Functional Programming and Theorem Proving for Undergraduates: A Progress Report

Rex Page
Department of Computer Science
University of Oklahoma
200 Felgar St, Room 119
Norman, OK 73019
page@ou.edu

Carl Eastlund Matthias Felleisen
College of Computer Science
Northeastern University
West Village H 202
Boston, MA 02115
{ cce, matthias } @ ccs.neu.edu

Abstract

For the past five years, the University of Oklahoma has used the ACL2 theorem prover for a year-long sequence on software engineering. The goal of the course is to introduce students to functional programming with "Applicative Common Lisp" (ACL) and to expose them to defect recognition at all levels, including unit testing, randomized testing of conjectures, and formal theorem proving in "a Computational Logic" (ACL2).

Following Page's example, Northeastern University has experimented with the introduction of ACL2 into the freshman curriculum for the past two years. Northeastern's goal is to supplement an introductory course on functional program design with a course on logic and theorem proving that integrates the topic with programming projects.

This paper reports on our joint project's progress. On the technical side, the paper presents the Scheme-based integrated development environment, its run-time environment for functional GUI programming, and its support for different forms of testing. On the experience side, the paper summarizes the introduction of these tools into the courses, the reaction of industrial observers of Oklahoma's software engineering course, and the feedback from a first outreach workshop.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.6 [*Software Engineering*]: Programming Environments; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; K.3.1 [*Computers and Education*]: Computer Uses in Education; K.3.2 [*Computers and Education*]: Computer and Information Science Education

General Terms Design, Reliability, Security, Verification

Keywords mechanical logic, test-driven development, formal methods, predicate-based testing, Dracula, DrScheme

FDPE'08, September 21, 2008, Victoria, BC, Canada.

1. Theorem Proving in Software Engineering

Five years ago, Page [18] started teaching a senior-level course sequence on software engineering in ACL2 [4, 14, 15]. The course sequence has three goals. The primary goal is to instruct students in the theory and practice of the construction of reliable software systems. An external industrial board reviews the student projects at the end of the year to ensure quality standards. The second goal is to expose students to functional programming in ACL2 and to demonstrate how functional programming eliminates some of the common headaches from the software engineering process. The third goal is to introduce theorem proving as a quality assurance technique (beyond testing).

The first of the two courses is mostly a theoretical introduction to software engineering concepts, functional programming at the list and structure level, and theorem proving. The programming and theorem proving projects help students build expertise and small libraries (e.g., packages for AVL trees or statistical functions) for the second course. There, students apply the principles to a semester-long programming project with many separate deliverables. The project varies from year to year. As a few examples, past projects have included a stock market analysis program, an interactive graphical game, and a graphics processor.

While teaching this course for the first year, Page discovered two problems with ACL2. First, students had difficulty coping with the complex, text-based programming environment for ACL2; most students were used to graphical IDEs. Second, while ACL2 supports batch-style file i/o, it does not include any facilities for building modern interfaces.

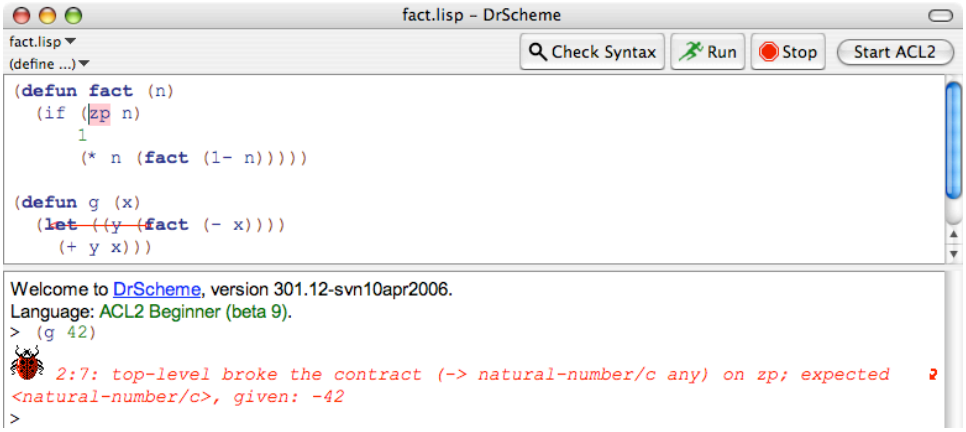

Figure 1. Safety check violation in DRACULA

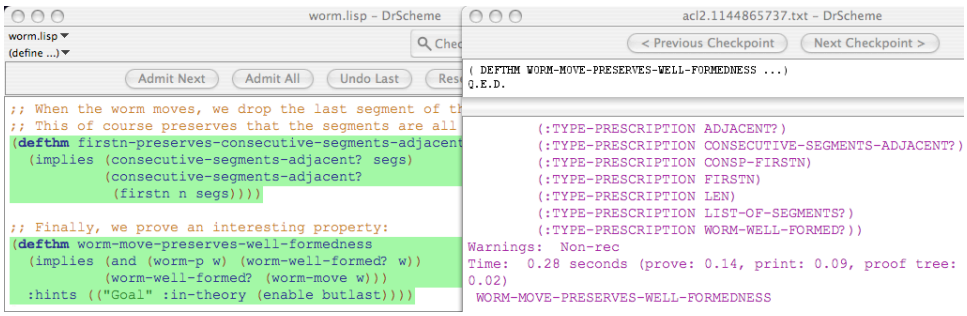

Figure 2. Interacting with ACL2 via DRACULA

Accordingly, Page's report at FDPE 2005 [17, 18] called for a GUI-based programming environment (IDE) for ACL2 that would include interactive, graphical i/o.

In response, Felleisen and Vaillancourt constructed such an IDE for Page in December/January of 2005/06. They created DRACULA, an environment for ACL2, using the DrScheme infrastructure [11, 12], and managed to deliver a prototype that was usable in the classroom [19]. Page and Felleisen's teams have continued their collaboration since then with the support of a joint NSF grant.

Over the past couple of years, we have improved this prototype, which now includes facilities for unit testing and automated, predicate-based testing (dubbed DoubleCheck). These tools have had a significant impact on the teaching of the software engineering course and have also been used in a freshman course at Northeastern [8]. The following sections discuss the IDE enhancements, in theory and practice, as well as recent accomplishments in the software engineering course. The paper concludes with a description of the project's outreach efforts and plans for future work.

2. DRACULA: Theorems and Programs

This section presents DRACULA, the suite of software tools that we developed in support of the software engineering course. Roughly speaking, DRACULA is an integrated programming environment built on top of DrScheme, with tools for functional graphical programming, unit testing, random testing, and a GUI connection to a theorem prover. The first subsection is a general introduction to DRACULA; the last two focus on its support for testing.

2.1 DRACULA

ACL2 stands for "A Computational Logic for Applicative Common Lisp". It consists of two parts: a first-order, purely functional subset of Common Lisp ("Applicative Common Lisp"), and an equational logic for reasoning about it ("A Computational Logic"). DRACULAis a partial embedding of the former into DrScheme. DRACULA thus inherits most of DrScheme's tools and some of its teaching libraries, including a library for creating interactive GUI games. Because we also wished to use DRACULA for the freshman logic course at Northeastern, we decided to make its primitives safe.[1] DRACULA leaves the computational logic to ACL2,

[1] In ACL2 terminology, the guards are enabled.

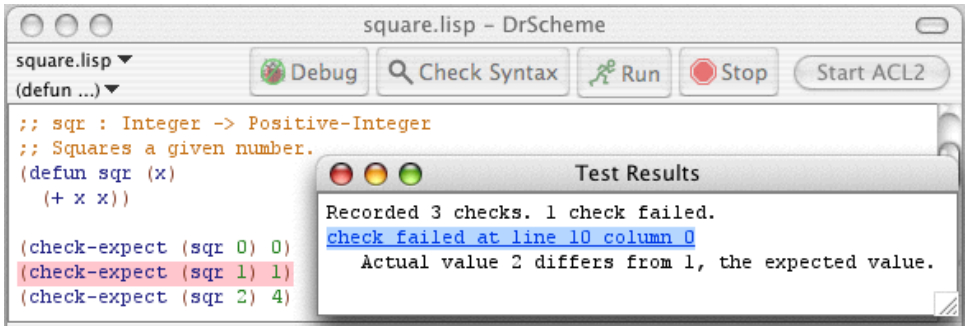

Figure 3. Unit test failure

providing an interface to the theorem prover when users need to reason formally about their program.

Figure 1 displays a snapshot of DRACULA. The IDE consists of three parts: a console with four buttons for controlling the IDE, plus an optional SAVE button; a definitions window; and an interactions window. The definitions window is a language-sensitive editor, which in this case contains function definitions for *fact* and *g*. Once the student clicks the RUN button, the definitions are evaluated and the focus shifts to the bottom pane, i.e. the interactions window. The interactions window roughly acts like a Lisp read-eval-print loop.

In the screen shot, DRACULA is requested to run (*g* 42). Since *g* then calls *fact* with -42, the safety check for *zp*—whose domain proper is the set of natural numbers—raises an exception. DRACULA highlights the use of *zp* that causes the error and, with a failure arrow, also shows the pending call to *fact* and where its result would be used (through *y* and **let**).

For the proof of theorems, DRACULA relies on ACL2's logic. A student can launch the theorem prover with a click on START ACL2 (top right). As the screen shot in figure 2 shows, doing so brings up a "report window" (on the right) and opens an additional command pane in DRACULA, with buttons for sending definitions and theorems from the definitions window to the theorem prover.

The additional console supports five pieces of IDE functionality. The first admits the next definition or theorem that hasn't been sent to the theorem prover yet. If the prover accepts it, DRACULA highlights it in green; otherwise it is turned red. The second admits the entire definitions window. The third undoes the last admit. The fourth button resets the theorem prover to its initial state, and the last one forces the theorem prover to shut down.

2.2 Unit Tests

Unit tests are an integral part of program design [3], and our courses teach students to systematically create examples and to turn them into tests [10]. Testing is also a crucial precursor to automated formal verification of a program, because it eliminates basic mistakes for which the use of theorem provers is too expensive. DRACULA's unit test mechanism allows students to express concrete applications of their functions that are checked on each run of the program. Before writing the function, students construct valid inputs and corresponding expected outputs for the function. Afterward, DRACULA reports any failures with a link to the "offending" example.

The example in figure 3 shows a flawed program—sqr doubles its argument instead of squaring it—and unit tests checking sqr on inputs 0, 1, and 2. DRACULA reports a failure in the second case, providing the actual result 2, the expected result 1, and the location of the test within a link that displays and highlights it. This provides students with an immediate, concrete counterexample from which to debug their program.

2.3 DoubleCheck

The DoubleCheck library, inspired by QuickCheck [2, 5], provides automated testing for program properties based on random inputs. Students write predicate-based properties about their program, providing a distribution of values for each input. DoubleCheck's name hints at dual modes of verification. ACL2 verifies properties as theorems; DRACULA's ACL simulation tests properties using inputs generated randomly from their distributions, reporting the chosen values in case of failure. Thus, students may use the full power of the theorem prover, but also have assistance finding counterexamples when theorem proving fails and they need to check whether their conjecture may not hold.

Programmers declare new properties using the **defproperty** syntactic form:

(**defproperty** *name count*
  ((*variable distribution hypothesis*) . . .)
  conclusion)

The property, called *name*, is evaluated *count* times when the program is run. During a trial, DoubleCheck binds each *variable* to a value chosen from the given *distribution*. A variable's *hypothesis* constrains its values, via implication during theorem proving and as a filter on the random dis-

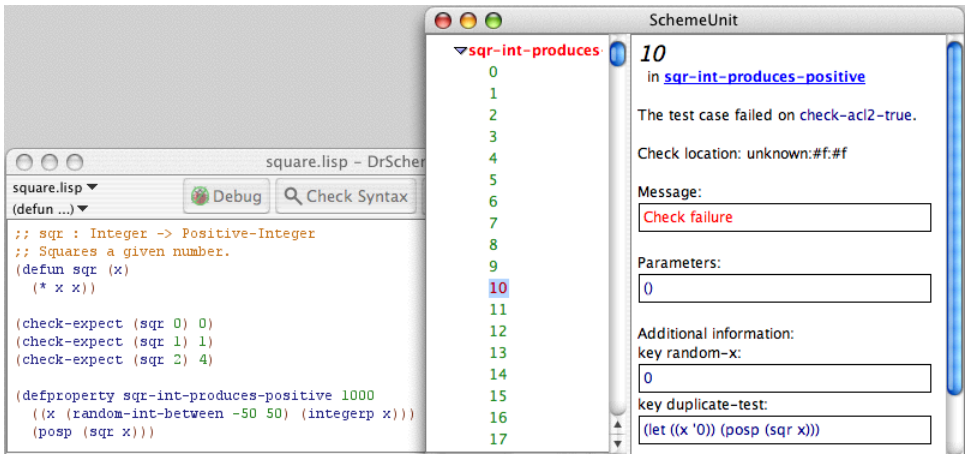

Figure 5. DoubleCheck failure

(random-int-between *i j*)
- generates n, $i <= n <= j$

(random-boolean)
- generates **t** or **nil**

(random-string)
- generates a string containing random characters

(random-list-of *d*)
- generates a list of elements from distribution *d*

(random-apply *f d* ...)
- applies *f* to values from each distribution *d*

(**defgenerator** *name* (*arg* ...) (*w d*) ...)
- defines a distribution *name* that chooses among potentially recursive sub-distributions *d*, each with weight *w*

Figure 4. Random distributions in DoubleCheck

tribution during random testing. The *conclusion* expression determines the final test.

DoubleCheck provides a library of random distributions of values, with tools for users to combine them and to create their own. Figure 4 lists some of the distributions. Random distributions may be used only within **defproperty**, as their behavior is both higher-order and side-effecting, and not compatible with ACL2's logic.

Figure 5 shows a correct implementation of sqr with a well-formed but untrue property. The **defproperty** form declares a property named *sqr-int-produces-positive*, to be given 1000 trials on each run. The second line declares the property's inputs; in this case, a single variable x. Values for x are chosen from a distribution of integers between -50 and 50, and must satisfy integerp. The final line declares the property's test: sqr must produce a positive integer.

The window on the right shows a counterexample to the conjectured property. At the bottom-right, the *random-x* key shows that x was 0; the *duplicate-test* key provides an expression that can be copied to create a new unit test. The user can add this unit test before fixing the function, building up a regression suite so old bugs won't creep back in.

3. Working with DRACULA

In this section, we present functional GUI programming and a simple interactive computer game in DRACULA: Worm (a.k.a. Snake or Nibbles). The first subsection sketches the underlying GUI library. The second subsection describes the construction of a reasonably large example and working with DoubleCheck and the theorem prover to validate conjectures about the program's behavior.

3.1 Functional Interactive GUI Programming

Northeastern's introductory course uses DrScheme to teach program design principles [10]. In order to make the course entertaining, the course software includes a library, dubbed a teachpack, for manipulating images (which are first-class values) and creating animations. As a matter of fact, the very first program—our "hello world" program—shows a rocket lifting off (at constant speed). DRACULA inherits this teachpack.

The GUI library is an "abstract machine" with five instructions:

1. The *big-bang* instruction creates a world and with it a visible canvas:

   ;; (*big-bang width height rate init*)
   ;; *width*, *height* : *Nat*
   ;; *rate* : *Rational*
   ;; *init* : *State*

The first two arguments determine the size of the screen; the third one specifies how often the clock ticks; and the fourth one is the initial state of the world.

2. Every time the clock ticks, the library performs a program-specific *tick* callback:

   ;; (*on-tick-event tick*)
   ;; *tick : State → State*

   The callback is registered via *on-tick-event*; its purpose is to transform the state at time t into the state at $t + 1$.

3. Every time the user presses a key, the library performs a program-specific *key-event* callback:

   ;; (*on-key-event key-event*)
   ;; *key-event : State Symbol → State*

   The callback is registered via *on-key-event*; it updates the state in response to each key pressed.

4. When it is time to refresh the image on the canvas, the library invokes the program-specific *to-image* callback:

   ;; (*on-redraw to-image*)
   ;; *to-image : State → Image*

   The callback is installed with a call to *on-redraw*; it consumes the current state of the world and returns an image. The library sends the image to the canvas.

5. After each *tick* or *key-event* callback, the library queries the program-specific *game-over* predicate:

   ;; (*stop-when game-over*)
   ;; *game-over : State → Boolean*

   The callback is installed by *stop-when*; it consumes the state of the world and reports whether the animation is done. Once *game-over* returns true, the library renders the final state on the canvas and stops the world.

From a semantic perspective, this form of i/o control is closely related to Clean's model of input and output [1]. It is also somewhat reminiscent of the Yale school of functional reactive programming [6, 9, 13].

In addition, the library provides basic primitives to manipulate images, which programmers can insert into the editor or which the program can create on the fly. Lastly, the library provides a pseudo-random number function.

The machine perspective is useful for proving theorems about such GUI programs. The typical approach is to show that the state transforming callbacks preserve some invariant and that the initial state also satisfies this property. We can then conclude that all game states have the property.

3.2 Worms, Theorems, and Properties

The Worm game is representative of the projects assigned to freshmen in their first programming course at Northeastern University; the courses at Oklahoma use larger projects than this but the principles remain the same.

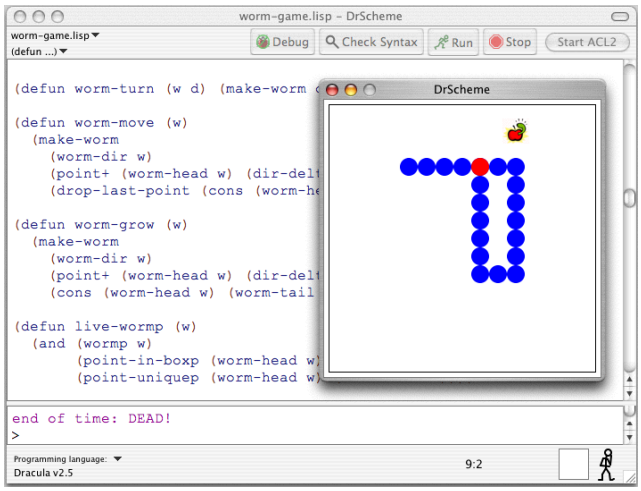

Figure 6. Interactive games in DRACULA

Here is a concise description of the problem:

> The player controls the direction of a constantly-moving worm on a grid. The grid has walls and contains a piece of food. If the worm eats the food, the player gains points, the worm grows in length, and a new piece of food appears. If the worm runs into a wall or its own tail, the game ends.

Producing the game requires only four (compound) data structures: Cartesian points, lists, game state, and images. The game has only three rules (move, grow, or die), and the player controls only the worm's direction. See figure 6 for a screen shot of a functioning Worm game.

Instead of proving the correctness of such a game, instructors usually focus on basic invariants that hold across the entire game. For the worm game, we choose to prove that at any point in the game, the food and the tail of the worm are inside the visible portion of the grid. Other candidate invariants would state that the worm consists of adjacent points and that the worm's tail does not cross itself.

The specific correctness predicate for the game state is shown in figure 7 in terms of two helper functions *point-in-boxp* and *point-list-in-boxp* for testing whether points lie in a bounding box. Verifying the game means proving the initial state satisfies the (partial) correctness predicate, then proving that each transition function preserves the predicate when called before *game-over* (i.e., on a game in which the worm has not yet run into a wall or itself).

When the theorem prover fails, we can attempt to assist it with lemmas or hints, or we can look for conflicts between the program and the conjecture. DoubleCheck facilitates the latter; we convert the final two **defthm** forms to **defproperty** to generate counterexamples; see figure 8.

```
;; in-bounds-gamep : Game → Boolean
;; Reports whether the food and worm tail of a game
;; are in bounds.
(defun in-bounds-gamep (g)
  (and (point-in-boxp
         (game-food g)
         0 0 (- *grid-width* 1) (- *grid-height* 1))
       (point-list-in-boxp
         (game-worm-tail g)
         0 0 (- *grid-width* 1) (- *grid-height* 1))))

;; Verify the game's initial state.
(defthm initial-game/in-bounds-gamep
  (in-bounds-gamep *initial-game*))

;; Verify the game's time transition.
(defthm game-tick/in-bounds-gamep
  (implies (and (gamep g)
                (not (game-over g))
                (in-bounds-gamep g))
           (in-bounds-gamep (game-tick g))))

;; Verify the game's user interaction.
(defthm game-key/in-bounds-gamep
  (implies (and (gamep g)
                (not (game-over g))
                (in-bounds-gamep g)
                (symbolp key))
           (in-bounds-gamep (game-key g k))))
```

Figure 7. Main theorems for Worm game.

```
;; random-uniform-list :
;;     (Random X) → (Random (Listof X))
;; Generates a list of elements taken from elem-dist.
(defgenerator random-uniform-list (elem-dist)
  (1 nil)
  (3 (random-apply cons
                   elem-dist
                   (random-uniform-list elem-dist))))

;; random-worm : → (Random Worm)
;; Generates a random worm.
(defgenerator random-worm ()
  (1 (random-apply build-worm
                   (random-dir)
                   (random-point-in-bounds)
                   (random-uniform-list (random-dir)))))

;; random-game : → (Random Game)
;; Generates a random game state.
(defgenerator random-game ()
  (1 (random-apply make-game
                   (random-int-between 1000000000
                                       2000000000)
                   (random-point-in-bounds)
                   (random-worm))))

;; Test the game's time transition.
(defproperty game-tick-in-bounds 100
  ((g (random-game) (and (gamep g)
                         (not (game-over g))
                         (in-bounds-gamep g))))
  (in-bounds-gamep (game-tick g)))

;; Test the game's user interaction.
(defproperty game-key-in-bounds 100
  ((g (random-game) (and (gamep g)
                         (not (game-over g))
                         (in-bounds-gamep g)))
   (k (random-dir) (symbolp k)))
  (in-bounds-gamep (game-key g k)))
```

Figure 8. DoubleCheck properties for the Worm game

The *random-uniform-list* generator duplicates the built-in generator random-list-of, but is presented here to show a recursive use of **defgenerator**. One out of four times, the generator produces **nil**; three in four times, it extends a recursively chosen tail list with an element chosen independently from *elem-dist*.

The *random-worm* and *random-game* generators use random-apply to pass randomly chosen inputs to *build-worm* and *make-game*, respectively. The **defgenerator** forms have only a single clause with a weight of 1. The *build-worm* function consumes the direction of a worm's motion, the position of its head, and the directions (up, down, left, or right) between adjacent worm segments. Note that *build-worm* converts the directions to a list of adjacent points. This is more straightforward than making a distribution of adjacent points directly, as the directions may be chosen independently. The *make-game* function consumes a random seed (in the sense of the functional pseudorandom library used by the game), a location for the food, and a worm.

Finally, the *game-tick-in-bounds* and *game-key-in-bounds* properties restate the hypotheses and conclusions of the original theorems, providing random inputs from the generators defined above. Note that the hypothesis predicate for random games tests *game-over*, constraining the distribution by filtering out games where the worm's head overlaps its tail. On the other hand, the predicate for random keys tests only *symbolp*, allowing more values than our distribution, as we test only the four cardinal directions (corresponding to the arrow keys).

4. Pedagogical Experiences

The two-course sequence in software engineering at the University of Oklahoma has completed its fifth year of incorporating functional programming and elements of mechanical logic as a significant part in two of three major themes typical of course offerings in software engineering: design, quality, and process. Functional programming has

the greater impact in the design theme, mechanical logic plays a strong role with the quality theme.

Major steps forward include moving from a dual programming language environment to an integrated environment (DRACULA), and most recently the addition of DoubleCheck. The addition of an automated, predicate-based testing component was the primary difference between the courses this year and those of previous years.

In the first course, students work individually and in teams to complete five to seven projects ranging from a hundred to a thousand lines of code, with supporting process documentation and validation records. The second course requires a project of a few thousand lines with a dozen separate deliverables. The first course is about two-thirds individual work and one-third teamwork, and those ratios reverse roles in the second course.

Over the years, a collection of about two dozen carefully designed and reusable projects have gradually emerged. Careful design of projects is necessary to make sure they are feasible for novices in functional programming and mechanical logic, in the same way that projects for novice programmers must be carefully designed for feasibility. Project topics include general purpose utilities for list processing and applications of such utilities in cryptography, graphics, parsing, statistical analysis, text analysis and formatting, transcendental functions, routing algorithms, audio signal processing, and image processing.

Course prerequisites include symbolic logic, and most students acquire that prerequisite in the form of a course that applies logic directly to the verification of properties of digital circuits and software [16]. For most students, the course serves as their first serious experience with functional programming. Almost all students now succeed in this aspect of the course. About a third say in an anonymous, end-of-course survey that they would be inclined to use functional programming in future projects, given the option.

Most students are able to express informal conjectures about software properties as formulas in logic and to test for them using DoubleCheck's predicate-based random testing. Put differently, DoubleCheck helps students with one of the trickiest task of using logic in programming, namely, the transition from ideas to formal statements. Unfortunately, the distance between DoubleCheck and the theorem prover is still too large.

The percentage of students who acquire an understanding and ability to use the ACL2 theorem prover effectively varies from ten to twenty percent, depending on the year. Proving theorems—mechanically or manually—calls for a higher degree of mathematical sophistication than many students have. Students with good mathematical backgrounds usually do well, and those without learn to appreciate its value. Almost all students leave the courses with a revised view of software correctness. This point is where the applications of predicate-testing and logic—as emphasized in the course—apparently have their greatest, long-term effects.

Since 2004 the course evaluation questionnaire that students have filled out at the end of the course has included five supplementary questions asking them to assess their understanding of ACL2 programming and their ability to use the ACL2 theorem proving technology, and also their inclination to use ACL2 or another system with a built-in mechanical logic in future work. The results are remarkably consistent, varying by a few percentage points up or down each year. Well over two thirds of students rate their ACL2 programming abilities highly. Over a third believe they can use the mechanical logic technology effectively (a higher figure than the instructor's estimate based on homework and exam results), and about the same percentage would welcome the opportunity to use ACL2-like technology in future work.

Near the end of the second course, student teams make a half-hour presentation of their software project. Eight to twelve members of the departmental advisory board (prominent members of the industrial software development community, mostly engineering managers, but also senior programmers and senior managers) attend the presentations and comment, both orally and in writing, during a question/answer session at the end of each presentation.

Presentation guidelines direct the teams to target an audience of engineering managers unfamiliar with the specific project, but familiar with company goals and procedures. Coverage must include software architecture, implementation problems and solutions, defect prevention, comparison of planned and actual schedules, remaining implementation problems, potential enhancements, and a demonstration of their software product in operation.

Reactions of advisory board members to both the course work and the presentations have been uniformly favorable over the years. The board members appreciate the emphasis on defect prevention as an important aspect of software quality and see the value in stating and validating software properties in terms of formal, testable and mechanically verifiable statements. Somewhat surprisingly, they also see the value in exposing students to the functional programming paradigm as an alternative to conventional programming, in spite of the fact that, in the past five years, only three of the approximately thirty different board members have been involved in projects making significant use of functional programming.

Last year, one board member, Stephen Mercer, Lead Developer in R&D for LabVIEW at National Instruments, was especially impressed with a bit-plane graphics property that a team verified using the ACL2 theorem prover:

> I saw ACL2 for the first time when observing Dr. Page's senior capstone course. I was blown away by the power of it. ACL2 provides a meta-language for programming. Students have been taught for years to

write comments about the pre-conditions and post-conditions of their functions. ACL2 gives us a way to write those comments in a way that is meaningful to the compiler and then have the compiler verify that they are true. This is the biggest development in programming that I've heard of in a decade.

But could students use this reasoning system? Proving program correctness, even just setting up the problem, was non-trivial in my experience. When watching the capstone presentations, I saw that students could use the system, and quite easily if they took the time to do it. Rigorously proving every single aspect of a function was asking a bit much, but simple proofs to show zero out-of-range errors for arrays or guarantee a floor to a recursion function were common. Proving just those small bits represents a huge step forward in quality. And one group showed the true power of this concept by writing a 3D rendering engine and then proving that their engine had no bit plane errors: ask two objects to draw and you'd never end up with the background object drawing in front of the other object. Proved. Not "yeah, we think so" or "we desk checked it thoroughly" or even "we've never seen a problem in years of empiric testing." This was a hard math proof of correctness. I was excited.

Since I work in language design, I was eager to find out if the ACL2 research could apply to the language I work on (LabVIEW). I got back to the office and found that we already had research ongoing in that area. LabVIEW is a dataflow programming language, which makes it "topologically" similar to functional languages. We already have a working transformer to generate ACL2 code from LabVIEW code for a large subset of our language. Someday we hope to expose in our graphical language the same power that ACL2 is exposing in text languages.

Comments on the presentations can be viewed online.[2]

5. Outreach

In May of this year, thirteen computer science instructors attended the Teaching Software Correctness Workshop and were able to experience some of the software engineering course materials themselves, including both lectures and projects. The workshop ran three days, eight hours per day. About a third of the time was spent in lectures and discussions and two thirds in project work. The course staff consisted of the first two authors, plus two students who were familiar with the software engineering course and had explored the use of DRACULA outside normal course work, using tools from the courses.

The workshop goal was to spread the use of mathematical logic, especially mechanical logic, in undergraduate computer science course work. The lectures focused primarily on material from the OU software engineering courses, to give participants a feeling for the educational impact on students. Projects had the same format and intent. Discussions made it possible for participants to share their own use of logic in the classroom and comment on how mechanical logic might be integrated into their educational environment.

During project work periods, the workshop staff continuously interacted with workshop participants. During these work periods, we observed that:

- Instructors run into the same problems as students in trying to work through projects. Future outreach workshops must start with low expectations and carefully introduce all basic notions. We may also extend the workshops from three to five days.
- It is easier to deal with formal statements of software properties (formulas in logic) than it is to convert informal statements to formal ones.
- A carefully constructed experience in stating and proving theorems expressing software properties is a gentler way to intitiate novices than predicate-based testing.

  In other words, students and instructors must first learn to translate informal claims into formal, universally quantified expressions. Given formal conjectures, let the theorem prover take over. When it fails, programmers should use DoubleCheck to inspect the relationship between the program and their conjecture.
- An understanding of the interplay between formal logic and informal notions of software properties gained from experience on the formal side improves the ability of novices to convert their informal notions into cogent tests and formal statements of software properties.
- Carefully constructed project work does lead to a reasonable facility with the use of mechanized logic for predicate-based testing and mathematical proof of software properties.

Workshop participants suggested additional ways to improve the DRACULA facility for predicate-based testing. Most importantly, the future syntax of DoubleCheck's predicates must be more closely aligned with ACL2's syntax for theorems than in the current version. We are now at work in making changes to respond to the participant's suggestions.

Another positive outcome of the workshop is the formation of a working group that will host periodic meetings to share progress on the infusion of mechanical logic and other formal methods into the computer science curriculum.[3]

The workshop website[4] contains all lecture and discussion notes, twelve small to medium sized projects with both exemplary solutions and solutions created by participants

[2] `http://www.cs.ou.edu/~rlpage/SEcollab/prescom/`

[3] `http://www.resource-aware.org/twiki/bin/view/ProgrammingLanguages/MEPLS`

[4] `http://www.cs.ou.edu/~rlpage/SEcollab/tsc/`

during the workshop, two larger projects with participant solutions, and responses by participants to a daily, workshop-assessment questionnaire. Solutions are protected on the website, but available to computer science instructors interested in using them in courses.

6. Conclusion

This report summarizes the progress we have made since our first FDPE presentation on the use of theorem proving in software engineering courses [17, 18].

After five years, we and our industrial partners confirm that software engineering courses can (and ought to) convince students that quality matters. Our experience demonstrates that the use of functional programming, a theorem prover, and a facility for random testing of predicates make a significant impression on students and changes their minds about the quality of software.

Both the courses at Oklahoma University, at Northeastern University and the outreach workshop also show that such a course benefits tremendously from a tightly integrated suite of tools. DRACULA's seamless support for test-driven design, random testing, and theorem proving has significantly improved the software engineering course at Oklahoma and has also enabled us to run the workshop.

Over the next couple of years, we intend to enrich the course in two different ways. On the pedagogical side, we will produce recipes for the joint use of the theorem prover and random testing. On the software side, we will equip the variant of ACL2 in DRACULA with modules, enabling student teams to work independently on separate parts of a project and reducing the cost of theorem proving.

References

[1] Achten, P. and M. J. Plasmeijer. The ins and outs of Clean i/o. *Journal of Functional Programming*, 5(1):81–110, 1995.

[2] Arts, T. and J. Hughes. Erlang/QuickCheck. In *Ninth International Erlang/OTP User Conference*, November 2003.

[3] Beck, K. and E. Gamma. Test infected: Programmers love writing tests. In *Java Report*, volume 3, pages 37–50, 1998.

[4] Boyer, R. S. and J. S. Moore. Mechanized reasoning about programs and computing machines. In Veroff, R., editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 146–176. The MIT Press, Cambridge, Massachusetts, 1996.

[5] Claessen, K. and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[6] Cooper, G. H. and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Sestoft, P., editor, *15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2006.

[7] Dillinger, P. C., P. Manolios, J. S. Moore and D. Vroon. ACL2s: The ACL2 Sedan. In *Proceedings of the 7th Workshop on User Interfaces for Theorem Proving*, volume 174(2) of *Electronic Notes in Theoretical Computer Science*, pages 3–18. Elsevier, 2006.

[8] Eastlund, C., D. Vaillancourt and M. Felleisen. ACL2 for freshmen: First experiences. In *ACL2 '07: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 200–211, New York, NY, USA, 2007. ACM Press.

[9] Elliot, C. and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, 1997.

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

[11] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[12] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In Glaser, H., P. Hartel and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 369–388, Southampton, UK, September 1997. Springer.

[13] Ignatoff, D., G. H. Cooper and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In Hagiya, M. and P. Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006*, volume 3945 of *Lecture Notes in Computer Science*, pages 259–276. Springer, 2006.

[14] Kaufmann, M., P. Manolios and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

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

[16] Page, R. Software is discrete mathematics. In Runciman, C. and O. Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003*, pages 79–86. ACM, August 2003.

[17] Page, R. Engineering software correctness. In *Proc. 2005 Workshop on Functional and Declarative Programming in Education*, pages 39–46, New York, NY, USA, 2005. ACM.

[18] Page, R. Engineering software correctness. *Journal of Functional Programming*, 17(6):675–686, April 2007. Preliminary presentation at FDPE '05.

[19] Vaillancourt, D., R. Page and M. Felleisen. ACL2 in DrScheme. In *ACL2 '06: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 107–116, New York, NY, USA, 2006. ACM Press.