JFP **17** (6): 675–686, 2007. © 2007 Cambridge University Press
doi:10.1017/S095679680700634X First published online 26 April 2007 Printed in the United Kingdom

EDUCATIONAL PEARL

Engineering Software Correctness

REX PAGE
School of Computer Science, University of Oklahoma, Norman, OK 73019, USA
(*e-mail:* page@ou.edu)

Abstract

Design and quality are fundamental themes in engineering education. Functional programming builds software from small components, a central element of good design, and facilitates reasoning about correctness, an important aspect of quality. Software engineering courses that employ functional programming provide a platform for educating students in the design of quality software. This pearl describes experiments in the use of ACL2, a purely functional subset of Common Lisp with an embedded mechanical logic, to focus on design and correctness in software engineering courses. Students find the courses challenging and interesting. A few acquire enough skill to use an automated theorem prover on the job without additional training. Many students, but not quite a majority, find enough success to suggest that additional experience would make them effective users of mechanized logic in commercial software development. Nearly all gain a new perspective on what it means for software to be correct and acquire a good understanding of functional programming.

1 Opportunities

Software engineering courses offer one of many opportunities for providing students with a significant experience in functional programming. This report, which is a revision of a presentation at a workshop on functional programming in education (Page, 2005), discusses a two-course sequence that exploits this opportunity by using functional programming as a framework for aspects of design and quality in the study of software engineering.

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 24 courses. Eight of these involve significant projects in software or hardware development. None of the courses with significant software development projects prescribe any particular technology in their official descriptions, but by agreement of the faculty, the first three courses use Java and C++ to specify computations. The other courses leave the choice of programming language to instructors and/or students.

Before 2003, no course in the curriculum afforded students a significant experience in declarative programming. Sometimes students in the required programming language course wrote short programs in a functional language such as Scheme

Software Engineering I. 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.

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 practice oral/written communication skills.

Fig. 1. Official descriptions of SE-I and SE-II.

or in a logic language such as Prolog. These 10- to 20-line programs could, in no way, provide students with enough background to apply declarative programming in future projects. In 2003, a new approach to a required, two-course sequence in software engineering employed functional programming as a central element.

The remainder of this report discusses this experiment and some of the results. Sections 2 through 6 describe the courses (SE-I and SE-II), including examples from lectures and descriptions of projects. Section 7 discusses student accomplishments along with their reactions and those of the faculty and observers from industry. The last three sections discuss comparable efforts and conjecture about future improvements and possible benefits.

2 History

Figure 1 describes two courses that provide the basis for an experiment combining functional programming, mechanized theorem proving, and software engineering education. The courses have three primary elements: design, software development processes, and defect control. Students work in teams in both courses. SE-I places more weight on individual work than on team projects, while SE-II puts the greater emphasis on teamwork.

There are many ways to put together educational material on design, software processes, and defect control. Accordingly, in the past several academic years, the courses have employed a variety of strategies for covering the required topics. In the beginning, the courses relied on traditional textbooks, such as Pressman (2005) or Sommerville (2004), and supplemented the material with experiences from industry. These textbooks cover a great deal of ground, but lack intellectual depth. More recent versions of the courses have used a textbook by Humphrey (1995). Humphrey covers less ground than the traditional textbooks, but 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 text is especially attentive to defect control in software development. Students employ a specific set of estimation and record-keeping activities that Humphrey calls the Personal Software Process (PSP).

SE-I is organized around 6 to 10 small software development projects carried out by individual students. SE-II is grounded in one medium-sized project (5,000–15,000 lines of code) carried out by teams of four to six students. In the past, students implemented this software in a variety of languages (C++ and Java primarily, sometimes supplemented by Tcl/Tk, Microsoft Word macros, HTML managers, or other tools), following a conventional (i.e., imperative) paradigm.

A few years ago, a new approach to the courses was introduced. It centered around the functional paradigm, with the intention of boosting the design and defect-control themes. During the 2003–2004 academic year, the students wrote the I/O portions of their software in Scheme, using the DrScheme environment (Findler *et al.*, 2002), but wrote computational functions (as distinguished from functions performing some sort of input or output) in ACL2, a purely functional subset of Common Lisp with a computational logic for verifying properties of defined functions (Kaufmann *et al.*, 2000a). 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 functions.

Two problems emerged from the combined Scheme/ACL2 approach. First, keeping two versions of a source code in synch (a running version in Scheme and a mechanical logic version in ACL2) is nearly impossible to manage. It is not a realistic way to do software development. Second, the two-language approach failed to give students a significant exposure to functional programming methods. Most students expanded their I/O 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, whether through grading or through discussions with individual students, proved to be impossible without seeming arbitrary or unreasonable.

On the basis of this experience, students were required to write all code in ACL2 in the following 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 I/O 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. Despite these problems, the advantage to the students of experiencing the benefits of functional programming outweighed the disadvantages.

3 SE-I

The ACL2-based version of SE-I requires each student to complete six small software development projects (100–500 lines each) working alone and one team project of modest size (typically 1,000–1,500 lines of code, including some reused code from the individual projects). Students also cooperate in teams in a prescribed manner in the development of a smaller piece of software earlier in the course. Teams of four to six students are formed according to criteria developed by Michaelsen (2002).

Each individual project requires the students to deliver four items: design, code, PSP report, and proven theorems. The design is presented as a boxes-and-arrows chart, together with some textual descriptions of data structures, interfaces, and decisions about algorithms. The code is written entirely in ACL2. The 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 scripts and reports.

Theorems stated as Boolean formulas in ACL2 express properties of functions. In the individual projects, specific theorems are given in the project assignment to keep the students from floundering. In the early going, the theorems, once correctly stated, are provable by the mechanized logic without human intervention. As the course progresses, the projects include theorems that require students to find supporting lemmas. After ACL2 has the supporting lemmas in its database, it can successfully prove the target theorems.

This assumes that the student has constructed a collection of definitions that facilitate reasoning. Convoluted definitions make it hard for the mechanical logic to succeed, so the use of ACL2 improves the code viewed as an artifact of human communication. The iterative approach, proceeding from lemmas to theorems, is part of what Kaufmann *et al.* (2000a) call “The Method.” It is one of many techniques that users of ACL2 must master to succeed in verifying significant software properties. The Method is the primary theorem-proving technique emphasized in the course.

The team software project in SE-I has the same four deliverables, but size estimates are based on averages of individual PSP data. In addition to the team software project, there are several team projects on software process issues.

About two thirds of the class time in SE-I is devoted to lectures, and the remaining third is used as meeting time for working on team projects. About 40% of the lectures focus on ACL2, and the remaining lectures are about equally divided between design issues and software processes.

4 ACL2 examples

Course lectures on ACL2 have three themes: (1) defining functions in the form of equations expressed in Lisp, (2) specifying properties of functions in the logic of ACL2, and (3) verifying properties with the ACL2 theorem prover. Some proofs exhibit the process of directing the theorem prover through supplementary lemmas.

The first lectures on ACL2 discuss nonrecursive functions from propositional logic and theorems with noninductive proofs, such as de Morgan’s laws. A step-by-step presentation of each proof is followed by a demonstration that ACL2 succeeds in mechanizing the proof.

Inductive examples come next, starting with the associativity of concatenation, the canonical example of the Boyer–Moore theorem prover (1998) from which ACL2 evolved. Most theorems discussed in the lectures relate directly to correctness, as in

```
(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)))
```

Fig. 2. Correctness of concatenation.

```
(defun flatten (tr)
   (if (consp tr)
       (append (flatten (car tr)) (flatten (cdr tr)))
       (list tr)))
(defun occurs-in (x tr)
   (if (consp tr)
       (or (occurs-in x (car tr)) (occurs-in x (cdr tr)))
       (equal x tr)))
(defthm flatten-conserves-atoms
   (iff (occurs-in x tr)
        (member x (flatten tr))))
```

Fig. 3. Flatten conserves atoms.

the relationships between the take, drop, length, and concatenation operations that confirm the correctness of concatenation (Figure 2), assuming the correctness of the other operations. Another early, inductive example has to do with conservation of elements for a function that flattens a tree (Figure 3).

ACL2 proves these early theorems without assistance. It is not entirely trivial, however, to state the theorems correctly. For example, without the true-list hypotheses (specifying nil-terminated lists) in the concatenation theorems of Figure 2, the conclusions do not hold. That is, theorems without the true-list hypotheses overstate the capabilities of these functions. A programmer who expects extended capabilities may apply the functions improperly. Knowing the limitations can help programmers avoid introducing bugs in software.

This is not theoretical. Students occasionally discover details of the domains of functions by finding that ACL2 cannot verify properties they expect their functions to have. When they are able to verify reformulated properties, they understand implicit limitations. In this way, students use theorem proving not only as a tool for quality assurance after delivering a software product but also as an aid in ongoing design.

One example defines a function that drops an initial segment of a list and illustrates the process of admitting it to the ACL2 logic and verifying some of its properties. A naïve definition of the drop function (Figure 4, left) 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

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

```
(defun drop (n xs)
  (if (or (not (posp n)) (atom xs))
      xs
      (drop (- n 1) (cdr xs))))
```

Fig. 4. Possibly nonterminating (left) and provably terminating (right) definitions.

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

Fig. 5. Correctness of packets.

might not be an integer. ACL2 proves the theorem when hypotheses constrain the argument to integral values.

For both inductive and noninductive theorems, the lectures present informal proofs, at the level of normal, mathematical argumentation, and point out that these informal proofs gloss over myriad details that ACL2 meticulously takes into account. 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, as De Millo *et al.* (1979) famously pointed out. Full mechanization gives software verification real value.

More advanced examples use “The Method.” Lemmas derived from the steps in an informal proof guide the mechanized logic. 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 5). One of the definitions is viewed as correct, and the mechanized logic is used to prove that the second definition is extensionally equivalent to the first.

5 Projects

Designing the first set of projects for SE-I involved a lot of care and experimentation. Projects aimed to give the students problems on which they could be successful, even though few had experience in declarative programming. Another goal was to give

Table 1. *Typical SE-I projects*

	Description	*Theorems*
0	Several small, familiar functions	No theorems
1	Linear encryption of text messages	Decryption inverts encryption
2	Fibonacci three ways: nested recursion, tail recursion, and Binet's formula	Equivalence of nested and tail-recursive definitions
3	Steganography in BMP files	Decryption inverts encryption
4	Word frequency charts for texts	Sum of frequencies is one
5	Image transforms for BMP files	Image reflection and rotation identities

Table 2. *Combined team/individual project phases*

1	Planning and design	Team
2	Design review/revision	Team
3	Implementation of revised design	Individual
4	Review of randomly selected implementation	Team
5	Completion of reviewed code	Individual

students opportunities not only to implement software but also to succeed in using the mechanized logic of ACL2 to verify at least a few properties of their code.

To increase the likelihood of meeting these goals, undergraduate students involved in a summer research program worked on 10 software development projects, including the verification of theorems expressing software properties. Not all of the projects were suitable as originally specified, but the student researchers modified them to produce a set of 10 feasible projects. These projects were assigned in SE-I in the fall term and revised, again with the help of undergraduate research assistants, during the next summer.

At this point, three offerings of SE-I have featured the use of computational logic for verifying correctness properties, and the reserve of student-tested projects has increased to about 20.[1] In general, all students succeed in writing ACL2 programs meeting basic project requirements. Most succeed in stating theorems that correctly express software properties, and a quarter to half succeed in getting ACL2 to prove their theorems. Table 1 outlines typical projects. They are not much different from the projects of standard software engineering courses, except for the following characteristics:

1. Projects use file-based input–output, exclusively, to accommodate ACL2.
2. Projects identify properties to be verified for which the instructor has found, in advance, straightforward ACL2 proofs.

One of the projects is split into individual and team phases (Table 2). Teams of students create a software design, then each individual implements the design. One of their implementations is selected at random, and the team performs a formal

[1] Materials for the courses described in this report, including lesson plans, lecture notes, projects, schedules, and supplied software are available at: `http://www.cs.ou.edu/ rlpage/SEcollab.`

code review. Finally, each individual revises the reviewed code to produce a working program. The project provides opportunities to experience the benefits of design and code review, to practice interpersonal skills, and to prepare for a second, larger team project.

The final project of SE-I requires the teams to develop a program of modest size (typically 1,000–2,000 lines of code) for carrying out computations such as stock market analysis, graphics operations, or symbolic differential calculus. In addition to delivering a working program, the team reports on planning, estimation, development time, testing, defects, and usage instructions. And, 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.

6 SE-II

SE-II is a project-based course. Students teams are required to deliver about a dozen separate items, culminating in a full implementation of a software product of moderate size. One typical project calls for the implementation of an "image calculator." The calculator interprets a lambda-like formula specifying image transformations using operations such as filtering, superposition, and scaling.

Deliverables include initial design and time estimates, engineering standards, detailed design, design and code reviews, product specifications and installation guide, unit and integration test suites, final design and code, meeting logs, written and oral presentations of the team's design and software, and a presentation reviewing another team's product. Individual students compile weekly progress reports, PSP documents, and mechanically verified properties for components contributed to the team's software product.

Because SE-II is a project course, most class periods are devoted to meeting time for the teams to keep their projects on track. Several class meetings are devoted to team presentations, and a few are devoted to formal lectures. Informal lectures occur occasionally throughout the course. Teams participate in scheduled, weekly meetings with the instructor to report progress and discuss problems.

7 Results

One of the benefits of introducing students to mathematically verified correctness properties of software is that students gain a new appreciation for what it means for software to be correct. Many students, probably most, have never given serious consideration to the possibility that software testing cannot ensure correctness. In some cases, they find through the use of mechanized logic that their software fails to have some of the properties they thought they had carefully tested for. That is, the process of expressing anticipated properties and attempting to verify them reveals flaws in their assumptions. This does not happen often, but it happens to most students at least once, and it makes a strong impression.

Improved software design is another benefit. Functions with convoluted definitions rarely make it past the ACL2 compilation process, which requires a proof of termination before admitting a newly defined function to the logic. This forces function definitions to be concise and easy to reason about.

One might expect complaints from students, members of the faculty, or observers from industry about the use of an unusual programming environment for a required course in software engineering. However, no such complaints have materialized. About half of the students express positive support for the ACL2 approach. So, the idea of using a functional paradigm and a mechanical logic in software engineering has been reasonably well accepted in this educational experiment.

On the basis of conversations with individual students and on evaluations of projects they turned in, it appears that all 158 of the students who have completed the ACL2-based version of software engineering learned enough about functional programming to be able to use it effectively in subsequent projects. About 60% of the students understand how to formulate theorems about software properties, and see some value in stating such theorems. Between 40% and 50% of them can use ACL2's mechanized logic to verify at least a few software properties. More than 20% can formulate theorems that, together, verify a coherent theme of software correctness. Not quite half would choose ACL2 or another functional language for a software project in the future, given the opportunity. A few students (3%–5%) gain competence in using the theorem prover well beyond expectations. They do not become full-fledged experts, but would be able to use ACL2 effectively on their own in new projects and gradually become experts through experience without further formal training.

The percentage of students who embrace functional programming and mechanized logic as useful and desirable tools for software development has increased a bit each semester. One can hardly project a trend based on only three semesters of informal data. However, a standard set of questions bearing on acceptance of functional programming and mechanized logic and personal success in using such methods was added to the course evaluation process in fall, 2005. In a few years, a formal assessment will be possible.

8 Fellow travelers

Graduate-level courses and textbooks on the use of mechanical logic exist (Bjorner, 2006), but the topic seems to have had little exposure at the undergraduate level. The IEEE/ACM guidelines for software engineering education (2004) recommend exposure to formal methods, but not to the use of theorem-proving tools. Nevertheless, the use of symbolic logic to express software correctness properties has found its way into baccalaureate courses, and in some cases these methods are backed up with tools that automate parts of the process.

For example, QuickCheck (Claessen and Hughes, 2000), a tool that generates random tests from specified software properties, and Alloy (Jackson, 2006), a system that supports formal methods in programming, are regularly used in undergraduate courses. These tools expose students to the idea of expressing software properties in terms of Boolean formulas, which is always a first step in using theorem provers to confirm such properties.

Theorem provers have the reputation of imposing a steep learning curve on new users. In the case of ACL2, at least, this reputation is undeserved. Its logic adds no syntax to the underlying language, and software properties are expressed in terms of a small set of logical operations familiar to almost all students (and, or, implication, etc.). Furthermore, the ACL2 theorem prover is powerful enough to complete proofs of significant properties with little or no human intervention. With carefully designed projects, this makes it possible for students to succeed in using the theorem prover with little more frustration than they encounter in learning to use new programming languages.

9 Wishes

9.1 Performance

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 C-like performance from ACL2 code has not yet been accomplished in this experiment.

9.2 Higher order functions

It would also be nice if ACL2 were a higher order language. It is not, and will not be, which is sometimes burdensome from a programming point of view. Many students ask for this feature and are 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, most students avoided higher order functions by using a switch to choose from a fixed collection of functions. So, they did not use higher order functions when they could, and should have. Now that they can not, they want to.

9.3 Interactive graphical user interfaces

ACL2 does not support interactive, graphical user interfaces, and this limits the range of course projects. Students find this disappointing. Fortunately, Vaillancourt and Felleisen are addressing this issue in an ongoing project at Northeastern University. They are creating language support in DrScheme that makes it possible for students to develop and run ACL2 code (and the theorem prover) from the DrScheme environment. SE-II students have used a prototype of this tool, and have given it positive reviews (2006).

9.4 Integrated development/verification

The prototype DrScheme/ACL2 programming environment makes it possible to learn to use ACL2 and its mechanical logic while benefitting from interactive, graphical operations in the DrScheme world.[2] Eventually, one might anticipate teaching software engineering in an environment that closely integrates DrScheme and ACL2. An important element of programming strategy in such an environment could be to design code with explicit contracts on function interfaces (Findler and Felleisen, 2002) and to gradually replace contracts with theorems, eliminating runtime checks. It might also be possible to integrate QuickCheck-style test generation (Claessen and Hughes, 2000) with logic-based contracts.

10 Guesses

Experience in courses like those described here and experience in commercial software projects (Kaufmann *et al.*, 2000b) suggest that a computational logic, integrated into a software development environment, can provide practical benefits in software projects today. The software development process, in this mode, includes use of the programming environment's logic to state important software properties in conjunction with coding. Properties are proved gradually, as a normal software development activity, in parallel with testing.

In other words, mechanized logic is ready for prime time. Unfortunately, few software engineers are ready for mechanized logic. If educators incorporate technologies like ACL2 in courses, the next generation of graduates could begin to reap a benefit that the functional paradigm facilitates more effectively than any other: using computational logic to engineer reliable software.

Acknowledgments

The author thanks the students and research assistants whose experiences provided much of the material of this report, and the referees, whose suggestions led to many improvements. The work was supported by the National Science Foundation, grant no. EIA0082849.

References

Bjorner, D. (2006) *Software Engineering 1: Abstraction and Modelling*. Springer.

Boyer, R. S. & Moore, J. S. (1998) *A Computational Logic Handbook*, 2nd *ed*. Academic Press.

Claessen, K. & Hughes, J. (2000) QuickCheck: A lightweight tool for random testing of Haskell programs. Pages 268–279 of. *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*. Pittsburgh: ACM Press.

[2] Another tool, the ACL2 Sedan (Dillinger *et al.*, 2006), shares the goal of making ACL2 accessible to students. It does not address the issue of developing interactive software, but does provide a modern programming environment for ACL2 and improves its ability to verify termination, a feature of great importance for students.

De Millo, R. A., Lipton, R. J. & Perlis, A. J. (1979) Social processes and proofs of theorems and programs. *Commun. ACM*, **22**(5), 271–280.

Dillinger, P. C., Manolios, P., Moore, J. S. & Vroon, D. (2006) ACL2s: the ACL2 Sedan. *User Interfaces for Theorem Provers Workshop*, August 2006, Seattle, WA. *To appear in: Electronic Notes in Theoretical Computer Science*. Available at: `http://www.cc.gatech.edu/∼manolios/research/uitp-acl2s.html`

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. & Felleisen, M. (2002) DrScheme: A programming environment for Scheme. *J. Funct. Program.*, **12**(2), 159–182.

Findler, R. B. & Felleisen, M. (2002) Contracts for higher-order functions. *Pages of 48–59 of. Proc. 7th ACM SIGPLAN International Conference on Functional Programming*. Pittsburgh: ACM Press.

Humphrey, W. S. (1995) *A Discipline for Software Engineering*. Addison Wesley.

IEEE Computer Society/ACM Joint Task Force on Computing Curricula. (2004) *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. Available at: `http://sites.computer.org/ccse/SE2004Volume.pdf`

Jackson, D. (2006) *Software Abstractions: Logic, Language and Analysis*. MIT Press.

Kaufmann, M., Manolios, P. & Moore, J. S. (2000a) *Computer Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Kaufmann, M., Manolios, P. & Moore, J. S. (2000b) *Computer Aided Reasoning: ACL2 Case Sudies*. Kluwer Academic Publishers.

Michaelsen, L. K. (2002) Getting started with team based learning. *Pages 27–29 of*. Michaelsen, L. K., Knight, A. B. & Fink, L. D. (eds), *Team-Based Learning: A Transformative Use of Small Groups*, Westport, CT: Praeger.

Page, R. L. (2005) Engineering software correctness. *Pages 39–46 of*. Findler, R., Hanus, M. & Thompson, S. (eds), *Proc. 2005 Workshop on Functional and Declarative Programming in Education, 25 September 2005, Tallinn, Estonia*. Pittsburgh: ACM Press.

Pressman, R. (2005) *Software Engineering: A Practitioner's Approach*, 6*th ed*. McGraw-Hill.

Sommerville, I. (2004) *Software Engineering*, 7*th ed*. Pearson.

Vaillancourt, D., Page, R. & Felleisen, M. (2006) ACL2 in DrScheme. *Pages 107–116 of*. Manolios, P. & Wilding M. (eds), *Proc. 6th International Workshop on the ACL2 Theorem Prover and Its Applications, 15–16 August 2006, Seattle, Washington*. Ruben Gamboa.