Seeking Trust Through
Specification, Verification, Evaluation, and Analysis

A Voting Machine Example

Warren A. Hunt, Jr. and Sandip Ray
Department of Computer Sciences
University of Texas at Austin
1 University Station M/S C0500
Austin, TX 78712-0233.

Email: {hunt,sandip}@cs.utexas.edu
Web: http://www.cs.utexas.edu/users{hunt,sandip}

HCSS 2006

Introduction

How can we increase the the confidence in correct and secure executions of commercial off-the-shelf systems?

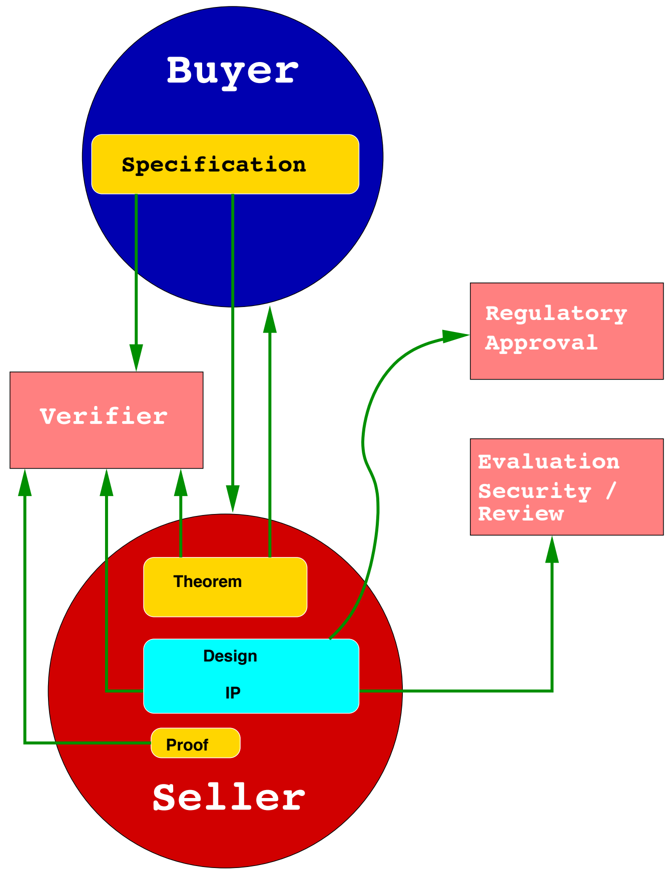
- Design details cannot be disclosed to protect IP.
- Requirement specifications are informal and ambiguous.
- Simulation and testing cannot catch all bugs.
- Security-sensitive designs must conform to further regulatory checks.

Our goal is to have a more formal definition of a buyer/seller process to enable uniform specification and analysis of system designs.

Talk Outline

- **Buyer-Seller Paradigm**
- Defining Specifications
- Formal definitions of netlist implementations
- Mechanical Reasoning
- Validation
- Computational Property Checks
- Concluding remarks

Buyer/Seller Process



The buyer/seller process must unambiguously specify each relationship.

Current Buyer/Seller Process

How does a Buyer get exactly what is desired and nothing else?

- Current specification are text, graphs charts.
- Designs are programs, netlists, IP, etc.
- Some testing.

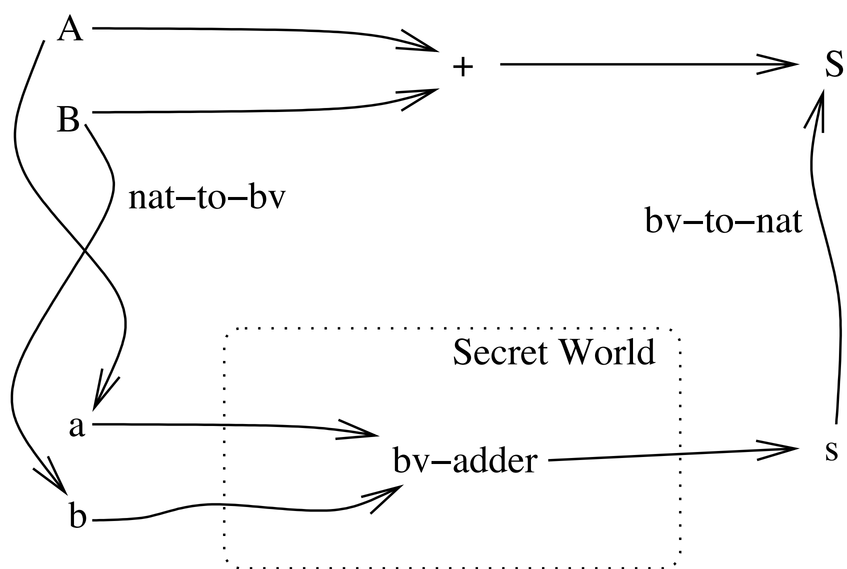
This process, although sometimes considered rigorous, doesn't provide a repeatable, mechanical procedure to verify the suitability, security, and correctness of a delivered design. Nor does this approach provide a rigorous evaluation procedure.

Current Buyer/Seller Process

- **Buyer:** Can you build **(+ x y)** ?
- **Seller:** No. We can build **(rem (+ x y) (expt 2 64))**.
- **Buyer:** Hmm. I guess that is OK. *Later, much later...*
- **Seller:** We have your system. We want it certified. Here is our theorem.

```
(implies
 (and (natp x) (natp y))
 (equal (rem (+ x y) (expt 2 64))
        (bv-to-nat (bv-adder (nat-to-bv x 64)
                             (nat-to-bv y 64))))))
```

Buyer/Seller Adder Diagram



- **Buyer:** What are `bv-to-nat` and `nat-to-bv` ?
- **Seller:** They are the usage instructions.

This diagram also identifies some of the problems when attempting to purchase third-party IP.

New Buyer/Seller Dialogue

- **Buyer:** We want a design that satisfies these terms.

```
(acceptable-design 'bv-adder netlist)

(implies (and (natp x) (natp y))
         (equal (rem (+ x y) (expt 2 64))
                (bv-to-nat (se 'bv-adder
                               (list (nat-to-bv x 64)
                                     (nat-to-bv y 64))
                               nil netlist))))
```

- **Seller:** You are specifying part of our practice.
- **Buyer:** Yes, a third party must be able to mechanically certify our purchase.

Product Acceptance and Regulatory Requirements

- **Buyer:** Since you choose not to reveal your design, we must have a neutral third-party to complete our transaction, especially if it requires evaluation.Therefore, we also want a design that satisfies these two formulas.

```
(security-properties-check 'bv-adder netlist)

(evaluation-properties-check 'bv-adder netlist)
```

- **Seller:** Wow. Can you send us the `check` codes?
- **Buyer:** We will provide you with our `check` codes;
  however, the evaluators may choose to use their own `check` codes
- **Seller:** We will not know when we are done.
  How can we price our product? Have you under-specified?

Our Thesis

A computational logic and mechanical reasoning technology provide a foundation for rigorously capturing different facets of the buyer/seller communication.

- Buyer's specification can be formally defined.
- Seller's implementation can be unambiguously described.
- Mechanical reasoning can be used to verify correspondence.
- Different environmental constraints can be checked by simulation.
- Many regulatory properties can be checked with computation.

Our Thesis

A computational logic and mechanical reasoning technology provide a foundation for rigorously capturing different facets of the buyer/seller communications.

- Buyer's specification can be formally defined.
- Seller's implementation can be unambiguously described.
- Mechanical reasoning can be used to verify correspondence.
- Different environmental constraints can be checked by simulation.
- Many regulatory properties can be checked with computation.

All these operations can be uniformly performed within the same formal framework.

Our Enabling Technology: ACL2

We use ACL2 to model, design, specify, and verify computing systems.

- ACL2 is a formal logic.
  - ▷ we model specifications and implementations in this logic.
- ACL2 has an automated proof checker.
  - ▷ this is used to certify implementation conformance.
- ACL2 is a programming language.
  - ▷ We can simulate formal designs, write analysis tools, etc.

In this talk, we illustrate the application of the buyer/seller process using ACL2 to develop a voting machine netlist design.

Talk Outline

- Buyer-Seller Paradigm
- **Defining Specifications**
- Formal definitions of netlist implementations
- Mechanical Reasoning
- Validation
- Computational Property Checks
- Concluding remarks

Specifications

Traditionally specifications are described in informal English or with charts and diagrams.

The machine is in one of states **:ready**, **:locked**, and **:frozen**, and has a counter for each candidate. It responds to the following user actions:

:vote 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.

:reset The voter can change her mind by performing **:reset**. This clears the tentative selection above.

:commit Once this action is selected, the system records the vote and transits to the state **:locked**.

:unlock The **:unlock** action is performed by a polling official after a vote has been cast and the voter has left. The machine then changes from state **:locked** to **:ready**.

:freeze The **:freeze** action is performed at the end of polling. The machine then provides a tally of votes.

Specifications

Traditionally specifications are described in informal English or with charts and diagrams.

The machine is in one of states **:ready**, **:locked**, and **:frozen**, and has a counter for each candidate. It responds to the following user actions:

:vote 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.

:reset The voter can change her mind by performing **:reset**. This clears the tentative selection above.

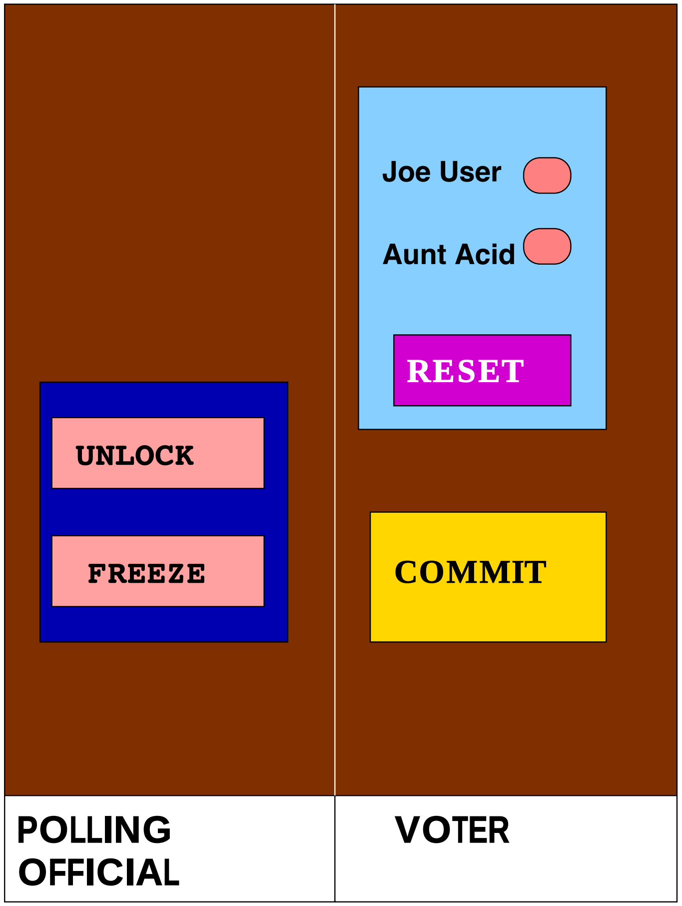
:commit Once this action is selected, the system records the vote and transits to the state **:locked**.

:unlock The **:unlock** action is performed by a polling official after a vote has been cast and the voter has left. The machine then changes from state **:locked** to **:ready**.

:freeze The **:freeze** action is performed at the end of polling. The machine then provides a tally of votes.

Such descriptions are
- ambiguous
- not readily amenable to mathematical reasoning.

Voting Machine Front Panel



Operational Specification Definition

We formalize specifications by a small-step operational model.

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

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

Operational Specification Definition

We formalize specifications by a small-step operational model.

- Precisely describe machine behavior for each state/action combination.
- Specification is executable, allows simulation.
  - ▷ Essential for validation: specifications of complex artifacts are complex.

Talk Outline

- Buyer-Seller Paradigm
- Defining Specifications
- **Formal definitions of netlist implementations**
- Mechanical Reasoning
- Validation
- Computational Property Checks
- Concluding remarks

Netlist Implementations

We formalize netlists via deep embedding.

- Netlists are described as constants in the logic.
- Enables the uniform use of theorem proving and other analysis tools.
- Avoids overloading logic in hardware description.

Traditionally, hardware implementations are described using languages like VHDL and Verilog.

- Any reasonable subset is too complex for effective formalization.

Our answer is the DE Hardware Description Language.

DE Language Features

Hierarchical, occurrence-oriented language

Modules defined hierarchically as a list of sub-module occurrences.

Deeply embedded in ACL2

ACL2 predicate checks syntactically well-formed netlists.

Execution semantics modeled by an interpreter.

Two-pass evaluation

First pass computes the outputs and the second pass computes next states.

Compact Core Semantics Definition

Four ACL2 functions (100 lines of formal definitions).

DE Semantics Definition

Function `se` computes the values of the output wires in a single pass.

```
(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)))))
```

DE Semantics Definition

Function de evaluates the next state in a second pass.

```
(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)))))
```

Voting Netlist in DE

```
(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 oout02 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)
               ...)))
```

Some Observations

Deep embedding allows us to accurately formalize gate-level hardware artifacts.

We use co-simulation to validate the formal model against fabricated designs.

Executability of formulas is essential for this purpose.

DE provides a rich enough language to model interesting hardware while still having simple semantics.

Simplicity is essential in the context of mechanical reasoning.

Talk Outline

- Buyer-Seller Paradigm
- Defining Specifications
- Formal definitions of netlist implementations
- **Mechanical Reasoning**
- Validation
- Computational Property Checks
- Concluding remarks

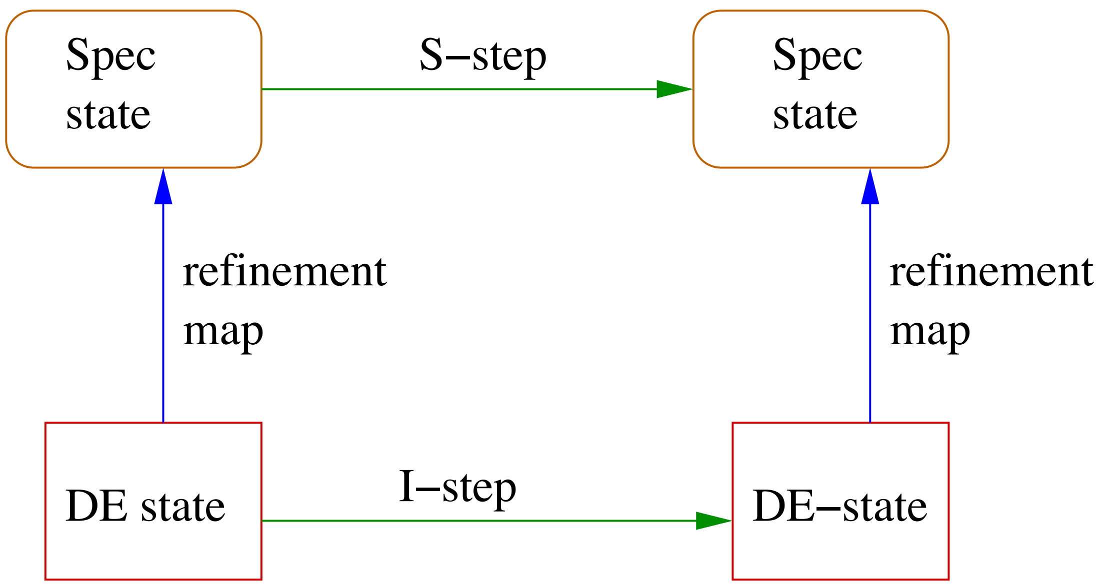
Formal Correspondence

To verify that the netlist satisfies the specification, we need a notion of correspondence between state machines.

We use a simple commutative diagram.

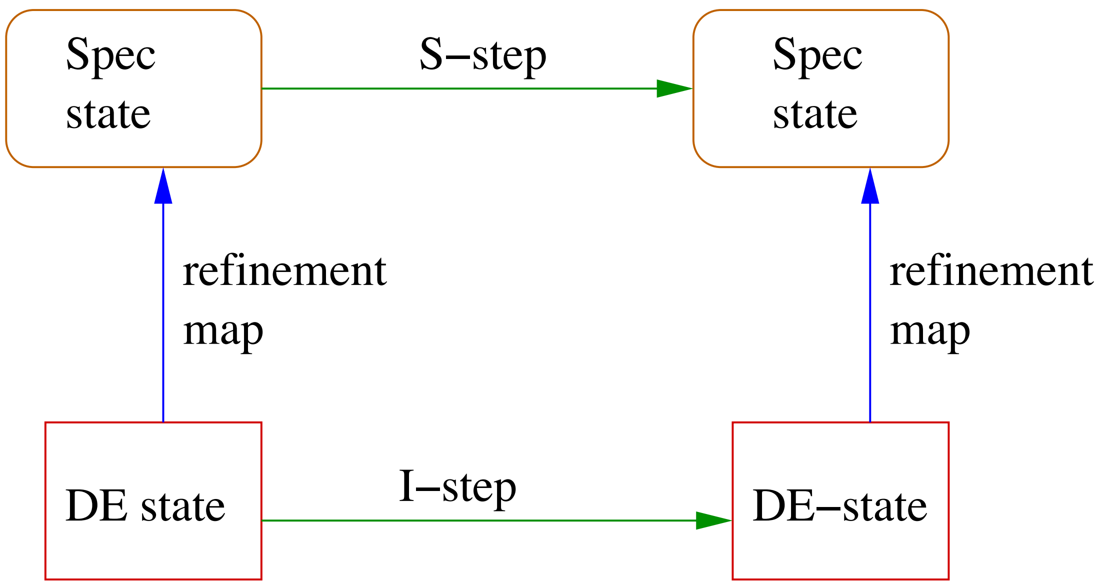
Formal Correspondence

To verify that the netlist satisfies the specification, we need a notion of correspondence between state machines.



Formal Correspondence

To verify that the netlist satisfies the specification, we need a notion of correspondence between state machines.




```
(defthm commutative
  (implies (and (good-state s) (good-input s i))
           (equal (rep (de 'vote s i *vnlst*))
                  (spec (rep s) (map-input i)))))
```

Formal Correspondence

```
(defthm commutative
  (implies (and (good-state s) (good-input s i))
           (equal (rep (de 'vote s i *vnlst*))
                  (spec (rep s) (map-input i))))))
```

The implementation and the specification machines are thus in lock-step for each good input.

`good-state` is a state invariant and `good-input` is an environmental constraint.

The notion of correspondence is generic and uniform.

Used for microprocessor verification, concurrent systems, etc.

[**Note:** In other more general contexts we might need to account for stuttering as well.]

Talk Outline

- Buyer-Seller Paradigm
- Defining Specifications
- Formal definitions of netlist implementations
- Mechanical Reasoning
- **Validation**
- Computational Property Checks
- Concluding remarks

Importance of Validation

Checking specification conformance is not sufficient!

Importance of Validation

Checking specification conformance is not sufficient!

Does the specification reflect the requirements?
Is the refinement mapping correct?
Are the environmental constraints practical?

Importance of Validation

Checking specification conformance is not sufficient!

Does the specification reflect the requirements?
Is the refinement mapping correct?
Are the environmental constraints practical?

These questions can be answered by validation of the formal models by simulation.

- This is the verifier's job (in addition to certification of the theorem).
- To achieve this, executability must be tightly integrated with the formal logic.
  - ▷ ACL2, which is a subset of Common Lisp, provides high-performance execution.

Validation on Voting Machine

Simulation was used to check input constraints.

A simple bug: If `reset` is pressed twice it clears all the votes.

- To prove `commutative`, the predicate `good-input` rules out this case.
  - ▷ Simulation can check if such input constraints are valid.

Note: Practical constraints might be more complex.

To facilitate validation in practice ACL2 provides several features for high performance execution, e.g., single-threaded objects, guards, etc.

- Such features have been used in processor verification by AMD, IBM, Rockwell, etc.

Talk Outline

- Buyer-Seller Paradigm
- Defining Specifications
- Formal definitions of netlist implementations
- Mechanical Reasoning
- Validation
- **Computational Property Checks**
- Concluding remarks

Regulatory Properties

Regulatory properties are of a different nature than functional specifications.

- Does the system contain hidden states?
- Are there trapdoors?
- Does it preserve privacy properties?
- ...

We have the full power of theorem proving to formalize and prove these properties.

- However, in some cases, we want a static property check by computation.
  - ▷ We want to design such checkers within the same formal framework.

A Simple Regulatory Property

The state bits for one candidate of the does not depend on those for the other candidate.

To check this property, we mark, for each state bit, the state bits it depends on.

- Logically this is cone-of-influence reduction.

A Simple Regulatory Property

To check this property, we mark, for each state bit, the state bits it depends on.

The semantics of DE makes it simple to write such checkers.

- Change the primitive evaluators so that instead of evaluating state values they mark the corresponding components.

The property can now be checked by computation.

Talk Outline

- Buyer-Seller Paradigm
- Defining Specifications
- Formal definitions of netlist implementations
- Mechanical Reasoning
- Validation
- Computational Property Checks
- **Concluding remarks**

Conclusion

Formal logic can provide the basis to make the buyer/seller process rigorous and trustworthy.

- Theorem proving, simulation, semi-formal analysis, all have a role.
- All the activities can be performed within a unified framework.

Conclusion

Formal logic can provide the basis to make the buyer/seller process rigorous and trustworthy.

- Theorem proving, simulation, semi-formal analysis, all have a role.
- All the activities can be performed within a unified framework.

The need for a uniform framework is being increasingly realized.

- The *common criteria* is a direct response to this need.

Conclusion

Formal logic can provide the basis to make the buyer/seller process rigorous and trustworthy.

- Theorem proving, simulation, semi-formal analysis, all have a role.
- All the activities can be performed within a unified framework.

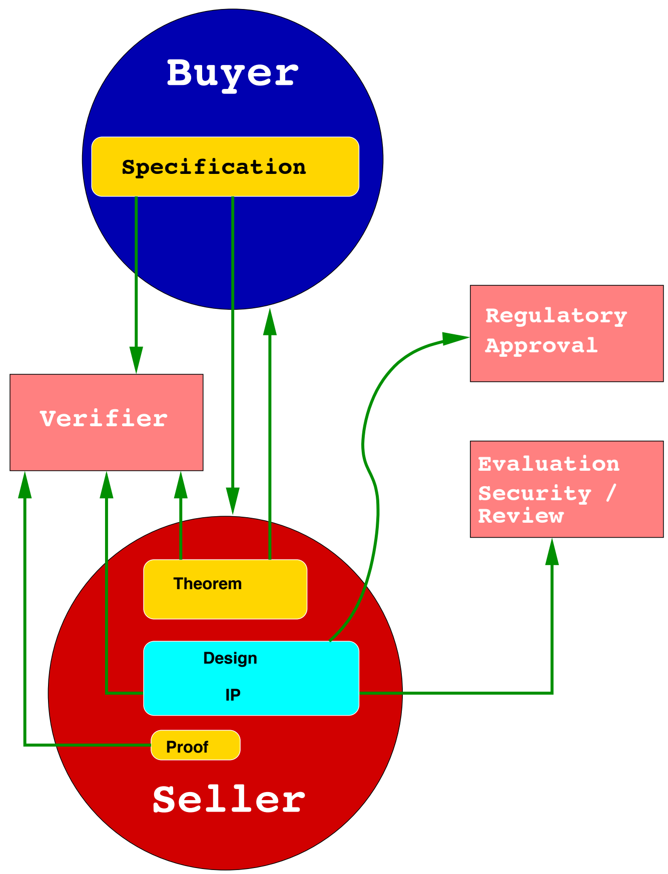
The need for a uniform framework is being increasingly realized.

- The *common criteria* is a direct response to this need.

We believe the process can be facilitated by

- **Formal, operational, executable specifications**
- **Deeply embedded design implementations with formal language semantics.**
- **The use of a *computational logic* to uniformly allow reasoning, simulation, and analysis on the same design artifact.**

Buyer/Seller Process



The buyer/seller process must unambiguously specify each relationship.