CDM: Teaching Discrete Mathematics to Computer Science Majors

KLAUS SUTNER
Carnegie Mellon University

CDM, for computational discrete mathematics, is a course that attempts to teach a number of topics in discrete mathematics to computer science majors. The course abandons the classical definition-theorem-proof model, and instead relies heavily on computation as a source of motivation and also for experimentation and illustration. The emphasis on computational issues is particularly attractive to computer science majors and increases their involvement and participation.

Categories and Subject Descriptors: K.3.1 [**Computers and Education**]: Computer Uses in Education; G.2.1 [**Discrete Mathematics**]: Combinatorics; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems

General Terms: Theory, Algorithms.

Additional Key Words and Phrases: Experimental mathematics, symbolic computation, mathematics education.

1. HISTORY AND MOTIVATION

In the mid-1990s, Dana Scott decided that the time was ripe to bring modern computational software into the computer science curriculum at Carnegie Mellon. He started the "ModMath Project" whose goal was to systematically exploit software to teach discrete mathematics to computer science majors in a novel way that was custom-designed for their specific needs. The central idea was to push computation into the foreground, both by presenting mathematical content from a computational point of view and by using computer algebra as a mathematical sandbox in which the students could experiment, find examples and counterexamples, and sharpen their intuition.

The classical definition-theorem-proof cycle is replaced by a different loop that begins with some problem or question, based on the student's current state of mathematical knowledge. Computational examples and experiments lead to a better understanding of the problem and, hopefully, to ideas and conjectures on how the issue might be tackled. The conjectures are then rigorously proven, added to the current level of knowledge, and the cycle then begins anew. Of course, some of the ideas will often not work out as planned, and there will be some back-and-forth between conjecture-formation and proofs, always with the potential aid of computation as a source of inspiration, examples, and counterexamples. The role of a proof in our framework is two-fold: First, it helps to ensure that the observations hold in general and are not just some peculiar coincidence. Second, the proof should provide some insight into why the observations hold, insight

The author is with the Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue Pittsburgh, PA 15213-3890
Author's address: sutner@cs.cmu.edu

that could not be gained by just observing the data. This empirical and experiential approach to mathematics can help greatly in coming to a solid understanding of the relevant concepts. Moreover, it is fun to make small discoveries of one's own and then try to formalize and establish them in a rigorous way.

One example from the trenches: some years ago students were asked in a homework assignment to determine the number of binary sequences of length n that contain no two consecutive 0's. The students knew how to tackle simple recurrence equations, and in particular were familiar with Fibonacci numbers. However, no further hints were given as to how the problem might be solved. Most students used a program to compute the number of such 00-free sequences for a few small values of n by brute force, then correctly conjectured the answer (the Fibonacci number F_{n+2}) and gave a proof by induction. Brute force here means that one first generates all binary sequences of length n and then eliminates the ones containing 00, a very simple task in our standard computational environment. Indeed, in Mathematica, augmented by a macro package for the course, the whole program can be written as a "one-liner":

```
ZZfree[n_]:=
  DeleteCases[Tuples[{0,1},n],{___,0,0,___}];
```

While this solution is perfectly acceptable, and indeed what the instructor intended, a few students worked a bit harder initially and wrote a recursive program that directly generates only 00-free sequences. The crucial insight here is that sequences of length n can be obtained by appending suffixes 1 and 10 to shorter sequences of lengths n - 1 and n - 2, respectively. The induction proof is then essentially replaced by a correctness argument for the generating program: one has to make sure that all 00-free sequences are so generated and that there are no duplicates. The connection between proofs and programs – where the latter notion is perfectly palatable to the students, while the former often is not – is one of the major themes of the course.

For a number of years, D. Scott, E. Clarke, P. Miller, and I taught a variety of ModMath courses at an introductory level [Miller and Sutner 1997] Alas, the students' response was decidedly mixed: while the stronger students embraced the new technology and used it to great effect, a large number of weaker ones felt intimidated and overwhelmed by the machinery. For them, the extra burden of having to learn how to cope with a specific computing environment never seemed to be quite compensated for by the benefits that all of us envisioned. Ultimately, the project went into hibernation and the curriculum returned to its traditional patterns. Predictably, the computer science majors resumed their standard gripes about too much abstraction and a general lack of connection between the mathematics and their core interests.

Motivated by the enormous progress in symbolic computing software, and in particular the emergence of document-centric computing environments, we decided in 2002 that another attempt should be made to redesign the discrete mathematics component of the computer science curriculum. We created a new course, titled CDM, for computational discrete mathematics. Unlike its ModMath predecessors, CDM is not required, but is offered as a choice towards the fulfillment of the mathematics requirement. The students are typically in their second year and have taken two prerequisite courses in discrete mathematics. The course is presently capped at 20 students, most of whom are computer science majors, though some mathematics majors also enrolled (and acquitted themselves very well). CDM will be in its fourth incarnation in Fall 2005.

In a drastic change from previous policies, the use of computing equipment is also no longer tightly controlled. By the time they take CDM, the computer science majors at Carnegie Mellon tend to be extremely opinionated about the relative merits and demerits of various systems, programs, and languages. There appears to be little benefit in forcing them into one particular environment when they are already comfortable and experienced in another. Specifically, from the perspective of the instructor, while Mathematica [Wolfram 2002] is used as the main workhorse in the course and some of the supplementary material is posted in the form of Mathematica notebooks, there is no requirement for its use. In the past, students have successfully employed C, C++, Java, Perl, and Maple to tackle the computational aspects of various assignments. As a matter of fact, not only is there no requirement for the use a specific software system, there is not even a requirement for the use of computation at all. The students are free to handle the given assignments any way they like – as long as the results are elegant and correct. In principle, a purely theoretical approach might succeed, though it would often be too time-consuming and tedious for serious consideration. The students are quick to realize this, and choose their weapons accordingly. Note that this freedom for students places a burden on the course staff, in particular in office hours and in the assessment of student work, but in our experience the results are well worth the effort.

One last comment about the role of programs in CDM. It was pointed out by B. Thurston in his thoughtful article [Thurston 1994] that writing correct programs is a difficult task, more difficult in some ways than pure mathematics:

> The standard of correctness and completeness necessary to get a computer program to work at all is a couple of orders of magnitude higher than the mathematical community's standard of valid proofs. Thurston, 1994

We largely agree with this assertion, though we believe that it actually works in our favor in this course. Unlike Thurston's work in low-dimensional topology, the computations required in CDM are mostly rather basic. Much of the burden of writing correct programs is alleviated by the ready availability of functioning, well-tested code for these fundamental operations. Hence, the challenge Thurston refers to can actually be exploited in a constructive way: dealing with programs and the results of their computations produces better insight into the underlying mathematical principles. Rather than removing precision and careful reasoning it emphasizes them. It leads to a concrete, experiential and integrated understanding of the concepts and their applications, rather than the more abstract, mechanistic, and isolated approach fostered by the traditional definition-theorem-proof method.

2. EXPLOITING COMPUTATION

In this section we will give a few characteristic examples that highlight the CDM approach to discrete mathematics. Most of the material is of considerable sophistication and would be difficult to transmit in a more traditional setting. A more complete listing of topics can be found at the website, complete with lecture slides and accompanying material [Sutner 2004]. While some of the material at the site can be expected to undergo changes and adjustments, the core of the course is now stable. At any rate, it will give a good indication of the philosophy of the course.

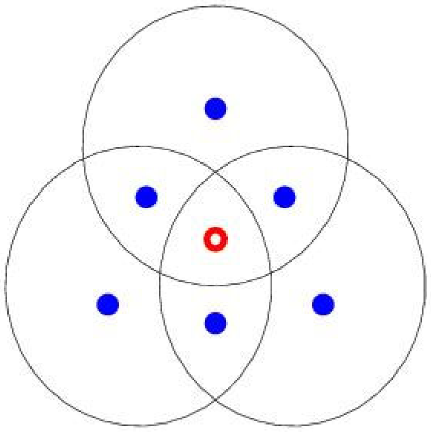

Fig. 1.

2.1 Closure Operations and Invariants

Here is a simple puzzle based on two colored pebbles that are placed inside three overlapping circles, as shown below. Originally, all pebbles are blue-side up, and the goal is to determine whether one can bring them into the configuration shown, where the central pebble is red-side up.

The only permissible operations are to flip all four pebbles in one circle over, or to arrange all pebbles in one circle red-side up. Since there are only 2^7=128 possible configurations, it is easy to compute by brute force all reachable configurations and to verify that the target configuration is not among them.

However, the assignment requires an explanation, not just a yes or no answer. So the program alone will not do: it provides no insight whatsoever as to why the target configuration fails to be reachable. In the second step, it is necessary to inspect the reachable configurations and to determine some appropriate invariant: some property that holds the initial configuration is preserved by the admissible operations and is violated by the target configuration. A slightly more ambitious project is to give a full characterization of the reachable configurations, a task that seems to require some linear algebra over the two-element field.

The operation of flipping a collection of pebbles easily leads to a class of related but more difficult problems. For example, we can think of the nodes of an arbitrary graph as the pebbles, and “flipping” one node will affect not just the node but also all its neighbors. Given some starting configuration, we would like to understand all reachable configurations. Even if we limit our attention to grid graphs, the questions become quite complicated. The nodes of the graph here are the squares of a generalized chessboard and each square is considered adjacent to its north, south, west, and east neighbor. In the following picture, dark squares indicate which pebbles have to be flipped to turn the all-blue configuration on an 8 by 8 chessboard to an all-red configuration. Note how the nodes are chosen so that each node of the graph has an odd number of neighbors, including itself, in the chosen set.

Needless to say, the solution is not found by searching through all possible 2^{64} configurations but is based on, first, a reduction of the search space to dimension 8 and, second, the use of linear algebra on the smaller space to avoid search altogether. The

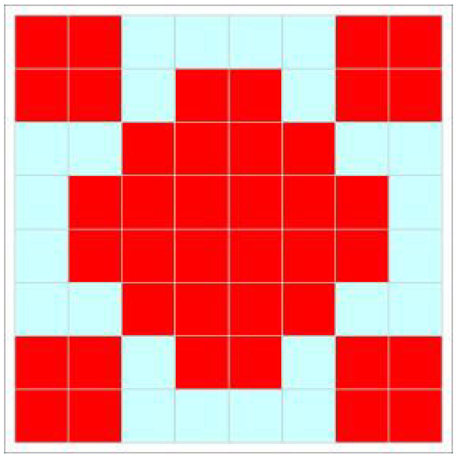

Fig. 2.

solution is essentially unique in this case. Determining the number of solutions for other grids leads to rather interesting problems related to divisibility properties of Fibonacci polynomials over the two-element field [Sutner 1988].

2.2 Pólya and Redfield

Students in CDM all have encountered basic group theory in previous courses, but seem never to have found the topic particularly relevant. A concept that they are better acquainted with, and more appreciative of, is the representation of equivalence relations in a way suitable for computation. For example, they are all familiar with the use of the union/find algorithm to construct mazes. We connect the two ideas and use symmetry groups to limit the size of search spaces in some games such as checkers or *nine men's morris*, apparently an excellent motivation for computer science majors. Group actions are introduced as yet another way to represent certain types of equivalence relations, albeit in a somewhat abstract fashion. It comes as a pleasant surprise to most students that the abstraction has an immediate computational pay-off: otherwise difficult questions about the number of equivalence classes can be reduced to questions about polynomials. The brute force approach, where all possible configurations are generated first and the equivalent ones identified later, becomes demonstrably useless for all but the smallest examples – though it does help greatly to build up some intuition for the basic ideas in Pólya-Redfield counting.

The students are given access to software that performs the necessary cycle-decomposition of elements of a permutation group and generates the cycle index polynomials. For example, the invocation

```
P4 = CycleIndexPolynomial[DihedralGroupSG[4]];
SubstitutePolynomial[ P4, 6, 2, Full -> True ];
```

will duly produce the polynomial,

$$\frac{1}{8}\left((c_1+c_2)^4+2(c_1+c_2)^2\left(c_1{}^2+c_2{}^2\right)+3\left(c_1{}^2+c_2{}^2\right)^2+2(c_1{}^4+c_2{}^4)\right).$$

In class, only two-dimensional examples such as the dihedral group above are discussed. For the assignments, the students are required to deal with symmetries in three-space. In particular, they have to discover the structure of the octahedral group and find a nice set of generators and relations. The actual calculations are easy to carry out – if one has the ability to manipulate polynomials of several hundred terms. Somewhat unexpectedly, a number of students elected to use systems other than our standard environment for polynomial arithmetic. In fact, some even implemented the requisite symbolic algorithms on their own, exploiting work from other courses.

There is some anecdotal evidence for the strength of this approach: after last year's notes were posted on the web, a researcher in the Netherlands working on a sophisticated checkers problem sent email to inquire about a number of problems in the actual implementation of search-space-reducing techniques. Some of the questions will be handed-off to the students in the next incarnation of the course.

2.3 Logic and Provers

Finding convincing motivation for the use of a formal languages such as predicate logic and formal systems that describe and regulate proofs is always a major challenge in any discrete mathematics course. While most students would concede that Fermat's original description of his infamous theorem, as follows:

> There do not exist four numbers, the last being larger than two, such that the sum of the first two, both raised to the power of the fourth, are equal to the third, also raised to the power of the fourth.

is inferior to, say,

> For no positive integers x, y, z, and n, where $n>2$: $x^n+y^n=z^n$.

very few would agree that the step to

$$\neg\exists x, y, z, n(n > 2 \wedge x^n + y^n = z^n)$$

provides any further advantages.

However, the last version is an expression in a formal language, albeit one whose validity over the natural number turned out to be exceedingly difficult to verify. But for many other, less complicated assertions, perhaps over other domains, validity can be established with the help of programs that operate on formal expressions. Indeed, (semi-) automatic verification of hardware and software systems using formal specifications in appropriate logics has become a major industry, and commercial programs use rather sophisticated tools from proof theory and model theory [Huth and Ryan 2000].

As it turns out, even in propositional logic there are some challenging computational problems to be solved. In CDM we begin with a discussion of various normal forms (negation normal form, conjunctive and disjunctive normal form) and algorithms that convert a given propositional formula into these forms. We then introduce the Davis-Putnam algorithm and discuss a few of the more obvious heuristics that can be used to guide the selection of appropriate literals in the splitting step. It is remarkable that even a simple implementation of the algorithm in a high-level environment such as Mathematica can typically handle formulas with thousands of variables. It also provides an opportunity to put the theory of computational hardware into perspective.

Of course, more powerful tools are needed for predicate logic, and the algorithms tend to be burdened with numerous technical details. We use Analytica II, a natural deduction-based theorem-prover that exploits algebraic simplification to establish results in what might loosely be called 19th century mathematics. Analytica produces eminently human-readable proofs, with formulas typeset in near-publication quality [Bauer et al.

1998; Clarke et al. 2003]. For example, the system can establish the Bernstein approximation theorem with only minimal help from the user. More importantly, Analytica has extensive tracing facilities that make it possible for students to observe its attempts at proving a theorem. For example, the following command will produce a proof of the fact that a function *g* is surjective whenever *g*□*g* is also.

```
Prove[imp[ all[ x, some[ y, x == g[g[y] ]]],
           all[ x, some[ y, x == g[y] ]]]]
```

We hasten to point out that the purpose of using Analytica is to demonstrate the importance of formal systems. It is not intended as a mathematics tutor of any kind, and would certainly fail miserably in this capacity. It does, however, provide many examples of computational problems in propositional and predicate logic that would otherwise appear hopelessly artificial and pointless. Skolemization of formulas in predicate logic is one case in point here. In the example above, the given input formula $\forall x \exists y (x = g(g(y))) \Rightarrow \forall x \exists y (x = g(y))$ is first converted by the system into a sequent using the negative Skolemized form: $v_1 = g(g(f_{v_1})) \supset c_1 = g(v_2)$. The latter is then established by finding suitable values for the variables v_1 and v_2 while the Skolem function c_1 and f_{v_1} remain uninterpreted, a process rather similar to natural reasoning. Analytica also has a mechanism to "un-skolemize" a formula, which adds greatly to the human-readability of the generated proof. Another example is a simplification strategy in Analytica that attempts to reduce the logical complexity of a sequent by eliminating "superfluous" parts. It is a very healthy exercise to discuss implementations of this method and to rigorously establish its soundness.

2.4 Finite State Machines

Students in CDM have usually had some cursory exposure to finite state machines, mostly from the programmer's more pragmatic point of view or perhaps from some simple string-matching applications. While applications of state machines provide ample motivation, they do tend to obscure the mathematics a bit. To exhibit the mathematical

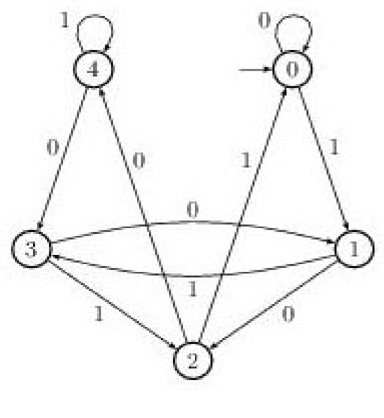

Fig 3.

structure, we start with a simple problem: Construct a deterministic finite state automaton that recognizes numbers written in base *B* that are divisible by *m*. One attractive feature of these machines is that they can be represented by labeled diagrams that are easily understandable to humans. The example below shows the automaton in diagrammatic form for *m*=5 when the input is given in binary. In algebraic form, the transition function is given by $\delta(p,a) = 2p+a \bmod 5$ where the state set consists of the modular numbers modulo 5.

The diagram is just one instance of a natural automaton that uses the modular numbers as a state set and essentially computes the value of the input modulo *m* by using a Horner scheme to evaluate the corresponding polynomial. However, in many cases the resulting machine fails to be minimal: there is a smaller DFA (deterministic finite automaton) that decides divisibility by *m* using fewer than *m* states. This effect can be observed easily in our environment by constructing the natural Horner automata and minimizing them for some values of *m* and *B*. Here is a table for $m \leq 12$ (row index) and $B \leq 16$ (column index) that demonstrates this effect. In our environment, the table can be generated easily with two lines of code.

This table may look rather forbidding at first, but a lot of information can be extracted from it. The ability to interpret data is one of the essential skills needed to profit from use of a computational environment. For example, for prime *m* it appears that the minimal machine always has size *m* unless *B* is a multiple of *m*, in which case the size drops to 2. To establish this and other observations, it is best to construct the minimal automaton by preserving the natural state set as a partition of the modular numbers. Incidentally, this provides another opportunity to discuss implementations of equivalence relations; in this case an implementation built on the notion of a kernel relation. At any rate, careful inspection of these partitions, together with some thought about the structure of the Horner automaton, leads to a connection to solutions sets for a class of modular equations and, ultimately, to a complete characterization of the minimal automaton for divisibility. Needless to say, our standard computational environment can handle the requisite modular arithmetic as easily as it deals with the finite state machines. Thus, we can solve the modular equations, use the solutions to construct a finite state machine, and then verify that the machine is indeed minimal. The seamless integration of computation in different areas of mathematics is of great importance to our approach.

Looking for generalizations we can ask about different number representations. For example, we may consider reverse base *B* where the least significant digit comes first, or Fibonacci representation. We know characterizations for some of these number systems, but mostly there are lots of interesting open problems.

3. THE AFTERLIFE

The primary purpose of mathematics-related courses in a computer science curriculum is to provide a foundation for material that will be presented later in the curriculum – material that relies heavily on this foundation and cannot be properly understood without it. Needless to say, the consumer courses may be separated from the preparatory mathematics courses by several semesters. As a consequence, it is very important that the students retain access to the material, at least during their stay at the university – although one naively hopes that they will actually make use of it beyond the acquisition of a degree. These concerns are addressed in an NSF-funded project with the suggestive title "Course Capsules: Persistent, Personalized Courseware (CCaps)"; [Kohlhase et al. 2002]. The goal of the CCaps project is to develop mechanisms that make it possible to package the whole content of a course, including student contributions, into an electronic repository that will be available indefinitely over the web for student use. This is in contrast to course management systems such as the popular Blackboard system that focus on delivery, but largely ignore archiving and long-term access. Persistent, intelligent access is a particularly pressing issue for a course like CDM which exists exclusively in computer-held form (there is no textbook) and whose content is to be applied later on in a number of other courses.

The target document format in CCaps is OMDoc, a specialized XML application that allows us to represent mathematical knowledge structures and extends the well-known OpenMath standard in several ways [Kohlhase 2002]. We have developed a number of software tools that help in the mark-up process, notably a PowerPoint to OMDoc converter, a Mathematica to OMDoc converter, and a powerful emacs mode to handle the final editing without endangering the mental health of the editor. Alas, it has turned out that the effort to perform the actual mark-up is rather significant, our tools notwithstanding, so progress has been slower than originally hoped for.

Ultimately, the students will be able to turn to their CDM capsule to gain intelligent access to the material. The semantic mark-up will provide the ability to index and search on content rather than just performing the customary string-matching. For example, all uses of the notion of "equivalence relation" will be easily identified and collated. It will be possible to extract all theorems and lemmata about equivalence relations or, alternatively, find a use for them in concrete algorithms. Moreover, the student will have access to examples and exercises that will allow her to regain mastery of the relevant concepts. The interactive nature of mathematics documents on the web is rather significant here: the definition of an unknown or forgotten term is only a few mouse-clicks away, hopefully complete with illuminating examples and further pointers. Assessment, at least at the superficial level of multiple-choice problems and simple computational exercises, can also be fully automated.

Another problem is created by the CDM computational approach itself: at least during the students' tenure at Carnegie, they should be able to rerun the sample computations in the standard, Mathematica-based environment. This seemingly innocent requirement actually causes substantial problems, as anyone who has ever tried to compile and run, say, a program downloaded from some web site will recognize. The problem of ever-changing compilers and libraries is somewhat ameliorated by using a high-level environment such as Mathematica, where changes to the language between versions are relatively minor (though the same cannot be said for the underlying mathematical algorithms). However, macro packages although written in a stable environment such as Mathematica are notoriously unreliable software artifacts, even when they come from reputable repositories such as Wolfram's Information Center. In our case, the core package contains several hundred functions dealing with combinatorics, graph theory, finite state machines, syntactic semigroups, and cellular automata. The package has grown by accretion over the years and is now rather solid; see Sutner [2002] for a recent application and Sutner [1994] for a predecessor version of the system.

In light of the persistence problem, we have developed a special packaging mechanism that will make it possible to preserve the old functionality, should changes be made in the future. The packaging system also provides for automatic testing of the code. More important for teaching applications is the extensive help browser that comes with the system: context-dependent help is available for each command, and there are numerous sample computations that demonstrate the use of the machinery, from simple one-liners to complicated applications more typical of research applications. Ultimately, we hope that web-based mathematics services will alleviate this problem, but at present local solutions are essential.

4. SOME PRELIMINARY CONCLUSIONS

As mentioned already, CDM currently has an enrollment cap of 20 students, and the course has been taught only three times thus far. One important goal of these first few rounds was to find a compelling selection of material. For example, regarding the central notion of computability, we felt obliged to provide some historical background and an introduction into the classical theory of computability. On the other hand, for the actual computations, some understanding of the more recent theory of computational complexity is necessary. Striking a balance between these competing interests without introducing inappropriate levels of technical difficulty turned out to be quite challenging. A typical student comment from last Fall seems to indicate that the proper balance has been found: "This class covers a lot of ground, but it isn't impossible to understand, and it is extremely interesting." Another central concern was to find a mode of presentation that engages the students to the point where they are prepared to invest considerable time and effort in mastering the material. Again quoting a student from last Fall: "[The instructor] could make studying the mathematics of a rock interesting."

Another positive indicator is the quality of the work submitted by the students. Overall, it is remarkably high – although we have to bear in mind that the students are self-selecting, and presumably somewhat better motivated than their peers. For example, the best solution to the characterization of reachable pebble configuration from the problem in Section 2 is due to a student in the course. Another student found a novel way to establish the fact that over the two-element field every function is polynomial. A few of the students have gone on to take more advanced courses such as constructive logic, and seem to be well-prepared, though this evidence is currently anecdotal.

To obtain a slightly less informal understanding of student response to the course, we used a standard faculty course evaluation instrument developed at the Eberly Center for Teaching Excellence [2004]**.** In a nutshell, the course ratings for CDM for the first three years were 4.52, 4.89, and 4.7, and the instructor ratings 4.61, 5.00, and 4.95, on a scale from 1 to 5. The ratings are significantly higher than those for comparable courses in our curriculum.

Overall, we feel that the first phase of our project has been completed successfully and that we can now turn to the crucial second step of turning the material into a course capsule as described in the previous section. To this end, it will be necessary to rethink and make explicit the learning objectives for each of the course topics. A more elaborate system of completed and partially completed exercises has to be developed, along with better and scalable machinery for assessment. At this point, most of the course material still exists in a strictly presentation-oriented format, and needs be converted into a semantically rich format. Since full semantic mark-up has proven to be an extremely labor-intensive task, efforts are currently under way to secure funding for a joint project with the Open Learning Initiative at CMU [Open Learning Initiative 2003]. The OLI project offers in particular a lot of experience in providing reliable, persistent web services as well as assessment and cognitive task analysis.

The most recent version of course notes and a number of notebooks are available at Sutner [2004]. The Mathematica package used in CDM is also available at the site, and alternatively at the WRI website http://library.wolfram.com/infocenter, see item 4603 in the MathSource section. The code contains extensive online help and a number of sample notebooks. Additional material such as assignments, tests, a printed manual, and various supplementary notebooks will be made available to interested parties.

REFERENCES

Bauer, A., Clarke, E., and Zhao, X. 1998. Analytica – An experiment in combining theorem proving and symbolic computation. *J. Automated Reasoning 21*, 3, 295–325.

Clarke, E., Kohlhase, M., Ouaknine, J., and Sutner, K. 2003. Analytica 2. In *Proceedings of the Calculemus Conference* (Rome) .

Eberly Center for Teaching Excellence. 2004. http://www.cmu.edu/teaching/eberlycenter.

Huth, M. and Ryan, M. 2000. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press.

Open Learning Initiative. 2003. http://www.cmu.edu/oli.

Kohlhase, M. 2002. OMDoc: An open markup format for mathematical documents. http://www.mathweb.org/omdoc

Kohlhase, M., Sutner, K., Jansen, P., Kohlhase, A., Lee, P., Scott, D., and Sudszik, M. 2002.Acquiring mathemematical content in an academic environment. In *Proceedings of the MathML Conference.*

Miller, P. and Sutner, K. 1997. Exploiting computer algebra systems in computer science courses. *ACM SIGCSE Bull. 29*, 377–378.

Sutner, K. 1988. On □-automata. *Complex Syst. 2*, 1, 1–28.

Sutner, K. 1994. Implementing finite state machines. In *Computational Support for Discrete Mathematics*, vol. 15, N. Dean and G. Shannon (eds.), DIMACS, 347–365.

Sutner, K. 2002. automata, a hybrid system for computational automata theory. In *Proceedings of the CIAA 2002 Conference* (Tours, France), J.-M. Champarnaud and D. Maurel (eds.), 217–222.

Sutner, K. 2004. Computational discrete mathematics. http://www.andrew.cmu.edu/course/15-354.

Thurston, W. 1994. On proof and progress in mathematics. *Bull. Am. Math. Soc. 30*, 161–177.

Wolfram, S. 2002. *The Mathematica Book*. Cambridge University Press.

Received June 2004; revised March 2005; accepted January 2006