Abstract—All future advances in computational capabilities will involve the use of parallel programming methods involving millions of tasks/threads running on multicore CPUs. Unfortunately, parallel programs are very difficult to reason about and debug using informal approaches that are often ad hoc as well. Our work addresses the challenges of incorporating formal approaches for parallel programming in existing CS curricula. We aim to cover a core set of principles that will be found relevant and valuable across a wide spectrum of concurrency paradigms. Our main contribution is a collection of hands-on practice software modules written in the Python language that will help enliven existing formal methods topics in CS, and build our proposed formal methods topics as extensions to this framework. We believe that this approach can be picked up and taught even by faculty whose primary research focus is not formal methods. We present a list of challenges all of which, we believe, can be overcome through collaborative effort that adds to our software module collection.

Keywords-Parallel/concurrent programming, and education; multicore computing; teaching parallelism concepts; formal methods/debugging tools.

I. INTRODUCTION

There is already a crisis in our ability to teach the formal foundations of computing. Despite the hard work of thousands of teachers who pour their hearts into teaching formal foundations of computing (that includes topics in discrete structures, automata, and logic):

- the learning/teaching is passive—emphasizing one’s ability to draw automata and think about their behaviors
- topics are not well connected—for instance, automata and logic are seldom shown to be tightly related
- the topics are never shown as “powertools” of modern reasoning. For instance, automata are often taught to highlight their use behind lexing and parsing (but not formal reasoning); and mathematical logic is seldom introduced in the context of modern SAT and SMT solvers that the industry has begun using for software analysis, test generation, and synthesis.
- Many opportunities to connect topics are lost—for instance, teaching SAT solving in under 100 lines of Python can vastly enhance a student’s appreciation for NP-completeness. Most faculty involved in teaching these topics don’t have access to books that readily provide such pedagogical material.

With parallelism/concurrency coming to the fore, we have a crisis at hand. It is insufficient to explain away crucial notions such as data races or volatiles (as in Java), nor teach crude software testing using random explorations of the execution space (as opposed to using partial-order reduction search or SMT-solving based concolic execution). We offer a structured approach to teach the formal foundations of these topics while not disrupting existing curricula. The “extra” material we need in our approach (e.g., teaching the basics of Python) address other well-recognized curricular needs.

A. Proposal-1: Make Students Teach on Day-1

If what we teach along the lines of “formal foundations” (discrete mathematics, graphs, automata, Turing machines, logic) can be supported using program modules in a notionally compact and widely accessible language such as Python, then students instantly become teachers: for instance, instead of reading about DFA, they will be teaching a computer how to express and manipulate DFA. This role-reversal is amazingly effective, as already demonstrated in legendary textbooks such as “The Structure and Interpretation of Computer Programs” (Sussman) where scores of CS concepts are taught by students to computers in Scheme! Instead of passively reading about NFA to regular-expression conversion, they will be coding these from scratch (or assembling a kit of “parts” to arrive at the code). They will immediately confront many fun facts and opportunities, learning a slew of topics on the side:

- We highly recommend the use of Python as the vehicle for these software modules—a choice already popular at many schools (not just in CS, but also in Bioinformatics, Scientific Computing, etc.)
- This allows students to learn scripting and declarative/functional-style programming—two skills that are essential for modern programmers.
There are many built-in teaching moments; for instance, when they try to parse regular expressions, they will realize that their grammar is context-free; when they try to parse context-free productions, they will realize that their grammar is regular.

They will be able to walk the loop of conversions (RE to NFA to DFA to minimal DFA, back to RE, ...), watch the often touted exponential size growth, etc.

One can teach all of “automata theory,” with details often glossed over actually confronted and solved by the students (e.g.: what configurations do you store in a hash-table to combat the potential non-termination of push-down automatons?)

Last but not least, they retain what they learn. It also does not take away from “maturity” they supposedly acquire by merely using their brains and paper/pencil (the programming components can be assigned as self-study material, with the conceptual chalk-board/pen/paper combo used in class).

B. Proposal-2: While at it, Slip in Formal Methods Topics

The above approach to rejuvenate existing “foundations” curricula immediately pays off. We can incorporate several topics as extensions to the basic fabric that we sketch above:

- Binary decision diagrams can be readily introduced as minimized DFA for the characteristic set of a Boolean formula.\(^1\)
- This immediately allows one to teach other uses of BDD (Knuth\(^1\) calls BDDs “one of the most important of data structures to emerge over the last 25 years”)
- One can introduce several topics in mathematical logic (Boolean satisfiability, how modern SAT solvers work, SMT solving, etc.) using these Python modules.

C. Proposal-3: Now Slip in Formal Parallelism Topics

With these course modules available, one is a few steps away from reaching into parallelism/concurrency topics. Some of this material can be taught using the Python modules while others may be best taught using dedicated formal reasoning tools:

- With a good introduction to NP-completeness, one can discuss the formal underpinnings of cache coherency quite naturally through reduction proofs from SAT.\(^3\)
- While on this topic, one can discuss memory consistency models.\(^4\)
- One can easily illustrate state-space traversal methods encoded using BDDs, thus introducing the notion of symbolic model checking

One can introduce Büchi automata and temporal logic, thus introducing explicit-state model checking

One can invoke SMT solvers from Python, learn about constraint solving, and understand how modern concolic testing methods can generate high quality test inputs for programs

Specialized tools for verifying MPI and CUDA programs can now be introduced (formal analysis tools for these concurrency models are distributed from the authors’ website).

II. WHAT WE HAVE: WHAT MORE IS NEEDED

We now detail how much of the above project has been finished, and the path ahead (where you may be able to help). Unless stated otherwise, all of this software is written in Python:

- We have developed a collection of automata-theoretic routines in Python and class-room tested them.
- Material on Büchi automata and linear-time temporal logic model checking are planned.
- We have a very compact BDD package that is being class-room tested now.
- We have a basic Boolean SAT implementation within 100 lines, and a modern DPLL SAT with clause learning and backjumping in around 600 lines.
- We have an interface to the Microsoft Research Z3 solver. Using it, we have encoded a number of popular puzzles (such as Sudoku and KenKen).
- We have developed an SMT-based verification methodology for CUDA programs, using a canonical scheduling method that is essentially a symbolic version of dynamic partial order reduction.\(^5\)
- Pedagogical material detailing the algorithms of CUDA formal analysis are planned.
- We have developed MPI formal analysis tools, making them available within the Eclipse Parallel Tools platform. Pedagogical material detailing this is planned.

III. CONCLUDING REMARKS

We propose to rejuvenate the teaching of foundational topics in CS using programming modules that emphasize a declarative programming style, and are based on the widely accessible language Python. We describe how such a curriculum can minimize the need for textbooks, and allow faculty to readily “grab and teach” many topics important for the era of parallelism that we are in. Our software is available upon request, and is being readied for wider distribution.


\(^3\)See “You Don’t Know Jack About Shared Variables or Memory Models,” by Boehm and Adve, Communications of the ACM, February 2012.