Engineering Software Correctness

Rex Page
University of Oklahoma
School of Computer Science
Norman OK USA
1 405 325 5048

page@ou.edu

ABSTRACT

Software engineering courses offer one of many opportunities for providing students with a significant experience in declarative programming. This report discusses some results from taking advantage of this opportunity in a two-semester sequence of software engineering courses for students in their final year of baccalaureate studies in computer science. The sequence is based on functional programming using ACL2, a purely functional subset of Common Lisp with a built-in, computational logic developed by J Strother Moore and his colleagues over the past three decades. The course sequence has been offered twice, so far, in two consecutive academic years. Certain improvements evolved in the second offering, and while this report focuses on that offering, it also offers reasons for the changes. The discussion outlines the topical coverage and required projects, suggests further improvements, and observes educational effects based on conversations with students and evaluations of their course projects. In general, it appears that most students enjoyed the approach and learned concepts and practices of interest to them. Seventy-six students have completed the two-course sequence, half of them in the first offering and half in the second. All of the students gained enough competence in functional programming to apply it in future projects in industry or graduate school. In the second offering, about forty percent of the students gained enough competence with the ACL2 mechanized logic to make significant use of it in verifying properties of software. About ten percent acquired more competence than might reasonably be expected, enough to see new opportunities for applications and lead future software development efforts in the direction of declarative software with proven correctness properties.

Categories and Subject Descriptors K.3.2 [**Computer and Information Science Education**]: *Computer science education, Curriculum*.
D.1.1 [**Applicative (Functional) Programming**]

D.2.4 [**Software/Program Verification**]: Correctness proofs, Formal methods.

General Terms Design, Languages, Verification.

Keywords Software engineering education, functional programming, Lisp, ACL2, mechanized logic, theorem provers

1. OPPORTUNITIES

Software engineering courses offer one of many opportunities for providing students with a significant experience in declarative programming. Many computer science programs require at least one course in software engineering, and some require more.

For example, the technical portion of the baccalaureate curriculum in computer science at the University of Oklahoma comprises seventy-three credit-hours of coursework. (One credit-hour is awarded for one fifty-minute lecture per week for a sixteen-week semester.) The seventy-three credits-hours are parceled into twenty-three, three-credit courses and one four-credit course. Half of these are mathematics courses (four of which — applied logic, discrete mathematics, theory of computation, and algorithm analysis — are taught by the School of Computer Science). The other half are engineering courses (all of which are taught by the School of Computer Science). Eight of the twelve computer science courses (or more, depending on electives) involve significant software or hardware development.

None of the courses with significant software development assignments prescribe any particular technology in their official descriptions, but by agreement of the faculty, the first three courses (introduction to computer programming, programming structures and abstractions, and data structures) use Java and C++ to describe computations. The other courses leave to the instructor and/or the student the choice of programming languages and other software development tools.

Before 2003, no course in the curriculum afforded students a significant experience in declarative programming. Sometimes students in the programming language course wrote short programs in a functional language such as Scheme or in a logic language such as Prolog. However, these ten- to twenty-line programs could in no way provide students with enough background to apply declarative programming to future projects. Substantial additional education would be required to meet that goal, and the curriculum did not provide an opportunity to get that education.

In 2003, I began using declarative programming in a two-course sequence in software engineering: Software Engineering I and Software Engineering II. The remainder of this report discusses this experiment and some of the results.

2. READING GUIDE

The following summary of topics may serve as a reading guide:

Section 3.	Evolution of the courses
Section 4.	Overview of first semester (SE-I)
Section 5.	Software verification examples from lectures

3. HISTORY

All baccalaureate students in our computer science program are required to take a two-course sequence (six credit-hours in all) in software engineering. Almost all of them take this sequence during the last year of their studies.

The official description of the first of these courses, Software Engineering I, is "Methods and tools for software specification, design, and documentation. Emphasis on architectural modularity, encapsulation of software objects, and software development processes such as design review, code inspection, and defect tracking. Students working in teams apply these ideas to design and document software products. Study of professional ethics, responsibility, and liability." The course catalog has the following description for Software Engineering II: "Methods and tools for software development, testing, and delivery. Emphasis on data abstraction and reusable components. Students working in teams implement a significant software product, including design documents, user's guide, and process reports, using methods and processes studied in Software Engineering I. Students will practice oral and written communication skills."

As taught (by me, as least), the courses have three primary elements: design, software development processes, and defect control. Students work in teams in both courses. Software Engineering I places more weight on individual work than on team projects, while Software Engineering II gives teamwork more weight.

There are many ways to put together educational material on design, software processes, and defect control. Accordingly, in the six academic years in which I have taught the two-course sequence in software engineering, I have put the material together in several different ways.

In the beginning I based the course on traditional textbooks, such as Pressman [5] or Sommerville [6] and supplemented the material with experiences from industry. These textbooks cover a great deal of ground, but in a way that I find unsatisfying. There is little material of intellectual depth, and the books are entirely noncommittal on software engineering methods. One way is as good as another. Well … maybe some ways are better suited for some applications and other ways for other applications, but the authors provide no useful information about how to choose. Students are encouraged to learn a litany of terms and techniques, but without much motivation to go to the trouble.

Later, I began to use a textbook by Humphrey [2], which emphasizes software processes, and supplemented it with material on design and implementation. The book covers less ground than the traditional textbooks, and it does a poor job in covering software design, but it provides excellent coverage of software processes, and in a form that makes it practical for students to experience some of the benefits of applying such processes. The material is especially attentive to defect control in software development. Processes are experienced by students in the form of a specific set of estimation and record-keeping activities that Humphrey calls the Personal Software Process (PSP). It is presented with about as much intellectual depth as is possible for software process coverage, and the book takes a specific, useful point of view about software development.

During all of this period (pre-Humphrey and post-Humphrey) Software Engineering I was organized around six to ten small software development projects carried out by individual students, and Software Engineering II was organized around one, medium-sized project (5,000 to 15,000 lines of code) carried out by teams of four to six students. Students implemented software in a variety of languages (C++ and Java primarily, sometimes supplemented by scripting languages such as Tcl/Tk, Microsoft Word macros, or HTML managers). In every case, programming followed a conventional (that is, stateful) paradigm.

Two years ago, I decided to try using a declarative paradigm to boost the design and defect-control elements in the course. During the 2003-2004 academic year, the students used Scheme (specifically, the DrScheme environment [1] with its associated user interface tools) to implement their software, but wrote computational functions (as distinguished from functions performing some sort of input or output) in ACL2 [3], a purely functional subset of Common Lisp with a computational logic (theorem prover) for verifying properties of defined functions. This made it possible for students to verify, by way of mathematical proof, certain properties of the software they were writing. They used a mechanical translator to convert the functions they had defined in ACL2 to Scheme, to integrate them with their i/o-performing functions.

The main problem with this approach was that it failed to give most students a really significant exposure to functional programming methods. Most students expanded their i/o-performing functions in every way they could think of. Then, they could avoid functional programming in most of their code and use conventional methods for the bulk of it. Discouraging this tendency to minimize the portion of the program written in functional form, whether through grading or through discussions with individual students, proved to be impossible (for me) without seeming arbitrary or unreasonable.

Based on this experience, I required the students to write all code in ACL2 in the 2004-2005 academic year. This made all of the code conform to the purely functional paradigm and gave the students a consummate experience in functional programming. The primary disadvantage was that the assigned problems had to be designed to avoid interactive input/output operations, since those would be clumsy, at best, in ACL2. A secondary disadvantage, compared with Scheme, was the lack of higher-order functions in the ACL2 subset of Common Lisp. Neither of these disadvantages turned out to cause any significant problems, and the advantage to the students of experiencing the benefits of functional programming easily outweighed the disadvantages.

The remainder of this report focuses on the 2004-2005 offering of Software Engineering I and II, with the all-ACL2 requirement.

4. SE-I

The 2004 edition of Software Engineering I required each student to complete six small software development projects (100 to 500 lines each) working alone. The course also required students working in teams to complete one software development project of modest size (about 1,500 lines of code, including some reused code from the individual projects) and to cooperate in a prescribed way in the development of a smaller piece of software earlier in the course. I formed the teams (five or six students per team) with a view to balancing the talent and following other criteria developed by Larry Michaelson in his work on team-based learning [4].

Each individual software development project in Software Engineering I requires the students to deliver four items: design, code, PSP report [2], and proven theorems. The design is presented as a boxes-and-arrows chart, together with some textual descriptions of data structures, interfaces, and algorithmic decisions. The code is written entirely in ACL2. The Humphrey-style, PSP report includes a project plan, a software size estimate using a statistical estimation method based on historical data (estimates get better as the course progresses), a time log, a defect log, and a collection of test designs and reports.

Theorems (stated in ACL2 logic) express properties of functions written for the project. In the individual projects, specific theorems are given in the project assignment, to keep the students from floundering around with things they are unlikely to be able to get ACL2 to prove. In the first couple of problems, these theorems, once correctly stated in ACL2 logic, are things that the computational logic of ACL2 can prove without special hints.

As the course progresses, the project assignments specify theorems that require students to find additional supporting lemmas that ACL2 can prove directly. After ACL2 has the supporting lemmas in its database, it can proceed successfully to prove the target theorems.

This approach (stating theorems, finding that ACL2 cannot prove them on its own, discovering lemmas that ACL2 can prove and that provide a basis for proving other theorems, and finally working up to successful proofs of the originally stated theorems) is part of what the authors of the ACL2 book [3] call "The Method." It is one of many techniques that users of ACL2 must master to succeed in verifying significant software properties. It is the primary theorem-proving technique emphasized in the course.

The team software projects have the same four deliverables, and size estimates are based on averages of individual PSP data [2]. Altogether there are seven team projects in the course (five of which primarily concern software process issues) and seven individual projects (six of which are software development).

Twenty-one of the thirty-one, seventy-five-minute, class periods in Software Engineering I are devoted to lectures, and the remaining ten class periods are used as meeting-time by the teams to work on team projects. Five lectures address primarily design issues, eight focus on ACL2 (both as a programming language and as a computational logic), and six concern software processes.

5. LECTURES ON ACL2

Lectures on ACL2 in the first semester (SE-I) have three themes: (1) defining functions in the form of equations expressed in Lisp (the first experience with declarative programming for many of the students), (2) specifying properties of functions in the logic of ACL2, and (3) getting the ACL2 theorem prover to verify the properties, sometimes by supplementing with lemmas and building gradually to proofs of the desired properties.

The first lectures on ACL2 discuss non-recursive functions from propositional logic and theorems with non-inductive proofs, such as de Morgan's laws. I go through the proofs by hand, and then demonstrate that ACL2 succeeds in mechanizing them.

```
(defthm take-append-identity
  (implies (true-listp xs)
           (equal (take (length xs) (append xs ys))  xs)))
(defthm drop-append-identity
  (implies (true-listp xs)
           (equal (drop (length xs) (append xs ys))  ys)))
```

Figure 1. Correctness of concatenation.

Inductive examples come next, starting with the associativity of concatenation, the canonical example of the Boyer-Moore theorem prover from which ACL2 evolved. Most theorems discussed in the lectures relate directly to correctness, as in the relationships between the take, drop, length, and concatenation operations (Figure 1) that confirm the correctness of concatenation (assuming the correctness of the other operations).

Another early inductive example has to do with conservation of atomic elements in a function that flattens a tree (Figure 2).

```
(defun flatten (tr)
  (if (atom tr)
     (cons tr nil)
     (append (flatten (car tr)) (flatten (cdr tr)))))
(defun occurs-in (x tr)
  (or (and (atom x) (atom tr) (equal x tr))
      (and (atom x)
           (not (atom tr))
           (or (occurs-in x (car tr))
               (occurs-in x (cdr tr))))))
(defthm flatten-conserves-atoms
  (iff (occurs-in x tr)
       (and (atom x) (member x (flatten tr)))))
```

Figure 2. Flatten conserves atoms.

ACL2 proves all of these early theorems directly from their statements. When numbers are involved, it needs a little help from some arithmetic theorems supplied with ACL2, but no special steps are required. In these examples, one simply states the correctness properties, and the rest is automatic. However, it's not entirely trivial to state the theorems correctly. For example, without the true-list predicate (specifying a nil-terminated list) in the hypothesis of either theorem on concatenation (Figure 2), the equality in the conclusion may fail.

This is consistent with programming experience. What is true for the test-and-debug approach to software development is also true for a regimen that includes mechanically verified software properties: Initial expectations of a piece of software often turn out to be wrong. Knowing the conditions under which a formula delivers the intended result makes software more reliable, and that is one kind of information provided by software properties verified through computational logic.

```
(defun drop (n xs)
  (if (or (<= n 0) (atom xs))
      xs
      (drop (– n 1) (cdr xs))))
```

Figure 3. Incorrect definition of drop.

One lecture is devoted to defining the drop function, getting ACL2 to admit it to its logic (that is, to prove that it terminates), and verifying some of its properties. A naïve definition (Figure 3) fails to specify that the numeric argument must be integral, and an examination of ACL2's attempt at a termination proof shows that it runs off track trying to deal with the possibility that the number might be complex. ACL2 is able to complete the proof when the argument is constrained to integral values.

For both inductive and non-inductive theorems, the lectures present informal proofs, at the level of normal, mathematical argumentation, and point out that these informal proofs gloss over thousands of details that are not overlooked by the mechanized logic of ACL2. Part of the point is that informal proofs of software properties have limited value because they are at least as likely to be defective as function definitions. It is only with full mechanization that software verification has real value.

In more advanced examples, it is necessary to derive several lemmas from the steps in an informal proof to lead ACL2 to a successful proof ("The Method" [3]). One such example is a function that parcels a list into packets. Each packet is a contiguous sublist of the original, containing the elements lying between occurrences of a specified delimiter. A notion of correctness in this example involves expressing the function two ways (Figure 4), one of which is viewed as a correct specification, then verifying that the two definitions are extensionally equivalent.

Two lectures discuss a more extensive design and verification example: AVL trees (insertion, deletion, and search) expressed in about 130 lines of ACL2, with roughly the same number of additional lines devoted to stating lemmas and correctness properties. The theme of these lectures is designing correctness into software from the beginning by stating the properties each function is expected to have. Most of the properties in the AVL case have to do with preserving order, preserving or restoring balance, and conserving keys in various operations on trees.[1]

```
(defun packets (d xs)
  (if (atom xs)
      '(nil)
      (let* ((split (break-at d xs))
             (first-packet (car split))
             (rest (cadr split)))
        (cons first-packet
              (if (atom rest)
                  nil
                  (packets d (cdr rest)))))))

(defun packet-n (n d xs)
  (take-to d (drop-past-n-delimiters n d xs)))

(defthm packets-thm
   (implies
     (and (true-listp xs) (integerp n) (>= n 0))
     (equal (packet-n n d xs)
            (nth n (packets d xs)))))
```

Figure 4. Correctness of packets.

6. PROJECTS IN SE-I

Designing the software development projects for Software Engineering I involved a lot of care and experimentation. I wanted to give the students problems on which they could be successful, even though none of them had prior experience with ACL2 and few had experience in declarative programming. I wanted students not only to design and implement software, but also to succeed in using the computational logic of ACL2 to verify at least a few properties of their code.

Initially, I was not at all confident that I could design projects on the fly to meet these goals. Fortunately, I was able to organize a summer research program for undergraduate students in 2003 to try out some ideas. I designed ten software development projects, with theorems expressing software properties that I thought students could verify with ACL2, and set the students involved in the research program to work on them.

Not all of the projects were suitable as specified, but the students were able to modify requirements so that, in the end, we had a set of ten software engineering projects that we knew students could successfully complete. These were the projects assigned in the fall, 2003 offering of Software Engineering I.

To prepare for fall, 2004, I scrapped some of the problems, rearranged others, developed some new ones, and had a pair of undergraduate students work on the problems during the summer of 2004. The software development projects used in fall, 2004 emerged from the efforts and suggestions of these students.

This represents a great deal more than the usual preparatory work for putting together a problem set for a course. But, using a computational logic as a principal element of a software engineering course was new to me. It seemed wise to find out, ahead of time, whether or not students could solve the problems with a reasonable amount of effort.

Having gone through the process twice has convinced me that, in addition to the usual guidelines that instructors follow in putting together software development projects for students, there are two tricks to designing projects when ACL2 is the implementation language:

1. Make sure all input/output is file based.
2. Identify properties to be verified, and find proofs of those properties, using ACL2, before assigning them.

These observations eliminate the need for students to test the projects before assigning them in the course. However, it's still a lot of work. I have to write the functions involved in the software properties to be verified and work through proofs using ACL2. This is part of the burden of using ACL2 in a software engineering course. One might hope that the burden could be reduced in the future by sharing projects among instructors.

[1] The project is incomplete. All the necessary properties and many supporting lemmas are stated, but proofs of order, balance, and key-conservation properties are complete only for primitive operations, such as rotations. Full verification awaits proofs of similar properties for insertion and deletion.

Project 0. The first of the six individual projects gives the students a chance to learn basic mechanics of the ACL2 system and to gain a little experience in specifying computations in the form of equations rather than sequences of commands. It consists of four small problems that students are likely to be familiar with from other courses.

The assignment requires students to define ACL2 functions specifying the following computations: Newton's method for approximating square roots, reversing a list, set operations (eliminating duplicates from a list, set union, set intersection, and set difference), and the towers of Hanoi problem. Students are not required to state or prove any theorems in this first assignment.

Almost all students succeeded in all parts of this project. A few failed to get ACL2 to admit their function for Newton's method for square roots. ACL2 will not admit a function to its logic unless it can prove termination, and termination is a bit tricky for Newton's method. Since ACL2 deals only in full-precision, rational numbers, a square root function needs an extra argument specifying a desired accuracy. The function computes an iteration count from this argument, and terminates based on this count.

Project 1. The second project requires students to define functions that compute the mean and variance of a sequence of numbers and the frequency count of each number in the sequence. Students are required to prove that the frequency-count function delivers the same result for every permutation of the input sequence. An example in the ACL2 book [3] proves a similar property for a sorting function, and this gives students a leg up in the software-verification part of the project. Otherwise, the proof would probably be too hard at this stage.

All students succeeded in completing a working program for Project 1. About half were able to verify that the frequency-count function is invariant with respect to permutations of the input sequence. This is a tricky property to state, and not all students were able to dig the material on permutations out of the textbook on their own.

Project 2. In the third project, students define three functions to compute the n^{th} Fibbonaci number (one using nested recursion, one using tail recursion, and one using Kepler's formula). They use the mechanized logic of ACL2 to verify that two of the functions (the nested recursion and the tail recursion) are equivalent, and they use a stopwatch to compare efficiency. Students also define nested and tail recursions for the Lucas sequence (Fibbonaci, generalized to arbitrarily specified starting values) and use ACL2 to prove that the Lucas sequence is non-decreasing if the starting values are nonnegative.

Finally, the students are required to write an analytic essay describing why they believe the approximation to the square root of five that their function uses in Kepler's formula is accurate enough to deliver the correct answer. They use their Newton's method program from the first project to approximate the square root, and they are allowed to assume that k^{th} iterate of Newton's method (starting from 2 as the zeroth iterate) delivers an approximation to the square root of five that is correct in the first 2^{k-1} decimal digits.

Most students got ACL2 to prove the equivalence of nested and tail recursions for Fibbonaci. About a quarter of the students succeeded in proving that their Lucas functions delivered non-decreasing sequences. This property is easier for ACL2 to deal with in terms of the Lucas implementation based on nested recursion. The tail recursive version complicates the proof. So, the trick is to choose the "simpler" of two equivalent functions when verifying properties, which isn't necessarily the more efficient implementation.

This approach is a good lesson for a general setting: Sometimes it is worth finding two representations of a function, one that specifies an efficient computation, and one that is more clearly correct. Then, use ACL2 to prove that the two representations are equivalent. Finally, verify correctness of the simpler function, and use the more efficient one in the running software.

One student (of nearly fifty) composed an essay with a first-rate analysis of the accuracy that Kepler's formula requires in the approximation to the square root of five to deliver the correct value for the n^{th} Fibbonaci number. About two-thirds of the essays had some engineering value, but danced around the main point without finding a real solution, and the remaining third missed the point entirely.

Project 3. The fourth project involves producing a concordance of a text. The project gives syntactic rules defining words in the text and specifies line formats for the output file (requiring, for example, that the principal words appear in a column near the middle of the line, with the amount of surrounding context varying according to the size of the lines and the size of the principal word on the line).

The project calls for an *n log n* sorting function and a proof that it delivers a permutation of the input sequence in which the elements occur in increasing order. All students put together a working program for this project, and over three-quarters of them succeeded in getting ACL2 to prove the correctness of the sorting function (mostly following an example in the ACL2 book [3] of a similar theorem for an n^2 sorting function).

Project 4. The fifth project converts a file of text into two word-frequency tables, one arranged alphabetically and one in decreasing order of word frequency. The program must include a function that converts a sequence of numbers arranged in increasing (or decreasing) order into a sequence of "run frequencies" — that is, a sequence of ratios between the length of each contiguous block of identical numbers in a given, ordered sequence and the total number of elements in the given sequence. This function has the property that the sum of the ratios in the sequence it delivers is one, and the project requires verification of this property using ACL2's computational logic.

All but a few of the students completed the program and over half succeeded in proving the required software property. This was the first proof in which it was necessary for students to explore supporting lemmas to get the proof to go through, and a fifty-percent success rate was better than might be expected.

Project 5. The last individual software development project calls for a function that compares the number of tokens in a given ACL2 program with the number of tokens it would have if invocations of defined functions were in-lined.

The project requires students to use a supplied software package implementing AVL trees to record function bodies (or token counts and argument reference counts) keyed by function names for later look-up while going through the program. It also requires them to choose two properties of the program to verify with ACL2's mechanized logic.

Almost all students succeeded both in constructing a working program and in carrying out the proofs. However, most of them chose rather minimal properties to verify. There was a wide variety in the complexity of the submitted programs, varying from a few hundred lines to over a thousand. The short programs were better.

Team Project 1. The concordance project (Project 3) is actually divided into five parts:

1. planning and design,
2. design review,
3. initial implementation of revised design,
4. code review, and
5. final implementation of reviewed code.

Parts 1, 3, and 5 are individual projects, and parts 2 and 4 are team projects. In part 1, students convert the problem description into a design, on an individual basis. Then, in part 2, the teams meet, choose one of the designs at random, conduct a design review in a team session, and then revise the design. In part 3, individual students implement the revised design through the unit testing phase, but without integration testing. In part 4, the teams meet again, select one of the implementations at random, and conduct a code review. In part 5, individual students complete the implementation of the reviewed code.

The project provides students with opportunities to experience the benefits of design and code review and to practice interpersonal skills. Most students seem to enjoy the project, and it serves, too, the purpose of providing practice for the second team project, which is a more substantial software development effort.

Team Project 2. The larger of the two team software projects in Software Engineering I requires the implementation of stock market analysis software. The software processes a file of inquiries describing analytic computations of stock market data. The syntax of inquiries is rudimentary, and the computations are not complicated, but the data file, which contains market data for S&P 500 corporations, is massive.

The software invokes functions in an AVL package to record data from the file, on the fly, to improve the response time of inquiries. In addition to the AVL package, students are encouraged to incorporate code from earlier projects (for statistical calculations, for example).

In addition to the usual reports on planning, estimation, development time, testing, and defects, the project requires documentation explaining how to use the software. Plus, teams choose two important properties of their code suitable for an ACL2 proof, write a short analysis of the benefits that proving the property might provide, and outline an approach to a proof. Finally, they choose one of the properties and prove it in ACL2.

The project gives students experience in organizing and cooperating in a modest software effort. (Implementations run one to two thousand lines.) It would be enhanced by explicit requirements for testing regimes and a prescribed effort in verified properties focused on some important aspect of correctness.

7. SE-II

Software Engineering II is a project-based course required of all seniors (final-year students) in computer science. In the 2005 offering of this course, I formed seven teams [4] of five or six students each to carry out team projects. There were thirteen separate items that each team was required to deliver, culminating in a full implementation of a software product of moderate size (3,000 to 5,000 lines of code implementing an “image calculator” that takes a formula specifying a computation that applies image operations to a sequence of images and generates a new sequence of images transformed by the operations in the formula).

The thirteen deliverables are due on a more-or-less weekly basis throughout the semester and include such items as initial design and time estimates, engineering standard, detailed design, design and code review reports, product specs and installation guide, unit and integration test suites following a testing strategy, final design and code, meeting logs, and three presentations.

There is one individual project consisting of a compilation of weekly progress reports, plus PSP documents [2] and ACL2-proven properties for each component the individual contributed to the team’s software product. These individual projects are typically about thirty pages long, although some are as short as twenty, and some as long as two hundred pages. Usually, but not always, the longer ones are better.

Because Software Engineering II is a senior project course, most class periods are devoted to meeting time for the teams to keep their projects on track. Four class meetings are devoted to team presentations, and two are devoted to formal lectures. Informal lectures occur occasionally throughout the course.

8. PROJECTS IN SE-II

Teams of students in Software Engineering II, spring 2005, built a program to carry out image transformations (filtering for feature enhancement, differencing for background removal, addition, scaling, etc) in combinations specified by formulas presented in a syntax based on lambda expressions. They delivered their projects in the form of thirteen separate items with due-dates spread throughout the sixteen-week semester.

After two weeks, teams completed high-level designs together with size and time estimates based on PSP data [2] collected during the first semester. Shortly afterward, they wrote engineering standards (document management procedures, design and code style sheets, testing procedures, etc). They worked on more detailed designs and estimates for the next three weeks. In the midst of this period, they conducted design reviews. They delivered completed designs and estimates in the sixth week, along with ten-minute, in-class presentations describing their plans. Five deliverables, six weeks.

For the next eight weeks, the teams worked on implementation. Along the way, they delivered code review reports, product specifications, and unit and integration test suites. Finally they delivered the code itself and a thirty-minute presentation covering specific points, such as software architecture, implementation problems and solutions, planned versus actual schedules, remaining implementation problems, and potential enhancements. The teams were instructed to address an audience consisting of engineering management familiar only with a short description of their software product’s goals. Six engineering managers from industry attended the presentations, asked questions, and left the students (and instructor) with written comments and evaluations. Five more deliverables, eight more weeks.

During the last two weeks, in addition to making their thirty-minute product presentations in class, the teams worked on a test suite for the software product of another team (assigned in round-robin fashion), based on that team's product specification, and a ten-minute presentation of the results of applying the test suite. Testing had to follow a known and documented strategy of the team's choice, such as statistical use-based testing or software reliability engineering.

The final item of the team project is a meeting log. All but a few class periods are devoted to meeting time for the teams. Before each meeting, each team submits an agenda. During the meeting, they annotate their agendas with discussion notes and decisions. This collection of annotated agendas comprises the meeting log. Altogether, thirteen team deliverables in a period of sixteen weeks.

During development of the team's software product, individual students put together reports on each software element (one function or a few related functions) they contribute to the product. These reports begin with a description of the software element and its role in the team's software design. They continue with a PSP log (plans, estimates, design, time log, defect log, testing templates, code, and summaries [2]). Each report also states at least one proven property of the contributed software element along with a summary of a proof of the property using ACL2.

The collected reports form one section of the sole individual project of Software Engineering II. The other section of the report consists of the collection of weekly reports each student makes throughout the semester. Each team meets with the instructor once a week for twenty minutes to discuss progress and problems, and the individual weekly reports serve as one form of input for those meetings.

In spring 2005, individual reports varied from twenty pages to two hundred, and quality had a similar range. Only a few students failed to take the individual project seriously. Two students completely omitted proofs of software properties, and about two-thirds of the software properties stated and proven in the reports had no perceptible theme or significant relevance to overall software correctness.

Nevertheless, most reports were of good quality. Five were outstanding, with insights about software processes, good explanations of the roles of individual software elements in the team's software product, and significant, proven software properties.

9. RESULTS

I expected some complaints from students and from the five representatives from industry who attended presentations in Software Engineering II about the use of an unusual programming environment, ACL2, for a project course in software engineering. To my surprise, I got none from either quarter, and actually got positive support from one industry representative and several students. So, the idea of using a functional paradigm in software engineering was reasonably well accepted.

In past years there has been considerable whining from students about the record keeping required for PSP reports [2]. There was still some of that, but substantially less, possibly because we now use a tool to reduce the burden of keeping logs, making estimates, and recording data.

Based on conversations with individual students and on evaluations of projects they turned in, I believe that all seventy-six of the students who completed both Software Engineering I and II in the 2003-2004 and 2004-2005 academic years managed to learn enough about functional programming to be able to use it effectively in subsequent projects.

Of the thirty-eight students who completed both courses in 2004-2005, about thirty understand how to formulate theorems about software properties, and see some value in stating theorems about software properties. Twenty-five can use ACL2's mechanized logic to verify at least a few software properties. Between ten and fifteen students can formulate theorems that, together, verify a coherent theme of software correctness. About the same number would choose ACL2 or another functional language for a software project in the future, given the opportunity. Five students acquired competence in using the ACL2 mechanized logic well beyond expectations and would be able to use it effectively on their own in new projects without additional training.

10. WISHES

Students would gain a better impression of functional programming if they could see it compete in speed of performance with programs written in C. This might be possible if, once ACL2 has been used to verify correctness, students could compile their code with a good Common Lisp compiler and run it from the resulting executable module. It seems that this would be easy, since ACL2 is a subset of Common Lisp, but so far, I have failed to find a way to do get C-like performance from ACL2 code. This is an improvement that I would like to make in the future.

It would also be nice if ACL2 were higher order. It isn't, and won't be, which is sometimes burdensome from a programming point of view. Many students asked for this feature and were somewhat disappointed that it is not available. Ironically, in earlier software engineering courses in which students were using C++ and passing a function as an argument to another function would have been advantageous (as in numerical quadrature, for example), most students avoided higher order functions by passing in a switch and choosing from a fixed collection of functions.

So, they didn't use higher order functions when they could, and should have. Now that they can't, they want to. Of course, the students involved are different, and not all students fall into either category. That accounts for the difference, but it seems ironic, anyway.

It would be even nicer if ACL2 had convenient support for interactive, graphical user interfaces. This may be feasible, perhaps with some sort of inter-language facility. It is something I would like to look into in the future, but probably not for the upcoming academic year. If I'm lucky, maybe someone else will provide a GUI solution.

Interestingly, two of the seven teams in the 2005 edition of Software Engineering II built a Java framework for automatically invoking their ACL2 code inside a GUI framework, thus solving the GUI problem themselves, but at a high expense. The Java harness was about as large (in terms of lines of code) as the ACL2 code implementing the main computation.

Two improvements of the coverage of defect control that I plan to introduce in the coming year are expanded usage of specified strategies for testing and the use of specific tools for configuration management and defect databases. During the first semester, students will be involved in planning for these changes, and the best of what they discover will go into the processes and tools they use in the second semester. Discussing a more comprehensive model for software quality would also improve the course, but there is some risk that it would make the course so broad that students would fail to get the intense experience that is one of the course's strengths.

A few of the best students from Software Engineering II made some recommendations about expanding the coverage of techniques for getting ACL2 to prove theorems. They suggested discussions of hints, inductive measures, and rule classes for type prescriptions and elimination. The students dug these ideas out of the ACL2 documentation and made good use of them during the year. I expect other students will find the methods useful and plan to introduce them in the next offering of Software Engineering I.

Finally, I wish I had given more guidance about what types of software properties student teams should address in their implementations. Only two of seven teams found coherent themes for applying ACL2's mechanized logic to correctness issues for their software. One of these teams dealt mostly with data-type issues and interfaces, and the other went far enough to consider their software proven correct for most intents and purposes. I think more teams would have this type of success if the project write-up were more explicit about productive uses of mechanized logic in the project.

11. GUESSES

I believe a computational logic like ACL2, integrated into a software development environment, would provide practical benefits in commercial software projects today. Theorem proving is ready for prime time.

The software development process, in this mode, would include using the development environment's logic to state important software properties at the same time that test sequences are designed — that is, ahead of or along with coding. The properties would be proved, gradually, as a normal software development activity, in parallel with testing.

Unfortunately few software engineers are ready for mechanized logic. If educators incorporate products like ACL2 in courses, the next generation of graduates could begin to reap the benefits of functional programming, especially a benefit that the functional paradigm facilitates far more effectively than any other: using mechanized logic to engineer reliable software.

12. MATERIALS

A package of materials for Software Engineering I and II, as described in this report, including syllabuses, schedules, lecture notes, assignments, and supplied software is available at http://www.cs.ou.edu/~rlpage/SEcollab

13. ACKNOWLEDGMENTS

I want to thank the students enrolled in software engineering courses at the University of Oklahoma whose experiences provided much of the material of this report. Especially, I want thank Nic Grounds, Isaac Harley, Jeff Kilpatrick, James Murphy, Elizabeth Murray, Stephen Pitts, and Jeff Sapp for their help in the design of projects and Butch de Berry and Zach Francks for pushing through ACL2 proofs of correctness properties of rotations and other operations in an AVL-tree implementation. Ryan Shepherd built the PSP automation engine used in the course, and the students and I are grateful for that.

Mike Brown (SAIC), Tony Caruso (MRE), David Franke (Trilogy, retired), Shane Merz (MRE), Boyd Nolan (PE), Ken Parker (Risk Metrics), Roger Rowe (Solarc), and Sunny Sethi (Xyant) attended in-class presentations by student teams. In addition to providing students with insight on applying concepts learned in an educational setting to commercial software development projects and contributing an industrial management perspective to the proceedings, they took the lead in developing guidelines for presentations. The software engineering courses at OU have benefited greatly from their efforts.

I also want to thank Rich Didday of INDEC Systems for helping develop major projects for Software Engineering II, including the image calculator project described in Section 8. Finally, I want to thank J Moore, Matt Kaufmann, and Jared Davis of the University of Texas, Austin, for helping my students and me learn to apply ACL2 in software engineering projects.

This report is based on work supported by the National Science Foundation under Grant No. EIA 0082849. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

14. REFERENCES

[1] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. *How to Design Programs*. MIT Press, 2001.

[2] Humphrey, W. S. *A Discipline for Software Engineering*, Addison Wesley, 1995.

[3] Kaufmann, M., Manolios, P., and Moore, J. S. *Computer Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[4] Michaelsen, L. K., "Getting Started with Team Based Learning" in *Team-Based Learning: A Transformative Use of Small Groups*, Praeger, Michaelsen, L.K., Knight, A. B., and Fink, L.D. editors, Stylus Publishing, Sterling VA, 2002.

[5] Pressman, R. *Software Engineering: A Practitioner's Approach, 6th Edition*. McGraw-Hill, 2005.

[6] Sommerville, I. *Software Engineering, 7th Edition*. Pearson, 2004.