The ACL2 Sedan Theorem Proving System

Harsh Raju Chamarthi, Peter Dillinger, Panagiotis Manolios, and Daron Vroon

College of Computer and Information Science
Northeastern University
360 Huntington Ave., Boston MA 02115, USA
{harshrc,pcd,pete}@ccs.neu.edu, daron.vroon@gmail.com

Abstract. The ACL2 Sedan theorem prover (ACL2s) is an Eclipse plug-in that provides a modern integrated development environment, supports several modes of interaction, provides a powerful termination analysis engine, and includes fully automatic bug-finding methods based on a synergistic combination of theorem proving and random testing. ACL2s is publicly available and open source. It has also been used in several sections of a required freshman course at Northeastern University to teach over 200 undergraduate students how to reason about programs.

1 Introduction

ACL2 is a powerful system for integrated modeling, simulation, and theorem proving [5, 4, 6]. Think of ACL2 as a finely-tuned racecar. In the hands of experts, it has been used to prove some of the most the complex theorems ever proved about commercially designed systems. Novices, however, tend to have a different experience: they crash and burn. Our motivation in developing ACL2s, the ACL2 Sedan, was to bring computer-aided reasoning to the masses by developing a user-friendly system that retained the power of ACL2, but made it possible for new users to quickly, easily learn how to develop and reason about programs.

Usability is one of the major factors contributing to ACL2's steep learning curve. To address the usability problem, ACL2s provides a modern graphical integrated development environment. It is an Eclipse plug-in that includes syntax highlighting, character pair matching, input command demarcation and classification, automatic indentation, auto-completion, a powerful undo facility, various script management capabilities, a clickable proof-tree viewer, clickable icons and keybindings for common actions, tracing support, support for graphics development, and a collection of session modes ranging from beginner modes to advanced user modes. ACL2s also provides GUI support for the "method," an approach to developing programs and theorems advocated in the ACL2 book [5]. Most of these features have been described previously, so we will not dwell on them any further [3].

The other major challenge new users are confronted with is formal reasoning. A major advantage of ACL2 is that it is based on a simple applicative programming language, which is easy to teach. What students find more challenging is the ACL2 logic. The first issue they confront is that functions must be shown to terminate. Termination is used to both guarantee soundness and to introduce induction schemes. We have developed and implemented Calling-Context Graph

termination analysis (CCG), which is able to automatically prove termination of the kinds of functions arising in undergraduate classes [7]. However, beginners often define non-terminating functions. A new feature of ACL2s is that it provides support for the interactive use of CCG analysis. In particular, we provide termination counterexamples and a powerful interface for users to direct the CCG analysis. This is described in Section 2.

Once their function definitions are admitted, new users next learn how to reason about such functions, which first requires learning how to specify properties. We have seen that beginners often make specification errors. ACL2s provides a new lightweight and fully automatic synergistic integration of testing and theorem proving that often generates counterexamples to false conjectures. The counterexamples allow users to quickly fix specification errors and to learn the valuable skill of generating correct specifications. This works well pedagogically because students know how to program, so they understand evaluation. Invalidating a conjecture simply involves finding inputs for which their conjecture evaluates to false. This is similar to the unit testing they do when they develop programs, except that it is automated. An overview of our synergistic integration of testing and theorem proving is given in Section 3.

ACL2s has been successfully used to teach novices. We have used ACL2s at Northeastern University to teach eight sections of a required second-semester freshman course entitled "Logic and Computation." The goal of the class is to teach fundamental techniques for describing and reasoning about computation. Students learn that they can gain predictive power over the programs they write by using logic and automated theorem proving. They learn to use ACL2s to model systems, to specify correctness, to validate their designs using lightweight methods, and to ultimately prove theorems that are mechanically checked. For example, students reason about data structures, circuits, and algorithms; they prove that a simple compiler is correct; they prove equivalence between various programs; they show that library routines are observationally equivalent; and they develop and reason about video games.

ACL2s is freely available, open-source, and well supported [1]. Installation is simple, *e.g.*, we provide prepackaged images for Mac, Linux, and Windows platforms. In addition, everything described in this paper is implemented and available in the current version of ACL2s.

2 Termination Analysis using Calling Context Graphs

Consider the function definitions in Figure 1, where `zp` is false iff its argument is a positive integer and `expt` is exponentiation.

This program was generated by applying weakest precondition analysis to a triply-nested loop. An expert with over a decade of theorem proving experience spent 4–6 hours attempting to construct a measure function that could be used to prove termination, before giving up. Readers are encouraged to construct a measure and to mechanically verify it. (It took us about 20 minutes.) We also tried our CCG termination analysis, as implemented in ACL2s: it proved termination in under 2 seconds, fully automatically with no user guidance.

We have found that if beginners write a terminating function, our CCG analysis will almost certainly prove termination automatically. Unfortunately,

```
(defun f1 (w r z s x y a b zs)
  (if (not (zp z))
      (f2 w r z 0 r w 0 0 zs)
    (= w (expt r zs))))

(defun f2 (w r z s x y a b zs)
  (if (not (zp x))
      (f3 w r z s x y y s zs)
    (f1 s r (- z 1) 0 0 0 0 0 zs)))

(defun f3 (w r z s x y a b zs)
  (if (not (zp a))
      (f3 w r z s x y (- a 1) (+ b 1) zs)
    (f2 w r z b (- x 1) y 0 0 zs)))
```

Fig. 1. An interesting termination problem.

beginners often write non-terminating programs, and then want to know why termination analysis failed. This lead us to develop an algorithm that generates a simplified version of the user's program that highlights the reason for the failure. We call the simplified program that is generated a *termination core*, and it corresponds to a single simple cycle of the original program which the CCG analysis was unable to prove terminating.

The termination core can be seen as an explanation of why the CCG analysis failed. After examining the termination core, a user has three options. First, the user can change the function definition. This is the logical course of action if the loop returned by CCG reveals that the program as defined is really not terminating. Second, the user can guide the CCG analysis by providing hints that tell the CCG analysis what local measures to consider. We provide a hint mechanism for doing this. The user can provide either the `CONSIDER` hint or the `CONSIDER-ONLY` hint. The former tells CCG to add the user-provided local measures to the local measures it heuristically guesses, while the latter tells CCG to use only the user-provided local measures. This is an effective means of guiding CCG if its heuristics fail to guess the appropriate local measures, and is much simpler than the previous alternative which was to construct a global measure. Finally, it may be the case that the CCG analysis guessed the appropriate local measures but was unable to prove the necessary theorems to show that those measures decrease from one step to the next. In this case, the user can prove the appropriate lemmas.

The result of integrating CCG with ACL2s is a highly automated, intuitive, and interactive termination analysis that eases the steep learning curve for new users of ACL2 and streamlines the ACL2 development process for expert users. The ACL2 Sedan includes extensive documentation of CCG analysis.

3 Random Testing and Proving: Synergistic Combination

Users of ACL2 spend much of their time and effort steering the theorem prover towards proofs of conjectures. During this process users invariably consider conjectures that are in fact false. Often, it is difficult even for experts to determine whether the theorem prover failed because the conjecture is not true or because the theorem prover needs further user guidance.

ACL2s provides a lightweight method based on the synergistic combination of random testing [2] and theorem proving, for debugging and understanding

conjectures. This has turned out to be invaluable in helping beginners become effective users of formal methods. We have integrated random testing into ACL2s in a deep way: it is enabled by default and requires no special syntax so that users get the benefit of random testing without any effort on their part.

Since ACL2 formulas are executable, random testing in ACL2 involves randomly instantiating the free variables in a formula and then evaluating the result. This is a small part of the picture because this naive approach is unlikely to find counterexamples in all but the simplest of cases. This is especially true in a theorem prover for an untyped logic, like ACL2, where every variable can take on any value. As might be expected, conjectures typically contain hypotheses that constrain variables. Therefore, we randomly instantiate variables subject to these constraints. We do this by introducing a flexible and powerful data definition framework in ACL2s which provides support for defining union types, product types, list types, record types, enumeration types, and mutually-recursive data definitions. It allows the use of macros inside definitions and supports custom data definitions (*e.g.*, primes). The data definition framework is integrated with our random testing framework in several important ways. For example, we guarantee that random testing will automatically generate examples that satisfy any hypothesis restricting the type of a variable.

Complex conjectures often involve many variables with many hypotheses and intricate propositional structure involving complex hierarchies of user-defined functions. Testing such conjectures directly is unlikely to yield counterexamples. We address this by integrating our testing framework with the core theorem proving engine in a synergistic fashion, using the full power of ACL2 to simplify conjectures for better testing. The main idea is to let ACL2 use all of the proof techniques at its disposal to simplify conjectures into subgoals, and to then test the "interesting" subgoals. This winds up requiring lots of care. For example, ACL2 employs proof techniques that can generate radically transformed subgoals, where variables disappear or are replaced with new variables that are related to the original variables via certain constraints. Finally, our analysis is *sound*, *i.e.*, any counterexamples generated truly are counterexamples to the original conjecture.

References

1. Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. ACL2 Sedan homepage. See URL `http://acl2s.ccs.neu.edu/`.
2. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.
3. Peter C. Dillinger, Panagiotis Manolios, Daron Vroon, and J. Strother Moore. ACL2s: "The ACL2 Sedan". *Electr. Notes Theor. Comput. Sci.*, 174(2):3–18, 2007.
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 and J Strother Moore. ACL2 homepage. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.
7. Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *Computer Aided Verification, CAV*, volume 4144 of *LNCS*, pages 401–414. Springer, 2006.