ACL2 in DrScheme

Dale Vaillancourt
Northeastern University
Boston, MA 02115
dalev@ccs.neu.edu

Rex Page
School of Computer Science
University of Oklahoma
200 Felgar St, Room 119
Norman, OK 73019-6151
page@ou.edu

Matthias Felleisen
Northeastern University
Boston, MA 02115
matthias@ccs.neu.edu

ABSTRACT

Teaching undergraduates to develop software in a formal framework such as ACL2 poses two immediate challenges. First, students typically do not know applicative programming and are often unfamiliar with ACL2's syntax. Second, for motivational reasons, students prefer to work on projects that involve designing interactive, graphical applications.

In this paper, we present DRACULA, a pedagogic programming environment that partially solves these problems. The environment adds a subset of Applicative Common Lisp to DRSCHEME, an integrated programming environment for Scheme. DRACULA thus inherits DRSCHEME's pedagogic capabilities, especially its treatment of syntax and run-time errors. Further, DRACULA also comes with a library for programming interactive, graphical games. The library interface forces students to think of a graphical user interface in terms of state-transition functions, enabling them later to prove theorems about their games in ACL2. DRACULA provides a graphical front-end to the ACL2 theorem prover for this purpose. In short, DRACULA allows the formulation of student projects that represent an important intermediate point between data structure exercises in theorem proving and industrial applications.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.6 [**Software Engineering**]: Integrated Programming Environments; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

DRSCHEME, ACL2, formal methods, pedagogy

Keywords

DRSCHEME, ACL2, TeachScheme!

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. TEACHING FORMAL METHODS

Over the past four years, the second author has designed and implemented a senior-level course sequence on the construction of sound software at Oklahoma University [16]. The courses employ ACL2 as a programming language and theorem prover. The first semester course teaches applicative programming and the basics of theorem proving. The second course focuses on a single project, continuing the training in formal methods in this larger context and also developing software management skills.

Both students and external industry observers consider the course a success. Students give the course high scores in evaluations, and industrial observers praise how much students learn about managing a large project and producing reliable software.

Unfortunately, two problems recur every year. First, students have little or no background in applicative programming. Although the syntax of Applicative Common Lisp is relatively simple compared to conventional languages, students have serious difficulties writing well-designed applicative programs and tracking down errors in their programs.

Second, due to ACL2's nature, the homework assignments and projects in this course are small or old-fashioned. Existing case studies [15] are either too complex for undergraduate courses or require domain knowledge that is beyond a typical computer science undergraduate. What students want is a language rich enough to build programs with interactive, graphical user interfaces, such as computer games. The result is that even though the undergraduate students like the course, they remain deeply skeptical about formal methods in the software development process. The students are left with the impression that formal methods are inapplicable for the kind of software development they have encountered before and that the applicative style of programming does not scale to modern software development.

In this paper, we present a software tool for correcting this impression, and we report on teaching experiences with the tool. In a sense, the problems of teaching Applicative Common Lisp are exactly the same as the problems of teaching Scheme or functional programming in a first semester course. The PLT research team under the direction of the third author has responded to this challenge with the development of DRSCHEME, a pedagogic and graphical programming environment, and TeachScheme!, a design-oriented curriculum. Based on the positive experience with DRSCHEME and TeachScheme!, we have developed DRACULA, an embedding of Applicative Common Lisp into DRSCHEME, with a connection to its theorem-prover companion, ACL2.

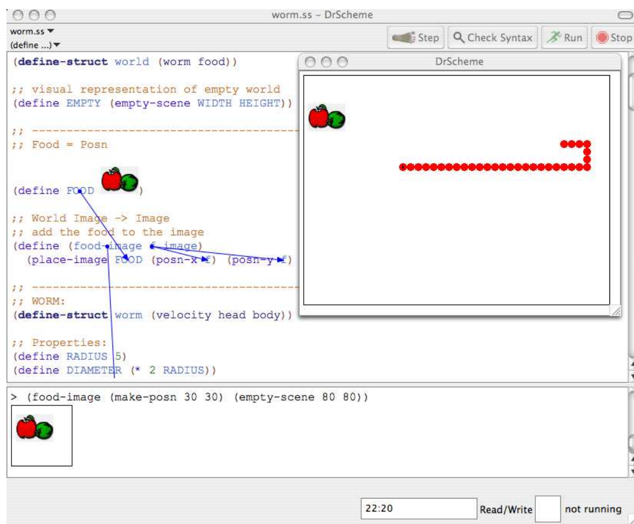

Figure 1: DrScheme

DrACuLa consists of two components: (1) an implementation of Applicative Common Lisp inside DrScheme for running programs, and (2) a graphical interface between DrScheme and a conventional ACL2 implementation for reasoning about programs. The language implementation provides a novice-oriented fragment of Applicative Common Lisp, and the graphical interface to ACL2 obviates the need for students to interact with the ACL2 REPL for theorem proving activities.

Because DrACuLa is inspired by DrScheme's support for teaching programming, the next section reviews the design of DrScheme. Section 3 describes DrACuLa, and section 4 reports on our first classroom experiences with this new environment. The remaining sections sketch implementation techniques and discuss related work.

2. A TOUR OF DrScheme

DrScheme is a graphical and pedagogic programming environment for PLT[1] Scheme [12, 13], originally intended for college freshmen courses and high school courses on program design. PLT constructed the environment in response to two observations. First, high school students are already used to applications with graphical user interfaces; they only reluctantly accept Emacs, the preferred programming environment in academia. Second, most available programming environments are too complex and confuse students. They represent a steep obstacle rather than a gentle slope to the novice programmer. Putting them in front of those is comparable to putting novice pilots into a Boeing 747 or Airbus 380 simulator rather than in a small twin-engine plane simulator.

This section provides a high-level overview of DrScheme. Readers who are familiar with the software and its role in teaching [5, 9] may safely skip it.

Figure 1 shows a screenshot of DrScheme, with a student program running a simple version of the well-known worm (or snake) game. The screenshot demonstrates how simple DrScheme looks to a novice student of applicative, parenthesized programming. There are three panels. The bottom panel is called an Interactions Window, roughly the familiar read-eval-print (REPL) loop from interactive Lisps, ACL2, and Scheme. The middle panel is a Definitions Window; we encourage students to define their functions and variables there instead of the REPL. The top panel is a "control" panel, with four carefully selected buttons:

Run evaluates the definitions in the editor and makes them available in the Interactions Window. There students can explore the workings of a function through experiments in the familiar Lisp manner.

Stop enables students to stop a run-away evaluation.

Check Syntax analyzes the syntax and the scope of the program in the editor. After a syntax check, the student can inspect the binding structure of the program. When the cursor moves over an identifier, DrScheme draws an arrow from the binding occurrence to the bound occurrence. As figure 1 shows, students can also tack these arrows as they explore the program. Finally, with a right click students can perform α-renamings of bound variables.

Step allows students to step through the algebraic evaluation of an expression. In contrast to conventional steppers, DrScheme's stepper explains an execution via a reduction sequence in the spirit of Plotkin's call-by-value λ-calculus [18]. Of course, to novices, this form of semantics looks just like their eighth-grade pre-algebra homework, though enriched with algebraic datatypes. Figure 2 displays one such reduction step. The redex on the left is green, and the contractum on the right is purple.

A fifth button (**Save**) shows up when the program in the Definitions Window is modified and hasn't been saved yet.

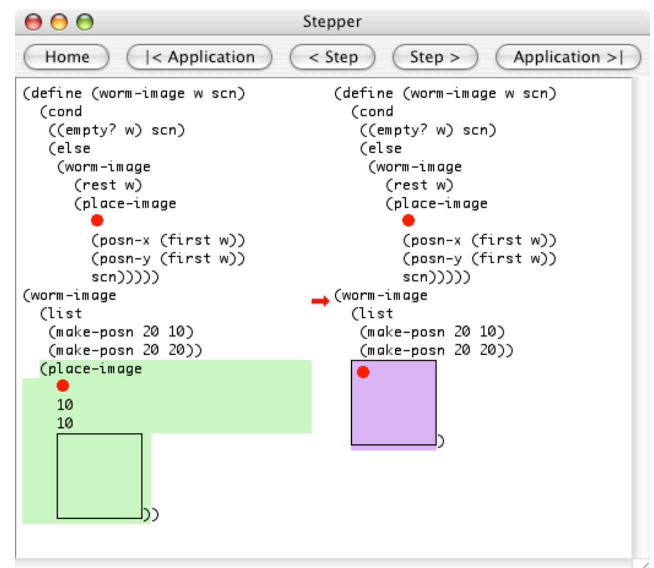

Figure 2: Algebraic Stepping with DrScheme

[1]PLT is a loosely coupled research group, founded by the third author during his time at Rice University.

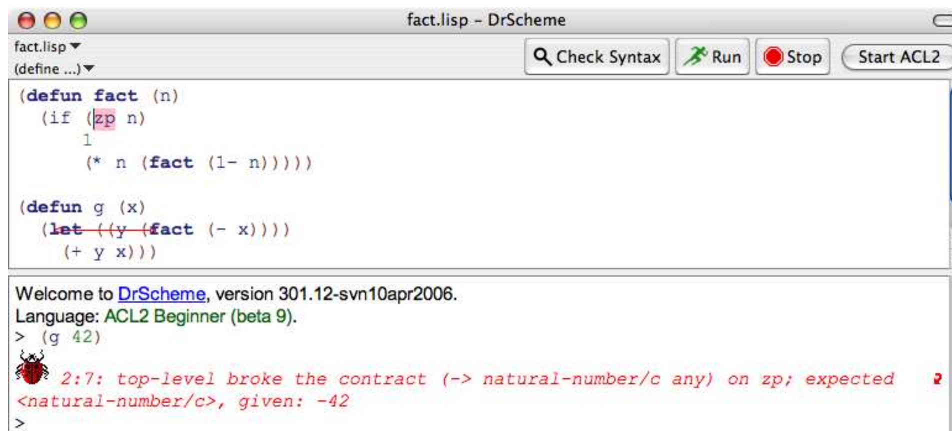

Figure 3: Safety check violation in DRACULA

In addition to being a graphical programming environment, DRSCHEME supports graphical programming in two senses. First, it supports images as first-class values; that is, images have roughly the same status as numbers, strings, and other first-order ("flat") values. In the screen shot of figure 1, the variable *FOOD* is just bound to an image value. In this case, the image is an imported GIF file. Programs can also build images from primitive image-generating functions and composition operators such as `overlay`. Second, DRSCHEME supports event-based GUI programming in a functional style. This makes it easy to design interactive games such as the worm program.

Unlike commercial and other traditional programming environments, DRSCHEME is pedagogic[2] by design. Here is a list of the key attributes that make DRSCHEME well suited for an introduction of novices to applicative programming:

1. Novices make mistakes. Therefore the primary task of the environment is to help them recover from mistakes. The pedagogic problem is that students must know the entire language to understand messages concerning syntax and run-time errors. This is equally true for Algol and C style languages as for syntactically simplistic languages such as Lisp and Scheme.

   While many first-semester approaches to programming use, and even define, a subset of a conventional language, nobody enforces them systematically, except for DRSCHEME. More precisely, DRSCHEME supports a series of five Scheme-like languages, each more powerful than the previous one. Each represents a cognitive stage in the learning process of novice programmers. Because the implementations are tailored to these subsets, they can issue error messages that are formulated at the given level of language and understanding.

   For a detailed explanation of the problems with using full language implementations and of the teaching languages, see the early papers on DRSCHEME [5, 9].

2. Both syntax errors and run-time errors are pin-pointed graphically.

[2]Commercial software producers (and others) sell so-called pedagogic IDEs but these are often just IDEs for professional programmers without a few of the advanced features.

3. Those portions of a program that students' test suites do not evaluate are highlighted in red. Instructors are encouraged to label such red portions as "errors."

4. The Interactions Window is transparent. Every click on **Run** re-initializes it so that students don't get confused about the state of the REPL.

5. Lastly the environment also supports a one-click Help Desk facility. A single key press brings up the help page for the function or variable on which the cursor currently rests, and the documentation produced is tailored to the current language level.

The enforcement of language levels for pedagogic purposes rules out the support of library or module constructs. Otherwise, students could use the module system and imports to circumvent curricular constraints. Still, instructors or book authors often wish to supply code infrastructure that cannot be expressed in the teaching language. To assist such clients, DRSCHEME implements the teachpack mechanism. Roughly speaking a teachpack is a module that the environment links into the student's program before running it. It is usually implemented in the full-fledged, graphical version of PLT Scheme. The interface language even allows the designers of teachpacks to inject higher-order functions and macros. Thus, teachpacks empower students to develop applicative graphics and applicative graphical user interface programs without syntactic distractions from the pedagogical goals.

Note: The DRSCHEME environment comes with a design-oriented approach to teaching applicative programming to novices. This is known as the TeachScheme! curriculum. We omit an explanation of this pedagogy and the symbiotic relationship between the pedagogy and the environment [6, 7].

3. DRACULA

The core of DRACULA consists of a subset of Applicative Common Lisp in PLT Scheme and a programming environment within DRSCHEME. It resembles the existing Scheme subsets for teaching novices. The implementation takes advantage of the aforementioned DRSCHEME features: it highlights source locations when guard checks fail; it uses Check-Syntax to display scope information; it allows α-renaming; it makes the original ACL2 documentation searchable via Help

Desk; it shows code coverage for test suites; and it provides libraries to enable development of interactive graphical programs in Applicative Common Lisp.

DRACULA supports reasoning about the GUI by including an ACL2 book that formalizes the provided functions. The structure of the interactive programs takes the form of a state transition function. Thus, students learn to use ACL2 in a manner that is consistent with successful industrial applications.

DRACULA also includes a mechanism for reasoning about code within the ACL2 theorem prover. Students can send portions of their program to ACL2, and DRACULA highlights the program in green or red to indicate admission or rejection.

3.1 The Language

DRACULA implements a modest subset of the Applicative Common Lisp special forms; it includes all of the documented library procedures. The set of implemented forms corresponds roughly to the Intermediate Student Scheme dialect:

- Definitions: **defconst**, **defun**
- Conditionals / Boolean operations: **cond**, **if**, **case**, **case-match**, **and**, **or**
- Local binding: **let**, **let***, **mv-let**, **mv**
- Structured data: **defstructure**
- Shortcut for constant lists: **quote**

In addition, DRACULA supports **deflist** because it automatically proves many lemmas that students might otherwise get stuck on.

For pedagogical reasons, the language always performs guard checks, even though ACL2 functions are total from the perspective of the logic. When a primitive is misapplied during program execution, it raises a run-time exception, and DRACULA highlights the origin in the source program. In addition, DRACULA draws an arrow that shows the remaining call chain.

We have chosen to use the "guarded semantics" because the goal of our effort is to teach software design with formal methods and not logic and theorem proving *per se*. In this context, students test software expecting it to fail on occasion. Put differently, students need constructive feedback when testing reveals erroneous or undefined behavior. Rendering the program total by producing arbitrarily selected values is unhelpful.

DRACULA extends Applicative Common Lisp by providing a teachpack that enables students to design interactive graphical programs in the same fashion as is supported by the pedagogical Scheme dialects. This teachpack supplies a framework that expects the student to develop one datatype and three event handling callbacks:

1. the datatype is the set of values that represents the (visual) "world";
2. a world transition function that is called for each clock tick;
3. a world transition function that responds to keyboard events; and
4. a rendering function for redraw events that produces an image from a given representation of a world.

Because the representation and transformation of the world is central, we call our teachpack the *World teachpack*.

Figure 4 shows a small interactive DRACULA program based on the World teachpack. The application opens a small window—like the one in the upper-right corner of figure 4—and displays the initial world (just a 0). At each clock tick, *tick* produces a new world by adding 1 to the current world. The up and down arrow keys increment and decrement the world, respectively. When a keypress is detected, the teachpack computes a new world by applying *respond-to-key* to the current world and the key event. The screen image is produced by *render-world*, which consumes the world and produces the image to be displayed. The (**big-bang** ...) form initializes the teachpack by specifying the width and height of the canvas, how often the clock ticks, and the initial world. The last three forms specify the callbacks that the teachpack should use to evolve and draw the world.

```
(include-book "world" :dir :teachpacks)

;; World = Natural Number
(defconst *world0* 0)
(defconst *width* 100)
(defconst *height* 20)
(defconst *clock-period* 1/2) ;; half a second

;; respond-to-key : World Key → World
;; Produce a new world given the current one
;; and a key event
(defun respond-to-key (w key)
  (cond ((equal key 'UP) (1+ w))
        ((equal key 'DOWN) (nfix (1− w)))
        (t w)))

;; tick : World → World
;; Produce a new world on each clock tick
(defun tick (w) (1+ w))

;; nat→string : Natural → String
;; Convert a natural number to a string
(defun nat→string (n)
  (coerce (explode-nonnegative-integer n 10 nil)
          'string))

;; render-world : World → Image
;; Produce an image of the given world
(defun render-world (w)
  (text (nat→string w) 12 'blue))

;; Set up the framework and run the program:
(big-bang *width* *height* *clock-period* *world0*)
(on-tick-event tick)
(on-key-event respond-to-key)
(on-redraw render-world)
```

Figure 4: A complete counter program

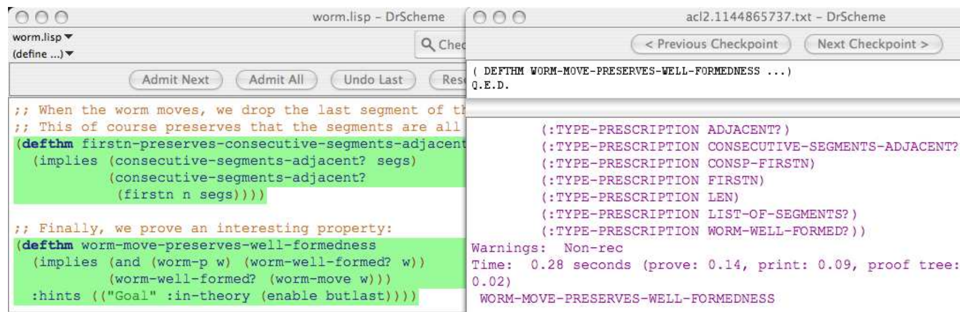

Figure 6: Interacting with ACL2 via DRACULA

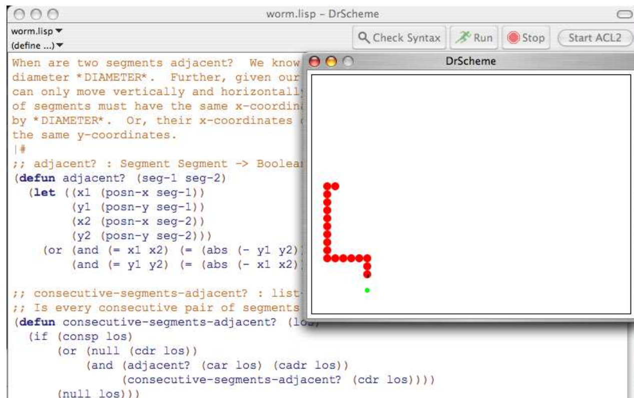

Figure 5: Worm in DRACULA

The Worm game has exactly the same structure, though the datatypes are more interesting than for our first example. For Worm, the World consists of a worm, a piece of food, and a bounding box for the play field. At each clock tick, the worm moves one step in the direction it is facing. If it is near food, it eats the food and grows by one segment at the end. The arrow keys change the worm's direction, and the game stops when the worm runs into itself or hits the wall of the bounding box. Figure 5 displays a screen shot from the DRACULA worm game with some of the actual code in the background.

The addition of graphical primitives necessitates a specification of both the callback functions and the datatype of images within ACL2. Our current specification is fairly weak: the theorems that we provide are essentially "types" for the operations. We can get quite far with this limited information, however, because most interesting theorems concerning interactive programs in our framework do not directly involve reasoning about images.

The World teachpack forces students to organize the program in the style of a state transition function. As the ACL2 community is well aware [2], this style facilitates the statement and proof of safety properties. For example, one may prove that the initial state satisfies some invariant and that the state transition functions preserve the invariant. For the example program in figure 4, it is easy to show that the world is always a natural number:

(**defthm** *natp-initial* (*natp* ∗*initial-world*∗) *:rule-classes* nil)
(**defthm** *tick-preserves-natp*
  (*implies* (*natp w*) (*natp* (*tick w*))))
(**defthm** *respond-to-key-preserves-natp*
  (*implies* (*natp w*) (*natp* (*respond-to-key w key*))))

This simple property is not very interesting, but the pattern applies even when the world is more complex. For the worm game, it is possible to state and prove the preservation of well-formedness for the worm, and fewer than ten lemmas are required before ACL2 can complete the proof. [3]

The World teachpack allows us to bridge the gap between toy exercises on algebraic datatypes and case studies because we can write programs whose complexity falls anywhere in between these two extremes. Programs can be as simple as that shown in figure 4 or as complex as the Worm game, and we do not demand any specialized domain knowledge. A student who is comfortable reasoning about programs based on the World teachpack is prepared to tackle case studies in the context of more advanced courses.

3.2 Interacting with the Theorem Prover

After students have developed and tested a piece of code, they can then interact with the theorem prover. For this purpose, DRACULA adds a START ACL2 button to the control panel. Clicking this button starts an ACL2 process and opens up a console window that shows ACL2's output in style similar to ACL2's Emacs mode[4] with proof trees enabled. In addition, a second row of buttons appears in the DRSCHEME control panel.

Figure 6 is a snapshot of an interaction with the theorem prover. The window on the left shows the program text, which is highlighted to indicate that it has been admitted into ACL2. Above the program text are the buttons that mediate between DRACULA and ACL2. Their functionality is described below. On the right, there is a second window, called the console window, that contains two panes and two more buttons. The top pane shows proof trees, and the lower pane shows the proof transcript produced by ACL2. The two buttons allow students to navigate the checkpoints identified by the proof tree when the theorem prover fails.

The console window of DRACULA is read-only, and so students cannot interact with the ACL2 REPL directly. In-

[3] Three of these lemmas simply prevent ACL2 from using accumulator-style definitions to reason about *reverse* and *firstn*, and another is a simple fact about *firstn*.

[4] See `acl2-sources/emacs/emacs-acl2.el`.

stead, they use a set of buttons in the DrScheme window just above the program text:

Admit Next Sends the next unprocessed expression in the definitions window to ACL2. If ACL2 rejects the expression for any reason, DrScheme highlights it red. If ACL2 accepts the expression, DrScheme highlights it green. Green expressions cannot be edited. The GUI state must always reflect the state of the ACL2 theorem prover.

Admit All Send the unprocessed expressions to ACL2 one at a time. If one expression causes an error, no further expressions are sent to ACL2.

Undo Last Removes the last event from ACL2 and removes the green highlighting from the corresponding text in the editor. This is equivalent to "`:u`" at the ACL2 REPL.

Reset ACL2 Put ACL2 into its initial state and unhighlights everything in the definitions window. This is equivalent to "`:ubt! 1`" at the ACL2 REPL.

Stop Prover Shuts down the ACL2 theorem prover, highlights the program text, and hides these five buttons. This is equivalent to "`(good-bye)`" at the ACL2 REPL.

Proof trees are enabled by default in the interface. When ACL2 fails to prove a conjecture, DrScheme displays the most recent proof tree and presents two buttons in the console window for navigating the proof checkpoints. The buttons refocus the proof text on the checkpointed goals.

This minimal interface may constrain an expert, but experience suggests that it suffices for novices in the context of a software engineering course.

4. TEACHING WITH DrACuLa

During the past two academic years, students in the second author's software engineering courses used Applicative Common Lisp as the implementation language for their software projects. For the first semester course, students work on six individual projects and two team projects:

Project 0. Introductory programming exercises: list reverse, set operations, Newton's Method, and towers of Hanoi. No theorem proving is required.

Project 1. Implement statistical functions over lists of numbers. Prove that the frequency of numbers in a list is invariant under permutation.

Project 2. Define three functions to generate Fibbonaci numbers: structurally recursive, accumulator-passing style, and Kepler's formula. Students prove the equivalence of the first two. There is an accompanying writing assignment that asks students to describe why they believe the approximation to $\sqrt{5}$ that they computed with Newton's Method from Project 1 is sufficiently accurate for Kepler's formula.

Project 3. Parse a text file and sort contents using an $O(n \log n)$ sorting algorithm. Prove that the sorting algorithm is correct.

Project 4. Parse a text file and produce two word frequency tables (the first sorted alphabetically, the second by word frequency). Write a function that computes run frequency. This function produces lists that sum to 1, and students are required to prove this fact.

Project 5. Compare the size of an Lisp program with the size of the program obtained by inlining some function applications. Students use a supplied implementation of AVL trees to represent programs. Students are required to formulate and prove two "interesting" properties of their choice.

Team Project 1. Emphasizes standard software development processes, building on Project 3 above. Students produce various design artifacts and participate in code reviews.

Team Project 2. Stock market analysis software. Students choose two properties of their software and handwrite proof outlines. They formally prove one of them in ACL2. Student submissions contained between 1000 and 2000 lines of code.

DrACuLa was released in time for the Spring 2006 course, and the primary course project took advantage of the graphical and audio libraries. The project called for the development of a variation on a Pachinko game in which two players compete to collect balls falling down through a maze of deflectors. A player attempts to direct balls to a designated collection bin by moving deflectors and adjusting their angles. Details of deflector layout, graphics, audio feedback, relative speeds of ball and player movements, and so on were left to the students, and there was a lot of variety in the games they delivered. Some focused on the physics of ball movement, and others focused on entertaining graphics or realistic arcade sounds.

Students were required to formulate and prove at least one theorem for each state transition function. They were required to follow a theme of their own selection in designing the theorems. The most common theorem style was of the form (*implies* (*well-formed-input X*) (*well-formed-output* (*f X*))). Student projects contained between 1800 and 6600 lines of code with an average of approximately 3600 lines of code (including documentation). Each project contained 62 theorems and lemmas, on average.

During the first semester of the sequence (Fall 2005), a few students used Emacs to interact with ACL2, but most relied on their favorite editors. In the second semester, the instructor introduced DrACuLa and then allowed students to evaluate and choose. [5]

4.1 Students' Perspective

About midway through the spring semester, the instructor sent a questionnaire consisting of six questions to his students, asking them to describe their experiences with ACL2 and DrACuLa. All 32 students responded in writing. They were asked what aspects of the DrScheme environment for ACL2 they preferred over plain ACL2 and vice versa. The

[5] The experience report that follows does not constitute a scientific comparison between DrACuLa and other modes of ACL2 interaction. Nevertheless, this feedback is an important source that will help drive the design of the next revision of DrACuLa.

questionnaire also gave the students an opportunity to describe improvements they would like to see in both products.

The students were impressed by the development environment provided by DrACuLa. During the first semester, all of the students had used ACL2 via its read-eval-print loop (REPL). Most of them did not use an Emacs-based environment. They chose instead to use their preferred editor and to paste definitions and theorems from their editor windows into an ACL2 REPL. In responses to the questionnaire, the DrScheme editor got high marks as an integrated development environment, compared to the mode in which students were accustomed to using ACL2.

Students were particularly pleased with two features of DrACuLa. In particular, students liked DrScheme's runtime error reporting because it graphically links the error to the source expressions in their code. Additionally, they strongly preferred interacting with the theorem prover via DrACuLa's graphical interface over a textual REPL-based interface.

Finally, students appreciated DrScheme's support for designing graphical programs in DrACuLa. One student went so far as to criticize as "primitive" and "low-level" those environments where programs cannot communicate with the outside world except through file I/O and character-based displays. Students apparently value coursework that allows and encourages them to produce software with a modern look and feel while learning and applying theorem proving techniques.

Graphical output versus file-based I/O may seem to be a trivial distinction when computational logic is the topic of study, but such capabilities of a development environment enable instructors to plan extended projects that interest and engage the students. Students are simply more familiar with GUI-oriented software than with text-based software. They enjoy developing software that supports this type of interaction, and they expect to be able to do so in a software engineering course, where design and project management, in addition to computational logic, comprise the bulk of the material. So, the ability to include graphical output in their software is a big advantage for software development projects, even when using a computational logic to verify properties of that software is an important component of the development process. Freeing the students from programming text-based software removes a source of distraction from the material.

About half of the students learned how to glean useful information from ACL2's proof trees, and they found the checkpoint navigation buttons of DrACuLa useful. This is not surprising because their primary tool for navigating proof trees before DrACuLa became available was to dump the proof to a file and view the file through an ordinary text editor. None of the students learned how to display and navigate proof trees via Emacs. DrACuLa lowers the barrier to entry here because proof trees are enabled by default, and the buttons obviate the need to look up and memorize keystrokes.

In commenting on the advantages of using the standard ACL2 environment, students cited a weakness in DrACuLa. Because it is an early prototype, DrACuLa evaluates Applicative Common Lisp programs that contain Scheme fragments, or it occasionally uses Scheme's semantics instead of Lisp's. For details, see section 5. Naturally, ACL2 strictly enforces the syntax and semantics of Lisp. While some students were initially happy to escape through these loopholes, they later found that using non-ACL2 features wasted time, because they eventually had to bring their software back in complicance with ACL2's syntax in order to prove theorems about their programs.

Two students found the convenience and speed of their accustomed editors, along with the copy-and-paste model of editing in a familiar environment, more attractive than DrACuLa. These students decided to stay mostly within that model, even when running their software in DrACuLa. More generally, because the copy-and-paste usage model and DrACuLa's mode of interacting with ACL2 match the needs of students in the courses well, students don't have sufficient motivation to learn to use the proof navigation facilities provided by the standard ACL2 environment.

Students commonly cited another advantage of the standard ACL2 environment: **include-book**. This facility is only partially available in DrACuLa. Students can include books prepared as teachpacks by instructors, but they cannot implement software as a collection of files specifying separate modules, and then use **include-book** to integrate the modules. Furthermore, they cannot use the **certify-book** facility, since it has not yet been incorporated either. The ability to organize software into separate modules, certify those modules, and make use of them in other components are important techniques to teach students proper software engineering techniques. These facilities, along with the DrScheme Stepper, were the most often requested improvements to DrACuLa.

By far the most often requested improvement to the standard ACL2 environment was an integrated development environment. Most of the students are clearly thinking along the lines of the DrScheme environment, but two students said they would like to see an Eclipse-based environment for ACL2. Dillinger, Manolios, and Vroon have developed such an environment, known as ACL2s [3]. So, future students in software engineering courses at OU will have the ACL2s option, as well as DrACuLa. They will need the latter for projects that involve graphical output, but they can, if they choose, stay within ACL2s for textual projects. We expect to report on such a head-to-head comparison in the future.

4.2 Instructor's Perspective

The software engineering courses include the use of ACL2 primarily as a way to expose students to powerful means for reducing the number of defects in software. In other words, the course intends to introduce students to the idea that the use of theorem provers can contribute positively to the software development enterprise, as one of many ways to improve software quality. This theorem proving portion comprises about one third of the course materials. In this time and at the given level of effort, students cannot be expected to become proficient users of ACL2. Because of these limits on scope, a smaller ACL2 environment that helps us focus on these goals is more beneficial for the software engineering courses than a comprehensive and fully-featured one.

For this goal, DrACuLa plays its role well. The experiences of students in the course puts them in a good position to become adept in the use of theorem provers (ACL2 or other products) at some point in the future, should the need arise. Thus, students clearly gain knowledge and skills that will make it possible for them to assess potential benefits of computational logic in software engineering projects.

In general, the introduction of theorem provers into software engineering courses relies on a number of attributes of theorem proving systems and may neglect others. Most importantly, a course like ours demands the full strength of ACL2 as an automated theorem prover. Students cannot become experts at automated theorem proving in one or two semesters, especially in a course like software engineering where there are many other topics to be addressed. A powerful theorem prover, one that can verify many properties without the need for hints or advanced methods, and without even the need for lemmas in many cases, allows students to experience some success without becoming theorem-proving experts. Without this success, it seems likely that students would form a negative assessment of the potential of theorem provers as a support technology for software engineering. Fortunately, the automated theorem-proving effectiveness of ACL2 is sufficient to give most students a positive experience.

Last but not least, the ability to factor software into modules is imperative for a software engineering course. Modules provide namespace management and can be reasoned about independently. Adherence to programming patterns involving books and packages can help mitigate the lack of modules [15], but DRACULA needs proper language mechanisms to adequately support software engineering courses.

5. IMPLEMENTATION

DRACULA is implemented in PLT Scheme; it is embedded into DRSCHEME as an environment tool [8]. This means that DRACULA automatically benefits from several DRSCHEME capabilities.

The translator from Applicative Common Lisp (Lisp, for short) into Scheme consists of a collection of PLT Scheme macros [11]. PLT Scheme's behavioral contract system [10] checks guards on Lisp primitives. Procedures are implemented using PLT Scheme macros. The expansion process checks that Lisp functions are used only in operator position, and DRACULA generates a helpful error message when a violation is detected. The procedures into which these macros expand manipulate Scheme representations of Lisp values. They perform conversions back and forth as necessary, but, for most values, the conversion is the identity function. The representation of booleans is complex because Lisp conflates the empty list and "false", whereas Scheme separates them.

In contrast to Lisp's macro system, PLT Scheme's macro system is quite sophisticated. These macros don't just manipulate raw S-expressions, but syntax objects that carry additional information. The macro expander automatically[6] propagates this information. Most importantly, this information includes source locations and lexical scope. Furthermore, names that are introduced into the expanded code take their bindings from the macro definition context rather than from the context of the expanded code. In short, macros act like an abstract API for the compiler. Due to these differences, DRACULA does not support Lisp macros.

For example, consider the definition of Lisp's **if** in figure 5, a distillation of the actual code. This form defines a macro named **acl2-if** and a helper function *null?*, which checks if its argument is the empty list. Here, the use of *null?* in the macro body refers to the definition that precedes it. Whenever the expression (**acl2-if** *X Y Z*) occurs in a program, the program behaves as though (**if** (*not* (*null? X*)) *Y Z*) had been written instead. In addition, it ensures that the meaning of *null?* is the procedure defined in the same (module) scope as the macro itself. In Lisp macro systems, the meanings would be taken from the context of the use of the macro. Thus, the expression

```
(let ([null? (lambda (x) (equal? x 3))])
  (acl2-if '() 1 2))
```

would not evaluate to 2 as the macro writer had intended.

With the powerful macro system, it becomes straightforward to adapt the available tools of DRSCHEME to a new language embedding. For example, the CheckSyntax tool extracts lexical information and source locations from programs. The lexical information—the point where the variable is introduced—allows it to find the source of binding arrows; the source location of identifiers determines the sink. Thus, DRSCHEME can draw lexical scope information in the form of binding arrows for Lisp programs *without any additional help from the* DRACULA *implementor*. See figure 8 for an example. This also applies to refactoring tools, such as α-renaming, that require nothing but static information. For dynamic tools, such as the stepper, additional work is required.[7]

PLT Scheme has an implementation of software contracts that allows developers to express constraints on the uses of values. In the case of a first-order language such as Applicative Common Lisp, these specialize to pre- and postconditions on functions. To implement Lisp guard checks, we simply translate guards into contracts that specify preconditions. When a contract is violated, the contract checking system automatically assigns blame to the guilty party and annotates the code to show the programmer where the error originated. Because macros preserve source information, these errors are of course reported in terms of the source language (Lisp in our case) and not the code into which the macros expand.

The first alpha version took about five man-weeks worth of effort to develop, test, debug, optimize, and deploy. We invested another two weeks to build detailed examples and

```
;; null? : any -> boolean
;; produces true iff the argument is Scheme's empty list
(define (null? x)
  (eq? x '()))

;; (acl2-if <acl2-exp> <acl2-exp> <acl2-exp>)
;; expands to a Scheme if plus value conversions
(define-syntax acl2-if
  (syntax-rules ()
    [(_ test consequent alternative)
     (if (not (null? test))
         consequent
         alternative)]))
```

Figure 7: Implementation of Applicative Common Lisp's if

[6] The macro system includes constructs for forcing information propagation where the flow is not obvious. We had no use for these constructs thus far.

[7] We have not yet completed the adaptation of the stepper to DRACULA.

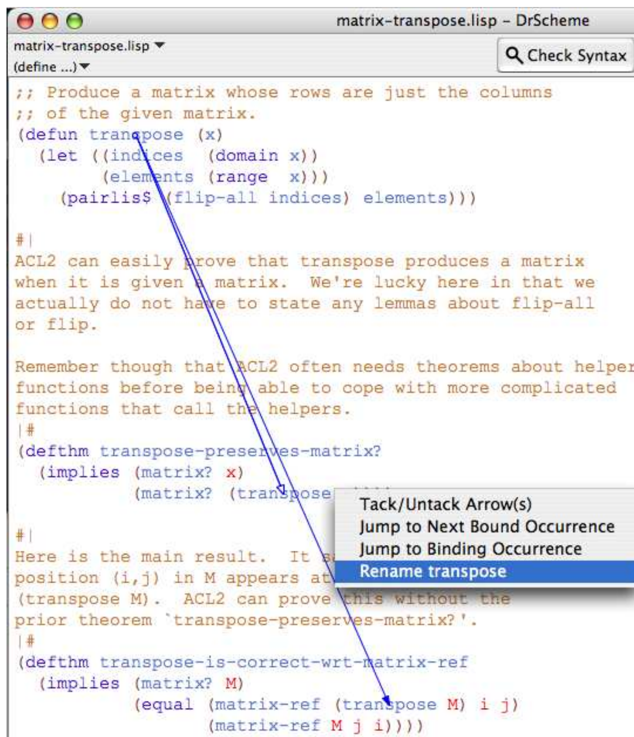

Figure 8: Check Syntax in DRACULA

documentation. Making the ACL2 documentation searchable via Help Desk necessitated writing a small script to generate a keyword index from the existing documentation.

The current version of DRACULA still has a few limitations. In particular, it does not support all of Applicative Common Lisp. Programmers cannot use **include-book** to import functions into DRACULA, but they may use it to import theorems into ACL2 as usual. The exception is for teachpacks provided with DRACULA; code is imported from them via **include-book**. Due to the semantic differences between Common Lisp (the source language) and Scheme (the target language), DRACULA currently uses a single environment for function bindings and local variable bindings. Finally, DRACULA implements the "guarded semantics" of ACL2 only, and so the primitive procedures raise errors when applied to the arguments outside of their intended domain.

6. RELATED WORK

To our knowledge, nobody else has attempted to use ACL2 to teach undergraduate software engineering courses, but the ACL2 developers routinely use ACL2 in many courses such as hardware verification, processor design, logic, and formal methods. The tool support that we have constructed for this course is informed by the PLT group's techniques for teaching program design [5, 6].

Our interface to ACL2 is modeled on both the ACL2 Emacs mode[8] and other theorem prover IDEs such as ProofGeneral and CoqIDE [1, 19]. The latter two environments feature highlighting of code upon admission or rejection and a graphical interface for issuing commands to the underlying theorem prover. Because our needs are focused on teaching undergraduates, our interface provides a strict subset of the features available in other environments.

DRACULA's closest competitor is Dillinger, Manolios, and Vroon's ACL2s [3]. ACL2s allows developers to design ACL2 models in Eclipse [4]. Rather than reimplement a portion of the language, they connect the user to a full version of ACL2 for both running and reasoning about code. ACL2s offers several modes that control the database of rules available to the theorem prover and whether or not to postpone termination proofs. Functions that have not yet been proved total may be evaluated at the REPL for testing purposes, and this is similar to the behavior of DRACULA. They do not attempt to provide a novice-oriented fragment of the language, and they do not attempt to provide a framework like our World teachpack for building graphical, interactive programs.

In contrast, we do not provide a comprehensive environment for expert ACL2 developers. Our environment is best viewed as a pedagogical stepping stone to more advanced environments. It is not our goal to produce a fully-featured ACL2 implementation, but rather we aim to provide software support for bridging the gap between toy examples and complex case studies for teaching software engineering with a formal methods component. This requires more than just small to medium sized examples because students must have proper environment support while learning how to design ACL2 models.

7. CONCLUSION

This paper introduces DRACULA, a programming environment for ACL2 in DRSCHEME. The environment provides a pedagagic framework for students, in which they can learn to design programs in an applicative manner and prove theorems about them. Due to its connection to DRSCHEME, DRACULA supports some GUI-oriented programming, which helps motivate students.

Despite the limitations mentioned in section 5 and despite the fact that DRACULA is alpha software (developed over a five-week period before the spring semester), students at Oklahoma University obviously enjoyed working with the environment, especially in comparison to plain ACL2. We therefore believe that DRACULA is on the right track.

Following the TeachScheme! precedent, we intend to use the feedback from instructors and from monitoring student usage to improve the system. One near-term goal is to provide a module-like facility, like those found in modern applicative languages [11, 14, 17], so that students can develop software in a modular manner and learn to prove theorems whose antecedents abstract over the deployment context. We conjecture that PLT Scheme modules with behavioral software contracts [10] are particularly suitable for this purpose.

Distribution: The software and a tutorial are available at

`http://www.ccs.neu.edu/~dalev/acl2/`

Page's software engineering course materials are available at

`http://www.cs.ou.edu/~rlpage/SEcollab/`

Acknowledgments: We thank Matthew Flatt for his suggestions on making DRACULA reasonably fast and his helpful modifications of PLT Scheme's JIT compiler for the same purpose. Also, we acknowledge John Clements's efforts with the

[8]See `acl2-sources/emacs/emacs-acl2.el`

adaptation of the DrScheme stepper to DrACuLa. Finally, we wish to express our gratitude to the anonymous referees for their helpful feedback.

8. REFERENCES

[1] Aspinall, D. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *LNCS*, 2000. http://proofgeneral.inf.ed.ac.uk/.

[2] 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.

[3] Dillinger, P., P. Manolios and D. Vroon. ACL2s: The ACL2 Sedan. http://naxos.cc.gt.atl.ga.us/acl2s/, 2006.

[4] Eclipse Consortium. Eclipse, 2000. http://www.eclipse.org.

[5] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. The DrScheme project: An overview. *ACM SIGPLAN Notices*, June 1998. Invited paper.

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

[7] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14:55–77, 2004.

[8] Findler, R. B. PLT DrScheme: Programming environment manual. Technical Report PLT-TR05-3-v300, PLT Scheme Inc., 2005. http://www.plt-scheme.org/techreports/.

[9] 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.

[10] Findler, R. B. and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[11] Flatt, M. Composable and compilable macros: You want it *when?* In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[12] Flatt, M. PLT MzScheme: Language manual. Technical Report PLT-TR05-1-v300, PLT Scheme Inc., 2005. http://www.plt-scheme.org/techreports/jfp01-fcffksf.

[13] Flatt, M. and R. B. Findler. PLT MrEd: Graphical toolbox manual. Technical Report PLT-TR05-2-v300, PLT Scheme Inc., 2005. http://www.plt-scheme.org/techreports/.

[14] Harper, R. and B. C. Pierce. Design issues in advanced module systems. In Pierce, B. C., editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2004.

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

[16] Page, R. L. Engineering software correctness. In Findler, R. B., M. Hanus and S. Thompson, editors, *Functional and Declarative Programming in Education*, 2005.

[17] Peyton-Jones, S., editor. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, Cambridge, UK, April 2003.

[18] Plotkin, G. D. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[19] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. LogiCal Project, INRIA, 8.0 edition. http://coq.inria.fr/doc/main.html.