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 an expert, 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, and novices tend have a very different experience: they crash and burn. As part of a project to make ACL2 and formal reasoning accessible to the masses, we have developed ACL2s, the ACL2 sedan. ACL2s streamlines the learning process with features not previously available for ACL2. Our 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. The latest version of ACL2s is a significant step in that direction.

1 Background

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." It is a programming language, a first-order mathematical logic based on recursive functions, and an "industrial-strength" automated theorem prover from the Boyer-Moore family [7, 4]. ACL2's power, however, comes with a steep learning curve. This is not an issue of documentation, which includes tutorials, a user's manual, academic papers [7], books [4], and heavily commented, free (GPL) source code. ACL2s, "the ACL2 sedan," is an Eclipse-based development environment for ACL2 that, in many ways, reduces the difficulty of learning specification and verification with ACL2 [3]. ACL2s is freely available on the web [2].

Besides the inherent difficulty of reasoning about a system that reasons about other systems, new ACL2 users would stumble in a few identifiable ways before the introduction of ACL2s [3]. Some first difficulties involved the GNU Emacs interface to ACL2. It takes a lot of learning to do interesting things in Emacs, and its connection to ACL2 is very rudimentary. For example, Emacs doesn't know whether a command sent to ACL2 passed or failed, whether it is expecting command input or some other kind of input, or even whether it is waiting for more input at all. With such limitations, it takes some cognitive effort for new users to avoid unintentional, "silly" interaction, which might not be easy to recover from. In Section 2, we discuss how ACL2s addresses these issues with its combination of *script management* and *command line* interfaces.

Another difficulty comes from the ACL2 logic, which is grounded in a simple programming language. But to accept functions for logical reasoning, users often need to help ACL2 prove termination. Thus, it can seem like proofs are required just to define what we want to prove things about! In Section 3, we describe a state-of-the-art termination analysis that ACL2s uses to automatically prove termination of almost any (correctly terminating) function a new user would write. Section 3 also describes some other features, before our conclusion in Section 4.

2 User Interface

Script Management *Script management* is a well-known and natural idiom for interacting with an interactive, extensible theorem prover [1]. Basically, the source code (or "proof script") editor tracks two "lines": the *completed* line and the *todo* line. Since the *completed* line is never beyond the *todo* line, these induce three (potentially empty) regions in this order: the *completed* region, which contains everything that has been accepted by the theorem prover; the *todo* region, which has everything currently being or scheduled to be processed; and the *working* region, which is for editing at will. In ACL2s, the *completed* region is read-only with gray highlight, the *todo* region is read-only with green highlight, and the editable *working* region has no highlight.

The user is granted free manipulation of the *todo* line, specifying how much of his work he wants the theorem prover to process. If the user wants to move the *todo* line above the current position of the *completed* line, then the *completed* line must move also, which is effected in the theorem prover by *undo*ing the forms no longer in the *completed* region. ACL2s also supports instantaneous *redo*ing of a sequence of *undo*ne forms, provided the abstract syntax of the input matches what was previously *undo*ne. The issues of line manipulation in a script management interface are too numerous to describe here, but ACL2s addresses them elegantly and robustly.

"Undo" Problem The reason ACL2s is the first and only effective script management interface for ACL2 is that there's no simple strategy for undoing the effects of arbitrary input forms submitted to ACL2. After all, the input could itself be one of several "undo" commands! A naive implementation would make it easily possible to get one's script into a state in which the completed region would not be accepted by a fresh session, breaking a key invariant of any good script management interface: reprocessing the completed region in a fresh session should result in the same observable theorem prover state.

ACL2s solves the "undo" problem by extending ACL2 with its own more powerful layer for remembering and recovering previous states. To be efficient, the states are incomplete, but they are able to undo the effects of a large, identifiable subset of possible ACL2 operations. Rather than restricting the user's freedom, ACL2s prints a warning when it *undo*es something that may have had effects it can't *undo*. Restarting a session to reprocess the completed region takes just one click.

With a Command Line The script management interface in the source editor is not enough when a user wants to know *why* ACL2 rejects a particular input form. The session editor provides ACL2's textual output in such cases, along with the entire input/output history for that session. Input can come from the *todo* region of the script management interface or it can be typed directly into the bottom of the session editor, making it a command-line interface to the same session.

The motivation for retaining a command line interface is that some ACL2 input, such as queries or stateless function calls, does not make sense from a script management interface. Likewise, "stateful" input belong in the script management interface, so that it can become part of the *completed* region. Instead of restricting the kinds of input that can be used from the two interfaces, ACL2s copies input submitted at the command line to the completed region if and only if it is "stateful" input.

3 Language Extensions

CCG Termination Analysis ACL2 includes an advanced termination analysis based on "Context Calling Graphs" (or CCGs) [6, 5], which significantly enhances ACL2's ability to recognize terminating function definitions with no user guidance. For example, against the ACL2 regression suite, which covers topics from set theory to processor verification, CCG analysis automatically proves 98.7% of the 10,000 functions terminating, including 68.2% of those that previously required explicit user hints.

Session Modes Analogous to "language levels" in DrScheme [8], ACL2s offers several "session modes" that configure ACL2 for development at a certain depth of understanding. This begins with **Programming mode**, which ignores anything related to proofs, soundness, or efficient implementation. **Recursion & Induction mode** adds proof capability without much concern for building coherent proof rules. **ACL2s mode** is just like ACL2 but has CCG termination analysis. **Compatible mode** retains compatibility with with official ACL2 releases.

4 Conclusion

ACL2s has put a modern, intuitive face on ACL2 development. In fact, we have required use of ACL2s in two graduate courses that previously recommended ACL2 with Emacs, and the sense has been that ACL2s lowers barriers to learning specification and verification in ACL2. After a short demonstration, students are able to go home, download the tool, go through our tutorial, and then focus on ACL2 the programming language, ACL2 the logic, and ACL2 the extensible theorem prover.

We plan to continue improving ACL2s so that it can be even more "intuitive" and "self-teaching", and hope to use it in an undergraduate environment soon.

References

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

[2] P. C. Dillinger and P. Manolios. ACL2s home page. http://www.cc.gatech.edu/∼manolios/acl2s/.

[3] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore. ACL2s: The ACL2 Sedan. In User Interfaces for Theorem Provers. ENTCS, 2006.

[4] M. Kaufmann, P. Manolios, and J. S. Moore. Computer-Aided Reasoning: An Approach. Kluwer Academic Pub., July 2000.

[5] M. Kaufmann, P. Manolios, J. S. Moore, and D. Vroon. Integrating CCG analysis into ACL2. In Eighth International Workshop on Termination, August 2006. Part of FLOC '06.

[6] P. Manolios and D. Vroon. Termination analysis with calling context graphs. In Computer-aided Verification (CAV) 2006, LNCS. Springer-Verlag, 2006.

[7] J. S. Moore and M. Kaufmann. ACL2 home page. http://www.cs.utexas.edu/users/moore/acl2/.

[8] PLT of Northeastern University. DrScheme, 2006. See URL http://www.drscheme.org/.