Introduction to a Computational Theory and Implementation of Sequential Hardware Equivalence*

Carl Pixley
Microelectronics and Computer Technology Corporation (MCC)
Computer Aided Design Program

Abstract

A theory of sequential hardware equivalence [1] is presented, including the notions of gate-level model (GLM), hardware finite state machine (HFSM), state equivalence ($\sim$), alignability, resetablility, and sequential hardware equivalence ($\approx$). This theory is motivated by (1) the observation that it is impossible to control the initial state of a machine when it is powered on, and (2) the desire to decide equivalence of two designs based solely on their netlists and logic device models, without knowledge of intended initial states or intended environments.

Binary decision diagrams are used to represent predicates about pairs of harware designs. Algorithms are given for computing pairs of equivalent states and sequential hardware equivalence as implemented in the MCC CAD Sequential Equivalence Tool (SET).

1 Introduction

A problem often encountered in commercial hardware design is to map an existing design from one technology to another (in some way, superior) technology. Differences in physical characteristics of the new and old technologies (e.g., different relative speeds or area characteristics) often cause designers to reimplement parts of the design to exploit the characteristics of the new technology. When reimplementation involves only purely combinational parts of the machine, tautology-checking algorithms can be used to decide equivalence.

However, sometimes sequential parts are reimplemented as well (e.g., combinational logic might be moved across storage elements). Unfortunately, the designer may not have an accurate specification (other than the existing design) of the individual part that is being replaced. For example, he may not know a reset state or reset sequence for the part. Furthermore, the designer may not have a specification of the part's intended environment to know what signals the environment will emit and how it should interact with the part. Therefore, the theory of sequential hardware equivalence presented in [1] (and elaborated here) is motivated by the desire to decide whether two gate-level designs are equivalent without reference to the intended environment, knowledge of initial (reset) states, and knowledge of reset (homing) sequences.

*This report is a revision of MCC Technical Report: CAD 448-89 (Q)

1.1 Related Research

In the classical theory of finite state machines [5], two FSMs are defined to be equivalent provided they "accept" the same input sequences. A FSM "accepts" a valid input sequence if, starting from a preferred initial state, the sequence leads to one of a set of designated final output states. Because, the theory presented here deals with comparison of two gate-level models (as defined by interconnections of logic gates and primitive storage elements), there is no notion of initial state, final state, or valid input to a state (though this notion of equivalence can be modified to reflect input constraints).

In the present theory, "state" is any one of the 2^n assignments of boolean values to the n many primitive storage elements of a design, and hardware is modeled in terms of its sequential input/output behavior from any initial state. This viewpoint is required because the initial state of a machine, when it is powered up, cannot be reliably predicted.

Also, the present theory is primarily concerned with equivalence of two hardware designs, rather than comparison of a design to a specification, though the latter can be accomplished with similar computational techniques. This viewpoint is dictated by the desire to understand when one design can be safely replaced by another, "equivalent", design.

Coudert, Berthet, and Madre [2] present a verification method based on characteristic functions of sets of tuples of boolean values. Starting from a pair of initial states, they compute the set of reachable states until either a discrepancy between output functions is recognized or all reachable state pairs have been examined. By contrast, the method presented here extracts the set of all pairs (possibly empty) for which the two machines have the same input-output behaviors. Surprisingly, our experiments indicate that our algorithm generally converges faster. However, their more incremental approach should be able to handle larger designs.

Devadas et al. [8] describe a method for comparing the state transition graphs of two finite automata (derived either from gate, register transfer, or ISP-like specifications) with boolean simulation. In contrast to the theory presented here, their comparison algorithm presupposes that the machines start in a valid initial state. The efficiency of their algorithm results from their ability to extract "don't care" information from the RTL or logic-level description, which allows them to greatly reduce the number of cases to simulate. In contrast, it is a remarkable property of the BDD representation of the state transition relation employed in the current work, that only dependencies among logic variables which affect the characteristic function are explicitly represented.

Symbolic simulation has been shown by Randal E. Bryant and his students [4] to be an effective method for checking that a hardware implementation responds correctly to a sequence of symbolic inputs. This approach can avoid the combinational explosion that would normally occur in boolean simulation when evaluating circuit operation over many combinations of input and initial state. Symbolic simulation can effectively compare the output behaviors of two designs with the same (finite) sequence of symbolic inputs but cannot, by itself, establish the equivalence of output behaviors for all possible sequences of inputs of arbitrary length as in the present theory. On the other hand, symbolic

simulation can complement the present approach. When two designs are found to be inequivalent, symbolic simulation can find a sequence of inputs that cause the outputs to diverge starting from a pair of initial states that are thought to be equivalent.

Bose and Fisher [10] use BDDs to characterize states of a machine and the machine's next-state function. They then check temporal logic properties of the design using algorithms similar to those in [1].

It is well known that the size of an ordered binary decision diagram is sensitive to the ordering of boolean variables employed. For combinational circuits, orderings of circuit inputs derived from the circuit interconnections are described in [6] and [7]. In the present work, there is the added complication of ordering both circuit inputs and storage element outputs (representing both state and next-state variables). In the present work this is accomplished by a rather simple depth-first heuristic. However, better ordering of variables is a continuing subject of investigation.

None of the previous works presents a theory of equivalent sequential hardware design based solely on netlists and logic models of devices.

2 Theory

2.1 Gate-Level Models and HFSMs

Synchronous designs are modeled at the gate level in terms of combinational elements and primitive storage elements. A *primitive storage element (PSE)* is a device that transports its input to its output on a clock event and holds the output value until the next clock event. An example of a primitive storage element is a simple D flip-flop (without enable or reset). Fortunately, most real storage devices, such as D flip-flops with enable, reset and both Q and Qn outputs, can be modeled as a network of these primitive storage elements and combinational logic.

A *gate-level model (GLM or design)* is defined to be an interconnection of purely combinational elements and primitive storage devices. Each interconnection (i.e., net) is required to have exactly one driver (design input or device output). A design may have no loops of purely combinational elements.

A *state* of a GLM is an assignment of boolean values (0 or 1) to the output of each primitive storage element of the design. Suppose a design (GLM), D, has i many inputs, n many PSEs, and o many outputs. Design D has 2^n many states. For each state, each output is a boolean function of the inputs of D (if, for each state, each output function is a constant function, the design is called a Moore machine; otherwise, it is called a Mealy machine). To account for quotient designs, which are defined later, we present the following notion.

Definition 1 *A* **Hardware Finite State Machine (HFSM)** *is a quadruple, (Ins, States, Transition, Outputs) where Ins is a non-empty set of symbols, States is any non-empty set, Transition is a total function from* $\{0,1\}^I \times S$ *into* S*, and Outputs is an n-tuple of functions* ($n \geq 0$)*, each of which has domain* $\{0,1\}^I \times S$ *and range* $\{0,1\}$.

We will think of the transition function as a relation, $Transition(\vec{q}, \vec{in}, \vec{next\text{-}q})$, that holds if and only if the circuit has state $\vec{next\text{-}q}$ when it receives inputs, $\vec{ins}$, while in state, $\vec{q}$. Note that this definition does not mention initial states or accepting states as in classical finite state machine theory. We define two designs to be **compatible** if they have the same set of inputs and outputs. This notion of hardware equivalence is defined only for compatible designs. We now define several notions that are useful in stating the theory.

Definition 2 *Let s0 be any state of design D, and let SEQ be any finite sequence of boolean input vectors.* **SEQ(s0)** *is defined to be the state of design D following n machine cycles with inputs SEQ starting at s0.*

Definition 3 *A set of states, S, is* **closed under (all) inputs** *means that if s is an element of S and $\vec{in}$ is an input vector, then $\vec{in}(s)$ is an element of S.*

2.2 Equivalent States and the Quotient Design

Definition 4 *Suppose s0 and s1 are states of compatible designs D0 and D1, respectively. We define s0 to be* **equivalent** *to s1 (i.e., $s0 \sim s1$) to mean that for the state pair (s0,s1) and for all state pairs reachable from (s0,s1) by sequences (of arbitrary length) of identical inputs, all corresponding output functions for the two designs are the same.*

Clearly $\sim$ is an equivalence relation among states of compatible designs. An algorithm for computing the set of all equivalent state pairs of two compatible design is given later.

Definition 5 *The* **quotient machine** *$(D/\sim)$ of design D, is the HFSM with the same inputs as D, with states being the set of equivalence classes induced by $\sim$, and with pthe induced transition relation and output functions.*

The proof that this definition makes sense is based upon the observation that if two states are equivalent, they have the same output functions, and for any input vector, their successor states are equivalent. The equivalence class of state $\vec{q}$ (i.e., set of states equivalent to $\vec{q}$) is denoted $[\vec{q}]$.

2.3 Alignability and Design Equivalence

Given that one cannot predict what state a design will be in when it is powered on, knowing that two designs have some pair of equivalent states is not enough to infer design equivalence. It must be possible to force the designs to behave the same no matter what their initial states are.

Definition 6 *A pair of states (s0,s1) of a design pair (D0,D1) is* **alignable**, *if there is a sequence, SEQ, of inputs (called an* **aligning sequence**) *such that $SEQ(s0) \sim SEQ(s1)$.*

Definition 7 *Given a compatible design pair (D0,D1),* **Alignable-State-Pairs(D0,D1)** *is the set of all pairs of alignable states.*

An algorithm for computing **Alignable-State-Pairs(D0,D1)** is given later. We now define a pair of designs to be equivalent if they have some equivalent states, and for every pair of initial states, there is some sequence of inputs (called an *aligning sequence*) that will force them to behave the same.

Definition 8 *Two designs, D0 and D1, are* **equivalent** ($D0 \approx D1$) *if and only if all state pairs are alignable (i.e., Alignable-State-Pairs(D0,D1) is the set of all pairs of states).*

The following fundamental theorem shows that if every state pair is alignable with some aligning sequence (that may depend upon the pair), then there is a *single* aligning sequence that will align all state pairs. More detailed proofs of the following theorems are given in [1].

Theorem 1 (fundamental alignment theorem) *If two design are equivalent, then there is a* **single** *aligning sequence (called a* **universal aligning sequence***) for all state pairs.*

The proof of the fundamental theorem is based upon the observation that the set of equivalent state pairs is closed under all inputs. So if p is a pair of non-equivalent states, let SEQ0 be a sequence that forces p to an equivalent pair (SEQ0 exists by hypothesis). Note that other pairs that were not equivalent may have been forced into equivalent states, but each pair that was already equivalent moved to equivalent states. Therefore, the number of non-equivalent pairs in the image of SEQ0 is fewer by at least one than the original number of non-equivalent pairs. This process is repeated until there are no more non-equivalent state-pairs. The concatenation of these sequences is the desired universal aligning sequence.

We intend to define hardware equivalence ($\approx$)in such a way that it is an equivalence relation (i.e., reflexive, symmetric, and transitive) on the set of reasonable designs.

Theorem 2 *The relation $\approx$ is symmetric and transitive.*

An application of this theorem is the observation that if a design is equivalent to any other design, then it must be equivalent to itself.

Corollary 1 *If $A \approx B$, then $A \approx A$.*

The following theorem shows the necessity of the hypothesis that all state pairs are alignable in Theorem 1.

Theorem 3 *There is a pair of machines having equivalent states such that no single aligning sequence will drive all alignable state pairs into the set of equivalent states.*

In fact, the example in [1] is of a single design such that not all state pairs are alignable. This example shows that, in general, hardware equivalence ($\approx$) is not reflexive. The next section characterizes reflexivity.

2.4 Resetability

The notion of resetability turns out to be fundamental to the present theory of sequential hardware equivalence.

Definition 9 *A state, s0, of a machine is a* **reset state** *if and only if there exists a sequence of inputs, SEQ, (called a* **reset sequence***) such that if s is any state then SEQ(s) = s0. The set of (all) reset states of machine D is denoted* **Reset-States(D)**. *A design is* **resetable** *if it has a reset sequence.*

The following theorem characterizes the set of reset states as the set of states reachable from any reset state.

Theorem 4 *Let D be any HFSM and let s0 be any reset state of design D. Then Reset-States(D) is the set of states reachable from s0.*

Note that this theorem may be applied when D is the quotient of an HFSM. The following theorem characterizes self-equivalence.

Theorem 5 (Reset Theorem) *Any design, D, is equivalent to itself (i.e. $D \approx D$) if and only if the quotient $(D/\sim)$ is resetable. Furthermore, an input sequence, SEQ, aligns all pairs of $D \times D$ if and only if SEQ is a reset sequence for $(D/\sim)$*

Definition 10 *A design is* **essentially resetable** *means that $D/\sim$ is resetable.*

Corollary 2 *(to Corollary 1) If $A \approx B$, then A and B are essentially resetable.*

Theorem 6 (Equivalence Theorem) *The relation $\approx$ is an equivalence relation on the set of essentially resetable designs.*

2.5 The Isopmorphism Theorem

We now present without proof an alternate characterization of hardware equivalence. This characterization requires the notion of isomorphism for HFSMs. As usual, isomorphism just means that the two structures are identical up to renaming. Two HFSMs $H0$ and $H1$ are isomorphic if and only if they are compatible and there is a one-to-one function F from the states of $H0$ onto the states of $H1$ satisfying the following two properties. For each state $S0$ of $H0$, corresponding output functions of the two designs are the same for states $S0$ and $F(S0)$. For any input vector $\vec{in}$ and state $S0$ of $H0$,

$$transition_{H1}(\vec{in}, F(S0)) = F(transition_{H0}(\vec{in}, S0)$$

The following theorem characterizes hardware equivalence ($\approx$) for the set of essentially resetable designs.

Theorem 7 (Isomorphism Theorem) *Suppose that D0 and D1 are essentially resetable designs, then the following are equivalent:*

1. $D0 \approx D1$.
2. *There exists some equivalent state pair for D0 and D1.*
3. *State equivalence ($\sim$) is an isomorphism from Reset-States($D0/\sim$) onto Reset-States($D1/\sim$)*

From the point of view of the theory of sequential hardware equivalence presented here, the essence of a design is captured by the reset states of its quotient, i.e., Reset-States($D/\sim$). For any essentially resetable design, D0, any other design, D1, is equivalent to D0 if and only if D1 is essentially resetable and the set of reset states of the quotient of D1 modulo $\sim$ is isomorphic to Reset-States($D0/\sim$). In fact, Reset-States($D/\sim$) is minimal in the following sense.

Theorem 8 *For any essentially resetable design, D, the Reset-States($D/\sim$) is a design with the smallest number of states that is equivalent to D.*

3 Algorithms

Binary decision diagrams (BDDs) [3] are used to represent characteristic functions of sets of n-tuples of boolean values. For example, suppose boolean values are assigned to the variables, $\vec{q}$, $\vec{in}$,and $\vec{nxt}$-q. Then $transition(\vec{q}, \vec{in}, \vec{nxt}\text{-}q)$ has value TRUE if and only if the values assigned to $\vec{nxt}$-q are the next values of the primitive storage elements of the circuit when the current values are $\vec{q}$ and the inputs are $\vec{in}$. The predicate *qs-are-next-qs* is true if and only if corresponding variables *qs* and *nxt-qs* have the same value. If *foo* is a predicate involving variables *qs*, then *exist qs:foo&qs-are-next-qs* is the same predicate in terms of variables *nxt-qs*. The programs below illustrate how to calculate the BDDs for the predicates *Transition*, *Equivalent-Outputs*, *Equivalent-State-Pairs*, and Equivalent-Designs.

3.1 Calculating *Transition* and *Equivalent-Outputs*

The transition predicate is derived as follows from a netlist for a design. For the input to each primitive storage element, q, the input to q is expressed as a boolean function, *in-q* of variables, $\vec{q}$ and $\vec{in}$. For each variable *nxt-q*, let $nxt\text{-}predicate(\vec{q}, \vec{in}, \vec{nxt}\text{-}q)$ be the predicate, $nxt\text{-}q \equiv in\text{-}q$. The transition predicate, $transition(\vec{q}, \vec{in}, \vec{nxt}\text{-}q)$, is then the conjunction of all the *nxt-predicate* predicates.

For corresponding outputs, Out_0 and Out_1, of the two designs, the output is expressed as a function, $Out\text{-}fun_0(\vec{in}, \vec{q_0})$ and $Out\text{-}fun_1(\vec{in}, \vec{q_1})$ of the inputs and q-values. Let $Out(\vec{q_0}, \vec{q_1})$ be the predicate $\forall \vec{in}, (Out_0(\vec{q_0}, \vec{in}) \equiv Out_1(\vec{q_1}, \vec{in}))$. $Equivalent\text{-}Outputs(\vec{q_0}, \vec{q_1})$ is the conjunction of all *Outs*.

3.2 Calculating *Equivalent-State-Pairs*

Let A_0 be the set of state pairs, $(\vec{q_0}, \vec{q_1})$, for which *Equivalent-Outputs* $\equiv 1$, i.e., corresponding outputs of the two machines agree. In general, let state pair $(\vec{q_0}, \vec{q_1})$ belong to A_{i+1} if and only if *Equivalent-Outputs*$(\vec{q_0}, \vec{q_1}) \equiv 1$ and the set of all states immediately reachable from $(\vec{q_0}, \vec{q_1})$ is in A_i. A simple induction argument shows that a state-pair, $(\vec{q_0}, \vec{q_1})$, belongs to A_i if and only if *Equivalent-Outputs*$(\vec{q_0}, \vec{q_1}) \equiv 1$ and for any sequence of inputs, $in_i, in_{i-1}, \ldots, in_1$, having length i, all of the states-pairs reached by that set of inputs satisfies Equivalent-Outputs. Furthermore, each A_{i+1} is a subset of A_i. Hence, the intersection of all A_is is *Equivalent-State-Pairs.*

```
char := 1
next_char := Equivalent-Outputs
loop until (char = next_char)
  char := next_char
  correct-next-qs := (exist qs: char&qs-are-next-qs)
  ins-qs-point-to-correct-next-qs
      := (exist nxt-qs:transition&correct-next-qs)
  qs-always-point-to-correct-next-qs
      := (all ins:ins-qs-point-to-correct-next)
  next_char := Equivalent-Outputs&qs-always-point-to-correct-next-qs
end loop
equivalent-state-pairs := char
equivalent-state-pairs-non-empty := exist qs:equivalent-state-pairs
```

3.3 Calculation of *Alignable-Pairs*

Suppose that the set of equivalent state pairs is non-empty (i.e., the characteristic function, *equivalent-state-pairs*, is not FALSE). Let B_0 be the set of equivalent state pairs. For all $i > 0$, let B_{i+1} be the union of B_i and the set of state-pairs, p, such that for some vector of inputs, $\vec{in_p}$, if $\vec{in_p}$ is applied to p, then the resulting state is in B_i. The set B_i is the set of state-pairs that can be transformed into equivalent-state-pairs in i or fewer cycles. Since B_i is a subset of B_{i+1}, let the union of all the B_is be called the **alignable-pairs**. The following algorithm computes the predicate *alignable-pairs*$(\vec{q})$ from *equivalent-state-pairs.*

```
char := 0
next_char := equivalent-state-pairs
loop until (char = next_char)
  char := next_char
  new-align := (exist in's nxt-qs:transition &
                    (exist qs: char & qs-are-next-qs))
  next_char := char or new-align
```

```
end loop
alignable-pairs := char
Equivalent-Designs := (all qs:alignable-pairs)
```

Two designs are equivalent if and only if *Equivalent-Designs* is True.

3.4 Experimental Results

The above algorithms together with an algorithm that computes all minimal reset sequences [1] are implemented in the MCC CAD Sequential Equivalence Tool (SET). The results using the 1988 version of the BDD routines in COSMOS system by R. Bryant of Carnegie Mellon University are reported in [1]. The longest comparison was of two four-bit serial multipliers (multipliers are known to generate large BDDs) with 18 and 15 storage elements, respectively, which took less than four cpu minutes on a Sun 4. For all 44 of the Berkeley/MCNC examples (from bbara through train4) two designs were synthesized from KISS2 descriptions by a commercial synthesis tool using different state encodings. Equivalence for each pair was decided in less than 23 cpu seconds and most were decided in less than two cpu seconds.

SET is being modified to use the more efficient Brace-Rudell-Bryant [11] program from Carnegie-Mellon University and Synopsys Inc.

4 Conclusion

An intuitively appealing definition of equivalence of gate level models (i.e., sequential hardware designs) was formulated which, for essentially resetable designs, is an equivalence relation. Several theorems were presented. A design is equivalent to itself if and only if its quotient is resetable. For two essentially resetable designs the following are equivalent. (1) The designs are equivalent. (2) The set of reset states of their quotients are isomorphic. (3) Their quotients have at least one pair of equivalent states. Algorithms were presented for computing the transition relation, the set of state pairs with equivalent outputs, the set of pairs of equivalent states, the set of alignable state pairs, and design equivalence. Experimental results show that deciding design equivalence for relatively small sequential designs is tractable.

References

[1] C. Pixley, A Computational Theory and Implementation of Sequential Hardware Equivalence, *DIMACS Technical Report 90-31, volume 2, Workshop on Computer-Aided Verification June 18-21*, Robert Kurshan and E.M. Clarke, eds.

[2] O. Coudert, C. Berthet, Jean Christolphe Madre, Verification of Sequential Machines Using Boolean Functional Vectors, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, November 13-16, 1989.

[3] R.E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, Vol. C35 No. 8, August 1986.

[4] R.E. Bryant, Can a Simulator Verify a Circuit?, *Formal Aspects of VLSI Design*, eds. G.J.Milne et al., North-Holland, 1966.

[5] J.E.Hopcroft, J.D.Ullman, *Formal Languages and Their Relation to Automata*, Addison-Wesley, 1969.

[6] M. Fujita, H. Fujisawa, N. Kawato, Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams, *ICCAD '88*, pp. 2-5.

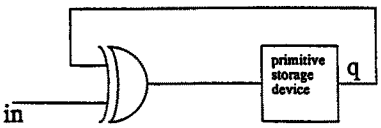
[7] S. Malic, A.R.Wang, R.K.Braton, A. Sangiovanni-Vincentelli, Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment, *ICCAD '88*, pp. 6-9.

[8] S. Devadas, H-K. T. Ma, A.R. Newton, On The Verification of Sequential Machines At Differing Levels of Abstraction, *24th ACM/IEEE Design Automation Conference*, Paper 16.2, pp. 271-276.

[9] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach, *Proceedings of the 10th Symposium on Principles of Programming Languages*, Austin, Texas, Jan. 1983, pp. 117-126.

[10] S.Bose, A.L.Fisher, Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, November 13-16, 1989.

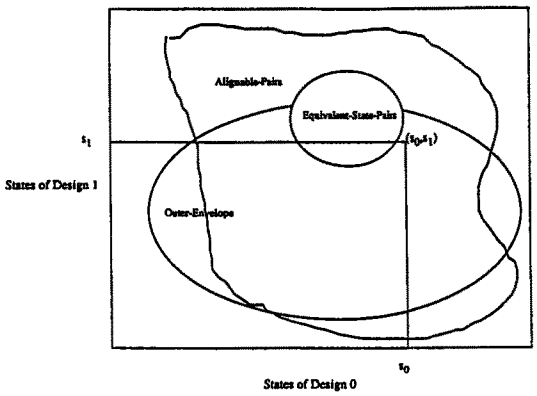
[11] K. Brace, R. Bryant, and R. Rudell, Efficient Implementation of a BDD Package, *Proceedings of the 27th ACM/IEEE Design Automation Conference*, June 1990, pp. 40-45.



transition(q, in, next-q) = in&(q xor next-q) or ~in&(q xnor next-q)
= ~(in xor q xor next-q)

extension = { (q, in, next-q)
(1, 1, 0),
(1, 0, 1),
(0, 1, 1),
(0, 0, 0) }

Figure 1



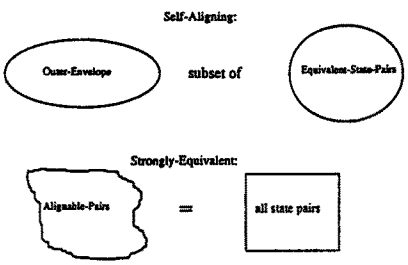


Figure 2

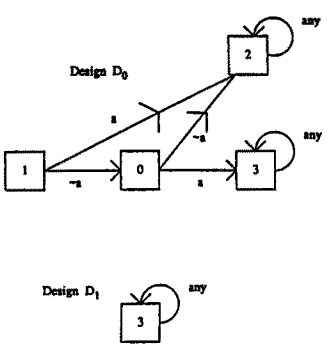


Figure 3