Random Testing in PVS

Sam Owre

SRI International, Computer Science Laboratory
333 Ravenswood Avenue, Menlo Park, CA 94025, USA
owre@csl.sri.com

Abstract. Formulas are difficult to formulate and to prove, and are often invalid during specification development. Testing formulas prior to attempting any proofs could potentially save a lot of effort. Here we describe an implementation of random testing in the PVS verification system.

1 Introduction

The PVS system has been used extensively for many verifications, both large and small. For true theorems there are many techniques available, including resolution, powerful decision procedures, and automatic rewriting. But interactive proving is often difficult and time consuming; and often during development one is uncertain if the formulas are actually true. More often than not the theorem being attempted is actually false. PVS has an interactive theorem prover, which is some use in finding where a theorem is false, as one can explore the proof tree in as much detail as desired. But this can be very time-consuming, and it is often difficult to relate the context of the given proof state to the original specification. What is needed is an easy way to test specifications in order to find trivial bugs before attempting any proofs.

To aid in such situations, we have recently added random testing to PVS. The work describe here was based on the work done in similar systems. Claessen and Hughes [CH00] developed the QuickCheck tool for random testing of Haskell programs, and used it on a specification of unification and the Lava embedded circuit description language. Berghofer and Nipkow [BN04] applied these techniques to Isabelle/HOL, including extensions to inductive datatypes and inductive predicates; they then applied it to a simple operational semantics and a specification of Red-Black trees. Dybjer, Haiyan,

and Takeyama [DHT03] implemented random testing in Agda/Alfa, where it is combined with theorem proving for dependent types; they used it to verify a BDD implementation.

In the rest of this paper we give a brief overview of PVS, describe the random test generator, and give some examples of its use. We conclude with some plans for the future.

2 A Brief Description of PVS

PVS (Prototype Verification System) is based on classical higher-order logic, with a rich type system including base types (boolean, integer, real, etc.), functions, tuples, records, cotuples, and recursive datatypes. It also allows subtypes derived from predicates, which means that typechecking may be undecidable. The typechecker does not attempt to prove everything, but outputs proof obligations in the form of *type correctness conditions* (TCCs).

The PVS system includes a number of components to aid development, including an Emacs-based user interface, parser, pretty-printer, typechecker, interactive theorem prover, model checker, ground evaluator, abstractor, and HTML generator. PVS is implemented in Common Lisp.

The components of primary interest for this discussion are the theorem prover and ground evaluator. The interactive theorem prover is based on the sequent calculus, the goal is for the user to construct a tree of sequents in which each leaf is true. A sequent consists of a set of antecedents (hypotheses) and a set of consequents (conclusions); the meaning is that the conjunction of antecedents implies the disjunction of consequents. Initially the proof tree consists of a single sequent with a single consequent which is the lemma to be proved.

The user guides the proof by issuing proof commands. In general a proof command, if it succeeds, adds one or more children to the current node of the proof tree, and makes one of the child leaves the new current goal. When a branch is proved, control moves to a new sibling of the current node, until there are no more unproved leaves.

The ground evaluator generates efficient Lisp code from a subset of PVS [Sha99]. It includes extensive analysis of array, record, and tuple updates to ensure that, where safe, destructive updates may

be used in place of copying. Crow et al. [COR+01] describe the use semantic attachments to compute values, create side effects, and animate specifications. This has been extensively augmented by Cesar Munoz in PVSio [Muñ03], in which PVS becomes almost programming environment, and can even be used as a scripting language.

3 Random Test Generation

The heart of the random test generator is a set of methods that create random test generators for each type class in PVS. For each type of PVS, a random test generator is created to generate values; each invocation of the random test generator will produce a random value of the associated type. The random test values are used to instantiate a given formula, which is then evaluated by the ground evaluator. If it is found to be false, the test values are reported.

For base types, values are generated using the Common Lisp `random` function. Booleans, enumerated types, etc. may be handled directly in this way. For unbounded base types such as integers, a bound may be given; the default is 100. Rationals are generated by generating two integers x and y; when $y/=0$ then x/y is returned.[1] Rationals are also used to generate random reals.

Random generators for subtypes presents a bit of a problem. Currently the generator generates a value for the supertype, then checks whether the result is in the subtype by evaluating the subtype predicate on that value. If unsuccessful, it continues doing this until a counter is decremented to zero. How well this works in practice depends both on how often the randomly generated value satisfies the predicate, and how computationally expensive the predicate is. An example where this would likely fail is generating a prime number below 10 million. Another example where this would fail is generating a list of integers, where each element is twice as big as the preceding element. To overcome this, some of the subtypes defined in the PVS prelude such as natural numbers, even and odd integers, and subranges, have specialized generators defined in order to quickly generate valid values.

[1] One advantage of doing this in Common Lisp is that there is no issue with overflow, as bignums are seamlessly integrated. Dividing one integer by another yields an exact rational, not an approximate floating point number.

The random generators for tuple and record types just build on the generators for the component types. The only slight difficulty is with dependencies, which are handled by substituting the randomly generated values for earlier types into subsequent dependent types as the components are processed.

Cotuples of the form $T_1 + \ldots + T_n$ are handled by first randomly selecting a number i between 1 and n, then invoking the random generator for that type to create a value v, returning $\texttt{in}_i(v)$.

Function types are handled by creating a closure that memoizes its values. Thus when a randomly generated function is applied, it checks to see if its argument has already been seen, in which case it returns the associated value. Otherwise, it randomly generates a value in the range type, and saves the argument, value pair for future invocations. This allows one to handle higher-order formulas, such as

```
FORALL (f: [int -> int], x, y, z: int):
  f(x) + f(f(y)) < f(f(f(z)))
```

Inductive data types are handled as described in [BN04]. Basically, there is a size parameter used to construct datatype elements whose term constructions are bounded by that size. Thus if the size is 4, lists of length at most 4 could be generated (including `null`), and trees with depth at most 4 would be generated. A List of trees could also be generated, where the list could have at most 4 elements, and each element is a tree of depth at most 4.

4 Using the Random Test Generator

The random test generator may be used as part of ground evaluation, or during a proof. In either case the test is driven by a universally quantified formula, and a test is run by generating a series of test vectors; where a test vector associates a value with each universally quantified variable of the formula, based on its type. A given test vector is checked by instantiating the formula with the vector, and invoking the ground evaluator on the result. It iterates through the vectors until one is found for which the ground formula returns FALSE, at which point it normally prints out the vector and terminates.

In the ground evaluator, the formula is typed in the form of the `test` command, for example

```
(test "FORALL (n: nat): even?(n)")
```

In the theorem prover, the command is `random-test`, and the formula is derived using the current sequent. By default, the formula it uses is the conjunction of the antecedents implies the disjunction of the consequents, universally closed over the Skolem constants. The user can select which formulas to include in the test. Note that it is easy to test arbitrary formulas in the prover without exiting—simply use the `case` command to introduce an arbitrary formula, and only include it in the random test.

There are a number of parameters to the random test commands; these are the same whether in the ground evaluator or the prover.[2] The *count* controls how many test vectors to try. The *size* and *dt-size* parameters control how big to make base types and inductive datatypes, respectively. The *all?* says to keep looking for counterexamples even if one is found, and the *verbose?* flag indicates that all test vectors and results should be displayed, not just the counterexamples. The *subtype-gen-bound* is used to control how hard to search for an element that satisfies a subtype predicate.

Recall that the ground evaluator only works with ground terms. The prover is perfectly happy with uninterpreted types and constants, but terms involving these cannot be evaluated. Most proofs, however, involve uninterpreted types and constants. For this reason, there is an *instance* parameter, that may be used to provide a theory instance, giving ground types and constants for the theory parameters, and mappings for the uninterpreted types and constants that your formula references. For example, if you are working on a formula in a theory of the form

```
Th[T: TYPE, c: T] =
BEGIN
 myT: TYPE
 myC: myT
 ...
END Th
```

[2] There is currently one important difference—the prover supports keyword arguments, while the ground evaluator only has positional arguments.

Then you could give an instance argument of

```
Th[int, 0]{{myT := [real, real -> real], myC := + }}
```

Of course, if you are not careful, you could pick an instance for which the formula is true, even though the fully general form may not be valid.

5 Examples

The first example we tried was red/black trees, which is the example done in [CH00]. In general, it was not very helpful, as PVS is primarily for specification, not implementation, so the algorithm was straightforward to define, and there were no errors that random testing could find. It was interesting in another way, however. The normal definition of a path of a Red-Black tree is a sequence of Red-Black trees, where the first element is the entire tree, and each subsequent element is a child of the preceding tree; the last element is a leaf. Thus it is a subtype of finite sequences of trees. But generating a random sequence of trees that just happens to be a path is very unlikely. To handle this, we instead defined a path over a tree as a list of booleans, where TRUE represents the left child, and FALSE represents the right child. Thus for a given tree, a path is a sequence of booleans that is the right length to reach a leaf node. With this definition, it became easy to randomly generate paths.

Another example we worked with is a flawed definition of `take` and `drop`, as described in [BN04]. Here is their specification in PVS.

```
ex1[T: TYPE]: THEORY
BEGIN
 i, j, n: VAR nat
 l: VAR list[T]

 take(n, l): RECURSIVE list[T] =
   CASES l OF
     null: null,
     cons(x, xs):
       IF n = 0 THEN null ELSE cons(x, take(n - 1, xs)) ENDIF
   ENDCASES
  MEASURE n
```

```
 drop(n, l): RECURSIVE list[T] =
   CASES l OF
     null: null,
     cons(x, xs):
       IF n = 0 THEN l ELSE drop(n - 1, xs) ENDIF
   ENDCASES
  MEASURE n

 take_drop_comm: THEOREM take(j, drop(i, l)) = drop(i, take(j, l))

END ex1
```

Here is the result of starting the proof and running `random-test`:

```
take_drop_comm :

  |-------
{1}   FORALL (i, j: nat, l: list[T]):
        take(j, drop(i, l)) = drop(i, take(j, l))

Rule? (random-test :instance "ex1[int]")
The formula is falsified with the substitutions:
 i ==> 4
 j ==> 3
 l ==> (: -4, -64, 0, -57, 39 :)
No change.
take_drop_comm :

  |-------
{1}   FORALL (i, j: nat, l: list[T]):
        take(j, drop(i, l)) = drop(i, take(j, l))

Rule?
```

It is easy to check that this is, indeed, a counterexample. In practice, this may need to be tried several times, or with the *count* argument set higher than the default 10. To simulate what QuickCheck does, simply run with *size* and *dtsize* set, for example,

```
Rule? (random-test :instance "ex1[int]" :dtsize 2 :size 2 :count 100)
The formula is falsified with the substitutions:
 i ==> 1
 j ==> 1
 l ==> (: 1, 0 :)
```

The last example illustrates higher-order functions.

```
  |-------
{1}   FORALL (f: [int -> int], x, y, z: int): f(x) + f(f(y)) < f(f(f(z)))

Rule? (random-test)
The formula is falsified with the substitutions:
 f ==> -89 -> -86, -92 -> -92, -86 -> -33, -4 -> 99
 x ==> -4
 y ==> -89
 z ==> -92
```

Here we see that the function built up has a finite number of pairs given. What this means is that any function of type `[int -> int]` that contains these pairs will falsify the formula. The key thing is that the random generator memoizes the function, so that if, for example, `z` is assigned to 2, and `f(2)` is 2, then `f(f(z))` is also 2.

6 Conclusion

The random test facility was only recently added to PVS, and we are still experimenting with it. It is in the PVS release candidate at `ftp://ftp.csl.sri.com/users/owre/pvs-release-candidate-2`. Some plans for the future include:

- Allow users to define random test generators for specific types. This should be done as PVS function attachments, rather than in Lisp. For example, for Red-Black tree paths, one could specify a random generator that randomly generates a Red-Black tree, then randomly selects a child, and continues until a leaf is reached. The result would be a path by construction, and it would be very fast, as the subtype constraint would not need to be tested.
- Allow for different random distributions, instead of the built-in uniform distribution.
- Provide handling for inductive definitions as described in [BN04].
- Handle more higher-order types, in particular, those arising from sets (e.g. $A = B \cup C$), where the domain is infinite (or simply very large).
- Try more serious experiments to see how effective this is as a proof aid.

References

[BN04] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *2nd International Conference on Software Engineering and Formal Methods*, pages 230–239, Beijing, China, September 2004. IEEE Computer Society.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279, Montreal, Canada, September 2000. Association for Computing Machinery.

[COR+01] Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001. Available from `http://www.csl.sri.com/users/rushby/abstracts/attachments`.

[DHT03] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203, August 2003.

[Muñ03] César Muñoz. *Rapid Prototyping in PVS*. National Institute of Aerospace, Hampton, VA, 2003. Available from `http://research.nianet.org/~munoz/PVSio/`.

[Sha99] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at `http://www.csl.sri.com/users/shankar/papers/PVSeval.ps.gz`.