Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 3–18
www.elsevier.com/locate/entcs

ACL2s: "The ACL2 Sedan"

Peter C. Dillinger Panagiotis Manolios Daron Vroon

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280, USA
{peterd,manolios,vroon}@cc.gatech.edu

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
moore@cs.utexas.edu

Abstract

ACL2 is the latest inception of the Boyer-Moore theorem prover, the 2005 recipient of the ACM Software System Award. In the hands of experts it feels like a finely tuned race car, and it has been used to prove some of the most complex theorems ever proved about commercially designed systems. Unfortunately, ACL2 has a steep learning curve. Thus, novices tend have a very different experience: they crash and burn. As part of a project to make ACL2 and formal reasoning safe for the masses, we have developed ACL2s, the ACL2 sedan. ACL2s includes many features for streamlining the learning process that are not found in ACL2. In general, the goal is to develop a tool that is "self-teaching," *i.e.*, it should be possible for an undergraduate to sit down and play with it and learn how to program in ACL2 and how to reason about the programs she writes.

Keywords: ACL2, Eclipse, theorem proving, script management

1 Introduction

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." It is the name of a programming language, a first-order mathematical logic based on recursive functions, and a mechanical theorem prover for that logic [9,5,4]. ACL2 is an industrial-strength version of the Boyer-Moore theorem prover [2] and was developed by Kaufmann and Moore, with early contributions by Robert Boyer; all three developers were awarded the 2005 ACM Software System Award for their work. Of special note is its "industrial-strength": as a programming language, it executes so efficiently that formal models written in it have been used as simulation platforms for pre-fabrication requirements testing; as a theorem prover, it has been used to prove the largest and most complicated theorems ever proved about commercially designed digital artifacts.

1571-0661
doi:10.1016/j.entcs.2006.09.018

ACL2's power comes with a steep learning curve. This is not an issue of documentation. ACL2 has extensive documentation, including tutorials, a user's manual, workshop proceedings, and related papers, all of which are available from the ACL2 homepage [9]. ACL2 is also described in a textbook by Kaufmann, Manolios, and Moore [5], and there is also a book of case studies [4]. The sources for ACL2 are freely available on the Web, under the GNU General Public License.

ACL2's steep learning curve is due to two major factors. The first factor is usability. Beginners have to use ACL2's command line user interface and are encouraged to learn GNU Emacs. Once they start proving theorems, they are confronted with the problem of developing a mental model of what ACL2 is doing, something that is inherently difficult: reasoning about a system that reasons about other systems. Driving a user interface that is unfamiliar, non-intuitive, and happily permits lots of illogical actions distracts new users from what is important.

The second factor is the ACL2 logic. ACL2 has this tremendous advantage over many other theorem provers: it is grounded in a programming language. This makes it very easy to introduce ACL2 to users with a computer science background. However, once the logic is introduced, termination becomes an issue. In order to guarantee soundness, ACL2 only accepts functions that are shown to terminate. While ACL2 can do this automatically in many cases, there are also simple cases that require user guidance. Termination reasoning in ACL2 is very powerful because it is based on the ordinal numbers. While students and beginners eventually understand (and are even sometimes fascinated by) the ordinal numbers, their introduction significantly increases the knowledge required for interesting interaction with ACL2. Note that termination is not only used to admit recursive functions, it induces sound induction schemes, which play a central role in ACL2.

To address the above two factors and thereby make ACL2 more accessible to beginners, we have developed and released ACL2s, the ACL2 sedan [3]. ACL2s is available at `http://www.cc.gatech.edu/home/manolios/acl2s` and is being developed with the goal of making formal reasoning accessible to the masses, with an emphasis on building a tool that any undergraduate can profitably use in a short amount of time.

To address the usability factor, ACL2s features a modern graphical integrated development environment in Eclipse that provides an intuitive, robust "script management" interface with an improved front-end to the familiar "command line" interface. The prior is good for augmenting or curtailing the current theory while the latter is good for querying, testing, or debugging the current theory. ACL2s permits the user to switch between the two without fear of either one misrepresenting the relevant logical history. Other features help to eliminate other simple misunderstandings that distract from the specification and proof process: full syntax highlighting, syntax error underlining, auto-indenting, character pair matching, and input command demarcation. In addition, "session modes" serve to hide complicated functionality from novice users.

To address the logic factor, we have developed and incorporated into ACL2s *CCG termination analysis* [8]. This is a powerful, state-of-the-art termination anal-

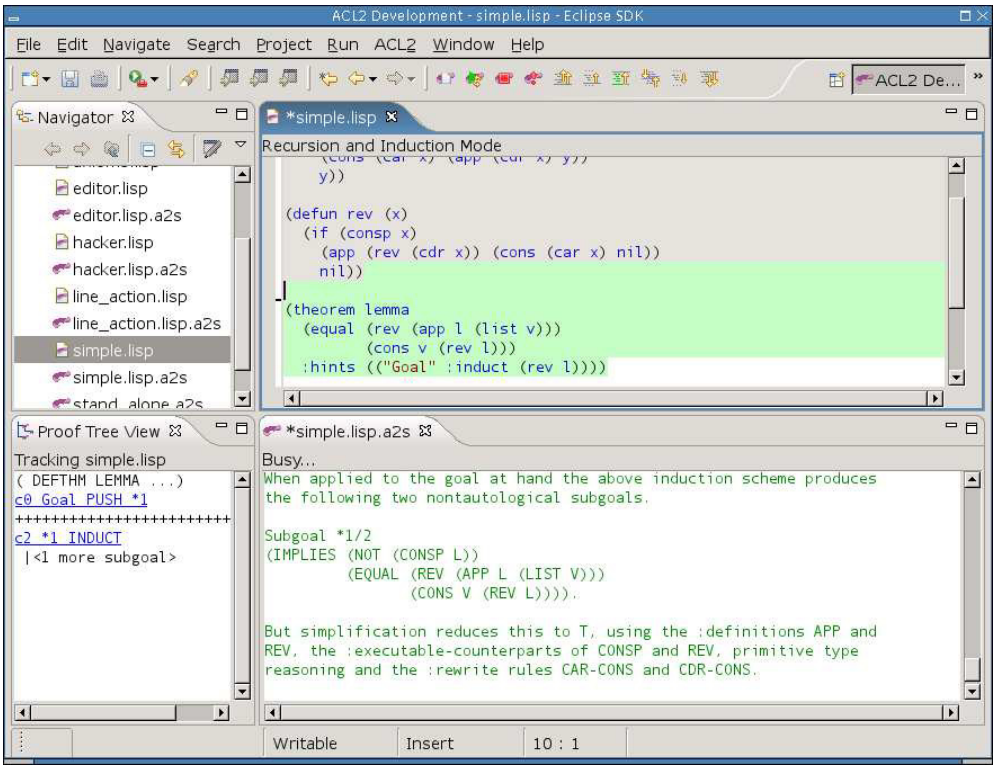

Fig. 1. **Snapshot of Eclipse workbench running ACL2s.** The top-left frame is Eclipse's file Navigator view. The top-right frame is an ACL2s source code editor. The bottom-right is an ACL2s session editor associated with that source code editor. The bottom-left is a "proof tree" view of proof happening in the session editor.

ysis method which significantly automates termination arguments. This eliminates the need for students to justify the kind of user-defined recursive functions and induction schemes that would be covered in an undergraduate class. We can therefore avoid discussing termination analysis completely or until well after students have become proficient ACL2 users. In addition, ACL2s includes several levels appropriate for beginners through experts. This allows us to introduce the major concepts in ACL2 in easy-to-understand modules that do not overwhelm beginners.

Together, the features of ACL2s lower barriers to learning specification and verification in ACL2—which was the sense after two graduate courses switched to requiring use of the tool.

2 GUI Overview

In the Eclipse *workbench*, development with ACL2s is centered around two types of *editors*: the source editor (also called "Lisp editor" or ".lisp editor"), and the session editor (or ".a2s editor"). In most cases, the user will use linked pairs of these editors, such as editing `somefile.lisp` and `somefile.lisp.a2s`. Based on their naming, the plugin links these so that each provides a consistent view of their shared logical *history*.

2.1 Line(s) in the Source Code

The source editor is where the user enters top-level definitions and commands as if programming offline, but the editor also provides *script management*[1] capabilities, providing what ACL2 literature calls "the line" [5]. The editor actually maintains two lines, which we call the "completed line" and the "todo line." Because the "completed line" is never beyond the "todo line", the lines induce three (potentially empty) regions in this order: the "completed region" (gray highlight and read-only), the "todo region" (green highlight and read-only), and the "working region" (no highlight, read/write). These regions can be seen in Figure 1. The lines and other metainformation are stored in special comments on disk, preserving source code compatibility.

User interface actions grant essentially free manipulation of the "todo line", regardless of the state of the associated ACL2 session, or whether it's even running. The "todo line" will only advance past syntactically well-formed ACL2/Lisp input and only at the granularity of whole commands (see Section 4.1). Moving the "todo line" can have consequences including initiating processing of "todo" forms, interrupting the processing of a form no longer in the "todo" region and "UNDO"ing of completed commands (see Section 4.3). As ACL2 is processing forms from the "todo" region, it advances the "completed" line on success and resets the "todo line" to the point of the "completed line" on failure. If ACL2 is restarted, the "completed line" is moved to the top. In each case, we are maintaining the simple invariant that the "completed" forms have been accepted by ACL2 (and have not been undone), the "todo" forms are being processed (if the ACL2 session is running), and the "working" area is freely editable.

"Script management"-style interaction usually involves the session editor as well, which gives ACL2's output in response to forms processed as a result of line motion. The session editor shows almost exactly what the user would see if she had been manually copying processed forms into a terminal running ACL2—though the session editor has some significant enhancements for browsing output (see Section 5.3).

2.2 Command Line

Even with a script-style interface, lots of ACL2 interaction does not make sense from such an interface. We therefore made the session editor much more than a provider of detailed output. The session editor implements a command line interface to the same session used by the script-style interface of the source code editor. Most importantly, the user cannot "trick" ACL2s into an inconsistent state by switching between the two interfaces. The basic mechanism for this is copying any successful, *relevant* commands (see Section 4.2) submitted at the prompt in the session editor to the "completed region" in the associated source code editor.

The session editor looks like a dump of the input and output to a sequence of ACL2 sessions, but input coming from the "todo" region and input typed at the prompt look the same in the history.

2.3 Other UI Pieces

ACL2s also incorporates a clickable *proof tree* view, much like the proof trees provided by the Emacs interface to ACL2. Our tool takes the view a step further, though, by remembering the final proof tree of all completed commands and bringing them up as the cursor is moved to corresponding sections of the session editor.

So far, the only wizard provided by ACL2s is a "New ACL2s/Lisp file" wizard, which allows the user to pick a session mode (see Section 3.1) and has some options for generating some skeleton code that commonly appears at the top of ACL2 files.

2.4 User Experiences

Our above description of some intricate interaction between the session editor and the source code editor do not translate to difficult understanding by users. ACL2s has been a required tool for two graduate courses with an introduction to theorem proving, and students have understood our merger of the script management and command line interfaces almost immediately.

3 Language Extensions

As part of ACL2s, we have made some extensions to the underlying ACL2 tool, but we have always made sure not to disable or obscure any ACL2 functionality available outside of ACL2s.

3.1 Session Modes

Analogous to "language levels" in DrScheme [10], ACL2s offers (at present) three modes of behavior for the underlying ACL2. In teaching ACL2, the modes can be introduced in this order:

- *Programming Mode.* This mode is intended to introduce new users to ACL2 as a programming language of untyped, total functions. None of the ordinary restrictions relating to logical soundness apply. With the exception memory exhaustion (heap or stack), no runtime errors are possible with functions defined in Programming Mode. Macro definition and usage is also available in this mode.

  Implementation Note: Readers with knowledge of ACL2 will note that this is similar to the built-in "program mode" for definitions, but there is at least one important difference: runtime checking of *guards*. Guards facilitate fast, raw lisp execution but are irrelevant to the logical language of ACL2. To novices, guard checking is a distraction, which is why our Programming Mode disables it. ACL2 version 3.0 has fixed the shortcoming in "program mode" by adding an option to turn off *all* guard checking. This will eliminate the need for the hack we currently use to implement our "Programming Mode."
- *Recursion & Induction Mode.* Defining functions and macros in this mode is just like in pure ACL2, except that the theorem prover is able to prove termination of most terminating functions with no help (using CCG termination analysis,

```
(defun sum-lists (x y)
  (if (or (consp x) (consp y))
      (cons (+ (car x) (car y))
            (sum-lists (cdr x) (cdr y)))
    nil))
```

Fig. 2. The function, `sum-lists`, takes two lists and returns the list whose elements are the sums of the elements of the inputs. If one input list is longer than the other, the returned list is as long as the longer and the smaller is considered to be padded with 0s.

described in Section 3.2). Theorem proving in this mode is accomplished with a macro that inhibits automatic, guessed induction and adding the rules generated by the theorem to the enabled *theory*. To perform induction, therefore, the user must provide an explicit hint with the scheme to use, forcing the user to think carefully about when and how induction should be applied. Utilizing user-defined lemmas (theorems) as proof rules also requires explicit hints. This helps users to focus on individual proofs rather than building a coherent theory, which is harder still.

- *Pure ACL2* This mode is just like regular ACL2, except that we offer CCG termination analysis as a convenience.

3.2 CCG Termination Analysis

Termination in ACL2. A significant stumbling block for new users and a source of frustration for experienced users of ACL2 is termination. Every function admitted to ACL2 must be proven to terminate for all inputs before ACL2 will accept it. First, this guarantees the logical consistency of function definitions—that every syntactically legal function application corresponds to exactly one value. Second, ACL2 derives induction schemes from recursive functions, and those induction schemes are sound as a consequence of termination of the corresponding function. Induction is an integral part of the theorem proving capability of ACL2, especially in proving properties over infinite classes of input.

To prove termination of a function, ACL2 uses a specified or guessed *measure* to map the function's inputs into values in the domain of a *well-founded* relation, such as the $<$ relation on the natural numbers. If the measure always returns a value in the relation's domain and recursive calls always use inputs that, according to the measure and well-founded relation, are "smaller than" the previous, it cannot go on forever. (No sequence decreasing according to a well-founded relation can be infinite.)

ACL2 uses only simple heuristics to guess measures when proving termination, so it is easy to define functions for which ACL2 is not able to guess the correct measure. Therefore, new users soon find the need to learn about engineering and justifying measures to ACL2, which tends to overwhelm those who are still struggling to prove simple theorems.

For example, ACL2 cannot guess the measure for the `sum-lists` in Figure 2. Us-

ing $<$ (the normal less-than relation) over the natural numbers as our well-founded relation, our measure can be `(+ (len x) (len y))`, where `len` returns the length of a list (0 for atoms, and $1 +$ `(len (cdr x))` for `cons`es, `x`).

Mechanics of CCG analysis. In [8], we introduce a new, more automatic termination analysis based on CCGs, or *context calling graphs*. Within a function (or set of mutually recursive functions) we augment each recursive call site with predicates that are needed to get there ("*rulers*") within the function. These augmented call sites are *calling contexts*, and a calling context graph (CCG) is a graph whose vertices are calling contexts and has an edge from e to e' if e calls the function containing e' and e can lead directly to e' during execution. Intuitively, a CCG is like a call graph but in terms of call sites instead of functions, giving it finer granularity.

Now we apply the notions of *measure* and *well-founded relation* CCGs. Suppose we can assign a set of measures—called *calling context measures*, or CCMs—to each context such that every infinite path through the CCG has a corresponding sequence of CCMs that never increase and decrease infinitely often. It follows that every computation must then terminate. Theorem proving plays a key role in this analysis as it is used to prune edges from CCGs and to determine when CCMs are non-increasing or decreasing as we traverse edges in CCGs.

Our algorithm uses heuristics to pick CCMs, together with a sufficient condition for the above path-related criterion that is based on [7]. More details and other improvements are described in our extended abstract [8].

Results of using CCG analysis. We ran our CCG implementation on ACL2's regression suite. This is a collection of ACL2 libraries on a variety of topics including arithmetic, set theory, processor verification, and model checking. These libraries were submitted by various members of the ACL2 community over a decade, and are therefore representative of typical ACL2 usage. The regression suite contains over 10,000 function definitions, a significant number of which required explicit user intervention to prove termination. When running our termination analysis on the regression suite, we discarded explicit user hints, and provided no manual assistance. Our analysis successfully proved 98.7% of the 10,000 functions terminating, including 68.2% of those that previously required explicit user hints.

We have implemented our algorithm into the current version of ACL2s [6]. The result is that ACL2s now proves termination automatically for a much higher proportion of functions, particularly among simpler functions that new users tend to define. The `sum-lists` function, for example, is easily proven to terminate by our analysis. With our analysis, a discussion of the complex concepts of termination analysis can be postponed for new users, allowing them to become more familiar and comfortable with the basic concepts of ACL2 first. In addition, advanced users can spend less time carefully engineering and justifying measures.

4 Script Management

Implementing a powerful, robust "script management"-style interface for ACL2 was non-trivial. First, we would need to be able to detect entire input forms for ACL2. Next, some input forms require explicit "undo" while others have no effect other than printing some result. Others still are not undoable with the regular "undo" command, but we do not want to restrict the commands available from the interface.

Another complication has to do with our command line interface. Recall that successful, "relevant" commands entered at the session editor's command prompt are inserted at the completed line in the source editor. We needed to come up with a notion of "relevance" that made sense.

4.1 Input Demarcation

To know where to move the todo line when the user asks to advance it one ACL2 form, we implemented a Common Lisp parser in Java. This parser (call it the "batch parser") was implemented before and independent of the parser that does syntax highlighting and checking for the editors (call that one the "online parser"; see Section 5.1). The batch parser also pulls entire ACL2 forms typed at the session editor's command prompt.

Another view of the job of the batch parser is to make sure that each time the ACL2 reader asks for an expression, it is given exactly one syntactically well-formed expression. "Well-formed" in this case means that it will not generate a *read error* by ACL2's reader. In the case of ill-formed input, the batch parser generates an appropriate error message with a relevant location in the input. With the batch parser in place, neither the plugin nor the user need worry about odd behavior from ACL2 in recovering from read errors or getting stuck with ACL2 expecting more input but not know exactly what is needed to complete an expression. These cases are particularly frustrating to novices.

ACL2's *keyword commands* are convenient, but are more prone to that "what else am I supposed to type" experience when used at a terminal because they potentially require several expressions to compose a single input form. Our tool imposes a stricter interpretation of keyword commands that, in a sense, fixes the confusion: keyword commands are terminated *only* by a newline outside of a Lisp expression, which corresponds to existing conventional usage. The mechanism enabling us to adopt this interpretation is our plugin's translation of keyword commands to their non-keyword equivalents before giving them to ACL2. This means that if the wrong number of parameters is given, instead of ACL2 blocking or misinterpreting input, it simply reports, "wrong number of arguments."

4.2 Input Classification

The next step in our solution was to classify each input form before letting the underlying session execute it. This would tell us (ACL2s) about the form's relevance and also provide some useful feedback to the user. In fact, the only real textual

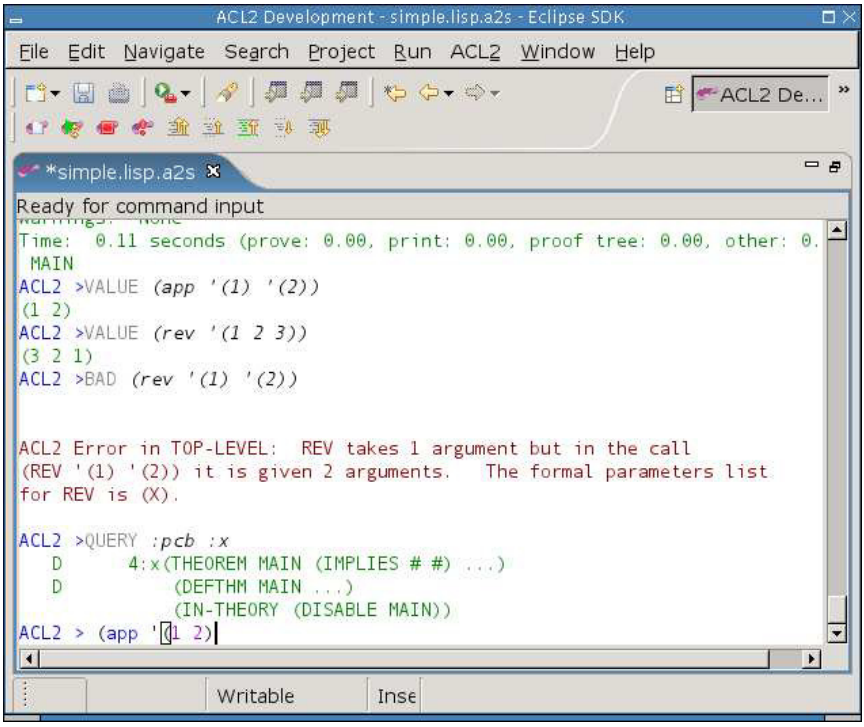

Fig. 3. **Close-up of the Session Editor.**

difference between a terminal dump and our session editor contents is that we prepend each input with a classification that categorizes the potential effects it could have. Figure 3 shows some examples.

Classifications can depend on the history of the particular session, so the easiest way to classify an input is inside of ACL2. Part of our extensions to the ACL2 core is some code for classifying inputs, that builds on some existing code in ACL2 for deconstructing and analyzing inputs. Also, we designed our Eclipse plugin code that interfaces with the ACL2 core to be able to handle requests that are hidden from the user. The output seen in the session editor is *not* all the output that the ACL2 session has generated; it's only the output that is relevant to the forms entered by the user!

Here are most of the possible classifications:

- *EVENT* is usually a definition of a function or theorem and is always considered relevant because it could modify or extend the logical *world*. Events also play an important role in book development, discussed in Section 5.2.
- *VALUE* is some computation that, by the top-level function's signature, is unable to change any *state*, including the logical world. The best example is testing a function on some input to see its result. Values are irrelevant.
- *QUERY* is one of many built-in commands that relate to the logical world, but are known not to change anything. Examples include printing the rules associated with a symbol or trying to prove an unnamed (thus, immediately

forgotten) conjecture. Queries are irrelevant.

- *BAD* is given to an input if the categorization code is able to detect an error the plugin's batch parser is not. Such semantic errors include trying to invoke a function with the wrong number of parameters. Because BAD inputs always fail, they are, in a sense, inherently irrelevant.
- *COMMAND* changes something in ACL2 that the built-in undo mechanism does not handle but ACL2s can undo cleanly and reliably. To be safe, commands are considered relevant.
- *IN-PACKAGE* is a special COMMAND that changes the current package. It is, thus, relevant, but is singled out for its role in book development (see Section 5.2).
- *ACTION* is a catch-all for inputs that could have effects we don't know how to undo. It would be rare for a novice to invoke such an input and rarer still for a non-undoable effect to affect soundness. Actions are considered relevant and cause a warning to be printed when they are undone (to the extent we are able).
- *UNDO* and *REDO* are generated by motion of the "completed" line, as discussed below.

These classifications give us a sane way of implementing script management and maintaining consistency between the two editors.

4.3 Undo

To have clean, powerful support for retreat of the "completed" line, the modified ACL2 core used by ACL2s includes an undo mechanism that is, in a sense, more primitive than the built-in undo mechanism. In other words, an ACL2 undo can be used as just another command on top of ACL2s's undo mechanism. Basically, our mechanism keeps a list of old pseudo-states, each of which contains a logical world and many other settings that affect the treatment of input. To perform an UNDO, the Eclipse plugin simply invokes the mechanism for restoring a previous pseudo-state. It is non-trivial for the ACL2s user to invoke this mechanism with a command (rather than causing the plugin to invoke it in response to line motion) because using the mechanism requires knowledge of a secret number chosen at random for each session. The secret is hidden in the ACL2 session such that only an expert who *really* wanted to usurp our undo structure would be able to do so.

4.4 Redo

Whenever ACL2s performs an UNDO, the pseudo-state it was in before the UNDO is not forgotten—nor is any pseudo-state that got us here by some sequence of UNDOs, REDOs, and irrelevant forms. Thus, we have a mechanism to return to such states. A REDO is actually invoked if, following the UNDO of a form x plus any sequence of matching UNDO/REDOs and irrelevant forms, a form y is submitted with the same abstract syntax as x. Two forms have the same abstract syntax if they parse to the same Lisp objects. Comments, for example, are irrelevant to abstract syntax, but an extreme example would be

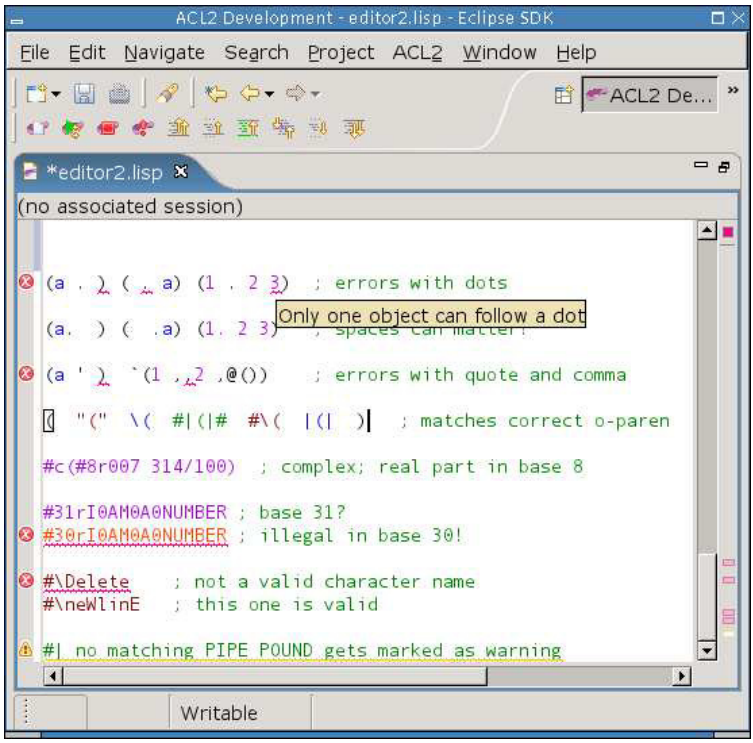

Fig. 4. **Close-up of the Source Editor.**

```
(1+ 42) ≅ (Ac|L|2::1\+ #c(840/20 -0.) . ())
```

A nice consequence of the REDO mechanism is the ability to modify comments and such above the line in a way that ensures ACL2 would process it the same way, but without having to wait for ACL2 to reprocess it. Simply retreat the line high enough, make the modifications, and move the line back to where it was. If the abstract syntax is unchanged, the completed line will move back to where it was almost immediately using REDOs. In fact, ACL2s even allows this with no session running! (It pretends to perform the UNDOs and REDOs that would be legal.)

5 Editor Features

5.1 Source Code

The source code editor can be used to edit .lisp files even when no corresponding .lisp.a2s file is present, meaning the editor is not paired with a session editor. The editor complains about some Common Lisp syntax that is illegal in ACL2, but an option to disable those cases might be included in future releases.

An important part of the source editor is the "online parser" it utilizes, which is a hand-coded incremental Lisp lexer with some parser-like capabilities as well. This part of the plugin is responsible for dividing up and classifying tokens for syntax-based coloring, depicted in Figure 4. It also computes matching character pairs and annotates the code in the case of any illegal syntax. The token representation also

plays a role in intelligent auto-indenting.

The editor matches open and close parentheses, as one would expect, but it also matches double-quotes around strings, pipes within symbols, potentially nested `#|`-style comments, and parentheses within comments. Particularly impressive to a crowd of ACL2 experts was the editor's matching of "`,`" and "`,@`" tokens to their respective backquote characters.

Only one type of potential ACL2 read error is not annotated in the source editor: the undefined package error. The set of defined packages can grow dynamically, so such errors are not identified until checked by the batch parser. All other errors show up as usual in Eclipse, with red or yellow underlining, a mark in the *overview ruler*, and a message that appears when hovered over. Examples are depicted in Figure 4.

The auto-indenting is much like Emacs or DrScheme. A notable, much-praised exception is indenting inside of string literals according to rules followed by ACL2's built-in functions for formatted output.

The editor scales nicely, for it performs acceptably fast on (ASCII text) Lisp files from the ACL2 source code that exceed a megabyte. The only exception is when a change causes the reformatting of almost the entire megabyte file—such as commenting out the whole file. This can take a few seconds to complete, but the bulk of that cost is Eclipse applying off-screen style information we give it.

5.2 Book Development

In many cases, ACL2 development is the development of a *book*, which is basically a reusable collection of definitions. Defining a book, though, can be tricky for several reasons, most of them relating to the requirement that books be processable by Common Lisp outside of ACL2. First, the *preamble* must be processed by ACL2 before the contents of the book, but it cannot simply appear above the book in the book's source file. Some users put the preamble into a special comment, some put it in a separate file, and many do both. This is a pain. In ACL2s, the preamble can be written directly at the top of the source file, though on disk it is stored in a comment (and possibly another generated file).

ACL2s has its own special construct for marking the end of the preamble: `(begin-book)`, which actually takes some optional parameters that are given to `(certify-book ...)` when the user asks to *certify* the book. When submitted to a running session, `(begin-book)` does not do anything special—nor does it need to.

What does happen when `(begin-book)` is submitted, is the source editor begins highlighting that part of the completed region with light purple instead of gray. As more forms are completed, the light purple highlight extends as long as the forms are legal for a book. After the `(begin-book)` this must be an `(in-package ...)`. After that, only EVENTS (see Section 4.2) are legal.

During book development, it is not unusual for the user to move the line past some forms that are not legal within books. This is fine, and we call this a "tangent".

When the user is ready to undo his tangent, the light purple highlighting indicates the point where legal book constructs were last abandoned.

We have not yet implemented "one click" certification of books within ACL2s, but the infrastructure is mostly there.

Finally, if `(begin-book)` is never used, no preamble is stored and no special highlighting of the completed region is done. The book development features do not complicate things if not utilized.

5.3 Session Editor

The session editor is the locus of our improved command line interface to ACL2 and captures much more than just a "dump" of the input and output. The saveable and restorable session history is a sequence of $\langle \mathit{Environment}, \mathit{Input}, \mathit{Output}, \mathit{Status} \rangle$ tuples. The *Environment* contains information such as the current Lisp package, the length of the logical history, the prompt printed to the user, and other settings that can influence the meaning of input. The *Input* captures the concrete and abstract syntax of an input form and a categorization describing its potential effects. The *Output* stores the text of the output, the location of *checkpoints* within the output, and the final *proof tree* associated with the command. The *Status* indicates whether the command was successful and, if not, whether it was interrupted.

The editor uses color to distinguish sources or types of text. For example, the prompt is blue, the input categorization is gray, the actual input is black, and the output is red on failure or green on success, as depicted in Figure 4.

The output can be navigated like any other read-only editor in Eclipse, but there are special shortcuts for traversing from input to input and among checkpoints within a single command's output.

The only editable region is beyond the final prompt, and it is only editable if ACL2 is waiting for input. This region, which we call the "immediate" region (for typing "immediate" commands), uses the same online parser and presentation scheme as the source editor. The command line interface, therefore, has character matching, syntax error highlighting, and even auto-indenting. The history of immediate commands is also navigable, much like in a UNIX shell.

The typed immediate command is submitted when *Enter* is pressed, though it's not quite that simple. When *Enter* is pressed, the batch parser checks to see if some prefix of the typed input is a syntactically well-formed input form. If not, the *Enter* simply causes a newline to be inserted. If a prefix is a full command, that prefix is removed and submitted as the next input. This could leave some text left over if an eager user decides to type more than one command at a time. The immediate text disappears while ACL2 is busy but reappears once another command is expected.

The session editor also supports typing input to ACL2 that is not command input. For example, ACL2 sometimes prompts the user for an answer to a yes/no type question. Another example is interacting with the *proof checker*, which has its own set of commands. All of these cases expect a single Lisp object as input, and our plugin is able to detect when ACL2 is expecting non-command input.

Non-command input is never taken from the source editor, but must be typed in the session editor, which, in fact, tracks an independent command line history for non-command input.

One never interacts with raw Lisp from ACL2s. There are cases in ACL2 in which errors take the user to a raw Lisp prompt and the user must manually break out of it to return to ACL2. ACL2s takes care of this for the user, partly for convenience but partly because it's hard to determine when raw Lisp is expecting input.

6 Related Work

6.1 ACL2 in DrScheme

Researchers at Northeastern University have hooked ACL2 into DrScheme [12]. Their preliminary system has some features we would like to have in ACL2s, but it also has some inherent limitations.

One part of the system is a DrScheme language for "ACL2 Beginner," which is an attempt to duplicate the ACL2 language using Scheme macros and Scheme functions. This simulated ACL2 benefits from the features of the DrScheme development environment, including its simple GUI, its debugging features, and its static checking features. The feature that would be most difficult to mimic is the "Check Syntax" feature, which performs macro expansion in a way that allows uses of lambda variables to be linked to their point of definition/declaration in the original source code.

The complication, however, is that Scheme is only partially compatible with Common Lisp, the basis of ACL2 proper. A clean embedding of full ACL2 in Scheme probably is not possible, due to incompatibilities such as packages and other namespace issues. The current "ACL2 beginner" language has other incompatibilities, including use of functions as first-class values and a restricted macro language.

Not necessarily an incompatibility but, in our opinion, a poor design decision for the current "ACL2 beginner" language was incorporation of *contracts* for functions, analogous to ACL2 *guards*. Guards are not part of the logical ACL2 language, so we feel such dynamic type errors complicate a beginner's ACL2.

ACL2 in DrScheme also provides a basic script management interface for interacting with the theorem prover. The implementation is still rough and easy to break, but of theoretical concern is the relative independence of the two interfaces. One can track two separate logical environments that pertain to the same input buffer. For example, defining a function in the *read-eval-print* interface does not cause it to be defined in the theorem prover.

6.2 PG/Eclipse

The Eclipse version of Proof General has some nice features [13,11]. The project is ahead of ACL2s in terms of utilizing the graphical interface for browsing help,

documentation, and the logical world. As we have overcome the technical challenges of hooking a robust script-management interface to ACL2, someone hooking ACL2 to PG/Eclipse using the *Proof General Interaction Protocol* could utilize our extensions to the ACL2 core.

7 Conclusion

In this paper, we have described ACL2s, the ACL2 sedan. ACL2s is a publicly available system that we have developed in order to make formal reasoning more accessible to the masses. One of our goals is to create a "self-teaching" system that enables undergraduates to learn how to prove theorems about the computing systems they design by "playing" with ACL2s. As an initial step in this direction, ACL2s includes many novel features for streamlining the learning process. This includes a modern graphical integrated development environment in Eclipse that provides an intuitive, robust "script management" interface with an improved front-end to the familiar "command line" interface. It also includes CCG termination analysis, a powerful, state-of-the-art termination analysis method which essentially eliminates the need for students to justify the kind of user-defined recursive functions and induction schemes that would be covered in an undergraduate class. Together, the features of ACL2s lower barriers to learning specification and verification in ACL2—which was the sense after two graduate courses switched to requiring use of the tool.

For future work, we are planning to use ACL2s to teach an undergraduate course on processor design. The goal is that students should learn more about processor design than they would have learned without the use of ACL2s. We plan to accomplish this by having student use ACL2s as a specification language and as an oracle that will be configured with the use of various libraries we are developing to either prove the correctness of the student designs or to provide useful information for finding errors. We also want to extend ACL2s so that it can provide more visualization and query support for proving theorems.

Acknowledgments

We would like to thank Matt Kaufmann for help and guidance in hacking ACL2, Qiang Zhang for bug hunting, and Alexander Spiridonov for providing ACL2s support to students at the University of Texas at Austin.

References

[1] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(2):161–194, 1998.

[2] Robert Stephen Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, second edition, 1997.

[3] Peter C. Dillinger, Panagiotis Manolios, J Strother Moore, and Daron Vroon. ACL2s, the ACL2 sedan. http://www.cc.gatech.edu/~manolios/-acl2s.

[4] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[5] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[6] Matt Kaufmann, Panagiotis Manolios, J Strother Moore, and Daron Vroon. Integrating CCG analysis into ACL2. In *Eighth International Workshop on Termination*, August 2006. Part of FLOC '06.

[7] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM Press, 2001.

[8] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In Thomas Ball and Robert Jones, editors, *Computer-aided Verification (CAV) 2006*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[9] J Strother Moore and Matt Kaufmann. The ACL2 home page. http://-www.cs.utexas.edu/-users/-moore/acl2/.

[10] PLT Scheme. DrScheme, 2003. See URL http://www.drscheme.org/.

[11] Proof General Project. PG/Eclipse, 2006. http://proofgeneral.inf.ed.ac.uk/eclipsewiki.

[12] Dale Vaillancourt, Rex Page, and Matthias Felleisen. ACL2 in DrScheme, 2006. http://www.ccs.neu.edu/home/dalev/acl2-drscheme/.

[13] D. Winterstein, D. Aspinall, and C. Lüth. Proof General / Eclipse: A generic interface for interactive proof. In *User Interfaces for Theorem Provers*. ENTCS, 2005.