Rewriting for Symbolic Execution of State Machine Models

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Office: TAY 4.140A
Email: moore@cs.utexas.edu

http://www.cs.utexas.edu/users/moore

Simulation Models

```
s.a = new-a(x,s);
s.b = new-b(x,s);
s.c = new-c(x,s);
return s;
```

may be formalized in ACL2 (i.e., Lisp) as

```
(let ((s (update-a (new-a x s) s)))
  (let ((s (update-b (new-b x s) s)))
    (let ((s (update-c (new-c x s) s)))
      s)))
```

```
(let ((s (update-a (new-a x s) s)))
  (let ((s (update-b (new-b x s) s)))
    (let ((s (update-c (new-c x s) s)))
      s)))
```

Applicative semantics allows theorem proving about the model.

The *single-threaded* use of `s` allows *destructive* modification at runtime so the formal model can be executed efficiently.

Let is just *lambda application*.

```
(let ((s (update-a (new-a x s) s)))
  (let ((s (update-b (new-b x s) s)))
    (let ((s (update-c (new-c x s) s)))
      s))) = {by abbreviation conventions}

((lambda (s x)
   ((lambda (s x)
      ((lambda (s) s)
       (update-c (new-c x s) s)))
    (update-b (new-b x s) s) x))
 (update-a (new-a x s) s) x)
```

```
(let ((s (update-a (new-a x s) s)))
  (let ((s (update-b (new-b x s) s)))
    (let ((s (update-c (new-c x s) s)))
      s))) = {by Beta reduction}

(update-c
 (new-c x (update-b
           (new-b x (update-a (new-a x s) s))
           (update-a (new-a x s) s)))
 (update-b
  (new-b x (update-a (new-a x s) s))
  (update-a (new-a x s) s)))
```

```
(defun phase1 (s)
 (let ((s (update-a
           (if (v0 1 (a s))
               (v1 1 (a s))
               (v2 1 (a s)))
           s)))
  (let ((s (update-a
            (if (v0 2 (a s))
                (v1 2 (a s))
                (v2 2 (a s)))
            s)))
    ... s...)))
```

```
(defun step (s)
  (let ((s (phase1 s)))
    (let ((s (phase2 s)))
      ...
       s...)))
```

It is not unusual for commercial simulation models of microprocessors to have *very many* assignment statements, i.e., very deep `lambda`-nesting.

At Rockwell-Collins Avionics, some of the models we look at have `lambda` nesting of depth 300 or more.

Beta reduction is not an option.

The typical question asked of a symbolic state expression is "what is the the value of slot x?", e.g., for slot b,

```
(b (let ((s (update-a (new-a x s) s)))
     (let ((s (update-b (new-b x s) s)))
       (let ((s (update-c (new-c x s) s)))
         s))))
```

A Simplifying Convention

We number the slots, so

```
(defun b (s) (nth 2 s))
(defun update-a (u s) (update-nth 1 u s))
```

Key Theorem:

```
(nth i (update-nth j u s))
  =
(if (= i j) u (nth i s))
```

Typically we want to simplify,
`(b (step s))`, where, e.g.,

```
(defun b (s) (nth 2 s))
(defun update-a (u s) (update-nth 1 u s))
(defun phase1 (s)
 (let ((s (update-a (if ... ... ...) s)))
    ... s...))
(defun step (s)
  (let ((s (phase1 s)))
    (let ((s (phase2 s)))
      ... s...)))
```

The First Key Idea: Facets

Terms are represented by *facets*, which are like `lambda` nests turned inside out.

Consider the `b` slot of the term

```
(let ((y (f x s))
      (s (update-a (new-a x s) s)))
  (let ((s (update-c (new-c x s) s)))
     (phase1 y s)))
```

= {*the facet representation*}

```
(phase1 y s), ([(s ◁ (update-c (new-c x s) s))
                (y ◁ y)]
               [(y ◁ (f x s))
                (s ◁ (update-a (new-a x s) s))
                (x ◁ x)])
```

Consider the `b` slot of the facet

```
(phase1 y s), ([(s ◁ (update-c (new-c x s) s))
                (y ◁ y)]
               [(y ◁ (f x s))
                (s ◁ (update-a (new-a x s) s))
                (x ◁ x)])
```

where

```
(defun phase1 (u s)
  (let ((s (update-b (h u s) s)))
    s))
```

the b slot of the facet

```
(update-b (h u s) s),
              ([(u ◁ y)
                (s ◁ s)]
               [(s ◁ (update-c (new-c x s) s))
                (y ◁ y)]
               [(y ◁ (f x s))
                (s ◁ (update-a (new-a x s) s))
                (x ◁ x)])
```

=

the facet

```
(h u s),        ([(u ◁ y)
                  (s ◁ s)]
                 [(s ◁ (update-c (new-c x s) s))
                  (y ◁ y)]
                 [(y ◁ (f x s))
                  (s ◁ (update-a (new-a x s) s))
                  (x ◁ x)])
```

The Second Key Idea: Reconciliation

Sometimes we wish to return a facet representing a term

```
(f a b)
```

where `a` and `b` are themselves represented by facets α, τ_α and β, τ_β.

To do this we *reconcile* the two facets:

compute elaborations of α and β,
say α' and β',

and a common stack σ,

such that $\alpha, \tau_\alpha = \alpha', \sigma$ and $\beta, \tau_\beta = \beta', \sigma$.

Then we return the facet `(f` α' β'`)`, σ.

The Third Key Idea: Caching or Memoization

In one application, the algorithm was called 216,524 times.

The cache hit rate was 6.2%.

But without the cache the algorithm would have required $\sim 3 \times 10^{26}$ calls.

Details are in the paper.

Experiments

```
(defun phase1 (s)
  (let ((s (update-a (if (v0 1 (a s))
                          (v1 1 (a s))
                        (v2 1 (a s)))
                     s)))
    (let ((s (update-a (if (v0 2 (a s))
                            (v1 2 (a s))
                          (v2 2 (a s)))
                       s)))
      ... s ...))) ; 6 levels
```

```
(defthm b-phase1
  (equal (b (phase1 s))
         (b s)))

(defthm b-phase1-phase1
  (equal (b (phase1 (phase1 s)))
         (b s)))
```

```
(defun next-a (a)
  (let ((a (if (v0 1 a) (v1 1 a) (v2 1 a))))
    (let ((a (if (v0 2 a) (v1 2 a) (v2 2 a))))
      ...
           a ...)))

(defthm a-phase1
  (equal (a (phase1 s))
         (next-a (a s))))
```

```
(defun phase0 (s)
  (let ((s (update-b (a s) s)))
    s))

(defun phase2 (s)
  (let ((s (update-a (b s) s)))
    s))
```

```
(defun machine (s)
  (let ((s (phase0 s)))
    (let ((s (phase1 s)))
      (let ((s (phase1 s)))
        (let ((s (phase2 s)))
          s)))))
```

```
(defthm a-machine
  (equal (a (machine s))
         (a s)))

(defthm b-machine
  (equal (b (machine s))
         (a s)))
```

Theorem	old	new
`b-phase1`	0.48	0.01
`b-phase1-phase1`	128.76	0.01
`a-phase1`	0.41	0.04
`a-machine`	139.39	0.02
`b-machine`	143.91	0.02

Figure 1: Seconds to Prove Theorems on 731 MHz Pentium III

old = ACL2 without new algorithm

new = ACL2 with new algorithm

Industrial Applications

At Rockwell-Collins, ACL2 is used to model the microcode engine of avionics microprocessors.

The ACL2 model of one microprocessor executes at approximately 90% of the speed of a hand-coded C model.

Using this algorithm, integrated into ACL2's rewriter, it is possible to symbolically step the microcode engine.

Without this algorithm, it was impossible.

Related Work

Hickey and Nogin's *term module* supports *delayed substitution*, e.g., lambda-expressions to represent instantiations. They support more general operations on terms.

Facets are inherently more efficient.

In addition, our algorithm uses caching.

Facets are suggestive but independent of *explicit substitution* logics.

I view facets as an efficient data structure for implementing certain simplification strategies for conventional logics.

Unrelated Work

We have an operational model of a JVM subset (supporting 138 bytecodes) excluding class loading and exceptions.

We have a mechanical translator from Java to this JVM model via `javac`.

We have proved theorems about Java programs, including mutual-exclusion and multi-threading examples.