: Springer Intl. Publishing, 2018.

[12] L. Lamport, "Teaching concurrency," *SIGACT News*, vol. 40, no. 1, pp. 58–62, Feb. 2009.

[13] A. Ferrari, F. Mazzanti, D. Basile, and M. H. ter Beek, "Systematic Evaluation and Usability Analysis of Formal Methods Tools for Railway Signaling System Design," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4675–4691, 2022.

[14] M. A. Kuppe, "Teaching TLA+ to Engineers at Microsoft," en, in *Proc. 5th Intl. Workshop on Formal Methods Teaching (FMTea)*, C. Dubois and P. San Pietro, Eds., Lübeck, Germany: Springer Nature Switzerland, Mar. 2023, pp. 66–81.

[15] M. Jaume and T. Laurent, "Teaching Formal Methods and Discrete Mathematics," *Electronic Proc. in Theoretical Computer Science*, vol. 149, pp. 30–43, Apr. 2014, arXiv:1404.6604 [cs].

[16] D. Mandrioli, "On the heroism of really pursuing formal methods: Title inspired by Dijkstra's "On the Cruelty of Really Teaching Computing Science"," in *Proc. Third FME Workshop on Formal Methods in Software Engineering*, ser. Formalise '15, Place: Florence, Italy, IEEE Press, 2015, pp. 1–5.

[17] A. Cerone, M. Roggenbach, J. Davenport, *et al.*, *Rooting Formal Methods within Higher Education Curricula for Computer Science and Software Engineering – A White Paper*, arXiv:2010.05708 [cs] version: 1, Oct. 2020.

[18] H. Liu and D. P. Gluch, "A proposal for introducing model checking into an undergraduate software engineering curriculum," *Journal of Computing Sciences in Colleges*, vol. 18, no. 2, pp. 259–270, 2002, Publisher: Consortium for Computing Sciences in Colleges.

[19] S. Liu, K. Takahashi, T. Hayashi, and T. Nakayama, "Teaching formal methods in the context of software engineering," *SIGCSE Bull.*, vol. 41, no. 2, pp. 17–23, Jun. 2009.

[20] Y. Tahara, N. Yoshioka, K. Taguchi, T. Aoki, and S. Honiden, "Evolution of a course on model checking for practical applications," *SIGCSE Bull.*, vol. 41, no. 2, pp. 38–44, Jun. 2009.

[21] P. Tavolato and F. Vogt, "Integrating formal methods into computer science curricula at a university of applied sciences," in *TLA+ workshop at the 18th international symposium on formal methods, Paris, France*, 2012.

[22] W. Schreiner, A. Brunhuemer, and C. Fürst, "Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models," *Electronic Proc. in Theoretical Computer Science*, vol. 267, pp. 120–139, Mar. 2018, arXiv:1803.01472 [cs].

[23] L. Aceto and A. Ingólfsdóttir, "Introducing Formal Methods to First-Year Students in Three Intensive Weeks," in *Proc. 4th Intl. Workshop and Tutorial on Formal Methods Teaching(FMTea)*, J. F. Ferreira, A. Mendes, and C. Menghi, Eds., Lübeck, Germany: Springer Nature Switzerland, Nov. 2021, pp. 1–17.

[24] P. Körner and S. Krings, "Increasing Student Self-Reliance and Engagement in Model-Checking Courses," in *Proc. 4th Intl. Workshop on Formal Methods Teaching (FMTea)*, J. F. Ferreira, A. Mendes, and C. Menghi, Eds., Lübeck, Germany: Springer Nature Switzerland, Nov. 2021, pp. 60–74.

[25] F. Freiberger, "Model Checking Concurrent Programs for Autograding in pseuCo Book," en, in *Proc. 5th Intl. Workshop on Formal Methods (FMTea)*, C. Dubois and P. San Pietro, Eds., Lübeck, Germany: Springer Nature Switzerland, Mar. 2023, pp. 51–65.

[26] P. Mauran, P. Queínnec, and X. Thirioux, "Teaching Transition Systems and Formal Specifications with TLA+," in *TLA+ workshop at the 18th international symposium on formal methods, Paris, France*, 2012.

[27] J. N. Reed and J. E. Sinclair, "Motivating Study of Formal Methods in the Classroom," in *Teaching Formal Methods*, C. N. Dean and R. T. Boute, Eds., Springer Berlin Heidelberg, 2004, pp. 32–46.

[28] J. Noble, D. Streader, I. O. Gariano, and M. Samarakoon, *More Programming Than Programming: Teaching Formal Methods in a Software Engineering Programme*, arXiv:2205.00787 [cs] version: 1, May 2022.

[29] N. Cataño, *Engaging Millennials into Learning Formal Methods*, arXiv:1806.03527 [cs] version: 1, Jun. 2018.

[30] M. Askarpour and M. M. Bersani, "Teaching Formal Methods: An Experience Report," en, in *Frontiers in Software Engineering Education*, J.-M. Bruel, A. Capozucca, M. Mazzara, B. Meyer, A. Naumchev, and A. Sadovykh, Eds., Cham: Springer Intl. Publishing, 2020, pp. 3–18.

[31] J. P. Gibson, "Formal methods — never too young to start," in *Formal Methods in Computer Science Education (FORMED 2008)*, Z. Istenes, Ed., Budapest, Hungary, Mar. 2008, pp. 151–160.

[32] A. Yadav, N. Zhou, C. Mayfield, S. Hambrusch, and J. T. Korb, "Introducing computational thinking in education courses," en, in *Proc. 42nd ACM Technical Symposium on Computer Science Education*, Dallas TX USA, Mar. 2011, pp. 465–470.

[33] J. F. Ferreira, *FME Education Course Database*, Maintained by the Formal Methods Teaching Committee, Oct. 2019.

[34] M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser, "SE 2014: Curriculum guidelines for undergraduate degree programs in software engineering," *Computer*, vol. 48, no. 11, pp. 106–109, 2015, Publisher: IEEE Computer Society.