Proof Pad: A New Development Environment for ACL2

Caleb Eggensperger
School of Computer Science
University of Oklahoma
Norman, Oklahoma
calebegg@gmail.com

Most software development projects rely on Integrated Development Environments (IDEs) based on the desktop paradigm, with an interactive, mouse-driven user interface. The standard installation of ACL2, on the other hand, is designed to work closely with Emacs. ACL2 experts, on the whole, like this mode of operation, but students and other new programmers who have learned to program with desktop IDEs often react negatively to the process of adapting to an unfamiliar form of interaction.

This paper discusses Proof Pad, a new IDE for ACL2. Proof Pad is not the only attempt to provide ACL2 IDEs catering to students and beginning programmers. The ACL2 Sedan and DrACuLa systems arose from similar motivations. Proof Pad builds on the work of those systems, while also taking into account the unique workflow of the ACL2 theorem proving system.

The design of Proof Pad incorporated user feedback from the outset, and that process continued through all stages of development. Feedback took the form of direct observation of users interacting with the IDE as well as questionnaires completed by users of Proof Pad and other ACL2 IDEs. The result is a streamlined interface and fast, responsive system that supports using ACL2 as a programming language and a theorem proving system. Proof Pad also provides a property-based testing environment with random data generation and automated interpretation of properties as ACL2 theorem definitions.

1 Introduction and Prior Work

1.1 ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) is a lisp dialect and theorem prover in the Boyer-Moore family of theorem provers. Functions and theorems in ACL2 are stated using an applicative subset of Common Lisp, where variables are read-only, and functions must be written using a recursive rather than an iterative style. Because of this, user-stated theorems can be more easily proven by the automated theorem prover from the collection of theorems and lemmas available in the system (both built-in and user-defined). Using this mechanism, it's possible to "steer" ACL2 to prove complex theorems with minimal interaction [8]. ACL2 has predominantly been used to model and verify the hardware and sometimes software of critical computer systems.

Recently, Carl Eastlund and Rex Page have used ACL2 as a pedagogic software development tool [6][9][10][11]. Page has been using ACL2 and DrACuLa (an ACL2 IDE) in the University of Oklahoma's software engineering course since 2003. ACL2 was brought into the course to aid the students with good design and defect control in their projects. The approach has been successful, with students able to create projects of significant complexity with a solid proof-based underpinning [9].

For this two-semester course, students were asked to approach a series of small individual and team projects, leading up to a final, semester-long team project. Projects begin with the creation of short functions and the formalization of simple theorems, in order to develop students' ability to state and

R. Gamboa and J. Davis (Eds.): ACL2 Workshop 2013 (ACL2 '13).
EPTCS 114, 2013, pp. 13–28, doi:10.4204/EPTCS.114.2

prove properties of their code. The course culminates with a project of 2,000 to 3,000 lines, along with ACL2 properties for individual components, documentation, unit tests, and integration tests [9].

1.2 Existing ACL2 development environments

The development of Proof Pad was informed by existing tools for working with ACL2. The most common of these is Emacs, which is suggested by the documentation. Additionally, two attempts have been made at more user friendly interfaces, both targeted at being easier to use for undergraduates: DrACuLa and ACL2s.

I have had some prior experience working with ACL2 user interfaces in the form of Try ACL2 (at `http://tryacl2.org`), an experimental website that I built that gives a restricted, web-accessible REPL for ACL2 along with a simple tutorial to help the user learn some of the basics of ACL2. Through this project, I got some experience working with ACL2 programmatically along with some feedback on this method of interaction from the programming community. Try ACL2 is more of an introductory tool rather than a fully featured development environment. Proof Pad builds on this work towards the goal of a more comprehensive environment, bringing it in line with the tools mentioned above.

1.2.1 DrACuLa

DrACuLa [12] is a plugin for DrRacket (formerly DrScheme) that acts as an IDE for ACL2. It is primarily intended for classroom use. Because DrACuLa is a DrRacket plugin, it has an excellent text editor for Lisp syntax. Its parentheses matching and auto-indentation are indispensable. DrRacket also has a history of pedagogic usage and, as such, is designed to be easy for students and inexperienced programmers to pick up and use [7]. DrACuLa also extends ACL2 with support for a modular style of programming inspired by the Scheme dialect that DrRacket uses. In this style, interfaces to modules are separated from the implementation of the modules. The modules also have contracts that implementations must uphold, which are mechanically verified by ACL2 [4].

In addition to supporting a sizable subset of ACL2's syntax and features, DrACuLa extends ACL2 with a set of "teachpacks" — short, easy-to-use libraries with specific functionality. One notable example is DoubleCheck, a QuickCheck-inspired library for running automated, randomized tests of ACL2 code. DoubleCheck provides both an accessible jumping-off point for students to start thinking in terms of verifiable properties before going straight to theorems, as well as a useful method of generating counterexamples for failed theorems. The alternative, reading ACL2's output for a failed proof attempt, is intimidating for many students [5].

However, a few aspects of DrACuLa pose problems for effective classroom use. Installation of DrACuLa is a complex process that involves acquisition and installation of three separate software components, interaction with the command line, and some potential pitfalls that are difficult to recover from, especially for beginning programmers. In order for this tool to be usable in courses at the University of Oklahoma, I have compiled and maintained a lengthy document for the installation process and investigated many potential ways to streamline the process, all to no avail. Installing the software tools is a frequent frustration in these classes.

Because DrACuLa executes ACL2 definitions in Scheme, only some of ACL2's functionality is available. For example, arrays and macros are not supported, and supporting them would likely involve a lot of work. This dual implementation can also lead to bugs where certain functions can be used in DrACuLa, but are rejected or incorrect in ACL2 (and vice versa). Additionally, this dual-implementation

can lead to error messages that are hard to follow or track down in many cases, especially in the included teachpack libraries where there are separate ACL2 and Racket implementations of each function.

1.2.2 The ACL2 Sedan

The ACL2 Sedan, or ACL2s, is an Eclipse plugin and set of additional features for ACL2 that seek to make ACL2 easier to use. Using the metaphor of a sedan, ACL2s intends to provide a simple, low-maintenance interface for ACL2 at the cost of high-level performance and customization. ACL2s includes some extensions of ACL2's functionality that add various useful features, including a method by which it can automatically attempt to generate counter-examples directly from theorems [3].

ACL2s has an automatic counterexample generation tool that works by looking directly at theorems to determine the types of data to try to bind to free variables in the theorem body. This means that ACL2s doesn't require a different syntax for specifying tests and data generators. This approach integrates more directly with the existing proof process, as opposed to the DoubleCheck approach of integrating proofs with an existing testing method [2].

However, ACL2s also has some drawbacks. The entire UI for ACL2s is placed in a single toolbar consisting of nine buttons and a single top-level menu. This is fine for Eclipse plugins that also re-use Eclipse's menu items and standard functionality, but, for the most part, ACL2s does not. When looking at a file, ACL2's status is shown only through colors, and no visual feedback is given for errors.

ACL2s requires that Eclipse be installed in a non-standard path on Windows systems. The initial placement of the REPL (which is treated as a type of file) is in a separate tab, which the user must flip back and forth between to use, instead of the docked window arrangement shown on the website. The default mode that ACL2s users are put into is called "Bare Bones" mode, where common macros such as addition, subtraction, and equality testing are undefined, with no clear indication of how to change that.

Modern development environments frequently provide complex, language specific syntax highlighting, that takes into account different categories and types of keywords and built-in functionality, data types, and comments. ACL2s, however, does not; syntax highlighting simply consists of one color for parentheses, one for code, and one for comments. This makes typos difficult to spot and built-in functionality difficult to recall (e.g. is it `equal` or `equals`?).

1.2.3 Emacs

Professional users of ACL2 commonly use Emacs along with a plugin specifically designed for ACL2. Emacs relies heavily on memorized keyboard commands and a minimal, code-focused UI. This usage style is not practical for a classroom environment where the students have learned programming on point-and-click IDEs, and do not want to learn the numerous key commands necessary to use Emacs proficiently. However, the emacs interface has several attributes that are conducive to ACL2's unique workflow. I have tried to incorporate some of those elements into Proof Pad.

2 Design

The design of Proof Pad was a carefully considered process. I sought to fix some of the usability problems with previous attempts without introducing new, worse problems in the process. I was also interested in producing a polished, good-looking application that users would feel comfortable and at home with. Careful design, fast prototyping and iterating of design ideas, and user feedback were all core to this process.

2.1 Goals and Constraints

When designing Proof Pad, I started by documenting several goals and constraints intended to make Proof Pad an effective, modern development environment. These goals are based heavily on the results of a questionnaire completed by students in the software engineering course at the University of Oklahoma. The questionnaire posed three free-form questions, soliciting up to three answers to each. The results of the questionnaire (see Table 1) were reviewed by Rex Page, Allen Smith, and myself as part of ongoing work in a project investigating pedagogic use of ACL2. As part of this review, we collected together responses we saw as duplicates, so that they could be prioritized. The results revealed several areas of the existing platform, DrACuLa, that seemed that they could be improved upon by taking a new approach. Many of these elements could not be easily addressed in the DrRacket framework.

2.1.1 User Interface

Inasmuch as is possible, Proof Pad should have a feature set inspired by industry standard IDEs such as Eclipse and Visual Studio. Professional and pedagogic IDEs frequently include syntax highlighting, automatic indentation, and bracket matching features to facilitate entering programs that are free of syntax errors. These features are a must, especially to students who are used to them in other environments.

Proof Pad should also encourage good use of ACL2 as a language. The Method is an approach to proving theorems in ACL2 that was constructed and is advocated by the developers of ACL2 [8]. A document set up using The Method consists of some proven theorems or lemmas, an invisible line separating the proven/unproven portions, and then the unproven lemmas or theorems. Progressing involves breaking down the first unproven theorem into lemmas to help steer ACL2 forward. Proof Pad can encourage users to follow this method of generating proofs in at least two ways. First of all, it can display the proof line prominently to help users keep track of where the proof process is. Another aspect of The Method is the advice to read the proof transcript from the top to see where ACL2 went wrong, instead of blindly trying to prove the last part of the proof process that failed. For advanced users, this might involve a lot of scrolling through irrelevant output, but for novice users attempting more simple proofs, the ability to read and interpret a proof narrative is important, so Proof Pad initially scrolls the proof output to the top.

2.1.2 ACL2 Integration

ACL2 can be frustrating to work with programmatically, but it is powerful and fast. Despite some benefits of a secondary interpreter such as the one DrACuLa uses (the possibility of step debugging and better memory management, for instance), it seems to me that the disadvantages outweigh the advantages, and that integrating directly with ACL2 is the better route.

The installation process is a user's first exposure to an application. As part of being tightly coupled with ACL2, and in order to give a good first impression, Proof Pad has a simple installation process that's comparable to the installation of other applications, and, more specifically, does not take the form of a separately installed plugin to another piece of software (such as Eclipse or DrRacket), and does not require the user to procure resources, such as the platform-specific ACL2 binary, from multiple sources.

2.1.3 Performance

Having a fast start-up time is an important part of the user's perception of the speed and performance of the program overall, and tends to leave a more lasting impression than other performance measures. In the development of Proof Pad, I paid particular attention to this metric.

Describe three services an IDE should provide:

Count	Service
10	Syntax error messages pinpoint type of error and location in source code
7	Runtime error messages pinpoint type of error and location in source code
6	Automatic indentation syntax highlighting, etc.
5	Syntax-error detection while typing
5	Runtime program stepper

Describe three things you like about DrACuLa:

Count	Aspect
9	Parentheses matcher during editing
8	DoubleCheck property based testing
6	ACL2's mechanized logic
5	Read-Eval-Print loop
5	`check-expect` tests

Describe three things you don't like about DrACuLa:

Count	Aspect
10	No runtime debugger or program stepper
8	Runtime error messages give little info about where or what the error is (stack trace would help)
7	Syntax error messages fail to pinpoint error in source code
5	Slow start-up
4	Multi-tab editor hard to use
4	Poor documentation of intrinsic functions
4	Lack of auto-completion and display of function parameters while typing

Table 1: Summary of results from the DrACuLa questionnaire administered in the University of Oklahoma's software engineering course in Fall of 2011. Only the top 5 most commonly mentioned items are shown (more if there's a tie)

2.1.4 Cross platform

Having an application that both runs on the user's operating system of choice and that also fits with their expectations for other applications on that platform is important to giving the user a good impression of the software and a solid framework for using it. Since ACL2 is already available on Windows, Mac, and Linux, Proof Pad should (and does) provide comparable cross-platform support. Additionally, even at the University of Oklahoma, which requires Windows for certain classes, a survey of Applied Logic (one of the target courses for Proof Pad) students' homework submissions determined that 36% of students use Macs, and 59% use Windows (the remaining 5% used Windows in a computer lab).

As much as possible, the tool should stay consistent with the platform it is running on, both graphically and interactively. As such, I worked towards compliance with the Windows User Experience Interaction Guidelines[1], the Mac OS X Human Interface Guidelines[2], and the GNOME Human Interface Guidelines[3].

2.1.5 User-focused process

I sought feedback from students and other ACL2 users at all stages of development, in both formal and informal contexts. User testing is an essential part of creating a high quality, usable application. I want Proof Pad to be as free of usability problems as I can make it, to allow users to focus on their code, not the interface. I used a combination of one-on-one testing with individual students completing tasks that I designed and more broad-scale testing by using the tool in a classroom setting in two semester-long courses, from which I collected more aggregate and quantitative data using a questionnaire.

2.2 User Interface Components

The working area of Proof Pad is logically split into a few different areas. A mockup of the main window is shown in subsection 2.2. The large area on top where most of the code editing occurs is called the definitions area, since the primary use for this part of the workflow is to define functions and theorems. To the left of the definitions area is the proof bar, which displays and allows manipulation of the state of ACL2 with respect to the definitions. Below these items is the read-eval-print loop, an interactive console where functions can be invoked to test or demonstrate their functionality.

2.2.1 Definitions Area

The definitions area is where files are edited. Functions, properties, and theorems are defined in this area. The space is visually divided into areas that denote the state of ACL2: the Proof Bar (see subsubsection 2.2.2), the actual syntax-highlighted text area, and a bar to the right that allows the user to view the output of ACL2 for specific items.

The two bars to the left and right visually segment the definitions area based on the top-level events or function calls in the definitions area. This segmentation is further emphasized if part of the definitions have been admitted to ACL2, which renders them with a grey background to indicate that they are not editable and with a dark line below them to separate admitted and unadmitted items.

---

[1] `http://msdn.microsoft.com/en-us/library/windows/desktop/aa511258.aspx`

[2] `http://developer.apple.com/library/mac/#documentation/UserExperience/Conceptual/AppleHIGuidelines`

[3] `http://developer.gnome.org/hig-book/3.0/`

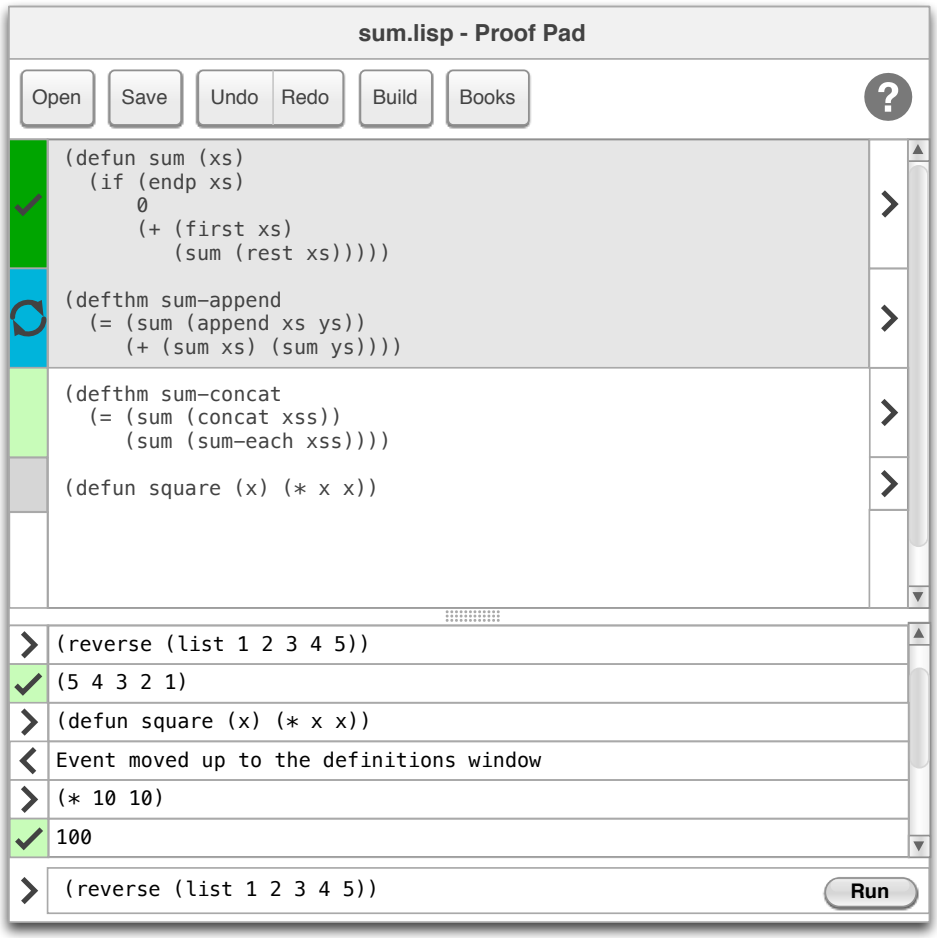

Figure 1: A mockup of the design of the main Proof Pad window. A screenshot of the most recent version is shown in Figure 5.

The syntax highlighting has been carefully tailored to ACL2. For example, ACL2 makes a fundamental distinction between regular functions (such as `equals`, `cons`, and `append`), and events, which modify the ACL2 state (such as `defun` and `defthm`). Proof Pad emphasises this difference by displaying the two types in different colors.

2.2.2 Proof Bar

Proof Pad introduces a user interface element for displaying and maintaining the status of the current document with relation to ACL2. Typical ACL2 workflow involves typing in a definition or theorem, attempting to admit it to ACL2, and either responding to errors or continuing with the next definition. Additionally, users occasionally need to return to a previously admitted expression and modify it to accommodate a case or error that they had not previously seen. Figure 2 shows some of the interactions the proof bar supports.

In order to accommodate these use cases, the proof bar responds to clicks in one of two ways:

1. If the click is to the left of an unadmitted expression, the proof bar queues for admission all of

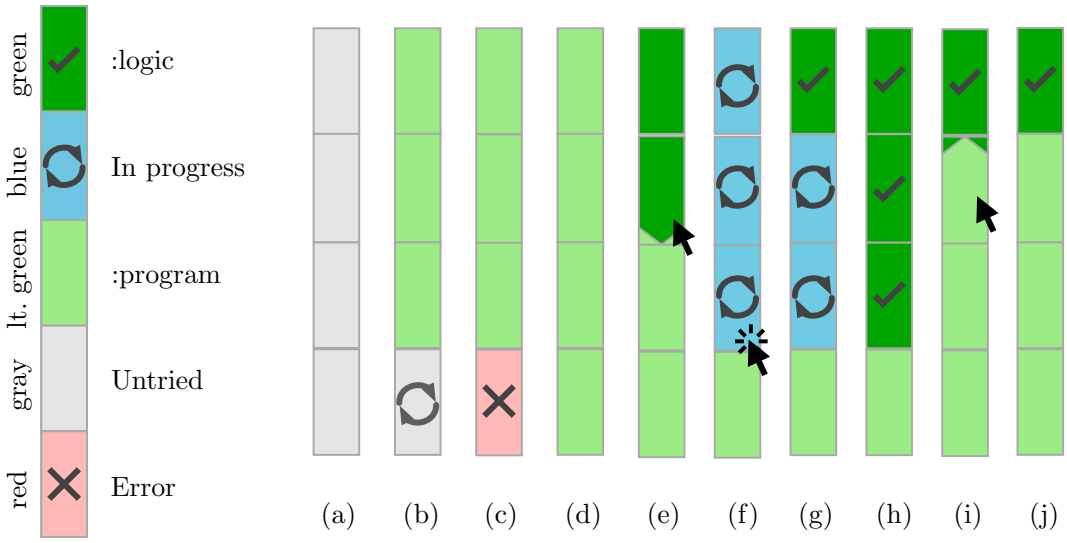

Figure 2: Diagram showing how the proof bar responds to user interaction in some routine scenarios as a series of state transitions. (a) Initial state. (b) Three expressions automatically admitted. (c) Fourth expression has an error. (d) The error has been corrected. (e) User hovers over the bar. (f) User clicks the bar. Admission begins. (g) One expressions has been admitted successfully to the logic. (h) All three requested expressions admitted successfully. (i) User hovers over second entry. (j) User has unadmitted two expressions.

the unadmitted top-level expressions down through (and including) the selected expression for admission. As ACL2 admits each one, the status changes from in progress to proven or failed.

2. If the click is to the left of an admitted expression, the proof bar undoes all of the admissions that changed the global state up through and including the selected expression. Undoing is done through the undo feature of ACL2, which has a fast response time.

Hovering over the proof bar previews the action that will be taken if the user clicks.

At all times, the proof bar indicates the current status of each top-level expression in the definitions area. There are five different statuses that are represented using different colors and symbols.

2.2.3 Read-eval-print loop

The read-eval-print loop (REPL) is a common feature of Lisp-like languages. Formulas are typed at a prompt and executed in the context of the current set of definitions. A REPL is a popular and effective part of other pedagogic IDEs, such as DrRacket [7] and DrJava [1].

One issue with the traditional REPL that the developers of DrRacket discovered is that novice users often have trouble keeping track of stateful changes to the environment made using the REPL. For instance, a student might fix a bug by redefining a function in the REPL, but then forget to incorporate that change back into the definitions area. This can lead to bugs that were thought to be fixed returning later [7]. Proof Pad addresses this problem by keeping definitions consistent with updates to the REPL. All functions in ACL2 are divided into two groups: events, such as `defun` or `defthm`, which modify the global state; and functions, such as `max` or `first`, which do not modify the state of the system, but just

return their result. Proof Pad automatically moves event formulas to the definitions area without passing them through ACL2, sidestepping the problem of state changes in the REPL.

Proof Pad's REPL has some unique enhancements over typical REPLs. For one, Proof Pad's REPL, like other parts of the UI, summarizes ACL2 responses to events, while still allowing the user to click a disclosure arrow in the UI to view the full ACL2 output.

2.2.4 Results window

```
ACL2 Warning [Compiled file] in ( INCLUDE-BOOK "book" ...):  Unable to load
compiled file for book
  <book path>
because that book is not certified.  See :DOC include-book.  No load was in
progress for any parent book.

ACL2 Error in ( INCLUDE-BOOK "book" ...):  There is no file named
"/Users/calebegg/Code/book.lisp" that can be opened for input.

Summary
Form:  ( INCLUDE-BOOK "book" ...)
Rules: NIL
Warnings:  Compiled file
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)

ACL2 Error in ( INCLUDE-BOOK "book" ...):  See :DOC failure.

******** FAILED ********
```

Figure 3: Full text of the error message shown when an included book can't be found. Collected from ACL2 version 4.3.

When Proof Pad encounters an error, or when the user clicks on one of the disclosure arrows either in the REPL or in the definitions area, the results window appears to the right of the main window. The format of the results window depends on the type of data being displayed. In most instances, it consists of a color-coded summary of error, warning, and success messages that Proof Pad has detected in ACL2's output, followed by a text area containing the raw ACL2 output.

A major part of this effort is the task of creating clear, concise summaries of common or frustrating ACL2 error messages. ACL2's output tends to be verbose and to suggest solution strategies that don't make sense for novice users. For example, consider the ACL2 error message shown in Figure 3. The first part of this message (where many students get stuck) suggests that the user needs to certify the book, which leads the user to research the complex certification feature of ACL2, even though, as is clear if the user keeps reading, the real problem is that the book simply could not be found. Proof Pad simplifies this to the error shown in Figure 4. The warning and error switch places (as they are sorted by importance), and the warning is de-emphasized with color coding and a different icon.

3 Implementation

In total, Proof Pad comprises approximately 11,000 lines of Java code developed over a period of 13 months. Figure 3 shows a timeline for the project. The project has expanded significantly as it pro-

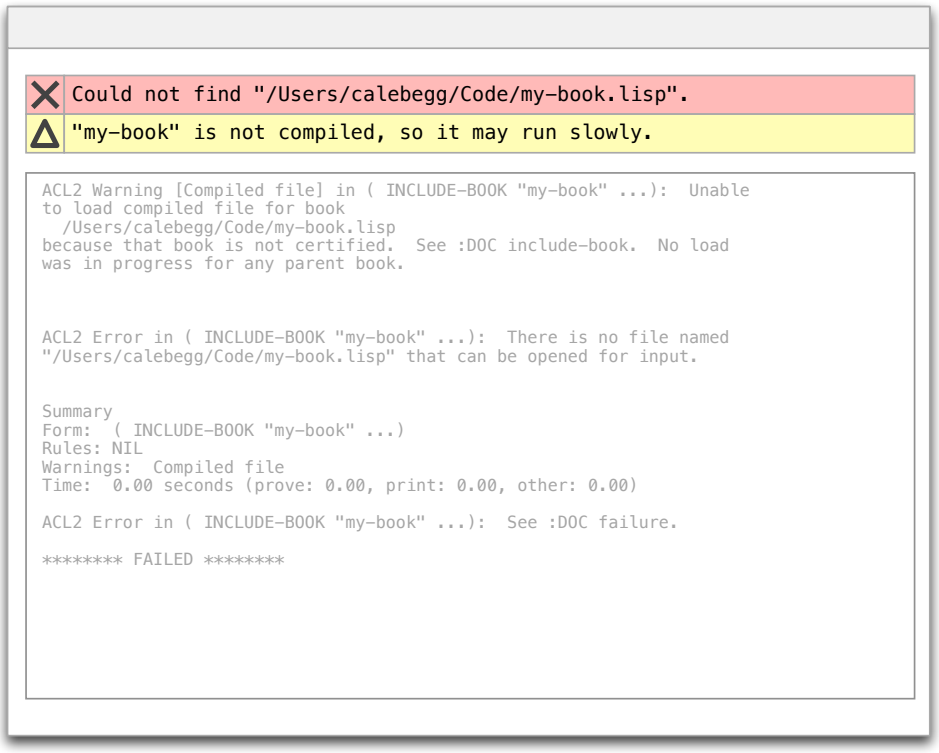

Figure 4: Mockup showing the results of trying to import a non-existent book; note that the warning is de-emphasized in favor of the more important error, but that the whole error message is also shown for the user's perusal.

gressed. The full source code can be found at `http://github.com/calebegg/proof-pad`, and is available under the GPLv3 license.

3.1 Code editor and syntax highlighting

Proof Pad uses the open-source Java code editor `RSyntaxTextArea` for the definitions pane and for the input to the REPL. `RSyntaxTextArea` makes certain familiar parts of code editing, such as syntax highlighting, easier to implement. I selected `RSyntaxTextArea` primarily for its speed and simplicity. The syntax highlighting parser works with a JFlex lexical analyzer generator. JFlex generates a minimal DFA from a regular expression based lexer definition so that the lexer code it generates is performant. As part of tailoring the syntax highlighting to ACL2, The JFlex lexer provided for Lisp needed to be heavily modified to make it aware of ACL2's keywords and some specific aspects of ACL2's primitive data types.

In addition to the syntax highlighting lexer, Proof Pad also has a slower, simple parser that runs after a few seconds of inactivity and scans the user's code, using the lexer's tokens, looking for several types of simple syntax errors, such as undefined functions or variables, too many or too few arguments for a built-in function or macro, or incorrect syntax for a built-in macro like `let` or `defthm`.

Proof Pad's auto-indentation feature watches for the user to enter a newline and determines the proper level of indentation using a full trace of the previous indent amounts and parentheses depth, taking into

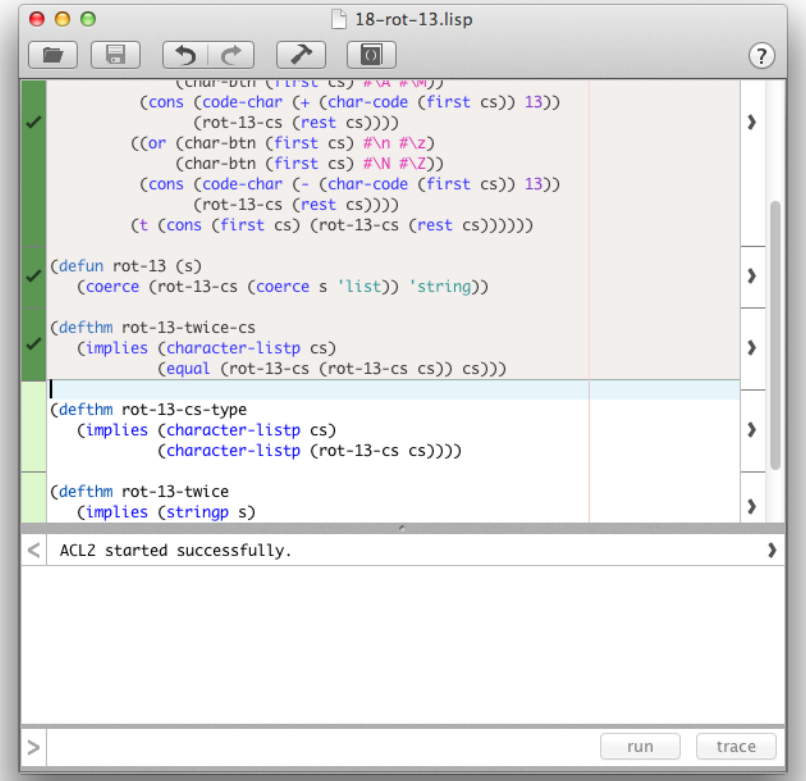

Figure 5: The main Proof Pad window, showing a file that is being actively worked on.

account specific indentation patterns for certain built-in macros. The user can also re-indent sections of code.

3.2 Wrapping ACL2

ACL2 is a complex tool with no public API. As such, it needs to be wrapped and its output parsed in order for the UI to give the user feedback on its progress and status. I used a separate Java thread and the Java standard library `ProcessBuilder` API to call and keep up with the progress of ACL2. The output is parsed and scanned for error or success messages, as well as for the prompt sequence (`ACL2 >`) to determine when the system is finished admitting an expression. Additionally, since some types of functions and macros can create multiple prompts, commands are sent to ACL2 to print identifying strings that can then be scanned for in the output.

3.3 DoubleCheck

I ported part of DrACuLa's DoubleCheck library to pure ACL2, so that DoubleCheck properties could be run in Proof Pad. This necessitated some changes to the system resulting from limitations in ACL2.

March 2012	Began work on prototype	7,400 lines of code
April 27 2012	First user evaluation	8,800
Fall 2012	Trial use in logic course	9,800
December 2012	Survey of students in logic course	
Spring 2013	Trial use in logic course	10,900
March 2013	Second evaluation	

Figure 6: Timeline showing some of the major events in the development of Proof Pad.

Probably most importantly, DoubleCheck relies on second order functions to allow users to define new random data generators. Since Proof Pad uses native ACL2 to interpret user code, Proof Pad cannot support user-defined generators. At this point, Proof Pad implements only a subset of the random data generators in Dracula's DoubleCheck system. This limits the usefulness of property based testing in large software development projects, but succeeds in allowing students to get some experience in property-based testing.

4 Evaluation

4.1 First user evaluation

Participants in the first Proof Pad evaluation came from The University of Oklahoma's software engineering course. This is the second semester of the capstone course for Computer Science, so many of the participants were seniors. Furthermore, because the course is taught using ACL2 in the DrACuLa environment, the participants already had a moderate amount of experience using ACL2, but with a different IDE. This made it possible to ask students participating in the evaluation to perform sophisticated evaluation tasks.

Some studies of usability testing have found that for qualitative usability studies like this one, small numbers of participants (four or five) are nearly as effective at finding most usability problems in an application as large numbers [13]. Five students participated in this initial evaluation.

The evaluation consisted of four tasks, each taking at most five minutes, followed by ten minutes allotted to discuss overall impressions of the tool, for a total of 30 minutes per student. Evaluations were performed individually with me on a Macbook Air, using either a Mac-like or Windows-like (with Windows-style keybindings and menus) build of Proof Pad. All students chose the Windows-style build.

For the first two tasks, the task was presented verbally alongside a Proof Pad window populated with some code. In the first task, the participant was asked to admit and run a provided function. In the second, a file that has three errors was shown, and the participant was asked to identify and fix any of them that they could, and then run the function.

For the third and fourth task, instructions were provided both verbally and on screen, along with an empty Proof Pad window for the student to work in. The first of these tasks asked them to write a function that computes the product of a list; one example is provided. The second task asks them to write and admit/run a theorem or property (DoubleCheck test) to demonstrate that `(prod (append xs ys)) = (* (prod xs) (prod ys))`, using either their `prod` function from the previous task, or a provided one if they did not complete the task.

From this first evaluation, I gained much valuable feedback. Nearly all participants had trouble discovering the use and purpose of the Proof Bar. Several participants expressed a desire for some of the

standard features of IDEs they've used before, such as line numbers, code folding, automatic insertion of parentheses, automatic saving, options to automatically fix highlighted errors, etc. In addition to this higher-level feedback, I also made several small usability changes to the application based on more minor problems experienced by only one or two participants.

4.2 Use in logic courses (Fall 2012 and Spring 2013)

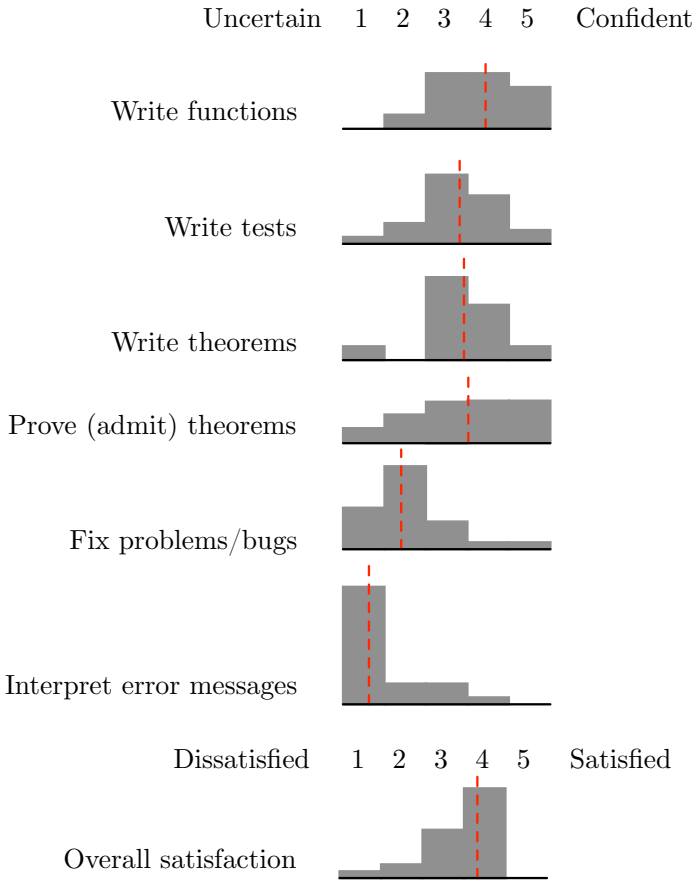

Figure 7: Graphical summary of the quantitative data collected from the one-page survey given to Applied Logic students at the end of the Fall 2012 semester.

Proof Pad was used as the primary software tool in the Fall 2012 section of Applied Logic and the Spring 2013 section of How Computers Work[11], a cross-listed honors course, at the University of Oklahoma. I provided support for the software throughout these two semesters, and the students completed several assignments using it. At the end of the semester, they were given an anonymous survey where they were asked to rate their confidence using certain parts of Proof Pad on a five-point scale, to supply (up to) three good things, bad things, and desired features for Proof Pad, and, finally, to rate their overall experience with Proof Pad on a five point scale. The results of the quantitative sections for the fall 2012 semester are summarized in Figure 7. For the three qualitative sections, I collected the

data in a spreadsheet, combining entries that I saw as duplicates. The top three items from each category are summarized in Table 2.

List up to three things you liked, found easy to do, or thought were useful about Proof Pad:

Count	Feature
5	Simple/elegant interface
5	Ability to prove theorems
3	Speed
3	Visual feedback/proof bar

List up to three things you did not like or found frustrating about Proof Pad:

Count	Aspect
18	Poor/difficult to understand error messages
7	Compatibility with individual computers
4	Resizing of the window is buggy
3	DoubleCheck syntax differs from textbook

List up to three features you wish Proof Pad had:

Count	Feature
9	Function documentation
6	Mark lines with errors
4	User manual/application help
4	Better error messages
4	Line numbers

Table 2: A summary of the qualitative results of the survey. Responses were collected in a spreadsheet, collated based on their similarity, and then sorted. Items with more than two responders are shown for each of the three qualitative questions.

4.3 Second evaluation

Another evaluation of Proof Pad is planned for the end of the spring of 2013 semester. I plan to recruit participants from the University of Oklahoma's second-level introductory programming course, give them an IDE-agnostic training session for the basics of ACL2, and then have them perform tasks one-on-one using either Proof Pad or DrACuLa (randomly selected).

5 Conclusion

Proof Pad is available now at `http://proofpad.org/`. It has been used in two classes at the University of Oklahoma, and will continue to see use there. By putting pedagogic goals and student expectations at the forefront of all my decisions, I hope to have developed a useful tool for teachers and students alike to take advantage of ACL2 in course work.

6 Future work

6.1 `.proofpad` files

```
(include-book "testing" :dir :teachpacks)
(include-book "doublecheck" :dir :teachpacks)

;; Write a definition for sum here. It must satisfy the below
;; tests and property.




(check-expect (sum nil) 0)
(check-expect (sum (list 1 2 3)) 6)

(defproperty sum-append
   (xs :value (random-integer-list)
    ys :value (random-integer-list))
   (= (sum (append xs ys))
      (+ (sum xs) (sum ys))))
```

Figure 8: A mockup of the definitions area of Proof Pad, showing an open `.proofpad` file.

Since use of Proof Pad in the classroom is my primary goal, I think it's important to take into account how ACL2 is used and how other pedagogic IDEs work, and to try to design Proof Pad to be as easy to use in the classroom as possible. In pursuit of this goal, and in line with some of the existing uses of ACL2 in the classroom, I have a planned feature to make distributing code for projects and giving students direct feedback on their progress possible.

Instructors will be able to create a special type of `.proofpad` file that, when opened in Proof Pad, is divided into some read-write regions and some read-only regions. Using this mechanism, an instructor can both provide existing code (to supplement ACL2's limited standard functionality, for instance) and set up tests or theorems that they expect the student to satisfy. An example is shown in Figure 8; in this example, I made use of DrACuLa's `testing` teachpack, which includes a function, `(check-expect left right)`, which passes if `left` = `right`, but fails and issues an error message otherwise.

The student's goal for this file is to modify the read-write (light background) portion of the file to make the following two tests and one property pass. In this case, they can see that they need to write a function, `sum`, that returns either 0 for an empty list, or 6 for the list containing 1, 2, and 3.

Through this mechanism, the instructor can easily work up in difficulty from simple tasks like modifying existing functions to cause failing tests to pass, all the way to advanced tasks like steering ACL2 to solve a complex theorem by adding lemmas to the logical world.

7 Acknowledgements

Rex Page is my Master's thesis advisor, under whom I've done this work. His support and feedback have been invaluable. Peter Reid provided feedback and several patches for issues he encountered early in the project. Matthew Kaney created the icon for Proof Pad, selected the color scheme for the proof bar, and gave feedback on other visual elements of the design. An early discussion about Proof Pad with Ruben Gamboa led to the idea for the `.proofpad` files. Proof Pad uses some open source components; in

particular, the previously-discussed `RSyntaxTextArea` (written and maintained by Robert Futrell), and, of course, ACL2 itself. The compiled and certified binaries of ACL2 that Proof Pad uses were created and provided by the ACL2s project. Parts of DoubleCheck and several other DrACuLa libraries, written by Carl Eastlund and others, are included with Proof Pad for compatibility with files that use these libraries.

References

[1] Eric Allen, Robert Cartwright & Brian Stoler (2002): *DrJava: a lightweight pedagogic environment for Java*. In: *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, SIGCSE '02, ACM, New York, NY, USA, pp. 137–141, doi:10.1145/563340.563395.

[2] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann & Panagiotis Manolios (2011): *Integrating Testing and Interactive Theorem Proving*. In David Hardin & Julien Schmaltz, editors: Proceedings 10th International Workshop *on the ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, November 3-4, 2011, *Electronic Proceedings in Theoretical Computer Science* 70, Open Publishing Association, pp. 4–19, doi:10.4204/EPTCS.70.1.

[3] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon & J. Strother Moore (2007): *ACL2s: "The ACL2 Sedan"*. In: *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006)*, *Electronic Notes in Theoretical Computer Science* 174, pp. 3 – 18, doi:10.1016/j.entcs.2006.09.018.

[4] C. Eastlund & M. Felleisen (2009): *Toward a practical module system for ACL2*. *Practical Aspects of Declarative Languages*, pp. 46–60, doi:10.1007/978-3-540-92995-6_4.

[5] Carl Eastlund (2009): *DoubleCheck your theorems*. In: *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, ACL2 '09, ACM, New York, NY, USA, pp. 42–46, doi:10.1145/1637837.1637844.

[6] Carl Eastlund, Dale Vaillancourt & Matthias Felleisen (2007): *ACL2 for Freshmen: First Experiences*. In: *ACL2 07: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, ACM Press, pp. 200–211.

[7] Robert Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi & Matthias Felleisen (1997): *DrScheme: A pedagogic programming environment for scheme*. In Hugh Glaser, Pieter Hartel & Herbert Kuchen, editors: *Programming Languages: Implementations, Logics, and Programs*, *Lecture Notes in Computer Science* 1292, Springer Berlin / Heidelberg, pp. 369–388, doi:10.1007/BFb0033856.

[8] Matt Kaufmann, J. Strother Moore & Panagiotis Manolios (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, doi:10.1007/978-1-4615-4449-4.

[9] Rex Page (2007): *Engineering Software Correctness*. *Journal of Functional Programming* 17, pp. 675–686, doi:10.1017/S095679680700634X.

[10] Rex Page (2011): *Property-Based Testing and Verification: a Catalog of Classroom Examples*. In: *Proceedings of the 2011 Symposium on Implemenation and Application of Functional Languages*, Lawrence, KS, pp. 134–147, doi:10.1007/978-3-642-34407-7_9.

[11] Rex Page & Ruben Gamboa (2012): *How Computers Work: Computational Thinking for Everyone*. In: *Proceedings of the First International Workshop on Trends in Functional Programming in Education*, St Andrews, UK, doi:10.4204/EPTCS.106.1.

[12] Dale Vaillancourt, Rex Page & Matthias Felleisen (2006): *ACL2 in DrScheme*. In: *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, ACL2 '06, ACM, New York, NY, USA, pp. 107–116, doi:10.1145/1217975.1217999.

[13] R.A. Virzi (1992): *Refining the test phase of usability evaluation: How many subjects is enough?* *Human Factors: The Journal of the Human Factors and Ergonomics Society* 34(4), pp. 457–468.