Symbolic Simulation in ACL2

Robert S. Boyer
ForrestHunt, Inc.
Austin, TX
boyer@centtech.com

Warren A. Hunt, Jr.
Centaur Technology and UT Austin
Austin, TX
hunt@centtech.com
hunt@cs.utexas.edu

ABSTRACT

We have created an experimental extension to ACL2 that provides a means to symbolically evaluate ACL2 expressions. This modified implementation can be used to compute the 'general' application of an ACL2 function to *generalized* data. In particular, we use uBDDs to represent functions from Boolean variables to finite sets of ACL2 objects, and for guard-checked ACL2 functions we can automatically create corresponding generalized functions to operate on such generalized data.

The `DEFTHM` hint mechanism has been extended to permit the direct application of symbolic simulation as a part of a proof attempt. This extension made it possible to directly verify the Legato Challenge using only symbolic simulation; this challenge involves proving the correctness of a 6502 assembly-language program that performs an 8-bit by 8-bit multiplication through repeated addition. We need only provide the initial symbolic data, i.e., two 8-bit, symbolic numbers, and we symbolically simulate the assembly-language program by symbolically simulating an ISA-level 6502-program interpreter that produces a symbolic result that we compare to its specification.

General Terms

ACL2, symbolic simulation, uBDDs

1. INTRODUCTION

Using the ACL2 theorem prover to prove the correctness of deeply-embedded system representations often requires significant amounts of symbolic simulation. Currently, to get ACL2 to perform symbolic simulation, the ACL2 rewriter is used to unroll functions and to simplify the results. Staged simplification has been used to help orchestrate this process, but the capacity and speed of such symbolic simulations is limited both by the size of the expressions being simulated and by the rewrite rules enabled to effect the symbolic simulation and simplification.

We have developed an experimental extension to ACL2 that provides a means to symbolically evaluate ACL2 symbolic expressions. We perform symbolic simulation by representing *generalized* ACL2 objects, and then manipulate such data with generalized versions of ACL2 functions. A user may access this capability through an extended version of the ACL2 `DEFTHM` hint mechanism or by escaping the ACL2 loop and evaluating the generalized version of the function. For any guard-verified ACL2 function definition, a user may request that the corresponding generalized version of the function be created. For every ACL2 primitive, we have implemented such generalized functions. A user may execute any generalized function on suitable generalized or explicit input objects at the Common Lisp prompt level.

We have used this extension for both hardware and software verification. We have verified the *Legato Challenge*; this verification demonstrated the correctness of an assembly program that implements multiplication. Although not discussed here, we have verified the floating-point addition and subtraction instructions of Centaur's floating-point media unit using symbolic simulation of our integer-based specification [2]. We begin by describing our internal data representation format for generalized data. Next, we define a evaluator of `IF` expressions with Boolean variables, and we show how symbolic simulation of this evaluator implements a tautology checker. Finally, we use symbolic simulation to solve the Legato Challenge.

2. SYMBOLIC SIMULATION

The symbolic simulation capability we have developed permits the application of an ACL2 function to generalized data; this involves defining a generalized version of each ACL2 function that operates on generalized data and a means to represent finite sets of ACL2 objects.

We describe a finite set of ACL2 data objects by generalizing the underlying Common-Lisp representation of ACL2 constants. Our data generalization approach is to generalize each binary digit (bit) of the underlying Common-Lisp representation for each object. We use a uBDD [1] to represent each generalized bit; a specific object is identified by an explicit assignment of Boolean values for each Boolean variable, evaluating each uBDD involved and finally yielding an explicit value. For example, the Common-Lisp representation of an integer is composed of binary digits (bits); we generalize an integer by generalizing each bit in its representation with a uBDD. The set of values represented by such

a generalized integer is found by considering all possible assignments of values to the variables of the uBDD(s) used to represent each underlying bit; particular integers in such a set are identified by assigning each uBDD variable a value which yields a collection of the binary digits of the integer in question. Thus, we map uBDD variable assignments, which are just a list of Boolean values, to particular explict values.

Consider the representation of the integers from zero to seven using three uBDDs, one to represent each of the least-significant three binary digits. For instance, by using a distinct uBDD variable for each digit, we can represent the collection of the integers values from zero to seven. Thus, uBDD variable values are the domain of a function that maps to integers; we assign each uBDD variable a value by giving a list of Boolean values, one for each variable. Given that the first three uBDD variables are used to represent (in least-significant order) the first three binary digits of a positive integer, we would have a mapping (represented by an association list) of uBDD variables to values as follows where `...` represents the values of other uBDD variables.

```
'(((nil nil nil ...) . 0)
  (( t  nil nil ...) . 1)
  (( nil t  nil ...) . 2)
  (( t   t  nil ...) . 3)
  ((nil nil  t  ...) . 4)
  (( t  nil  t  ...) . 5)
  ((nil  t   t  ...) . 6)
  (( t   t   t  ...) . 7))
```

For this example, the assignment `(nil ?  t ...)` to the first three uBDD variables yields the set of integers {4,6}, where `?` represents free assignment. To represent two sets of the integers from zero to seven would require six uBDD variables. Thus the first three uBDD variables might represent the integers from 0 to 7 as shown above and the second set of integers might be represented using the next three uBDD variables as follows where `?` represents an unknown assignment for each of the first three uBDD variables.

```
'(((? ? ? nil nil nil ...) . 0)
  ((? ? ?  t  nil nil ...) . 1)
  ((? ? ? nil  t  nil ...) . 2)
  ((? ? ?  t   t  nil ...) . 3)
  ((? ? ? nil nil  t  ...) . 4)
  ((? ? ?  t  nil  t  ...) . 5)
  ((? ? ? nil  t   t  ...) . 6)
  ((? ? ?  t   t   t  ...) . 7))
```

Thus, uBDD variables are the domain of functions that map uBDD variables to collections of finite sets of ACL2 objects.

Using '`g-<`' symbol for the generalized less-than predicate

```
(g-< {(? nil t   ? ? ? )} ; Set 1, range {4,5}
     {(?  ?  ?   ? t t )} ; Set 2, range {6,7}
     )
```

always evaluates to `T`; for every assignment of uBDD variables, the image of under the first function is less than that of under the second function. Or, more informally, the (first set of) integers {4,5} are everywhere less than the (second set of) integers {6,7}. If we symbolically execute `(g-< {5,6} {6,7})`, we produce a uBDD that is `NIL` when the assignment of uBDD variables maps each function range to `6` and is otherwise `T`.

A finite set of ACL2 objects is represented as a `CONS` pair where its `CAR` is one of nine symbols from the `ACL2_INVISIBLE` (abbreviated `I`) package; otherwise, it represents pair of possibly general ACL2 objects. Each symbolic object is of the form `(CONS I::`*object_type object*`)`; such `BAD-ATOM` symbols cannot be created by an ACL2 user and are internally used to identify symbolic objects. For the descriptions below, a uBDD is recognized by the `NORMP` function and a list of uBDDs by `NORM-LISTP`. Function call `(EVAL-BDD bdd lst)` produces the value of `bdd` given variable assignment `lst`, which, in turn, identifies one element of the function range.

- **BIT**: A general bit `g` has the form `(LIST* 'i::bit n)` and represents `(EVAL-BDD n lst)`.
- **INTEGER**: A general integer `g` has the form `(LIST* 'i::integer cadr cddr)` and represents the integer n whose sign is what `(CADR g)` represents and whose absolute value is what `(CDDR g)` represents, viewing `(CDDR g)` as an unsigned nonnegative integer, in binary, least significant bit first, with `T` meaning binary 1 and `NIL` meaning binary 0. `(NORM-LISTP (CDR g))` is required. `T` means 'negative' as a sign.
- **CHARACTER**: A general character `g` has the form `(LIST* 'i::character c)` and represents the result of calling function `CODE-CHAR` on what `c` represents, which must be a nonnegative integer.
- **STRING**: A general string `g` has the form `(LIST* 'i::string s)` and represents the result of calling `COERCE` upon (a) what `s` represents, which must be a true list of characters, and (b) `'STRING`.
- **SYMBOL**: A general symbol `g` has the form `(LIST* 'i::symbol p n)` and represents the result of calling `INTERN$` on (a) what `n` represents, which must be a string and (b) the result of calling `FIND-PACKAGE` on what `p` represents, which must be a string that is the name of an ACL2 package.
- **RATIO**: A general ratio `g` has the form `(LIST* 'i::ratio n d)` and represents the result of calling division (/) on (a) what `n` represents, which must be an ACL2 integer, and (2) what `d` represents, which must be a positive ACL2 integer.
- **IF**: An general if-expression `g` has the form `(LIST* 'i::if c tb fb)`, where `c` is a `NORMP` and `tb` and `fb` are general objects. `g` represents what `fb` represents if `c` represents NIL; otherwise, it represents what `tb` represents.
- **UNEVALUATED**: A general UNEVALUATED `g` has the form `(LIST* 'i::unevaluated x)` and represents an ACL2 object, but we assume nothing about which ACL2 object. Of course, two UNEVALUATED objects that are equal represent the same thing.

Given an ACL2-accepted, Common-Lisp compliant, guard-checked function, say FN, the user may symbolically compute the value of the automatically generated generalized version of FN, call it G-FN, on such generalized objects by escaping from the ACL2 read-eval-print loop. This modification greatly reduced the complexity of writing the Centaur floating-point specifications as we did not need to create a bit-level specification; our integer-based specification was directly symbolically simulated. We used this approach in our verification of the floating-point addition/subtraction operations for the Centaur CN media unit [2].

For the 31 ACL2 primitives, we have defined their symbolic simulation counterparts. For instance, for BINARY-+, we have implemented something like a symbolic version of a ripple-carry adder. That is, G-BINARY-+ has been defined to accept two symbolic data arguments and it computes their symbolic sum. Of course, the actual definition of G-BINARY-+ is somewhat more complicated as it needs to deal with the various cases that are possible, such as the arguments being complex numbers.

Perhaps it is simpler to visualize the code for G-NOT, the symbolic version of NOT. Its implementation starts by asking if its input argument is symbolic or explicit. If its argument is explicit, then this argument is provided to a call of NOT and that result is returned; otherwise, we return a new symbolic object that is T whenever the assignment of Boolean variables would produce a NIL for the given input argument, and otherwise T. When given explicit objects, all of our symbolic primitive definitions just call their corresponding ACL2 functions.

The key operation in our symbolic simulator is when we must deal with an IF expression. If the test argument can be determined, then we will return either the false (NIL) branch or the not false branch as appropriate. However, when the test is itself a symbolic object, then we have to merge the two (possibly) symbolic return objects into a single object; this involves computing a new mapping from Boolean variables to objects that respect the semantics of IF.

For improved performance, we have defined symbolic versions of a number of non-primitive ACL2 functions. For instance, the symbolic version of ASH only needs to perform simple list operations on the CDDR of symbolic integers when the shift amount is explicit. We spent many hours carefully tuning the symbolic definitions of a number of non-primitive ACL2 arithmetic operations, eventually allowing us to build a uBDD representing the single-precision, floating-point add function.

For any ACL2 function that does not have a built-in symbolic definition, we compile it by replacing each internal function call with their corresponding symbolic version. Thus, creating the G- version of a new ACL2 function is nothing more than creating a new function where every internal function call is replaced with its symbolic counterpart.

The ACL2 DEFTHM hint mechanism has been extended so that a user may specify generalized data to be symbolically simulated in pursuit of a theorem. For the DEFTHM example below, we introduce functions AND2 and OR2 so that we can disable them, thereby avoiding the possibility that ACL2 is doing the proof using its native reasoning about AND and OR.

```
(defun and2 (x y) (and x y))

(defun or2  (x y) (or  x y))

(in-theory (disable and2 (and2) or2 (or2)))

(defthm and-implies-the-or
  (implies (and (booleanp x)
                (booleanp y))
           (implies (and2 x y) (or2 x y)))
  :hints (("Goal"
           :clause-processor
           (:function
            g-clause-processor
            :hint
            (list
             'gs-specs-env ; function to run on
             (list 'x `(:bit . ,(qvar-n 0)))
             (list 'y `(:bit . ,(qvar-n 1)))))))
  :rule-classes nil)
```

This lemma looks like an ordinary ACL2 DEFTHM event except that the hint instructs the theorem prover to perform this proof using the G-CLAUSE-PROCESSOR where free variables X and Y represent two generalized ACL2 Boolean values. This theorem is proved entirely by symbolic simulation – no part of the waterfall is used.

3. MINI THEOREM PROVER

Using this symbolic simulation capability, it is possible to construct a decision procedure for propositional logic with only a very few lines. IF-TERMP is a recognizer for the IF-expression language in which propositional logic statements are to be written.

```
(defun if-termp (term)
  (declare (xargs :guard t))
  (if (atom term)
      (eqlablep term)
    (let ((fn (car term))
          (args (cdr term)))
      (and (consp args)
           (consp (cdr args))
           (consp (cddr args))
           (null  (cdddr args))
           (eql fn 'if)
           (if-termp (car args))
           (if-termp (cadr args))
           (if-termp (caddr args))))))

(defun if-evl (term alist)
  (declare
   (xargs :guard (and (if-termp term)
                      (eqlable-alistp alist))))
  (if (atom term)
      (cdr (assoc term alist))
    (if (if-evl (cadr term) alist)
        (if-evl (caddr term) alist)
      (if-evl (cadddr term) alist))))
```

Now, the validity of an IF expression can be determined by creating an association list where each variable, recognized by EQLABLEP in IF-TERMP, is paired with a unique generalized Boolean value, and then symbolically executing the generalized IF-EVL function. Thus, the function G-IF-EVL is a decision procedure for propositional logic given that TO-IF2 translates logic expressions into IF expressions.

```
:q  ; exit from ACL2

(maybe-gify 'if-evl)  ; creates function G-IF-EVL

; We now show that the created function G-IF-EVL is
; a theorem-prover for the propositional calculus.

(let ((term
       (to-if2
        '(implies (and x y) (or x y))))
      (alist
       `((nil . nil)
         (t   .   t)
         (x   . ,(make-bit (qv 0)))
         (y   . ,(make-bit (qv 1))))))
  (g-if-evl term alist))  ; Evaluates to T
```

4. THE LEGATO CHALLENGE

The Legato Challenge, proposed by Bill Legato in around 1990, is to prove the correctness of a multiply algorithm for the Mostek 6502. In chronological order, J Moore, Matt Wilding, Robert Krug, and Sandip Ray, have all verified this program using either NQTHM (Moore) or ACL2. Legato proposed this problem as a challenge for the automated theorem-proving community, and it has served as an interesting test for the ACL2 community.

The 6502 multiply assembler program developed the product through repeated addition, eventually producing a 16-bit product given two, eight-bit operands. Each successful verification has involved modeling each instruction used in the program as a function, and then showing that the repeated composition of these functions does indeed perform multiplication. Recently, the authors have also verified the Legato Challenge using our symbolic simulation capability.

The original Legato Challenge was 10 instructions; one carry-clear instruction was added so the program worked no-matter how the carry flag was initialized. We added the third instruction to make our final theorem more general; later this will prevent us from needing to initialize the carry flag.

```
(defg *mult*
  '((LDX 8)    ;  1  ; Initialize X to 8
    (LDA 0)    ;  2  ; Initialize A to 0
    (CLC)      ;  3  ;   Added for Initialization
    (ROR F1)   ;  4  ; Rotate Carry through F1
    (BCC 8)    ;  5  ; Branch if Carry Clear
    (CLC)      ;  6  ; Clear Carry flag
    (ADC F2)   ;  7  ; Add F2 with Carry to A
    (ROR A)    ;  8  ; Rotate A right
    (ROR LOW)  ;  9  ; Rotate Carry into Low reg
    (DEX)      ; 10  ; Decrement X
    (BNE 4)))  ; 11  ; Branch not-equal to 4
```

Legato also produced an ISA-level ACL2 simulator for a subset of the 6502 that was sufficient to execute the program above. The core of this ISA specification is a single-step function that transforms an ISA state to a new ISA state by updating the state in consideration of the instruction pointed to by the program counter. The state accessors for Legato's 6502 assembly-level model are:

```
(defmacro icntr  (s) `(car ,s))   ; program counter
(defmacro c      (s) `(nth 1 ,s)) ; carry flag
(defmacro a      (s) `(nth 2 ,s)) ; accumulator
(defmacro low    (s) `(nth 3 ,s)) ; memory variable
(defmacro x      (s) `(nth 4 ,s)) ; index register
(defmacro f1     (s) `(nth 5 ,s)) ; memory variable
(defmacro f2     (s) `(nth 6 ,s)) ; memory variable
(defmacro z      (s) `(nth 7 ,s)) ; zero flag
(defmacro f1save (s) `(nth 8 ,s)) ; shadow value
(defmacro p      (s) `(nth 9 ,s)) ; the program
```

Legato's 6502 ISA-level simulator is a "classic" FM8501-style instruction interpreter.

```
(defun mult-stp (s)
  (if (or (zp (icntr s))
          (< (len (p s)) (icntr s)))
      s    ;; Get Instruction
    (let ((inst (nth (1- (icntr s)) (p s))))
      (case (identity (car inst)) ; op code
        (LDX (update
              (update (nxt s) 4 (getarg s (cadr inst)))
              7 (if (zp (getarg s (cadr inst))) 1 0)))
        (LDA (update
              (update (nxt s) 2 (getarg s (cadr inst)))
              7 (if (zp (getarg s (cadr inst))) 1 0)))
        (ROR (update
...
        (otherwise s)))))
```

Legato's challenge was simple – prove that his program multiplies two 8-bit numbers and produces a correct 16-bit (represented as two 8-bit values) result. We represent each component of the state as a generalized data object. Of course, the program counter has to be initialized to 1, and the LOW memory value is 0. The carry and zero flags and the accumulator can be arbitrary values. The generalized data objects *F1* and *F2* represent two distinct data objects where each represents all of the natural numbers from 0 to 255.

```
(defg *general-start-state-mult*
   (list 1    ; program counter
         *c*  ; carry flag
         *a*  ; accumulator
         0    ; memory location LOW
         *x*  ; x - counter
         *f1* ; operand f1
         *f2* ; operand f2
         *z*  ; zero flag
         *f1save* *mult*))
```

DEFG defines a constant. The symbolic version of the function RUN-IT is allowed to symbolically execute as many as 67 step, each time RUN-IT recurs, the function MULT-STP is used once. Our goal theorem is this relationship.

```
(= (* (f1 *general-start-state-mult*)
      (f2 *general-start-state-mult*))
   (let ((*last-state*
          (run-it 67 *general-start-state-mult*)))
     (+ (* 256 (a *last-state*))
        (low *last-state*))))
```

Our specification is the multiplication of the initial symbolic number *F1* and *F2*. We execute Legato's program on generalized data by repeatedly executing MULT-STP symbolically, and when finished, we collect the LOW memory value to which we add 256 times the A (accumulator) register; this result is compared to the specification.

5. PROOF DISCUSSIONS

The authors asked each of the people that had previously verified Legato's challenge for their recollections about their efforts. We found these descriptions typical for an ACL2-style proof, but all were labor intensive. Moore, Wilding, and Krug didn't actually verify Legato's program using the MULT-STP ISA-level simulator, but instead, wrote functions that model the state changes each instruction produces. Ray's proof actually uses the MULT-STP ISA-level simulator. We have edited for brevity.

It took J Moore about four days to do his proof. Moore recalls:

> ... the thing that made my first proof different was that I formalized the "real" reason it was correct, which had to do with how it mimics ordinary shift-and-add binary multiplication. That is, I didn't do an arithmetic proof. I mapped everything to bit vectors and then established a simulation between the easy-to-prove correct shift-and-add and Legato's algorithm.
>
> I remember arguing with Bill [Legato] about the approach. He wanted an automatic arithmetic-based proof. I felt that no sane programmer would think of it arithmetically, he would think of it the way I described and that the "right" proof was to follow the derivation of the algorithm.

Over the 2001 Christmas break, Matt Wilding thought he would check out Rockwell's new SUPER-IHS arithmetic library on the Legato Challenge. Wilding's effort mentions:

1. Understanding the algorithm	1	hr
2. Modeling the program in ACL2	2	hrs
3. Lost time before decomposing proof	4	hrs
4. Proving that loop-spec works	3	hrs
5. Proving loop-spec works correctly	3.5	hrs
6. Cleaning and documenting	1.5	hrs

Robert Krug wanted to see how his evolving arithmetic library would perform on this proof. His effort involved:

1. Trying weakest precondition method	4.5	hrs
2. Trying to imitate Wilding's proof	2	hrs
3. Writing and testing specifications	2	hrs
4. Constructing the main helper lemma	2	hrs
5. Getting ACL2 to do the proof	4.5	hrs
6. Some clean up	2	hrs

In 2008, Sandip Ray developed a proof that uses the 6502 operational semantics as provided by MULT-STP. This effort was based on the recently developed cutpoint mechanism for performing interpreter proofs. Sandip writes:

> It took me about 4 hours. Most of the effort was in looking at failed arithmetic subgoals, and struggling with arithmetic libraries to determine what arithmetic lemmas were necessary.
>
> I don't think Wilding (or Legato, originally) used an interpreter definition. Rather, they used their own tools (Legato used his WP generator and Wilding probably used vFAAT) to spit out formulas which were then discharged in ACL2. So, if they wanted to do an operational-semantics-style proof, it would have taken them longer.

We don't have anything much to say. We just stated the theorem and attempted a symbolic simulation. Of course, it didn't work the first time – we had bugs in our symbolic simulator implementation. So instead of trying to debug our proof attempt, we used Legato's Challenge to debug our symbolic simulator! Once we wrung out our bugs, the symbolic simulator proves the Legato Challenge in just a few seconds. Finally, may the Legato Challenge rest in peace.

6. CONCLUSIONS

We believe symbolic simulation is a valuable tool for ACL2 users. We have an experimental version of ACL2 with our symbolic simulation capability; however, given our gradual transition to using Sol Swords' approach of embedding symbolic terms directly with user-level, symbolic variables, we may not further develop this approach.

7. ACKNOWLEDGEMENTS

We want to thank Matt Kaufmann for extending the ACL2 DEFTHM hint mechanism. We wish to acknowledge support from Centaur Technology and ForrestHunt, Inc.

8. REFERENCES

[1] Robert S. Boyer and Warren A. Hunt, Jr.. Function Memoization and Unique Object Representation for ACL2 Functions. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, ACM Digital Library, Seattle, Washington, 2006.

[2] Warren A. Hunt, Jr. and Sol O. Swords Centaur Technology Media Unit Verification. To appear in proceeding of the *2009 Computer-Aided Verification Conference (CAV 2009)*, Lecture Notes in Computer Science, Grenoble, France, 2009.