The DE Language

Erik Reeber

6/30/04

Overview

- Introduction & Motivation
- Examples
- Implementation and Static Checking Overview
- Verification & Tool Flow
- Verification Example
- Summary & Conclusion

Introduction

- DE is a Hardware Description Language
  - Simple FSM semantics
- Embedded in ACL2
- Predecessor: DUAL-EVAL
  - FM9001 proof
  - No combinational loops
  - Primitives are modules
  - No types

Motivation

- Need to get design into ACL2
  - Advantages to embedded approach:
    - Closer to actual design
    - Can verify optimization and verification tools
    - Can write and verify generation functions
- Need Finite Decision Procedures
  - Type system
- Explore HDL design
  - Simple semantics
  - May put specification in the code

Example 1: XNOR

```
`(xnor2
   (ins (x 2) (y 2))
   (outs (q 2))
   (wires (xq 2))
   (occs
     (xor-occ (xq)
              (bufn 2)
              ((xor 2 x y)))
     (xnor-occ (q)
               (bufn 2)
               ((not 2 xq)))))
```

- No one would build this module in DE; it is too simple.
- Declare widths of inputs, outputs, and wires to be 2.
- An occurrence is an instantiation of a module. Bufn plays the roll of an assignment.
- Inputs are ACL2 expressions using a small set of primitives.
- No combinational loops in the occurrence list.

Modeling a Synopsis MUX

```
'(any-mux-n-w
   (params n w)
   (outs (q w))
   (ins (sel n) (x (expt 2 w)))
   (occs
    (occ (q)
         ((lambda (n w sel x)
           (list
             'nil
             (s-m-n-w n w sel x)))
          n w)
         (sel x)))))
```

- Synopsis has a built-in (2^n:1) * w mux module that is sometimes used in TRIPS
- We implement an ACL2 version named s-m-n-w.
- We add this function to our list of allowed primitives.
- Lambda functions need the parameters and return the state as well as the outputs.
- We implement the module with parameterized bit widths.

Example 2: A simple ALU

```
'(simple-ALU
    (params w)
    (outs (q w))
    (ins (op 2) (x w) (y w))
    (wires (m-in (* (expt 2 2) w)))
    (occs
     (op0 ((m-in 0  (1- w))) (bufn w) ((or w x y)))
     (op1 ((m-in w  (1- (* 2 w)))) (bufn w) ((and w x y)))
     (op2 ((m-in (* 2 w) (1- (* 3 w)))) (bufn w) ((xor w x y)))
     (op3 ((m-in (* 3 w) (1- (* 4 w)))) (bufn w)
           ((not (and w x y))))
     (alu (q)
          (any-mux-n-w 2 w)
          (op m-in)))))
```

Example 2 Comments

- Using parameters to generalize a little bit
  - Need actual parameters to synthesize
- Using a lot of buffers and ACL2 functions
  - This is a lot more similar to the TRIPS style than using only module instantiations

An n bit register

```
`(reg-n
   (type primitive)
   (params n)
   (ins (x n))
   (outs (q n))
   (sts st)
   (st-decls (st n))
   (occs
     (st (q)
         ((lambda (st x)
          (list x st)))
         (st x))))
```

- Declared to be a primitive: allowed to use state like a wire, but not allowed to instantiate modules
- State st is declared twice. The first declaration means it takes state, the second means that it is finite and a bit-vector.

Example 3: Accumulator

```
`(accumulator
   (params n)
   (ins (op 2) (x n))
   (outs (q n))
   (sts st)
   (wires (y n))
   (occs
     (st (y)
         (reg-n n)
         (q))
     (alu (q)
          (simple-ALU n)
          (op x y))))
```

- Here we use the ALU and the register to build an accumulator
- The state is only declared once, and it has no type.
- State is passed automatically to register, by virtue of its instance name.
- The “loop” here is not combinational

Example 4: Memory Block

```
`(memory-block
  (type primitive)
  (params n w)
  (ins (wr w)
       (wr-en 1)
       (addr n))
  (outs (q w))
  (sts st)
  (occs
    (st (q)
        ((lambda (n w st wr wr-en addr)
         (list (mem-b-ns n w st wr wr-en)
               (mem-b-q n w st addr)) n w)
        (st wr wr-en addr))))
```

- Here we intend to model a large block of memory.
- State is not implemented as a bit vector, and may not be finite.
- ACL2 functions mem-b-ns and mem-b-q are added as next state and output primitives.

The SE and DE functions

- The semantics of DE are implemented as a single-pass **output** evaluator, SE, and a dual-pass **state** evaluator, DE.
- These functions take in the following arguments: flg, params, ins, sts, env, netlist.
- Example call and output:

```
> (se `flg `simple-ALU `(2)
      `((bv-const 2 0) (bv-const 2 1)) (bv-const 2 2))
      `() (simple-ALU-net))
((t t))
```

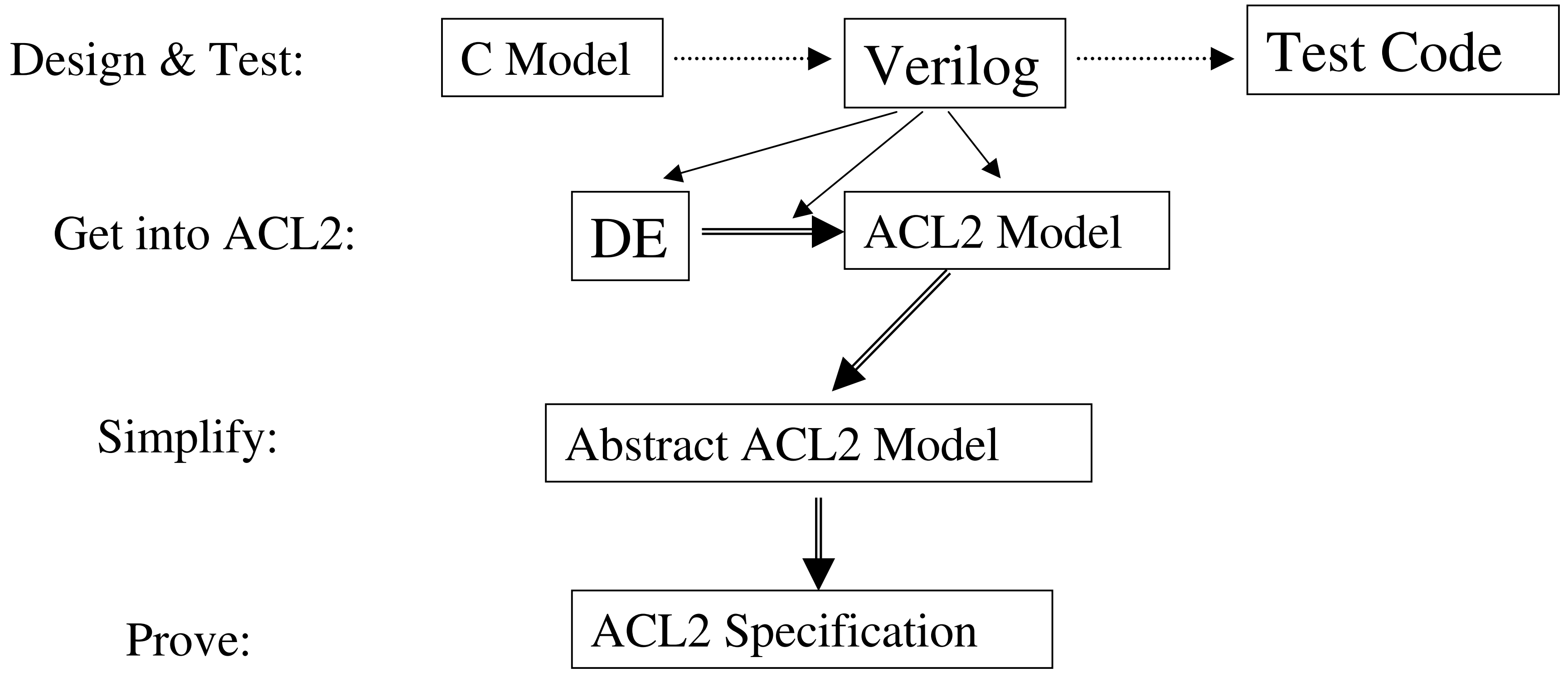
Static Checking

- **syntaxp**, given fn and netlist checks the following of fn and its components:
  - Is an alist with **outs** and **occs**
  - Each **occ** has the proper format.
  - Declarations in **outs**, **ins**, **sts**, and **st-decls** properly formatted**.**
  - No duplicate names
  - Names in **sts** occur in the occs entry (once)
  - All wires that are referenced are declared
  - Etc.

Static Checking (continued)

- **Syntax-with-params**, given fn, netlist, and params checks the following:
  - Every wire has constant width
  - Instance wire widths correspond with module declarations
  - Each bit of each wire is defined exactly once
  - A wire is completely defined before any part of it is used
- We also have **well-formed-inputs** and **well-formed-finite-state** functions.

Verification and Tool Flow



Verification Example -- Verilog

```
module dt_lsq_dsn_valid_blocks
(valid_block_mask, youngest, oldest, empty);
   output [7:0] valid_block_mask;
   input [2:0]  youngest;
   input [2:0]  oldest;
   input        empty;
   wire [7:0]   youngest_set_up, oldest_set_down;
   wire         youngest_lt_oldest;

   …  …  …
   assign valid_block_mask =
          empty ?              8'd0 :
          youngest_lt_oldest ? youngest_set_up | oldest_set_down :
                               youngest_set_up & oldest_set_down;
endmodule // dt_lsq_dsn
```

Verification Example -- DE

```
(add-module
 (quote (|dt_lsq_dsn_valid_blocks|
 (OUTS (|valid_block_mask| 8))
 (INS (|youngest| 3)
      (|oldest| 3)
      (|empty| 1))
 (WIRES (|youngest_set_up| 8)
        (|oldest_set_down| 8)
        (|youngest_lt_oldest| 1))
 (OCCS
   …  …  …
  (ASSIGN_3 ((|valid_block_mask| 0 7))
            (BUFN 8)
            ((BV-IF (G |empty| 0 0)
                    (BV-CONST 8 0)
                    (BV-IF (G |youngest_lt_oldest| 0 7)
                           (BV-OR 8 (G |youngest_set_up| 0 7)
                                  (G |oldest_set_down| 0 7))
                           (BV-AND 8 (G |youngest_set_up| 0 7)
                                   (G |oldest_set_down| 0 7))))))))))
```

Verification Example – ACL2

```
(defun |acl2-dt_lsq_valid_blocks| (|youngest| |oldest| |empty|)
  (let* (… … …
         (|valid_block_mask|
          (BV-IF (G |empty| 0 0)
                    (BV-CONST 8 0)
                    (BV-IF (G |youngest_lt_oldest| 0 7)
                           (BV-OR 8 (G |youngest_set_up| 0 7)
                                  (G |oldest_set_down| 0 7))
                           (BV-AND 8 (G |youngest_set_up| 0 7)
                                   (G |oldest_set_down| 0 7))))))
    (list |valid_block_mask|)))
```

Verification Example – Abstract ACL2

```
 (defun make_valid_mask (n youngest oldest ans)
  (cond ((zp n) ans)
        ((car (bv-eq 3 youngest oldest))
         (bv-or 8 ans (bv-lshift 8 3 (bv-const 8 1) oldest)))
        (t
         (make_valid_mask
          (1- n) youngest (increment 3 oldest)
          (bv-or
           8 ans (bv-lshift 8 3 (bv-const 8 1) oldest))))))

(defun valid_blocks (youngest oldest empty)
  (if (car empty)
      (bv-const 8 0)
    (make_valid_mask 8 youngest oldest (bv-const 8 0))))
```

Verification Example (theorems)

```
(defthm dt_lsq_dsn_valid_blocks-se-rewrite
  (implies (|dt_lsq_dsn_valid_blocks-&| netlist)
           (equal
            (se flg '|dt_lsq_dsn_valid_blocks| params
                ins st env netlist)
            (|acl2-dt_lsq_valid_blocks|
             (get-value flg ins env)
             (get-value flg (cdr ins) env)
             (get-value flg (cddr ins) env))))
  :hints (("Goal" :in-theory
                  (e/d (|dt_lsq_dsn_valid_blocks-EXPAND|
                        |dt_lsq_dsn_valid_blocks-&|)
                        ())))))

(thm (car (bv-eq 8 (valid_blocks youngest oldest empty)
                 (car (|acl2-dt_lsq_valid_blocks|
                       youngest oldest empty))))
     :hints (("Goal" :sat nil)))
```

Current System Summary

- Current System:
  - DE semantics and static checkers
  - Verilog to DE compiler
  - First-pass rewriting book
  - SAT decision procedure integrated with (my) ACL2
- (Near-term) Plans:
  - Verilog to low-level ACL2 compiler
  - Improved rewriting book
  - SAT-based Equivalence Checker
  - Proof of larger TRIPS components

Issues

- Combining the finite theorems with ACL2 is still a bit cumbersome.
- Is Embedded the right approach?
- Is SAT in the right place?
- Can we make DE more compact?
- Should we put more power in the parameters?
  - Could we then eliminate the need for defining ACL2 functions for each Synopsis primitive?
- Are we handling state right?
- Is the type system too limiting?

Conclusion

- DE is a simple HDL embedded in ACL2.
- We can implement complex hardware in it.
- We can quickly verify small hardware in it.
- We hope to verify complex hardware using a hierarchical approach.
- It now has a strong parameterized type system.