DoubleCheck Your Theorems

Carl Eastlund
Northeastern University
Boston, MA, USA
cce@ccs.neu.edu

ABSTRACT

Theorem proving in ACL2 is a complex undertaking. Initial attempts to admit a lemma often fail, in which case the programmer must either redirect ACL2's efforts or change the lemma. ACL2's output does not always indicate whether the formulation of the lemma or the proof process is at fault.

In this paper we present the automated testing framework DoubleCheck as an extension of Dracula, the ACL2 development environment for DrScheme. DoubleCheck creates randomized inputs for ACL2 conjectures and uses those to test the conjecture. If these tests fail, the programmer is presented with a list of counterexamples to the conjecture. DoubleCheck can be used to guide the theorem proving process or, in a classroom setting, as a gentle introduction to automated program verification.

Categories and Subject Descriptors

D [**2**]: 4—*Software/Program Verification*; D [**2**]: 5—*Testing and Debugging*; F [**3**]: 1—*Specifying and Verifying and Reasoning about Programs*

General Terms

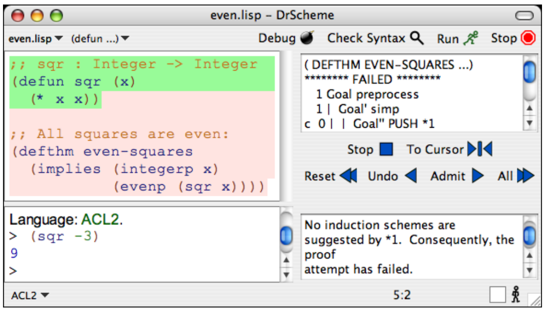
Reliability, Verification

Keywords

ACL2, Dracula, QuickCheck, Randomized testing

1. SEEING IS DISBELIEVING

Program verification in ACL2 is an iterative process. Kaufmann, Manolios, and Moore [4] present "The Method" for tackling a large proof in four steps. First, write down a to-do list of lemmas. Second, submit the first lemma to the theorem prover. Third, if the theorem prover verifies the lemma, remove it from the list and repeat. Fourth, if the theorem prover fails, adjust the lemmas in the list to try to correct the lemma that failed, and repeat.



```
;; sqr : Integer -> Integer
(defun sqr (x)
  (* x x))

;; All squares are even:
(defthm even-squares
  (implies (integerp x)
           (evenp (sqr x))))
```

```
Language: ACL2.
> (sqr -3)
9
>
```

```
( DEFTHM EVEN-SQUARES ...)
******** FAILED ********
  1 Goal preprocess
  1 |  Goal' simp
c  0 |  |  Goal'' PUSH *1
```

```
No induction schemes are
suggested by *1.  Consequently, the
proof
attempt has failed.
```

Figure 1: The Dracula graphical user interface.

Step four is the trickiest part of using ACL2: when the theorem prover does not arrive at a proof as expected, does the problem lie in the soundness of the lemma or in the proof search heuristics? The programmer must choose to debug one fault or the other, but the wrong choice might waste hours or days attempting to prove an unsound lemma or find nonexistent bugs in the program or the conjecture.

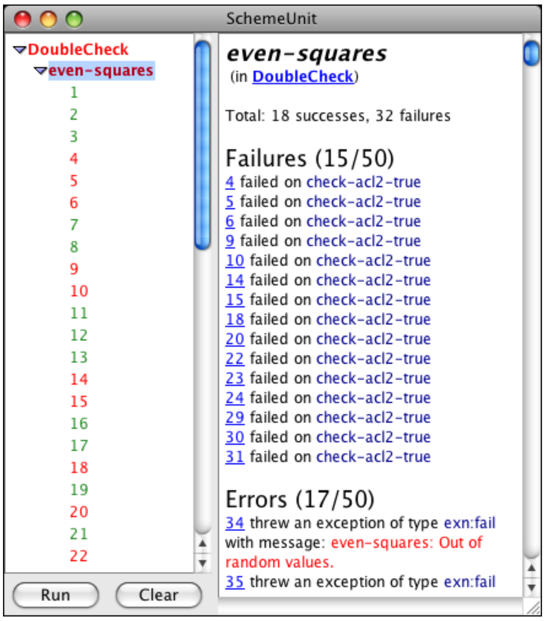
This problem is exacerbated in a classroom setting. Students are often unfamiliar with formal reasoning and discouraged by the task of deciphering the theorem prover's output. In 2007, Eastlund et al. experimented with teaching ACL2 to freshmen [3]. Their report demonstrated a need for quick diagnosis of a failed proof attempt.

The course used Dracula [9], the ACL2 programming environment for DrScheme. Figure 1 shows its user interface. The program is on the top left, with output from interactive evaluation below. Controls for ACL2 are on the right, with a proof tree above and detailed output below.

Figure 1 shows a simple program with a flawed conjecture: the square of any integer is even. The theorem prover rejects this claim with the message, "No induction schemes are suggested." This describes where ACL2's proof search went wrong, but is equivocal about the lemma. Students (and others) need concrete suggestions of how to proceed.

Libraries such as QuickCheck [2] provide a way for programmers to find concrete counterexamples to properties expressed as predicate functions. QuickCheck's testing framework generates random inputs for predicates and presents a report to the programmer of how many trials were run, how many failed, and the specific inputs for which each property failed (if any).

We present DoubleCheck, an adaptation of QuickCheck for ACL2, as an extension of the Dracula programming environment. DoubleCheck allows ACL2 programmers



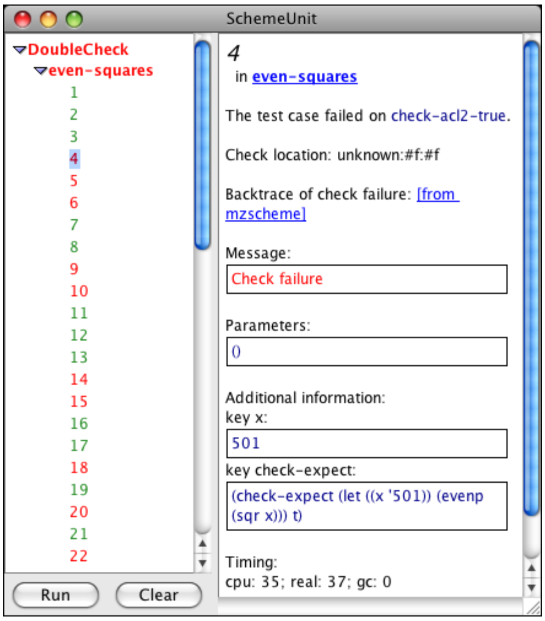
even-squares
(in DoubleCheck)

Total: 18 successes, 32 failures

Failures (15/50)
4 failed on check-acl2-true
5 failed on check-acl2-true
6 failed on check-acl2-true
9 failed on check-acl2-true
10 failed on check-acl2-true
14 failed on check-acl2-true
15 failed on check-acl2-true
18 failed on check-acl2-true
20 failed on check-acl2-true
22 failed on check-acl2-true
23 failed on check-acl2-true
24 failed on check-acl2-true
29 failed on check-acl2-true
30 failed on check-acl2-true
31 failed on check-acl2-true

Errors (17/50)
34 threw an exception of type exn:fail with message: even-squares: Out of random values.
35 threw an exception of type exn:fail

Figure 2: Summary of a property's trials.



4
in even-squares

The test case failed on check-acl2-true.

Check location: unknown:#f:#f

Backtrace of check failure: [from mzscheme]

Message:
Check failure

Parameters:
()

Additional information:
key x:
501
key check-expect:
(check-expect (let ((x '501)) (evenp (sqr x))) t)

Timing:
cpu: 35; real: 37; gc: 0

Figure 3: Data gathered from a single trial.

to quickly check whether a lemma is sound or unsound, generating a high degree of confidence in the former case and concrete counterexamples in the latter.

DoubleCheck may be used as a supplement to "The Method", aiding during step four in determining how to change the to-do list of lemmas when a proof attempt fails. It can also be used as a lightweight form of software verification. This is especially compelling in a classroom setting, where DoubleCheck presents the benefits of automatic validation without the burden of a full logical proof.

2. CHECK AND DOUBLECHECK

DoubleCheck is provided as a "teachpack" in Dracula. A teachpack is similar to an ACL2 book, but with additional interactive behavior (such as graphics) when run in Dracula.

To use DoubleCheck, a program must include it as a book:

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

The book provides facilities for declaring randomly-tested properties, distributions for generating random data, and tools to construct new distributions.

Properties are declared with the **defproperty** form. In the simplest case, the programmer translates a **defthm** conjecture to an equivalent property. For example, the conjecture *even-squares* from figure 1 can be rewritten as the following property:

```
(defproperty even-squares
  (x :where (integerp x))
  (evenp (sqr x)))
```

This translation expresses the same conclusion from the same premise. The key difference is that **defproperty** requires x and its precondition to be declared explicitly.

DoubleCheck records properties until the programmer gives the directive to check them:

```
(check-properties)
```

At this point, the body of each property is run repeatedly on random assignments to its free variables. Dracula displays a graphical summary of the results to the user.

Figure 2 shows graphical output summarizing 50 trials of the *even-squares* property above. The left side of the display lists the trials in red if they fail and green if they succeed. On the right, DoubleCheck displays the property's name, success and failure counts, and a summary of exceptional cases. First it lists failures—in this case, 15 trials failed to satisfy the property; DoubleCheck provides links to more detailed information. Lastly it lists errors that prevented running a trial. In 17 trials, DoubleCheck "ran out" of random values (it failed to produce values satisfying the precondition in an allotted number of attempts).

To see where the property went wrong, a user can inspect individual test cases. Selecting trial 4 (the first failure) changes the display as shown in figure 3. Details of the trial are displayed on the right. Listed under "Additional information", we see the "key" x was 501. This is a concrete counterexample: (*sqr* x) does not satisfy **evenp** when x is 501. Below that information, the key **check-expect** provides an expression that may be copied into the program as a repeatable (deterministic) regression test using **check-expect** [5] (similar to **assert-event**).

Like the ACL2 theorem prover, DoubleCheck sometimes needs programmer assistance. The **defproperty** form has several options for refining a counterexample search. Its full grammar (with brackets denoting optional keywords) is:

```
(defproperty name [:repeat count] [:limit max]
  (var [:where pred] [:value generator] [:limit max] ...)
  expression)
```

The :repeat option sets the number of trials DoubleCheck runs. The :limit option sets the maximum number of attempts DoubleCheck makes to generate random values that satisfy their preconditions. By default, every property is given 50 trials with a limit of 2500 attempts to construct random inputs.

Each free variable of the property may optionally have :where, :value, or :limit keywords. The :where clause sets a precondition; DoubleCheck discards random inputs that do not satisfy it. The :value clause provides a generator expression, selecting a custom random distribution for the variable. The :limit clause overrides the property's :limit number for a single variable. By default, a variable has no precondition, a distribution including all types of ACL2 data, and uses the property's generation limit.

Everything in Dracula has two meanings: its executable behavior (including any interactive content not present in ACL2) and its logical semantics. DoubleCheck is no exception. When passed to the theorem prover, each **defproperty** is interpreted as a logically equivalent **defthm**. The variable declarations and random distributions are stripped off, the hypotheses are joined to the conclusion with **implies**, and hints are retained.

Choosing the right distribution of random inputs is critical for effective random testing. DoubleCheck includes a library of random data generators. These generators are designed to produce a wide range of commonly-used value sets with minimal effort, though at present their distributions are not necessarily exhaustive or finely tuned. We expect to improve these distributions in future releases based on our own experience and feedback from users. In the meantime, we provide facilities for defining custom distributions.

The following generators produce atomic data:

```
(random-boolean)  (random-natural)
(random-symbol)   (random-integer)
(random-char)     (random-rational)
(random-string)   (random-number)
```

The randomly chosen characters, symbols, and strings are based on a uniform choice of lower-case letters. Natural numbers are chosen from a geometric distribution with an average of 1,000; integers, rationals, and complex numbers are constructed from random natural numbers and signs.

The next two generators provide integer distributions:

(**random-data-size**)   (**random-between** *lo hi*)

The sizes of random data structures (strings, symbol names, lists, and s-expressions) are taken from **random-data-size**. Initial attempts to base their size on **random-natural** (mean 1,000) produced intractably large values, so we added **random-data-size** (mean 4).

The **random-between** generator provides a uniform distribution over integers between *lo* and *hi*, inclusive.

These generators combine the previous ones to make distributions of generic atoms and s-expressions:

(**random-atom**)   (**random-sexp**)

The **random-atom** function chooses among booleans, symbols, characters, strings, and numbers. The **random-sexp** function builds a **cons**-tree of a random size (based on **random-data-size**), with a random atom at each leaf.

The **random-element-of** generator makes a uniform choice among the elements of a non-empty list:

(**random-element-of** *lst*)

Parametric distributions of lists and s-expressions are created by the **random-list-of** and **random-sexp-of** macros, with members and leaves (respectively) generated by the given sub-expression:

(**random-list-of** *expr* [:size *size*])
(**random-sexp-of** *expr* [:size *size*])

Each takes an optional *size* argument which defaults to (**random-data-size**). They evaluate *expr* once for each list or s-expression element; for instance, (**random-list-of** (**random-symbol**)) might generate '(a b c), and (**random-sexp-of** (**random-symbol**)) might generate '(a . (b . c)).

Finally, the **random-case** macro provides nondeterministic choice among multiple expressions:

(**random-case** *expr* [:weight *weight*] . . .)

Each expression has an associated weight, defaulting to 1.

Programmers can write new generators:

(**defrandom** *name* (*arg* . . .) *body*)

This defines functions like **defun**, but the function may refer to generators, and is treated as a generator itself.

For example, the following generator constructs a random multiset with elements from a provided list:

```
(defrandom random-multiset (elements)
  (random-case
    nil :weight 1/4
    (cons (random-element-of elements)
          (random-multiset elements))))
```

Our formulation of randomization violates the axioms of ACL2; for instance, two identical calls to a random function may generate different values.[1] To preserve the soundness of ACL2, generators may only be used in the :value clause of **defproperty** or in the definition of other generators.

3. BEHIND THE CURTAIN

Dracula presents a two-part interface to ACL2: for execution, programs are compiled to PLT Scheme code that simulates ACL2's runtime behavior; for verification, programs are marshalled and submitted to the (unmodified) ACL2 theorem prover. In keeping with this duality, every teachpack in Dracula has two parts to its implementation. One defines its interactive behavior for Dracula's simulation; the other is an ACL2 book describing its logical representation.

The runtime implementation of DoubleCheck uses the SchemeUnit [10] testing framework as a front-end. DoubleCheck constructs the code that generates random inputs, runs each trial, and annotates failed cases with relevant data. SchemeUnit provides tools to express hierarchical test suites, collect annotations, and display a graphical summary.

The **defproperty** macro constructs a new SchemeUnit test suite containing one test case for each trial to be run. DoubleCheck constructs a lazy stream of random inputs for the property based on its random generators, preconditions, and repetition limits. The stream generates new values from the generators when requested, skips (but records) any that do not satisfy their precondition, and stops when the repetition limit is reached.

Each trial of a property reads its inputs from the associated stream. If the stream is empty, the test case reports an error and describes previously rejected values so the programmer may adjust the random generators. Otherwise,

[1] We experimented with sound formulations of randomness using explicit seed arguments, but found they needlessly complicated the interface.

if the property fails for a set of (valid) inputs, DoubleCheck reports a failure to SchemeUnit annotated with the choice of inputs.

DoubleCheck collects the test suites generated by **defproperty** until the program calls **check-properties**. At this point, the individual properties are collected in one master test suite and run. SchemeUnit opens a window with a graphical display of the test suite and runs each trial in turn. As each trial starts, DoubleCheck's lazy stream generates its input; when it finishes, SchemeUnit updates the display with the new results.

SchemeUnit provides a convenient framework for exploring the results of DoubleCheck properties, but also presents some design constraints. SchemeUnit tests have a static layout: the entire test hierarchy must be constructed before any tests are performed. With a dynamic framework, DoubleCheck could add random trials as their inputs are generated and give them informative names based on their input, so the user could see counterexamples on the top-level summary page. This would also obviate the need for **check-properties**, as DoubleCheck could add each property to the test suite as it is declared. Finally, SchemeUnit does not collect any data from successful test cases, so DoubleCheck cannot help the user compare inputs from failing cases to those from successful cases.

Generator functions ensure that they are not used in a logical context by performing a dynamic check each time they are run. The DoubleCheck random input stream sets a flag during its execution; if a generator runs with the flag down, it raises an exception.

The ACL2 book representing DoubleCheck's logical implementation translates each property to a logical conjecture. Random generators and **check-properties** have trivial, program-mode definitions.

The **defproperty** macro assembles a **defthm** from the source property's hypotheses and body. For example,

```
(defproperty plus-comm :repeat 100 :limit 1000
  (x :where (integerp x) :value (random-integer)
   y :where (integerp y) :limit 200)
  (= (+ x y) (+ y x))
  :hints (("Goal" :in-theory (theory 'minimal-theory))))
```

expands into this ACL2 code:

```
(defthm plus-comm
  (implies (and (integerp x) (integerp y))
           (= (+ x y) (+ y x)))
  :hints (("Goal" :in-theory (theory 'minimal-theory))))
```

This definition expresses the logical premises and conclusion of the source property, but lacks the (logically irrelevant) randomized testing annotations.

4. PAST WORK, FUTURE DIRECTIONS

There are many other systems in existence for automated testing and counterexample generation. The example that initially inspired DoubleCheck was QuickCheck [2], the randomized testing library for Haskell.

Like DoubleCheck, QuickCheck uses randomized inputs to test predicate-based properties in an applicative language. Unlike DoubleCheck, QuickCheck uses typeclasses to build random distributions, so that input generation is type-directed. This means that default distributions in QuickCheck are more precise than in DoubleCheck and always match the type portion of a precondition.

The initial QuickCheck implementation had some statistical features, such as marking "trivial" trials, that DoubleCheck has not yet incorporated. QuickCheck has since developed many new features, such as failure shrinking—which starts at a failing input and tries to derive the smallest related input that fails—that should be evaluated for inclusion with DoubleCheck.

SmallCheck [6] generates test cases for properties exhaustively, rather than randomly, testing all the "small" inputs for a given type. The hypothesis behind SmallCheck is that most errors can be found with a small input size. The simple nature of ACL2's value set makes SmallCheck-style testing a natural tool to adopt in addition to random distributions.

There is a precedent for applying QuickCheck-style randomized testing to generate counterexamples for automated theorem proving, specifically for the Isabelle/HOL theorem prover [1]. Random distributions are created automatically for each datatype via polytypic programming, rather than being customized by the programmer.

The ACL2 community has produced other tools for automatic counterexample generation; for instance, by using SAT solving [8] and model checking [7] to generate counterexamples for ACL2 conjectures. These tools automatically inspect the property under consideration to direct their search; DoubleCheck users must perform this task manually by choosing appropriate random distributions.

DoubleCheck constitutes the first randomized testing toolkit for ACL2. It provides a lightweight complement to formal verification. Random testing is also beneficial for students learning to use ACL2. As Rex Page stated in a recent educational report [5], "DoubleCheck helps students with one of the trickiest task of using logic in programming, namely, the transition from ideas to formal statements." In other words, DoubleCheck provides automatic validation without the burden of constructing a logical proof.

DoubleCheck is still under development. The simple but frustrating task of translating conjectures to properties by declaring free variables and separating hypotheses might be alleviated by better choice of syntax or incorporation with **defthm** itself. Similarly, the arbitrary nature of the built-in random distributions and the trade-offs of SchemeUnit as a front-end may be revisited in future releases.

Acknowledgements.

Thanks to my adviser, Matthias Felleisen, for his inspiration and support in developing DoubleCheck; to Rex Page for putting the tool to good use and providing insightful feedback; and to Carter Schonwald for the prototype implementation. This paper is brought to you by the letters ACL, the number 2, and a research grant from the NSF.

5. REFERENCES

[1] Berghofer, S. and T. Nipkow. Random testing in Isabelle/HOL. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods*, p. 230–239. IEEE, 2004.

[2] Claessen, K. and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, p. 268–279. ACM, 2000.

[3] Eastlund, C., D. Vaillancourt and M. Felleisen. ACL2 for freshmen: first experiences. In *Proceedings of the 7th International Workshop on the ACL2 Theorem Prover and its Applications*, p. 200–211. ACM, 2007.

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

[5] Page, R., C. Eastlund and M. Felleisen. Functional programming and theorem proving for undergraduates: a progress report. In *Proceedings of the 13th Workshop on Functional and Declarative Programming in Education*. ACM, 2008.

[6] Runciman, C., M. Naylor and F. Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, p. 37–48. ACM, 2008.

[7] Spiridinov, A. and S. Khurshid. Pythia: Automatic generation of counterexamples for ACL2 using Alloy. In *Proceedings of the 7th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2007.

[8] Sumners, R. Checking ACL2 theorems via SAT checking. In *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and Its Applications*, 2002.

[9] Vaillancourt, D., R. Page and M. Felleisen. ACL2 in DrScheme. In *Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and its Applications*, p. 107–116. ACM, 2006.

[10] Welsh, N., F. Solsona and I. Glover. SchemeUnit and SchemeQL. In *Proceedings of the 3rd Workshop on Scheme and Functional Programming*, Technical Report GIT-CC-02-28, College of Computing, Georgia Institute of Technology, p. 21–30. ACM, 2002.