Mechanized Certification of Secure Hardware Designs

Sandip Ray
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712. USA.
sandip@cs.utexas.edu

Warren A. Hunt, Jr.
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712. USA.
hunt@cs.utexas.edu

Abstract— **We develop a framework for mechanized certification of secure hardware systems built out of commercial off-the-shelf (COTS) components purchased from untrusted vendors. Certification requires a guarantee that the fabricated system satisfies the requisite safety and security properties. Our framework facilitates this by (1) providing an unambiguous description of the requirements specification in a formal, computational logic, (2) a formalized hardware description language (HDL) to describe the implementation, and (3) mechanical tools and techniques for providing a certification of correctness and security. We illustrate the use of the framework in certifying the correctness and security properties of the netlist implementation of a voting machine using the ACL2 theorem prover.**

I. Introduction

Society today is dependent on computing systems built out of commercial off-the-shelf (COTS) components for critical applications. It is therefore crucial that such systems perform in a secure and reliable manner. Consequently there has recently been significant interest, both in the government and in the industry, in developing mechanized tools for certification of security-critical systems built out of COTS components.

Certification of a COTS system requires interaction between designers, consumers, and regulatory evaluators to guarantee that the fabricated system satisfies the requisite safety and security properties. To achieve this, one must have (1) an unambiguous specification of the system requirements, (2) a formal semantics of the language in which the system is implemented, and (3) mechanized mathematical tools to assist the analyst in the certification process.

We are developing a framework, based on machine-assisted formal reasoning, to facilitate mechanized certification of secure hardware designs. Our motivation is to provide the following key ingredients:

- A formal mathematical logic as a basis for requirements specification.
- A formalized Hardware Description Language (HDL) as a means for describing hardware implementation.
- Use of scalar and symbolic simulation, and automated formal analysis, on the same formal artifact.

In this paper we describe different facets of our framework, using, as a simple illustrative example, its application in the mechanized certification of the netlist implementation of a voting machine. The process illustrates the nature of mechanized infrastructural support necessary for the certification of security-critical hardware designs.

The formal foundational basis for our framework is provided by the ACL2 system [12]. ACL2 is a general-purpose theorem prover based on an applicative subset of Common Lisp. ACL2 has been successfully used in the formal analysis of a slew of computing systems, ranging from pipelined microprocessors to JVM byte codes [18], [20], [22], [16]. In our framework we make critical use of the mechanical reasoning engine of ACL2, and in particular its support for efficient function execution which facilitates validation of the formal models by simulation. However, the key ideas are independent of the nuances of a specific theorem proving system.

The rest of the paper is organized as follows. In Section II, we provide a brief overview of the ACL2 logic and theorem prover. Section III deals with the development of specification of a hardware device. In Section IV, we discuss a formalized hardware description language (HDL) called **DE**, and show how it can be used to succinctly and unambiguously describe hardware implementations. In Section V, we describe the key analysis steps in a certification, pointing out the respective roles of simulation and formal reasoning; we also discuss how our approach affords effective computation of information flow properties of the system. We conclude the paper in Section VI. A modicum of familiarity with Lisp is assumed in the presentation. However, no previous familiarity with ACL2 is required; the relevant aspects of ACL2 are mentioned in Section II.

II. Overview of ACL2

In this section, we briefly describe the ACL2 logic. This provides a formal notational and reasoning framework to be used in the rest of the paper. We refer the reader interested in a thorough understanding of ACL2 to the ACL2 Home Page (`http://www.cs.utexas.edu/users/moore/acl2`), which contains an extensive hypertext documentation and pointers to several published books and papers.

ACL2 is a first-order logic of recursive functions. The inference rules constitute propositional calculus with equality and instantiation, and well-founded induction up to ϵ_0. The language is an applicative subset of Common Lisp; instead of writing $f(a)$ as the application of function f to argument a, one writes `(f a)`. Terms are used instead of formulas. For example, the following term represents a basic fact about lists in the ACL2 syntax.

```
(implies (natp i)
         (equal (nth i (update-nth i v l))
                v))
```

The syntax is quantifier-free; formulas may be thought of as universally quantified over all free variables. The term above specifies the statement: "For all i, v and l, if i is a natural number, then the i-th element of the list obtained by updating the i-th element of l by v is v."

ACL2 provides axioms to reason about Lisp functions. For example, the following axiom specifies that the function `car` applied to the `cons` of two arguments, returns the first argument of `cons`.

Axiom:
```
(equal (car (cons x y)) x)
```

The Lisp axioms of ACL2 together constitute the ACL2 *Ground Zero Theory* ($\mathcal{GZ}$ for short). $\mathcal{GZ}$ characterizes about 170 functions described in the Common Lisp Reference Manual [23], which are (i) free from side effects, (ii) independent of the state or other implicit parameters or data types other than those supported by ACL2, and (iii) unambiguously specified on their intended domains in a host-independent manner. The return values predicted by the axioms agree with those specified in the Common Lisp Manual for arguments in the intended domains.

Theorems can be proved for axiomatically defined functions in the ACL2 system. Theorems are proved by the `defthm` command. For example, the command:

```
(defthm car-cons-for-2
  (equal (car (cons x 2)) x))
```

directs the theorem prover to prove that for every `x`, the output of the function `car` applied to the `cons` of `x` and the constant `2`, returns `x`.

ACL2 provides *extension principles* that allow the user to introduce new function symbols and axioms about them. The extension principles constitute (i) the *definitional principle* to introduce total functions, (ii) the *encapsulation principle* to introduce constrained functions, and (iii) the *defchoose principle* to introduce Skolem functions. We briefly sketch these principles here.[1] Kaufmann and Moore [14] present a detailed description of these principles along with their soundness arguments. Any ACL2 theory is an extension of $\mathcal{GZ}$ through applications of the extension principles.

Definitional Principle:: The *definitional principle* allows the user to define new total functions. For example, the following form defines the factorial function `fact` in ACL2.

[1] ACL2 has another extension principle, namely the *defaxiom principle*, which permits specifying any formula in the current theory as an axiom. The use of this principle is discouraged since can lead to inconsistent theories. We ignore defaxiom principles in our framework.

```
(defun fact (n)
  (if (zp n)
      1
    (* n (fact (- n 1)))))
```

The effect is to extend the logic by the following *definitional axiom*:

Definitional Axiom:
```
 (fact n)
=
 (if (zp n) 1 (* n (fact (- n 1))))
```

Here `(zp n)` returns `nil` if `n` is a positive natural number, and otherwise `T`. To ensure consistency, ACL2 must prove that the recursion terminates [5]. In particular, one must exhibit a "measure" m that maps the set of arguments in the function to some set W, where $\langle W, \prec \rangle$ forms a well-founded structure. The proof obligation, then, is to show that on every recursive call, this measure "decreases" according to relation $\prec$. ACL2 axiomatizes a specific well-founded structure, namely the set of ordinals below ϵ_0: membership in this set is recognized by an axiomatically defined predicate `o-p`, and a binary relation `o<` is axiomatized in the logic as an irreflexive partial order in the set.

Encapsulation Principle:: The *encapsulation principle* allows the extension of the ACL2 logic with partially defined constrained functions. For example, the command below introduces a function symbol `foo` with the constraint that `(foo n)` is a natural number.

```
(encapsulate
  (((foo *) => *))
  (local (defun foo (n) 1))
  (defthm foo-returns-natural
    (natp (foo n))))
```

Consistency is ensured by showing that some (total) function exists satisfying the alleged constraints. In this case, the constant function that always returns `1` serves as such "witness". The effect is to extend the logic by the following *encapsulation axiom* corresponding to the constraints. Notice that the axiom does not specify the value of the function for every input.

Encapsulation Axiom:
```
(natp (foo n))
```

For a constrained function f the only axioms known are the constraints. Therefore, any theorem proved about f is also valid for a function f' that also satisfies the constraints. More precisely, call the conjunction of the constraints on f the formula ϕ. For any formula ψ let $\hat{\psi}$ be the formula obtained by replacing the function symbol f by the function symbol f'. Then, a derived rule of inference, *functional instantiation* specifies that for any theorem θ one can derive the theorem $\hat{\theta}$ provided one can prove $\hat{\phi}$ as a theorem. In the example, since the constant `10` satisfies the constraint for `foo`, if `(bar (foo n))` is provable for some function `bar`, functional instantiation can be used to prove `(bar 10)`.

Defchoose Principle:: The *defchoose principle* allows introduction of Skolem functions in ACL2. To understand this principle, assume that a function symbol `P` of two arguments has been introduced in the ACL2 logic. Then the form:

```
(defchoose exists-y-witness y (x)
  (P x y))
```

extends the logic by the following axiom:

Defchoose Axiom:
```
(implies (P x y)
         (P x (exists-y-witness x)))
```

The axiom states that *if* there exists some `y` such that `(P x y)` holds, then `(exists-y-witness x)` returns such a `y`. Nothing is claimed about the return value of `(exists-y-witness x)` if there exists no such `y`. This provides the power of first-order quantification in the logic. For example, we can define a function `exists-y` such that `(exists-y x)` is true if and only if there exists some `y` satisfying `(P x y)`. Notice that the theorem `exists-y-suff` below is an easy consequence of the defchoose and definitional principles.

```
(defun exists-y (x)
  (P x (exists-y-witness x)))

(defthm exists-y-suff
  (implies (P x y) (exists-y x)))
```

ACL2 provides a construct `defun-sk` that makes use of the defchoose principle to introduce explicit quantification. For example, the form:

```
(defun-sk exists-y (x)
  (exists y (P x y)))
```

is merely an abbreviation for the following forms:

```
(defchoose exists-y-witness y (x)
  (P x y))

(defun exists-y (x)
  (P x (exists-y-witness x)))

(defthm exists-y-suff
  (implies (P x y)
     (exists-y x)))
```

Thus `(exists-y x)` can be thought of specifying as the first-order formula: `(∃y : (P x y))`. Further, `defun-sk` supports universal quantification `forall` by exploiting the duality between existential and universal quantification.

A. Executability in ACL2

ACL2 provides strong support for executing functions introduced through the definitional principle. To support this, the invocation of a definitional principle in ACL2 entails performing several operations, in addition to introducing the definitional axiom. In particular, a new function symbol is defined (and generally compiled) in the host Common Lisp. For example, the above `defun` for `fact` is executed directly in Common Lisp. We refer to this definition as the *executable counterpart* of `fact`. Because Common Lisp is a model of the ACL2 axioms, ACL2 may exploit the Common Lisp counterpart and the host Lisp execution engine as follows: when a ground application of the defined symbol arises during the course of a proof or when the user submits a form to ACL2, its value under the axioms may be computed with the Common Lisp counterpart in the host Lisp. For example, should `(fact 15)` arise in a proof, ACL2 can use the Common Lisp counterpart of `fact` to compute `1307674368000` in lieu of deriving that value by repeated reductions using instantiation of the definitional axioms.

The story above is made somewhat more subtle by the fact that the Common Lisp functions are partial, while ACL2 functions are total. For instance, in Common Lisp, `(car 7)` is undefined while in the ACL2 logic the value is provably `NIL`. ACL2 provides a mechanism, called *guards* [13] to enable the use of the Common Lisp counterpart only on ground terms where the arguments for each function f are in the intended domain of application of f. ACL2 contains contains several other constructs to support efficient executability, such as (1) single-threaded objects [6], and (2) mbe [9]. Single-threaded objects enable destructive updates to certain data structures in an applicative context. `Mbe` (or `must-be-equal`) allows the user to attach different function bodies to the same function definition; one body is used for logical reasoning and the other for executability, and the user proves the logical equality of their return values using the theorem prover.

III. Specification

The first crucial step in the development of a certifiable design is the definition of its *specification*. Unfortunately, in current practice, little attention is given to making the specification formal or unambiguous: system requirements are typically described with charts and diagrams, together with ambiguous English. In this section we will discuss an informal specification of the requirements of a voting machine, point out the the inadequacy of such a description as a basis of certification, and then show how to refine such descriptions to a formal specification that can be mechanically analyzed.

An informal description of the voting machine is as follows. The machine has a counter for each candidate. At any instant it has status `:ready`, `:locked`, or `:frozen`, and responds to the following user actions:

- At the `:ready` state, the voter performs a `:vote` action to tentatively select a candidate. The system records the vote, but does not change state.
- The voter can change her mind by performing `:reset`. This clears the tentative selection above.
- Once the `:commit` action is selected, the system records the vote and transits to the state `:locked`.
- The `:unlock` action is performed by a polling official after a vote has been cast and the voter left. The system changes state from `:locked` to `:ready`.

- The `:freeze` action is performed when polling is completed. The machine then provides a tally of votes.

The above sounds like a reasonable description of the specification of a voting machine. Nevertheless, it is not hard to find omissions. For instance, what should happen if `:unlock` is performed when the machine is `:ready`? We tacitly assumed that `:unlock` occurs only in the `:locked` state. This requires that (i) the voter does not leave without casting a vote, or (ii) if she does then the polling official does not unlock the machine.

To avoid omissions, we define specifications operationally with (i) the initial state of the machine, and (ii) a state transition function (spec s i) which defines the next state for each state s and input i. Attempting to formalize the above description immediately detects our omission: since ACL2 functions are total, `spec` must define the next state when `:unlock` is performed when the machine is `:ready`. We therefore refine the specification by stipulating that in this case the machine clears the tentative votes of the undecided voter. We further stipulate that if it encounters an "unexpected" input at any state, no change of state occurs. A fragment of our formalized `spec` function with these stipulations is shown in Fig. 1.

A specification requires consideration of the possible system behaviors and involves several design choices: instead of rejecting an unexpected input, an alternative could be for the machine to transit to an error state. Note that we can use encapsulation in ACL2 so that `spec` is defined only for expected inputs. While this approach is sometimes convenient, we prefer executable definitions whenever possible, since it facilitates validation of the specification via simulation.

The specification above is defined as a state machine rather than by formulas representing properties of the implementation. In addition to simulation, this affords intuitive specifications in practice. Most implementations are elaborations of simpler protocols to achieve execution efficiency, match a given architecture, etc. The simpler protocol then succinctly captures the behaviors of the elaboration. On the other hand, most modern systems are *reactive* and their properties are naturally described in a temporal logic. Defining such formulas thus requires a semantic embedding of temporal logic, which is cumbersome because of the first-order nature of ACL2 [19]. However, to use operational specifications, we must additionally formalize a notion of correspondence between the state machines. We address this in Section V.

```
(defun s-init ()
  (>_ :status :ready
      ...))

(defun spec (s i)
  (let ((satus (status s))
        (c0 (candidate0 s))
        (c1 (candidate1 s))
        (opcode (opcode i)))
  (case opcode
    (:vote
      (case status
        (:ready
          (case (candidate i)
            (0 (>s :tvote0 1
                   :tvote1 0))
             ...))
          ...
        (t s)
    (:commit
      (case status
        (:ready
          (>s :candidate0
              (+ c0 (tvote0 s))
              :candidate1
              (+ c1 (tvote1 s))
              :status :locked))
        ....
        (t s)))
   (:unlock
     (case status
       (:ready (>s :tvote0 0
                   :tvote1 0))
        ....
       (t s)))
   (:freeze
     (>s :status :frozen
         :tally ...))
   (t s))))
```

Fig. 1. Fragment of a Voting Machine Specification. Here we use the ACL2 records book [15] to update and access machine components; `(status s)`, `(opcode i)`, etc., are accessors, `>s` is a macro for updating fields of record `s`, and `>_` updates the empty record.

IV. Implementation

Having described the specification of our voting machine, we turn now to our approach to describing the implementation in a way that affords mechanized certification.

In order to certify that an implementation satisfies a specification, the implementation must be represented in a language with formal and unambiguous semantics. However, in practice, hardware designs are typically implemented in some commercial Hardware Description Language (HDL) such as VHDL [4] and Verilog [24]. These HDLs need to satisfy several disparate goals other than formal verification, namely ease of use, simulation speed, etc. As a result, most commercial HDLs are large, unwieldy, and in parts poorly specified [21]. Therefore, formal analysis of a hardware design written in a commercial HDL has been traditionally restricted to some alternative encoding of the underlying algorithm written (typically by a human) in some formal language. The utility of such a verification then rests upon the assumption that the encoding faithfully reflects the actual implementation.

Our solution to this problem is the development of the **DE** language [11].[2] **DE** is a hierarchical, occurrence-oriented HDL with a formal semantics defined by a deep embedding in the logic of ACL2.

Figure 2 shows a fragment of the netlist representation of our voting machine. Note that the netlist is represented as a constant (declared by the `defconst` construct) in the ACL2 logic. The name of the constant is `*vnlst*`. The netlist has five modules `vote`, `status`, `cmtvote`, `4-bit-ctr`, and `1-bit-ctr`. A module has input and output wires, state holding elements, and a set of occurrences; module `cmtvote` has three inputs (`candidate`, `commit`, and `reset-`), eight outputs (`out00`, `out01`, etc.), two state elements (`vote0` and `vote1`), and five occurrences (`vote0`, `vote1`, `g0`, `g1`, and `g2`). Some modules like `and`, `not`, etc., are *primitive*. In other modules, *occurrences* describe connections by instantiating other modules: in `cmtvote`, `g2` represents connection of (input) wire `candidate` and (internal) wire `ncandidate` by instantiating the `not` module. The top module `vote` instantiates two modules `status` and `cmtvote`; `vote` has input bits `op0`, `op1`, and `op2` encoding user actions, and a `candidate` input. The `status` module updates the status values **:ready**, **:locked**, etc., encoded in two state bits. Module `cmtvote` updates vote counts. Counting is done using 4-bit counters for demonstration purposes.

We now discuss the semantics of the **DE** language. The semantics is provided by defining a formal language interpreter in the ACL2 logic. The interpreter functions `se` and `de` are shown in Fig. 3. Function `se` returns the outputs of a module `fn` of a netlist `n` as a function of its inputs and state elements, and `de` returns the next state. Here `primp` determines if `fn` is a primitive module; `se-primp-apply` and `de-primp-apply` are primitive module evaluators; `se` crawls over the module structure recursively evaluating each signal occurrence and finally filtering the outputs; `de` performs a second pass to evaluate the next states. Note that unlike commercial HDLs, **DE** has a compact semantics: the above definitions together with the primitive evaluators constitute the *entire* language definition. The regularity and economy of **DE** makes it suitable for mechanically analyzable hardware implementations.

V. Analysis and Discussions

The key analysis step involves showing that the executions of the netlist satisfy the specification. Since the specification itself is a state machine, formalizing this requires a notion of correspondence between two state machine executions.

We formalize correspondence with functions `rep`, `good`, `pick`, and `inv` so that the formulas in Fig. 4 are theorems. The theorems imply that every `good` execution of the implementation is matched by the specification and essentially formalize the notion of *trace containment* [2] in ACL2, where

[2]The **DE** language is the successor of the DUAL-EVAL HDL [7], and is an evolving project [10], [11]. The version of **DE** used in the analysis described here is called **DE4**. Recently, Boyer and Hunt have developed a version of the language called **E**, with a number of sophisticated analysis capabilities.

```
(defconst *vnlst*
  '((vote
     (op0 op1 op2 candidate)
     (sout0 sout1
      out00 out01 out02 out03
      out10 out11 out12 out13)
     (votes stat)
     ((stat (sout0 sout1)
            status
            (op0 op1 op2))
      (votes (out00 out01 out02
               out03 out10
               out11 out12 out13)
              cmtvote
             (candidate commit reset))
      ...))
    (status
     (op0 op1 op2)
     (sout0 sout1)
     (s0 s1)
     (...))
    (cmtvote
     (candidate commit reset-)
     (out00 out01 out02 out03
      out10 out11 out12 out13)
     (vote0 vote1)
     ((vote0 (out00 out01 out02 out03)
              4-bit-ctr
              (commit0 reset-))
      (vote1 (out10 out11 out12 out13)
              4-bit-ctr
             (commit1 reset-))
      (g2 (ncandidate)
           not
          (candidate))
      (g0 (commit0)
          and
          (commit ncandidate))
      (g1 (commit1)
          and
         (commit candidate))))
    (4-bit-ctr
      (incr reset-)
      (out0 out1 out2 out3)
      (h0 h1 h2 h3)
      ((h0 (out0 carry0)
           1-bit-ctr
           (incr reset-))
       (h1 (out1 carry1)
           1-bit-ctr
          (carry0 reset-))
       (h2 (out2 carry2) ...)
       (h3 ...)))
    (1-bit-ctr ...)))
```

Fig. 2. Fragment of the netlist representation of a voting machine.

```
(mutual-recursion
 (defun se (fn ins sts n)
  (if (primp fn)
      (se-primp-apply fn ins sts)
    (let ((m (assoc-eq fn n)))
    (if (atom m) nil
      (assoc-eq-values
       (md-outs m)
       (se-occ (md-occs m)
               (pairlis$ md-ins ins)
               (pairlis$ md-sts sts)
               (delete-eq-module
                 fn n)))))))

 (defun se-occ (occs w-alst s-alst n)
  (if (endp occs) w-alst
   (let* ((occ (car occs))
          (ins (assoc-eq-values
                (occ-ins occ)
                w-alst))
          (sts (assoc-eq-value
                (occ-name occ)
                s-alst)))
    (se-occ (cdr occs)
      (append
        (pairlis$ (occ-outs occ)
                  (se (occ-fn occ)
                      ins sts n)
        w-alst)
      sts n)))))

(mutual-recursion
 (defun de (fn ins sts n)
  (if (primp fn)
      (de-primp-apply fn ins sts)
   (let ((m (assoc-eq fn netlist))
         (n-n (delete-eq-module fn n)))
    (if (atom m) nil
      (assoc-eq-values md-sts
        (de-occ (md-occs m)
          (se-occ
            (md-occs m)
            (pairlis$ (md-ins m) ins)
            (pairlis$ (md-sts m) sts)
            n-n)
          (pairlis$ (md-sts m) sts)
        n-n)))))

 (defun de-occ (occs w-alst s-alst n)
  (if (endp occs) w-alst
    (let* ((occ (car occs))
          (ins (assoc-eq-values
                 (occ-ins occ)
                 w-alst))
          (sts (assoc-eq-value
                (occ-name occ)
                s-alst)))
    (de-occ (cdr occs)
          (acons (occ-name occ)
                 (de (occ-fn occ)
                     ins sts n)
                 w-alst)
          s-alst n)))))
```

Fig. 3. Definition of Semantics for **DE** Language

```
(defthm rep-matches
 (and
   (equal (rep *init*)
          (s-init)))
   (implies
     (and (inv s)
          (good s i))
     (equal
      (rep (de 'vote s i *vnlst*))
      (spec (rep s) (pick i)))))

(defthm inv-invariant
  (and
    (inv *init*)
    (implies
      (and (inv s)
           (good s i))
      (inv (de 'vote s i *vnlst*)))))
```

Fig. 4. Theorems showing that the netlist implementation of the voting machines is a refinement of `spec`. Here `*init*` is the valuation of the state elements at the initial state, `rep` maps the design states to specification states, `pick` is the input mapping, `inv` is an invariant on the design, and `(good s i)` checks if `i` is a valid design input at state `s`.

containment is restricted to `good` traces.[3]

Note that the specification needs to match the implementation *only* for `good` transitions. Contrast this with our approach of defining `spec` as a total function. While we could similarly insist that the specification must match *each* implementation step, this often complicates definitions. For instance, `spec` uses unbounded additions above while `*vnlst*` uses 4-bit counters. Modifying `spec` to use bounded arithmetic would complicate its definition, and furthermore, the definition would no longer be applicable if we re-design the netlist with (say) 64-bit counters. We prefer generic specifications and use `good` to impose input constraints.

Proving the above theorems is a two-step process. The first step is what is referred to as *semantic simplification*. In this step, we define functions in ACL2 (called *semantic functions*) that mimic the workings of each module, and prove theorems relating the `se` and `de` expressions with these functions. Fig. 5 shows the theorems for the module `4-bit-ctr`.

The theorems relating `se` and `de` expressions are proven hierarchically. Since `4-bit-ctr` instantiates `1-bit-ctr`, we first prove analogous theorems for the latter; the theorems shown in Fig. 5 are then proven by symbolic expansion of `se` and `de` functions and applying the `1-bit-ctr` theorem for the corresponding occurrence. The process can be automated with Lisp macros [11].

In the second step we define `rep`, `good`, `pick`, and `inv`. The first three definitions are typically easy; for instance, `rep` maps the bit configurations of state elements `stat` and `votes` to keyword-based status values and numerical vote

[3]It is sometimes more convenient to use trace containment under *stuttering* to relate to machines at different abstractions [17]. We do not discuss stuttering in this paper.

```
(defun 4btnt (n)
  (and
   (equal (assoc-eq '4-bit-ctr n)
          '(4-bit-ctr (incr reset-)
                      ...))
  (1btnt
   (delete-eq-module '4-bit-ctr n))))

(defthm 4-bit-ctr-se-eval
  (implies
    (4btnt n)
    (equal (se '4-bit-ctr ... n)
           ...)))
(defthm 4-bit-ctr-de-eval
  (implies
    (4btnt n)
    (equal (de '4-bit-ctr ... n)
            ...)))
```

Fig. 5. Semantic simplification of `4-bit-counter` module. The "`...`" at the right hand side of each equality contains an ACL2 semantic function for the behavior of the module.

counts. Theorem `rep-matches` requires showing correspondence between single steps of two machines; by virtue of our having performed the first step, proving this does not involve reasoning about the **DE** semantics. A harder problem in practice is defining `inv` and proving `inv-invariant`; the theorem shows that `inv` is an inductive invariant, and allows us to assume `inv` in the proof of `rep-matches`. The standard approach is to define a predicate `suff` that is sufficient to prove `rep-matches`; we then incrementally strengthen `suff` to an inductive invariant. However, since netlists are *finite* state machines, decision procedures can be used to derive such invariance, and recent work integrating **DE** with SAT-solvers [11] provides substantial automation.

The definitions of `rep`, `good`, and `pick` are integral parts of system specification. For purposes of certifying the implementation, it is convenient to view them as "usage instructions" supplied by a seller of a component to augment the specification provided by the buyer. For instance `rep` describes how the valuations of the netlist state elements are viewed as abstract states. Certification then requires us to (i) check the validity of the theorems above (possibly with assistance from the seller), and (ii) validate that the usage instructions do indeed correspond to the environments in which the design is deployed. Note that it is possible to have buggy implementations satisfying the theorems above, under the wrong definitions of `rep`, `good`, etc. To illustrate this, consider a machine that clears the votes of all candidates when encountering the sequence **:vote**, **:reset**, **:reset**. If the predicate `good` specifies that such a sequence does not occur then it is possible to "verify" the implementation, although the implementation is obviously buggy. The problem is that the environmental assumptions used in the verification (in particular the predicate `good`) are falsified in the deployment environment. Checking such violations involves simulation of both the usage functions (in this case the definition of input constraint `good`) and the definition of `spec`.

We now consider regulatory checks. Regulatory checks are different from functional correctness, for instance requiring the guarantee of of privacy, absence of trapdoors, etc. One can (and often does) apply theorem proving to prove such properties. However, sometimes we can do automatic checks of structural or information-flow properties by computation.

An information-flow property we prove using computation is that the votes of one candidate do not affect those of the other. To check this property, we need a cone-of-influence analysis. **DE** facilitates such analysis by the following observation.[4] Note from the definitions of `se` and `de` that the core interpreter semantics is given by the primitive evaluation functions `se-primp-apply` and `de-primp-apply`; the remainder of the definitions involves recursive crawling over the netlist. Thus we can define a different interpreter by modifying the primitive evaluators. For cone-of-influence, we modify them to return a list of the *state bits necessary for evaluation* rather than the evaluation itself; the check then involves recognizing that all state bits are included in the evaluation of `votes`. The same function is used to show that there is no "hidden state" in the netlist that does not pertain to vote counts. This guarantees the absence of trapdoors.

VI. Conclusion

Developing COTS systems in current practice contain a number of informal components, namely requirement description as text, graphs, and charts, implementations in languages with incomplete or complicated semantics, and incomplete testing as the primary validation procedure. This does not afford a repeatable, mechanical means to verify the security and correctness of a delivered design. Recently there has been interest in a uniform formal framework to guarantee high assurance in correct and secure system executions. The Common Criteria [1] requires a uniform *lingua franca* for communication of designers, consumers, and evaluators. Rockwell Collins has used ACL2 to achieve the highest level of assurance (EAL7) provided for by the Common Criteria in the AAMP7$^{\text{TM}}$ processor design [8]. We have found that ACL2 is well-suited to serve as a mechanized framework for designing high-assurance systems for several reasons. The language of ACL2 is a programming language, namely Applicative Common Lisp, which facilitates implementation of different analysis tools in the same formal framework; secondly, the logic has high execution support; third, the theorem prover has been extensively used in the verification of systems at different levels of abstraction [3]. However, we believe that it is possible to port the framework to any other theorem prover that provides strong support to executability and symbolic rewriting.

[4]Currently, in the **E** language, this observation has been used to develop different built-in interpreters of the same module, including information-flow interpreter discussed here.

We have illustrated an approach to mechanize the different facets of mechanized certification of the implementation of a security-critical artifact in ACL2. The formal language provides a basis for unambiguous communication among the different parties. Deep embedding enables the use of different analysis tools to be applied to the same design artifact, namely a netlist, within the same formal system. Executable specifications and usage functions afford easy requirements validation via simulation. Refinements facilitate compositional proofs of correspondence via single-step theorems. Note that all these individual steps have been extensively studied by the formal methods community; our approach shows how to effectively orchestrate the steps in increasing assurance in correct executions of highly secure systems.

The definition of our framework is under development. In future work, we plan to provide more automation in specification design. We are also planning to apply the paradigm to design integrity checks on binary code developed for practical machine architectures.

VII. Acknowledgements

This material is based upon work supported in part by DARPA and the National Science Foundation under Grant No. CNS-0429591. We thank Matt Kaufmann for several illuminating comments and suggestions in course of this research.

References

[1] Common Criteria for Information Technology Security Evaluation. See URL: `http://csrc.nist.gov/cc/CC-v2.1.html`.

[2] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[3] W. R. Bevier, W. A. Hunt, Jr., J S. Moore, and W. D. Young. An Approach to System Verification. *Journal of Automated Reasoning*, 5(4):409–530, December 1989.

[4] J. Bhasker. *A VHDL Primer*. Prentice-Hall, 1992.

[5] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[6] R. S. Boyer and J S. Moore. Single-threaded Objects in ACL2. In S. Krishnamurthy and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *LNCS*, pages 9–27. Springer-Verlag, 2002.

[7] B. Brock and W. A. Hunt, Jr. The Dual-Eval Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor. *Formal Methods in Systems Design*, 11(1):71–104, 1997.

[8] D. Greve, R. Richards, and M. Wilding. A Summary of Intrinsic Partitioning Verification. In M. Kaufmann and J S. Moore, editors, 5*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, November 2004.

[9] D. A. Greve, M. Kaufmann, P. Manolios, J S. Moore, S. Ray, J. L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient Execution in an Automated Reasoning Environment. *Journal of Functional Programming*, To Appear.

[10] W. A. Hunt, Jr. The DE Language. In P. Manlolios, M. Kaufmann, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 119–131, Boston, MA, June 2000. Kluwer Academic Publishers.

[11] W. A. Hunt, Jr. and E. Reeber. Formalization of the DE2 Language. In W. Paul, editor, *Proceedings of the* 13*th Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, LNCS, Saarbrücken, Germany, 2005. Springer-Verlag.

[12] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.

[13] M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic Incorporated (CLI), 1717 West Sixth Street, Suite 290, Austin, TX 78703, 1994.

[14] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[15] M. Kaufmann and R. Sumners. Efficient Rewriting of Data Structures in ACL2. In D. Borrione, M. Kaufmann, and J S. Moore, editors, *Proceedings of* 3*rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*, pages 141–150, Grenoble, France, April 2002.

[16] H. Liu and J S. Moore. Executable JVM Model for Analytical Reasoning: A Study. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines, and Emulators*, San Diego, CA, June 2003.

[17] P. Manolios, K. Namjoshi, and R. Sumners. Linking Model-checking and Theorem-proving with Well-founded Bisimulations. In N. Halbwacha and D. Peled, editors, *Proceedings of the* 11*th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 369–379. Springer-Verlag, 1999.

[18] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Kernel of the AMD5K86 Floating-point Division Algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[19] S. Ray, J. Matthews, and M. Tuttle. Certifying Compositional Model Checking Algorithms in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and J S. Moore, editors, 4*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, Boulder, CO, July 2003.

[20] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-point Multiplication, Division, and Square Root Instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, December 1998.

[21] D. M. Russinoff and A. FLatau. RTL Verification: A Floating Point Multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 161–187. Kluer Academic Publishers, 2000.

[22] J. Sawada and W. A. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification. In O. Grumberg, editor, *Proceedings of the* 9*th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[23] G. L Steele, Jr. *Common Lisp the Language*. Digital Press, 30 North Avenue, Burlington, MA 01803, 2nd edition, 1990.

[24] D. E. Thomas and P. R. Moorby. *The Verilog® Hardware Description Language*. Kluwer Academic Publishers, Boston, MA, 3rd edition, 1996.