An Integrated First-Year Curriculum for Computer Science and Computer Engineering

David Cordes, Allen Parrish, Brandon Dixon, Richard Borie, Jeff Jackson and Patrick Gaughan
University of Alabama
Tuscaloosa, Alabama 35487

Abstract *- The University of Alabama is an active participant in the NSF-sponsored Foundation Coalition, a partnership of seven institutions who are actively involved in fundamental reform of undergraduate engineering education. As part of this effort, the University of Alabama has developed an integrated first-year curriculum for engineering students. This curriculum consists primarily of an integrated block of mathematics, physics, chemistry, and engineering design. The engineering design course is used as the anchor that ties the other disciplines together.*

While this curriculum is highly appropriate (and successful) for most engineering majors, it does not meet the needs of a computer engineering (or computer science) major nearly as well. Recognizing this, the Departments of Computer Science and Electrical and Computer Engineering recently received funding under NSF's Course and Curriculum Development Program to generate an integrated introduction to the discipline of computing.

The revised curriculum provides a five-hour block of instruction (each semester) in computer hardware, software development, and discrete mathematics. At the end of this three-semester sequence, students will have completed the equivalent of CS I and CS II, a digital logic course, an introductory sequence in computer organization and assembly language, and a discrete mathematics course.

The revised curriculum presents these same materials in an integrated block of instruction. As one simple example, the instruction of basic data types in the software course (encountered early in the freshman year) is accompanied by machine representation of numbers (signed binary, one and two's complement) in the hardware course, and by arithmetic in different bases in the discrete mathematics course. It also integrates cleanly with the Foundation Coalition's freshman year, and provides a block of instruction that focuses directly upon the discipline of computing.

Introduction

The University of Alabama is undertaking an effort to revitalize its freshman year in computing. This project, sponsored in part by NSF under the Course and Curriculum Development (CCD) program, is designed to institute educational reforms in our first-year computer science and computer engineering curricula. These reforms principally involve implementing an *integrated* freshman core curriculum common to computer science and computer engineering majors. This integrated curriculum will heavily involve the coordination of content and assignments for its courses. We will utilize "just-in-time" instruction [5,6], where topics are introduced in given courses at precisely the appropriate time for those topics to be applied in other, concurrent, courses.

Our goal of curriculum integration allows us to utilize results from the *Foundation Coalition*, an NSF-funded coalition of seven institutions, including The University of Alabama, dedicated to the improvement of engineering education. For several years, the Foundation Coalition has engaged in engineering curriculum development activities in three focus areas: curriculum integration, active learning, and technology-enabled problem solving. Most notably, the Foundation Coalition integrated mathematics, physics, engineering design and chemistry from the engineering freshman year. Our goals for integration are similar. We are simply substituting the disciplines of mathematics, physics, chemistry and engineering design with computer software, computer hardware, and discrete mathematics.

The Status Quo

At the University of Alabama, the computer science major is housed in the Department of Computer Science (CS), while the computer engineering major is housed in the Department of Electrical and Computer Engineering (ECE). There are approximately 150 computer science and 150 computer engineering majors. The two majors are distinguished by the proportion of hardware to software exposure, as follows.

Computer science majors:

- The complete computer science core (CS I and II, data structures, software engineering I and II, programming languages, operating systems, algorithms);
- A limited computer engineering core (digital logic, assembler, and computer organization and design);
- Various advanced computer science electives

Computer engineering majors:

- The complete computer engineering core (circuits, digital and analog electronics, digital logic, assembler and computer organization and design);

- A limited computer science core (data structures, software engineering I and II);
- Various advanced computer engineering electives.

Although computer engineering majors do not currently take CS I and II, they are instead required to complete two introductory programming courses which are oriented toward engineering applications. Currently, the intersection of our curricula results in a common core as follows:

- *Mathematics*: Calculus I, II and III, Linear Algebra, Discrete Mathematics;
- *Computer Science*: Data Structures, Software Engineering I and II;
- *Computer Engineering*: Digital Logic, Assembler, Computer Organization and Design.

The above "common core," while representing a useful set of courses for computer scientists and engineers, did not provide a common foundation for freshman students in the two disciplines. Moreover, quite the opposite is currently true: the computer engineering freshman year consists entirely of computer engineering courses, while the computer science freshman year consists entirely of computer science courses. Thus, students are entering the above core courses with substantially different backgrounds.

The New Course Sequence

To address the problems associated with the current system, we are constructing a common freshman year for computer science and computer engineering students. This freshman year is built around a series of five-hour courses on computing that includes: a hardware (computer engineering) sequence, a software (computer science) sequence, and a mathematics sequence. The computer science and computer engineering sequences are typical introductory sequences in these disciplines; details are provided below. However, we deviate significantly from classical norms with the mathematics sequence. In particular, both computer science and engineering students currently take calculus first, during the freshman year. While we agree with the arguments that the rigor of calculus provides a good mathematics foundation for our students, we do not feel that calculus provides the best domain material for computer science and engineering students. Instead, we feel that discrete mathematics provides much more useful domain knowledge for our students, and early exposure to this material should promote a better understanding of fundamental computer science and engineering concepts. This view is confirmed by textbooks such as [1,8] that are intended for freshman and sophomore computer science students; textbooks such as these will be very useful in assisting in this effort.

Thus, our proposed effort involves teaching discrete mathematics, rather than calculus, as part of the common freshman year. Calculus will still be required, but will not be mandated during the freshman year. We believe that the most novel aspect of our efforts is integration of these topics into a single course. The idea is that the three basic topics (software, hardware, discrete mathematics) are taught concurrently and are coordinated so that related topics are introduced together. To help define the organization of each semester (and illustrate the course integration), we have identified the following coordinated set of topics for each group of courses:

Table 1: First Semester Courses

Week	*Software*	*Hardware*	*Discrete Mathematics*
1-2	Concepts of programming, Prog. language introduction	Concepts of digital representation	Binary systems, Boolean algebra
3-7	sequence, selection, and iteration (S/S/I)	Hardware analogs of S/S/I (high-level data paths, micro-program execution)	Predicate & propositional calculus, Induction, proofs
8-10	functions, procedures	Combinational Logic	Sets, functions, relations
11-13	data types (arrays, records, strings, files)	Memory organization, Data storage, Address decoding	Big-O notation, Combinatorics
13-15	software design & testing	Hardware design & testing	Algorithm analysis, more proofs (proving basic properties of algorithms)

Thus, we are exploiting a number of relationships among these areas. For example, digital logic courses often traditionally re-teach binary number systems and Boolean algebra, topics that should have been covered in the discrete mathematics course. With this approach, material in these areas is taught properly within the mathematics course at precisely the time that it is needed for the computer engineering course. In addition to eliminating redundancy, coordination also resolves problems associated with material being taught in the wrong order. For example, students in a traditional CS I course often have difficulty evaluating complex conditions when constructing predicates for **if** statements and **while** loops. Since our students are taught

truth tables and rules such as DeMorgan's Law at approximately the time that program conditions are introduced, we find it much easier for them to understand conditional statements in programs.

Table 2: Second Semester Courses

Week	*Software*	*Hardware*	*Discrete Mathematics*
1-3	Pointers, dynamic allocation	Instruction addressing modes	Graph theory, Directed/undirected graphs
4-6	Data abstraction, Data structures (stacks, queues, trees, lists, sets)	Assembly implementation of data structures	Trees (rooted trees, binary trees, spanning trees, weighted trees)
7-9	Memory models (static, stack-based, heap-based)	Stack frames, Segmented memory, Protected modes	Sets, functions, relations POSETs, equivalence relations, composition, closure
10-12	Recursion	Assembly coding of procedures & functions	Recurrence relations
13-15	Case Study: medium-scale application w/ advanced data structures	Case Study: assembly and machine level version of the CS application	Algorithm analysis Case Study: analysis of the CS application

The above coordination of topics also illustrates how relationships are exploited to give students a better understanding of how concepts fit together across discipline boundaries. A case in point is our periodic presentation of the same example. During the second semester, we develop a case study involving a program in a high-level language, show the translation of this example into assembly and object code (effectively we show the material at the machine level), and conduct a formal efficiency analysis of the algorithms used. This allows students to anchor related concepts in the same problem. Similarly, our parallel discussions of software and digital design toward the end of the first course allow students to visualize certain commonalities and parallels.

Nonetheless, coordination is really the foundation of our efforts, and to further illustrate the coordination, we conclude this section with a detailed example of all of the activities during the first two weeks of the first course. This example better illustrates the level at which we conduct coordination activities on a daily basis.

This table illustrates two distinct types of coordination that we are utilizing. First, the idea of just-in-time delivery of concepts is illustrated on Day 1 with respect to the hardware and discrete mathematics. Since the math is providing a number of concepts to support the hardware and software concepts, we provide a "just in time" approach to the discrete mathematics topics. Thus, on Day 1, bases are introduced. Students then use this information in the hardware discussion where they apply the concept of binary numbers (something they just learned) as an abstraction of digital signals. Note that just-in-time delivery does not have to occur on the same day; we are also utilizing just-in-time delivery of Boolean algebra concepts during Week 2; this material is utilized during Week 3 when students learn about conditions in "**if**" statements.

Table 3: Detailed Look at the First Two Weeks

Day	*Software*	*Hardware*	*Discrete Mathematics*
Mon. *Day 1*	Basic program concepts and constructs	Digital signals Unsigned fixed length binary numbers	Bases (10, 2, 8, 16) Base conversion
Wed. *Day 2*	Simple I/O Integer variable declarations/expressions Simple assignment	Signed binary numbers Signed magnitude One's complement Two's complement	Base X arithmetic (`+, -, *, /`)
Fri. *Day 3*	Characters Simple programs involving integers & characters	ASCII Gray codes Excess codes BCD	
Mon. *Day 4*	Floating-point variables	Bases Fractional forms	Boolean algebra Basic operators & axioms
Wed. *Day 5*	More on assignment Operators & precedence	Floating-point representation	Tautologies, Contradictions, Boolean expressions
Fri. *Day 6*	Wrap-up; lots of examples involving straight-line code	Floating-point representation, Machine precision	

A second type of coordination is observed on Days 2 and 3 with respect to computer hardware and software.

Students are learning about integer variables at the same time they are learning about integer representation (Day 2); the same concept is true with respect to character variables and ASCII representation on Day 3. This is not just-in-time delivery, but is instead learning about the same concept at different levels of abstraction. We have found that, in general, the hardware and software components are coordinated most frequently using this technique, while the coordination with discrete mathematics topics tends to be of the just-in-time variety.

Active Learning Emphasis

We also utilize the concepts of active learning in constructing classroom materials for this effort. Our philosophy behind active learning involves students actively taking responsibility for their own learning, as opposed to being in the position of a mere "recipient" of instruction. We utilize two techniques in this area: *discovery learning* and *cooperative learning*.

Discovery learning involves motivating students to "discover" problem solutions themselves. One application of discovery learning to computer science involves providing the students with data and a problem to solve, but no algorithm. The students are then given an opportunity to find the algorithm themselves. Current research shows that students who discover a concept by themselves remember it much longer and find it easier to transfer the concept to other problem-solving situations than students who are just presented with the concept through classroom lectures [2,4,5,7].

The second strategy employed within the new curriculum is cooperative learning, which has demonstrated pedagogical benefits relative to traditional learning environments [5,9]. Students interact and collaborate in ways that require each individual to meet stated competency goals. With cooperative learning, student questions and answers stimulate involvement, multiple perspectives help individuals to grasp concepts, and peer accountability motivates team members.

Our notion of cooperative learning involves formalized classroom group activities and out-of-class group projects. For example, discovery exercises are presented that can be solved by individuals or groups. Ideally, cooperative learning rewards both individual accountability and positive interdependence among group members. Individual accountability should be guaranteed through individual testing and peer grading; the effect of peer pressure in this setting has been well documented and should not be underestimated [3]. Positive interdependence simply means that group members, although individually accountable, must also be able to depend on other group members for positive reinforcement. This can be enforced by mechanisms such as giving a bonus on tests to groups where all members of the group score above some threshold [5].

Throughout the proposed curriculum, we intend to structure all class meetings so that we eliminate the traditional lecture, and to follow a specific protocol involving discovery and cooperative learning. This protocol is structured around the four-quadrant learning cycle [6], and is organized as follows for a typical 50-minute class meeting:

- **Quadrant #1**: *Why study this*? (Motivate the topic, 5-10 minutes). Establish a "feel" for the subject to motivate the material to be covered during this class. This can include stories, classroom demonstrations, simulations, discussion, and group problem solving. The instructor serves as motivator for the topic.
- **Quadrant #2**: *What is it*? (Inform the students, 15 minutes). Present the basic materials necessary for the students to utilize this concept. The instructor serves as expert on the topic during this phase, and delivers a "mini-lecture."
- **Quadrant #3**: *How does it work*? (Apply this principle, 15 minutes). Present problems and exercises designed to ensure the students are capable of working with the material. Many of these exercises will be discovery exercises, although not always. Design the exercises to promote interaction between team members. The instructor serves as coach during this period.
- **Quadrant #4**: *What if*? (A self-discovery process as students learn how to apply this topic, 10 minutes). This stage is designed for the students to start applying the knowledge learned, where the previous stage simply focused on the problem-solving technique itself. The instructor serves as evaluator/remediator.

All classes are structured in this fashion. In addition to appealing to sound pedagogical foundations, this structure provides the student with a classroom environment that is constantly changing. A large part of any freshman course is capturing, and keeping, the students' attention. We have found the above format works extremely well in this regard.

Summary

The curriculum described in this paper represent a common 15-hour freshman year for computer science and computer engineering. Institutionalization plans are under development to replace the current EE digital logic and assembler courses (which are intermediate courses in the existing curriculum) with this sequence.

This project presents a number of unique benefits. First, we know of no computer science and engineering curricula where discrete math is taught before calculus (although some probably do exist). Our curriculum materials provides a model for other programs to follow when considering and

implementing this type of reform. Second, we feel that the areas being integrated (software, hardware and mathematics) form the foundation for both computer science and computer engineering. The possible methods for developing connections between these areas seem endless, and our model provides a basis for other institutions to explore these interconnections.

Additionally, although there is a progression toward active learning in education, the practice of active learning techniques is far from ubiquitous. By scripting each active learning based class meeting in detailed fashion, we are able to provide a suite of materials to be used in a variety of settings in teaching introductory computer science and engineering. Even if our scripts are not used in detail, the concept should provide a useful model for institutions seeking to adopt active learning techniques.

References

1) Aho, A. and J. Ullman, *Foundations of Computer Science*, Freeman, 1992.
2) Bransford, J.D., J.J. Franks, N.J. Vye, & R.D. Sherwood, "New approaches to instruction: Because wisdom can't be told," *Similarity and Analogical Reasoning*, S. Vosniadou and A. Ortony, eds., Cambridge Univ. Press, Cambridge, England, pp. 470-497, 1989.
3) Brown, A. & A. Palinscar, "Guided cooperative learning and individualized knowledge acquisition," In Resnick, L.(ed.), *Knowing, Learning, and Instruction*, Hillsdale, NJ: Lawrence Erlbaum Associates, 1989.
4) DIMACS, "In Discrete Mathematics: Using Discrete Mathematics in the Classroom".
5) Felder, R., "Reaching the Second Tier: Learning and Teaching Styles in College Science Education," *Journal of College Science Teaching*, Volume 23, Number 5, pp. 286-290 (1993).
6) Felder, R.M. and Silverman, L.K., "Learning and Teaching Styles in Engineering Education," *Engineering Education*, vol. 78, no. 7, April 1988, pp. 674-681.
7) Fellows, M. R., "Computer Science and Mathematics in the Elementary Schools'," *Report on the Megamath Project of the U.S. National Laboratories in Los Alamos, New Mexico*, 1991.
8) Gersting, J., *Mathematical Structures for Computer Science*, Third Edition, Freeman, 1992.
9) Johnson, D.W., R. Johnson, and K. Smith, *Active Learning: Cooperation in the College Classroom*, Edina, MN: Interaction Book Company, 1991.
10) Wineke, W.R., *et.al.*, *The Freshman Year in Science and Engineering: Old Problems, New Perspectives for Research Universities*, Alliance for Undergraduate Education, Ann Arbor, MI.