A Study of Some Flawed Adders

*** Work In Progress ***

Robert S. Boyer
Department of Computer Sciences
1 University Station, M/S C0500
Austin TX 78712-0233, USA
boyer@cs.utexas.edu

Warren A. Hunt, Jr.
Department of Computer Sciences
1 University Station, M/S C0500
Austin TX 78712-0233, USA
hunt@cs.utexas.edu

ABSTRACT

We are developing analysis and verification mechanisms for models composed of hierarchically specified, cooperating finite-state machines. We have used this system to represent and compare a number of flawed adder specifications to a reference adder specification in an effort to determine the usefulness our tools to analyze flawed-circuit models; this determination is still in progress.

We discuss a mechanism to count the number of different behaviors between two circuits. We present two different kinds of flawed adder specifications, and we count the number of different behaviors between these adders and our reference adder. We are attempting to determine if the number of different behaviors can be a useful metric when attempting to discover subtle, possibly induced, flaws. We also investigate the use of dependency information as another means to determine whether a circuit might have been inadvertently or maliciously altered. The novelty, if any, of our approach lies in our ability to exactly count the number of cases where two Boolean functions differ.

Categories and Subject Descriptors

Hardware Specification [**Formally specified HDL, Fault Analysis**]: ACL2-based Specifications; Circuit Verification [**Mechanical Proofs**]: Fault discovery—*performance measures*

1. INTRODUCTION

As more and more of our supply chain for integrated circuits moves off shore, it is important that we possess the means to ensure that foreign designs and remotely manufactured devices do indeed meet their specifications. We are investigating the use of formal modeling and mechanical verification to this end. We present the use of the ACL2 [15, 14, 12, 11, 13] toolsuite to determine whether various flawed adders seem maliciously altered, just incorrectly designed, or maybe contain internal wires "stuck-at" specific values. This work was originally conceived as a part of our work with the formalization of the **E** language [3]. Our **E**-language effort is a refinement of our previous work on the `DUAL-EVAL` language [5] and the **DE2** language [10]. We are experimenting with different tools in support of formal-methods-based CAD tools [9].

Our approach deeply embeds [1] a BDD system [6] within the ACL2 theorem-proving system in such a manner that it allows a user the means to directly manipulate Boolean functions represented as BDDs. Our ability to include such a BDD system is based on our definition of a Hash CONS operation [7, 8, 16, 17] that we call `HONS`. The logical definition of `HONS` is exactly the same as the pairing function `CONS`, but its implementation ensures that only a single copy of each distinct pair is created [2]. In addition to our definition of `HONS`, we have implemented a mechanism that permits any ACL2 function to be memoized; such a memoization capability allows the standard, recursive version of the Fibonacci function to execute in linear time. It is this feature that allows us to actually count the exact number of differences between two functions represented as BDDs.

Our study is empirical, and we note that this effort is a work in progress. We have investigated several thousand different flawed adders and determined the exact number of incorrect answers that each adder produces. Our study revolves around comparing the 65 output bits of flawed 64-bit adders to the 65 output bits of a known good 64-bit adder. We do this by introducing flaws and then calculate the number of incorrect answers as a function of all 2^{129} possible input configurations. Our use of BDDs to represent Boolean functions is nothing new; however, our ability to count the exact number of differences between Boolean functions represented with BDDs may provide some insight in determining the kind of flaws discovered.

In some cases, we have produced flawed adders that only exhibit a single incorrect answer bit; that is, we have produced adders that are flawed in such a manner that the sum is correct for all of the 2^{129} possible input combinations except for exactly one of the input cases where the answer is incorrect in only one bit position. To find such a difference using simulation alone is practically an impossible task. However, in

*An abbreviated, altered version of this paper with the title Circuit Specification, Abstraction, and Reverse Engineering appeared in the unpublished proceedings of the 2007 High Confidence in Computing Systems conference.

such cases, the input to output information flow is radically different than in other cases where, for instance, a single gate is flawed.

We begin by informally describing the problem we are investigating. We briefly describe our adder specifications and then we outline how we create flawed-adder specifications. We discuss our mechanisms for measuring the differences between our known good adder specification and the various flawed adders, and we present our findings. The technical presentation herein assumes some knowledge of using a formal logic, in this case ACL2, for specifying computations; the findings can be read independently.

2. ADDER SPECIFICATION EQUATIONS

We have defined several recursive functions that operate like variable-width adders. For the purposes of this study, we do not actually generate thousands of different adder netlists, but if necessary we could. Instead, we compare the equations that our different adder specifications represent. We compare the outputs on a bit-by-bit basis with two mechanisms: counting and cone-of-influence information flow.

2.1 Boolean Function Representation and Manipulation

Before we can specify our adder functions, we need a method to represent Boolean equations that we can later compare. We represent Boolean functions as binary trees terminating with `T` or `NIL`. We construct such trees using ACL2's `CONS` pairing operation, where each level of the tree represents a variable. For example, consider the disjunction of `A` and `B`. Actually, we use the `HONS` pairing operation, which is logically identical to `CONS`; ACL2's implementation guarantees that only one copy of each unique `HONS` pair actually exists.

```
   O    <-- A       ;  This simple tree represents
  / \               ;  the disjunction of A and B.
 /   \
T     O   <-- B
     / \            ;  The tree is represented by
    /   \           ;  (HONS T (HONS T NIL))
   T    NIL         ;  which prints as:  (T T).
```

The interior nodes above labeled with `A` and `B` are actually `CONS`, and `T` and `NIL` represent terminal nodes that represent logic true and false, respectively. We only accept such trees as recognized by the `NORMP` function.

```
(defun normp (x)
  (if (atom x)
      (booleanp x)
    (and (normp (car x))
         (normp (cdr x))
         (if (atom (car x))
             (not (equal (car x) (cdr x)))
           t))))
```

`NORMP` does not recognize trees containing any internal node having two identical terminal (`T` or `NIL`) children.

The meaning of a Boolean function is given by the `EVAL-BDD` function, which given a (function) tree and an assignment of Boolean values to variables, produces the output of the function for that assignment.

```
(defun eval-bdd (x values)
  (if (atom x)
      x
    (if (car values)
        (eval-bdd (car x) (cdr values))
      (eval-bdd (cdr x) (cdr values)))))
```

Next, We define `Q-ITE`, a three-argument argument function that is essentially a BDD-based, symbolic If-Then-Else (ITE) operation. We will use this function to define all other logic functions.

```
(defmacro qcar (x)
  `(cond ((atom ,x) ,x) (t (car ,x))))

(defmacro qcdr (x)
  `(cond ((atom ,x) ,x) (t (cdr ,x))))

(defmacro qcons (x y)
  `(cond ((or (and (eq ,x t)   (eq ,y t))
              (and (eq ,x nil) (eq ,y nil)))
          ,x)
         (t (hons ,x ,y))))

(defun q-not (x)
  (if (atom x)
      (if x nil t)
    (hons (q-not (car x))
          (q-not (cdr x)))))

(defun q-ite (x y z)
 (cond
  ((null x) z)
  ((atom x) y)
  (t
   (let
    ((y (if (hqual x y) t y))    ; Simplify Left
     (z (if (hqual x z) nil z))) ; Simplify Right
    (cond
     ((hqual y z) y)             ; (if x y y)   => y
     ((and (eq y t) (eq z nil))
      x)                         ; (if x T NIL) => x
     ((and (eq y nil) (eq z t)) ; Optimization
      (q-not x))          ; (if x NIL T) => (NOT x)
     (t (let
            ((a (q-ite (car x) (qcar y) (qcar z)))
             (d (q-ite (cdr x) (qcdr y) (qcdr z))))
          (qcons a d)))))))))
```

A more thorough definition and the statement of correctness for our `Q-ITE` function can be found in our companion paper [4].

Given the definition of the `Q-ITE` operation, we can implement other common Boolean functions; here are the 16, two-input Boolean functions. The `FN` argument lets any two-input Boolean function to be selected.

```
(defun q-fn (fn x y)
  (case fn
    ((0  clr)      nil)
    ((1  set)      t)
    ((2  v1)       x)
    ((3  v2)       y)
    ((4  c1)       (q-ite x nil t))
    ((5  c2)       (q-ite y nil t))
    ((6  and)      (q-ite x y nil))
    ((7  ior or)   (q-ite x t y))
    ((8  xor)      (q-ite x (q-ite y nil t) y))
    ((9  eqv nxor) (q-ite x y (q-ite y nil t)))
    ((10 nand)     (q-ite x (q-ite y nil t) t))
    ((11 nor)      (q-ite x nil (q-ite y nil t)))
    ((12 andc1)    (q-ite x nil y))
    ((13 andc2)    (q-ite y nil x))
    ((14 orc1)     (q-ite x y t))
    ((15 orc2)     (q-ite y x t))
    (otherwise nil)))
```

2.2 Reference Adder Specification

We specific three different groups of adders. The first group has a single adder; we consider this adder to be our reference specification. We have mechanically verified that such an adder specification does indeed add numbers when represented as bit vectors [5]. The second group is like the first adder, but every two-input gate used in the the original adder specification may be altered to any other two-input Boolean function. The specification for the third group of adders allows one to specify an adder that works correctly except for exactly one pair of numbers; in fact, adding 3 to 7 is actually a different pair than adding 7 to 3.

Our reference adder specification is given below. We first give the specification for a one-bit, full adder, which accepts three BDDs as inputs and produces two BDDs that represent the sum and carry outputs. The function `Q-FN` implements the logic function identified with its first argument and expects its second and third arguments to be BDDs, which are represented as ACL2 data objects [2].

```
(defun q-full-add (c x y)
  (let* ((xor-x-y   (q-fn 'xor x y))
         (sum       (q-fn 'xor c xor-x-y))
         (generate  (q-fn 'and x y))
         (propagate (q-fn 'and c xor-x-y))
         (carry     (q-fn 'ior propagate generate)))
    (mv sum carry)))
```

This one-bit adder specification is composed of five, two-input Boolean functions (gates). We specify an arbitrary-sized adder by repeatedly applying our one-bit, full adder to each bit position of two, equal-length bit vectors until all input bits have been processed; at each stage we pass along the carry output from the previous bit position as the carry input to the next bit position. Finally, when we have exhausted the input bit-vectors, we include the final carry as the last bit of the answer.

```
(defun q-bv-adder (c a b)
  (if (atom a)
      (hist c)
    (mv-let
     (sum carry)
     (q-full-add c (car a) (car b))
     (hons sum
           (q-bv-adder carry (cdr a) (cdr b))))))
```

This ripple-carry-style adder specification is our reference adder. In our investigation we compare our reference adder specification to other adder specifications to see how flaws manifest themselves by counting the number different functionalities and by observing cone-of-influence alterations in the outputs.

2.3 Single-Bad-Gate Adder Specification

We have created an adder specification that allows any two-input function (gate) used in our reference adder to be altered to another two-input function. We make this possible by creating another one-bit, full-adder adder specification that takes two additional arguments that are used to specify an alternative two-input function to be used in place of one of the internal two-input Boolean functions. We also elaborate our function that composes these, possibly-flawed, full adders so we can alter a specific gate in an arbitrary-sized adder.

We call this series of adders the single-bad-gate (SBG) adders. We begin by assigning a number to each of the 16 two-input Boolean logic functions. We then define the `NEW-FN` function that given some gate type and an integer from 0 to 14, it actually selects a different two-input Boolean function that the gate type provided. Our SBG full adder requires two additional arguments: one identifies which gate, of the five, should be altered and the second new argument specifies the new operation for the gate being altered.

```
(defun sbg-full-add (gate fn c x y)
  (let
   ((xor-0 (if (eql gate 0) (new-fn 'xor fn) 'xor))
    (xor-1 (if (eql gate 1) (new-fn 'xor fn) 'xor))
    (and-2 (if (eql gate 2) (new-fn 'and fn) 'and))
    (and-3 (if (eql gate 3) (new-fn 'and fn) 'and))
    (or-4  (if (eql gate 4) (new-fn 'or  fn) 'or )))
   (let* ((xor-x-y   (q-fn xor-0 x y))
          (sum       (q-fn xor-1 c xor-x-y))
          (generate  (q-fn and-2 x y))
          (propagate (q-fn and-3 c xor-x-y))
          (carry     (q-fn or-4 propagate generate)))
     (mv sum carry))))
```

To complete our SBG adder specification, we define a function that applies our `SBG-FULL-ADD` function repeatedly to an arbitrary-sized bit vector. This function also requires the user to identify the bit position where the flawed gate is to manifest itself.

```
(defun sbg-bv-adder (c a b pos bit gate fn)
  (if (atom a)
      (hist c)
    (mv-let
     (sum carry)
```

```
    (if (equal pos bit)
        (sbg-full-add gate fn c (car a) (car b))
      (q-full-add c (car a) (car b)))
    (hons sum
          (sbg-bv-adder carry (cdr a) (cdr b)
                        (1+ pos) bit gate fn)))))
```

2.4 Single-Bad-Value Adder Specification

Our third adder group specification involves making it possible to create an adder specification that produces any desired answer when exactly a single pair of inputs is provided. This kind of single-bad-value (SBV) adder is most unusual, but it is precisely what one might to insert into a larger design as a Trojan Horse. For instance, if such an adder were embedded in a pre-defined ALU (IP) library element, an unsuspecting user might inadvertently include such a very subtly flawed adder in their design that could later be exploited by software.

Our SBV-adder specification involves selecting between our reference adder specification and a pre-selected result when the inputs match two pre-selected input values. Before we define our SBV adder, we we first define a function that recognizes two identical bit-vector inputs.

```
(defun q-ite-cmp (a b)
  (if (atom a)
      t
    (q-and-ite (q-fn 'eqv (car a) (car b))
               (q-ite-cmp (cdr a) (cdr b)))))
```

Next, we define the `QV-IF-ITE` bit-vector multiplexor that selects one of its two input vectors depending on a control input. Note, that the `Q-ITE` function is the BDD If-Then-Else function.

```
(defun qv-if-ite (c a b)
  (if (atom a)
      nil
    (hons (q-ite c (car a) (car b))
          (qv-if-ite c (cdr a) (cdr b)))))
```

And finally, we define our SBV adder specification. Inputs `C`, `A`, and `B` have the same purpose as they do in our reference `Q-BV-ADDER` specification. But this adder specification produces `ANS-VAL` when input `A` is exactly the value provided to input `A-VAL` and when input `B` is exactly the value provided to input `B-VAL`; otherwise, `SBV-BV-ADDER` works exactly like our reference adder. Generally, we provide constants for `A-VAL` and `B-VAL`.

```
(defun sbv-bv-adder (c a b a-val b-val ans-val)
  (let ((bv-adder  (q-bv-adder c a b))
        (cmp-a-val (q-ite-cmp a a-val))
        (cmp-b-val (q-ite-cmp b b-val)))
    (qv-if-ite (q-fn 'and cmp-a-val cmp-b-val)
               ans-val
               bv-adder)))
```

One might believe that such a change to an adder would be obvious to detect. Of course, if such an adder is converted into a netlist specification, which we often do, one might be able to detect the lack of regularity. But, when such a change is made to a *sea-of-gates* design, such as pre-defined ALU, it may not be at all obvious. Later, we will compare these last two adder specifications to our reference adder.

3. COMPARISON MECHANISMS

We have defined two comparison mechanisms for comparing the behavior of our two, flawed adder specifications to our reference adder. Our first comparison mechanism compares two bit vectors on a bit-by-bit basis, and for each result bit it returns a count of the number of correct answers and a count of the incorrect answers; the sum of the correct and incorrect answers totals the number of different possible Boolean input configurations. In addition, for each pair of result bits (65 such pairs are compared when comparing the output of two 64-bit adders), our second comparison mechanism determines the inputs uniquely used by each answer; that is, this comparison mechanism computes the set of inputs that effect each output.

Before we can count the differences between two adders, we symbolically form the bit-wise Boolean difference between the output equations of our reference adder and one flawed adder.

```
(defun qv-ite-cmp (a b)
  (if (atom a)
      (if (atom b)
          nil
        (cons nil (qv-ite-cmp nil (cdr b))))
    (if (atom b)
        (cons nil (qv-ite-cmp (cdr a) nil))
      (cons (q-fn 'eqv (car a) (car b))
            (qv-ite-cmp (cdr a) (cdr b))))))
```

The result returned by `QV-ITE-CMP` is a long as the longer of the two bit vector being those being compared. The excess bits of the longer bit vector are given non-equal values.

Once we have symbolically computed the bit-by-bit symbolic difference between two bit vectors being compared, we compute point-wise the number of times the answers agree and the number of times the answers disagree, and return both results. Thus, for our adder specifications we perform this computation 65 times, once for each of the 65 Boolean outputs.

```
(defn count-tip-values (x depth)
  (declare (xargs :guard (integerp depth)))
    (if (atom x)
        (mv (if x (expt 2 depth) 0)
            (if x 0 (expt 2 depth)))
    (mv-let
     (left-cnt-1s left-cnt-0s)
     (count-tip-values (car x) (1- depth))
     (mv-let
      (right-cnt-1s right-cnt-0s)
      (count-tip-values (cdr x) (1- depth))
      (mv (+ left-cnt-1s right-cnt-1s)
          (+ left-cnt-0s right-cnt-0s))))))
```

COUNT-TIPS-VALUES returns two values: number of agreements and the number of disagreements. The base case involves determining whether the Boolean atom discovered is NIL. If T result is found, the significance is 2^{depth} times greater than just 1, depending the significance of this (possibly) "short circuit" value. Memoizing the COUNT-TIPS-VALUES function improves its performance, but because of the inclusion of the DEPTH parameter, two structurally identical subtrees at different levels in the tree will not share a memoization entry.

We have defined another version of COUNT-TIPS-VALUES. This version takes full advantage of our function memoization machinery, and it is presented below. We do not actually memoize the COUNT-TIPS-VALUES function, but instead we memoize the single-argument functions MAX-DEPTH and CTV. Single-argument functions are memoized in a more efficient manner than multi-argument functions.

```
(defun max-depth (x)
  (if (atom x)
      0
    (1+ (max (max-depth (car x))
             (max-depth (cdr x))))))

(defun ctv (x)
  (cond
   ((eq x t) (mv 1 0))
   ((atom x) (mv 0 1))
   (t (let* ((da (max-depth (car x)))
             (dd (max-depth (cdr x)))
             (m (+ 1 (max da dd))))
        (mv-let
         (a1 a0)
         (ctv (car x))
         (mv-let
          (d1 d0)
          (ctv (cdr x))
          (let ((a1 (if (eql da (+ 1 m))
                        a1
                      (* (expt 2 (1- (- m da)))
                         a1)))
                (a0 (if (eql da (+ 1 m))
                        a0
                      (* (expt 2 (1- (- m da)))
                         a0)))
                (d1 (if (eql dd (+ 1 m))
                        d1
                      (* (expt 2 (1- (- m dd)))
                         d1)))
                (d0 (if (eql dd (+ 1 m))
                        d0
                      (* (expt 2 (1- (- m dd)))
                         d0))))
            (mv (+ a1 d1) (+ a0 d0)))))))))

(defun count-tip-values (x depth)
  (let* ((max-depth (max-depth x))
         (multiplier (expt 2 (- depth max-depth))))
    (mv-let (1s 0s)
            (ctv x)
            (mv (* multiplier 1s)
                (* multiplier 0s)))))
```

Our second comparison mechanism involves computing the *cone of influence* for each output bit and then comparing them. This can be done by trivially discovering the variables present in each output equation or, as we choose, by computing with "gates" that actually take lists of names as inputs and produce a list of names. By computing with such special gates, we can compute a cone-of-influence for each output of very large circuits; this is something we cannot always do when attempting to inspect the variables present in the BDD output equations. This is because the representation size of some Boolean functions (e.g., multipliers) using BDDs is known to grow exponentially [6].

4. ADDER COMPARISONS

We compare our two flawed adder specifications to our reference adder using our counting and cone-of-influence comparison mechanisms. We first compare our SBG adder specification before we turn our attention to our SBV adder specification.

For the purpose of our study, we make our comparisons using a 64-bit reference adder. All of our adder specifications have two 64-bit, bit-vector inputs and a single-bit, carry input; each adder produces a 65-bit, bit-vector output. Our comparisons are performed point-wise; that is, the same output bit position from both adders is compared, thus each output bit from every flawed adder is compared bit-by-bit to the reference adder.

With our SBG adder specification, we can cause any particular two-input gate to behave like any other gate. To facilitate comparing the SBG adders to the reference adder, we created a set of functions that allow us to compare every flawed adder to the reference adder. Since there are five gates used to implement each bit of our SBG-adder specification, we have 4800 ($64 * 5 * 15$) different adders to compare. We computed the number of differences between our reference adder and all 4800 SBG adders. In all cases, substituting any different gate for any of the five gates in any of the bit positions produces a different answer, except for substituting a XOR gate for the OR gate in the SBG adder, where no difference is detected. Instead of presenting the number of differences detected for each bit position, we compute the minimum number of non-zero differences across all bit positions. In the 4800 flawed adders compared to our reference adder, there were 64 cases when no operational change was detected; this case involved the substitution of a XOR gate for the OR gate in each of the 64 bit positions. For the remainder of the cases, the minimum number of non-zero differences was 85070591730234615865843651857942052864; that is, we see 2^{126} differences in at least some output bit. We compute the non-zero minimum because if there is an unaffected bit position (which is a regular occurrence in an adder with a single bad gate), then our overall answer would be zero even though there are bit positions where differences are detected. On a 1.67 GHz PowerPC-based Apple laptop with 2 GBytes of memory we can create, compare, and count the differences between all 4800 different SBG adders in under a minute. For us, the remarkable thing is that even though the counts are enormous, we are still able to compute the exact difference counts for all of the 312,000 ($4800 * 65$) Boolean equations being compared.

Using our SBV-adder specification, we select exactly one pair of input bit patterns for which the adder produces a supplied answer; otherwise, the SBV adder will produce the same result as our reference adder. This kind of alteration is subtle. For instance, $3+7$ might be specified to return 11, but $7+3$ would return the correct answer of 10. Our SBV-adder specification only alters the output when it recognizes exactly one pair of numbers; the user of this adder may specify the specific output desired for this pair of input numbers. Just as we did for the SBG adder, we can count the correct and incorrect answers for the SBV adder given a specific configuration of our SBV adder. So, by using our `SBV-BV-ADDER` specification with bit vectors corresponding to 3 for the `A-VAL` argument, 7 for the `B-VAL` argument, and 11 for the `ANS-VAL`, and finally, the same input values we use in our reference adder for inputs `C`, `A`, and `B`, we can produce an adder specification that is incorrect exactly when its `A` is 3 and its `B` is 7. When we compare this SBV adder to the reference adder, we find that the first bit position of the output bit vector is incorrect in a single case and otherwise correct in the other 680564733841876926926749214863536422911 ($2^{129}-1$) cases. Discovering such a subtle difference would be practically impossible using testing, and knowing that there is only a single different case might be revealing.

If we investigate the cone-of-influence contributions from the inputs to each output bit position, a very different picture emerges. We have written a function that allows us to recover the input variables that affect each output bit. For instance, in our reference adder, the input carry, the first bit of bit-vector `A`, and the first bit of bit-vector `B` are the only bits that should effect the first output bit. In the case of the second output bit position, the output should only depend on the first two input bits of `A` and `B` and the input carry `C`. Using this kind of comparison mechanism, we sometimes find that the SBG adders sometimes lack expected contributions in one output bit position (when the circuitry that computes the output bit is flawed) or in all bit positions larger than some specific bit position (because the flaw is affecting the carry chain); in other cases, such as when we exchange a two-input logic gate for a different two-input logic gate, we may detect no cone-of-influence difference.

The cone-of-influence differences between the reference adder and the SBV adder are much more dramatic. For the SBV adder, every output bit position depends on every input – obviously, the SBV adder is a very different kind of function. It may be impossible to count the functional differences between very large circuits because it may not be possible to build BDDs representing the functions of interest. However, cone-of-influence computations can be done quickly even for very large circuit specifications.

5. CONCLUSION

Our empirical study comparing various flawed adders suggests that being able to use multiple comparison techniques may allow search for subtly inserted flaws. Any novelty in our approach lies in our ability to exactly count the number of functional differences when comparing two sets of BDD equations; this capability is derived from our newly developed function memoization mechanism. Such counts may provide insight for designers that would like to investigate IP or remotely designed circuits, and where appropriate, cone-of-influence analysis may provide additional insight. Note, these techniques are not limited to combinational logic.

We believe that these techniques may also be useful in a reverse engineering context; being able to view a circuit from multiple perspectives may provide hints as to the actual operation of a circuit under investigation. As we increase our reliance on third-party designs and manufacturing organizations, there is an increased need to have tools that allow evaluators to thoroughly investigate designs for flaws, both inadvertent and malicious.

6. ACKNOWLEDGMENTS

We gratefully acknowledge support from DARPA and from the NSF (CyberTrust Award: CNS 0429591).

7. REFERENCES

[1] R. J. Boulton, A. D. Gordon, M. J. C. Gordon, J. R. Harrison, J. M. J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP TC10/WG 10.2 International Conference*, IFIP Transactions A-10, pages 129–156. North-Holland, June 1992.

[2] 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.

[3] Robert S. Boyer and Warren A. Hunt, Jr.. The E Language. Presented at Hardware Design using Functional Languages (HFL) 2007 in Braga, Portugal, March, 2007.

[4] Robert S. Boyer, Warren A. Hunt, Jr., and Qiang Zhang. A Verified BDD Package. In *Proceedings of the Seventh International Workshop on the ACL2 Theorem Prover and its Applications*, ACM Digital Library, Austin, Texas, 2007.

[5] Bishop C. Brock and Warren 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, 1997.

[6] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, Volume C-35, Number 8, pages 677-691, August, 1986.

[7] A. .P. Ershov. On Programming of Arithmetic Operations. In the *Communications of the ACM*, Volume 118, Number 3, August, 1958, pages 427–430.

[8] Eiichi Goto. Monocopy and Associative Algorithms in Extended Lisp. University of Toyko, Technical Report TR-74-03, 1974.

[9] Warren A. Hunt, Jr. Decomposing the Verification of Pipelined Microprocessors with Invariant Conditions. In *Proceedings of CAV 2004*. Springer Verlag, LNCS 3114, 2004.

[10] Warren A. Hunt, Jr. and Erik Reeber. Formalization of the DE2 Language. In *Proceedings of CHARME*

2005, volume LNCS 3725, pages 20–34, 2005.

[11] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.

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

[13] M. Kaufmann and J S. Moore. ACL2: An Industrial Strength Version of NQTHM. Proceedings of the *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23-34, IEEE Computer Society Press, June 1996.

[14] M. Kaufmann and J S. Moore. A Precise Description of the ACL2 Logic. In *http://www.cs.utexas.edu/users/moore/publications/km97a.ps.gz*. Department of Computer Sciences, University of Texas at Austin, 1997.

[15] M. Kaufmann and J S. Moore. A flying demo of ACL2. Technical Report *http://www.cs.utexas.edu/users/moore/publications/flying-demo/script.html*, Department of Computer Sciences, University of Texas at Austin, 2000.

[16] Donald Michie. Memo functions: a Language Feature with Rote Learning Properties. Technical Report MIP-R-29, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1967.

[17] Donald Michie. Memo Functions and Machine Learning. In *Nature*, Volume 218, 1968, pages 19–22.