Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder (Extended Abstract)[⋆]

Jasmin Christian Blanchette and Tobias Nipkow

Fakultät für Informatik, T. U. München, Garching, Germany
{blanchette,nipkow}@in.tum.de

1 Introduction

Anecdotal evidence suggests that most "theorems" initially given to an interactive theorem prover do not hold, typically because of a typo or a missing assumption, but sometimes because of a deep flaw. Modern proof assistants for higher-order logic (HOL) provide counterexample generators that can be run on putative theorems or on specific subgoals in a proof to spare users the Sisyphean task of trying to prove non-theorems.

Isabelle/HOL includes two such tools: Quickcheck [1] generates functional code for the HOL formula and evaluates it for random values of the free variables, and Refute [5] searches for finite countermodels of a formula through a reduction to SAT (Boolean satisfiability). Their areas of applicability are almost disjoint: Quickcheck excels at inductive datatypes but is restricted to the executable fragment of HOL (which excludes unbounded quantifiers) and may loop endlessly on inductive predicates. In contrast, Refute copes well with logical symbols, but inductive datatypes and predicates are mostly out of reach due to the state space explosion.

Our new tool, Nitpick [6], is designed to bridge this gap. Instead of using a SAT solver directly, it builds upon the Kodkod first-order relational model finder [4].[1] As a result, it benefits from Kodkod's optimizations (notably its symmetry breaking) and its richer logic. Inductive datatypes are handled following an Alloy idiom [3], and inductive predicates are unrolled as in bounded model checking [2]. Infinite datatypes are approximated by a finite fragment augmented with an undefined value, embedded in a three-valued logic. The current prototype outperforms Refute in nearly all benchmarks while enjoying wider applicability than Quickcheck.

2 Our Approach

Enumeration of domains. Nitpick uses Kodkod to find a finite model (a satisfying assignment to the free variables) of the formula $A \wedge \neg P$, where P is a putative theorem and A is the conjunction of axioms relevant to P. Nitpick, like Refute and Alloy, systematically enumerates the possible domain cardinalities for each type variable, so that if a formula has a finite counterexample, the tool finds it (unless it runs out of memory).

[⋆] This work is supported by the DFG grant NI 491/11-1. Thanks to Mark Summerfield and Tjark Weber and to an anonymous reviewer for suggesting textual improvements.

[1] The name Nitpick is shamelessly appropriated from a now retired Kodkod ancestor.

Nonuniform representations. The encoding of HOL terms as Kodkod expressions is designed to take advantage of Kodkod's relational calculus. Functions are normally represented as relations constrained to be left-total and functional. However, predicates and other functions whose range has cardinality 2 are naturally represented as sets. Higher-order quantifiers are skolemized away when practicable. Any remaining quantified function is replaced by a k-element vector of values, where k is the cardinality of the function's domain. Vectors are also used for functions passed as arguments to other functions. Abstractions $\lambda x.\, f(x)$ are encoded as comprehensions $\{(x,y) \mid y = f(x)\}$.

Partiality. Because of the axiom of infinity, the type *nat* of natural numbers does not admit any finite models. To work around this, Nitpick considers prefixes $\{0,1,\ldots,K\}$ of *nat* and maps numbers beyond K to the undefined value ($\perp$). Formulas of the form $\forall n :: \mathit{nat}.\ P(n)$ become $(\forall n \leq K.\ P(n)) \wedge P(\perp)$. Undefined values lead to a three-valued logic, which is encoded in terms of Kodkod's binary logic. If the formula evaluates to $\perp$, Nitpick reports the model as a potential counterexample and continues looking for a model that makes the formula evaluate to *True*. The user can instruct Nitpick to use Isabelle's automatic tactics to ascertain whether a potential counterexample is genuine.

Inductive datatypes and recursive functions. A typical example of an inductive datatype is α *list*, the type of lists with elements of type α generated by the constructors $\mathit{Nil} :: \alpha\ \mathit{list}$ and $\mathit{Cons} :: \alpha \to \alpha\ \mathit{list} \to \alpha\ \mathit{list}$. Nitpick's approach to inductive datatypes is inspired by the Alloy modeling of infinite datatypes done by Kuncak and Jackson [3]. Conceptually, each constructor is seen as constructing a different type, for which the user can specify a cardinality. For example, using a cardinality of 1 for *Nil* and 10 for *Cons*, Nitpick looks for all counterexamples that can be built using at most 11 different lists. The constructors are constrained relationally to ensure freedom and induction.

Functions that operate on inductive datatypes are often implemented by recursion. Instead of using the internal representation of functions synthesized by Isabelle's **primrec** and **function** packages, Nitpick relies on the more natural equational specification entered by the user.

Inductive predicates. Besides recursion, Isabelle also supports induction as a definition principle. An inductive predicate corresponds to a least fixed point. If there is only one fixed point (which can usually be proved automatically using Isabelle's termination tactics), Nitpick simply takes the fixed point equation as the specification of the predicate; for example, given the introduction rules $\mathit{even}\ 0$ and $\mathit{even}\ n \Longrightarrow \mathit{even}\ (n+2)$, Nitpick uses the equation $\mathit{even}\ n = (n = 0 \vee (\exists m.\ n = m+2 \wedge \mathit{even}\ m))$. In the general case, Nitpick unrolls the predicate a given number of times, as in bounded model checking [2]. Because the unrolling is incomplete, the predicate typically evaluates to *True* or $\perp$, rarely *False*.

Function specialization. A simple but effective optimization is to eliminate fixed arguments. Consider the *map* function specified by $\mathit{map}\ f\ \mathit{Nil} = \mathit{Nil}$ and $\mathit{map}\ f\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs})$. To falsify the formula $\mathit{map}\ h\ [y] = [h\ y]$, Nitpick introduces a new function map' defined by $\mathit{map}'\ \mathit{Nil} = \mathit{Nil}$ and $\mathit{map}'\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{Cons}\ (h\ x)\ (\mathit{map}'\ \mathit{xs})$ and attempts to falsify $\mathit{map}'\ [y] = [h\ y]$ instead. This drastically reduces the search space, especially for higher-order functions.

3 Evaluation

To assess Nitpick, we used a database of mutated formulas consisting mostly of non-theorems, as was done for Quickcheck [1]. Figure 1 summarizes the results of running Nitpick, Refute, and Quickcheck on 3 000 formulas derived from three theories: *List* is the standard theory of lists, *AVL* is a theory of AVL trees, and *POPLmark* is a formalization of System $F_{<:}$. Many of the formulas were not executable, in which case Quickcheck was not applicable. Refute's three-valued logic is unsound, so all counter-examples for formulas that involve an infinite type are potentially spurious. In contrast, Nitpick fared well on all kinds of formulas.

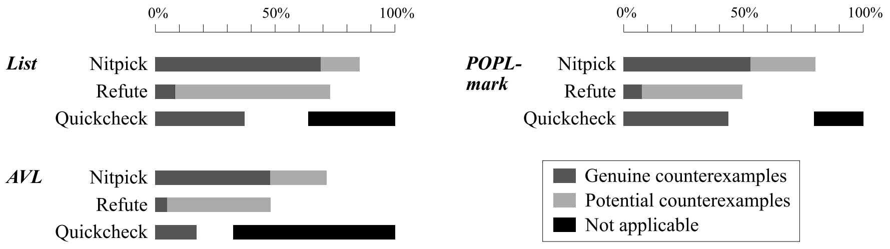

Fig. 1. Success rates of the counterexample generators on three theories

4 Conclusion and Future Work

Thanks to Kodkod and a nonuniform encoding scheme, Nitpick generates more counter-examples than similar tools, without restrictions on the form of the formulas. This means that Isabelle users can avoid a lot of wasted time spent trying to prove non-theorems. In addition, using Isabelle and Nitpick together provides a viable higher-order alternative to Alloy. Future work includes looking for encodings that would make it possible to handle much larger states, which arise when formalizing programming language semantics, and finding ways to rule out certain domain sizes through static analysis of the formula.

References

1. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) *SEFM 2004*, pp. 230–239. IEEE C.S. (2004)
2. Biere, A., Cimatti, A., Clarke, E. M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) *TACAS 1999*, LNCS vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
3. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Gall, H. (ed.) *Proc. ESEC/FSE 2005*, pp. 207–216 (2005)
4. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*, LNCS vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
5. Weber, T.: *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, T. U. München (2008)
6. Software: Nitpick. `http://isabelle.in.tum.de/~blanchet/#nitpick`