The Verification Problem for Safe Replaceability

Vigyan Singhal *
Computer Science Division
University of California
Berkeley, CA 94720
vigyan@ic.eecs.berkeley.edu

Carl Pixley
Motorola Inc., MD OE321
6501 Wm Cannon Drive West
Austin, TX 78735
pixley@math.sps.mot.com

Abstract. This paper addresses the problem of verifying that a sequential digital design is a safe replacement for an existing design without making any assumptions about a known initial state of the design or about its environment. We formulate a safe replacement condition which guarantees that if an original design is replaced by a new design, the interacting environment cannot detect the change by observing the input-output behavior of the new design. Examples are given to show that safe replacement ($\preceq$) allows simplification of the state transition diagram of an original design. It is showed that if D_1 is a safe replacement for design D_0 then every closed strongly connected component of D_1 is contained in D_0. We present a decision procedure for determining whether a replacement design satisfies our safe replacement condition.

1 Introduction

This paper addresses the problem of implementation verification for synchronous sequential designs. Although we will address the problem for gate-level designs, our theory is equally applicable for designs at the state transition level. We want to guarantee that if we replace a sequential design then no surrounding environment will be able to detect the change based on the input-output behavior of the replacement (a *safe* replacement). The problem comes up because frequently designers work separately on separate pieces of a large design, and the objective is to modify one's design so that any of the interacting designs will not notice the modification (Figure 1). The designers do not want to make any assumptions about the surrounding designs outside their own, not even about any initializing sequence coming from outside when their design is powered up. Latches (or memory elements) in the designs may not even have reset lines. In Figure 1, the designer working on design $D0$ would like to replace it by design $D1$ without making any assumptions about the other interacting designs. In this paper, we will refer to the design outside of $D0$ as the environment of $D0$.

The problem of implementation verification for sequential designs is not a new one. Efficient methods exist for the verification of sequential designs [3, 8, 1].

* Research supported by NSF/DARPA Grant MIP-8719546 and a summer internship from Motorola, Inc.

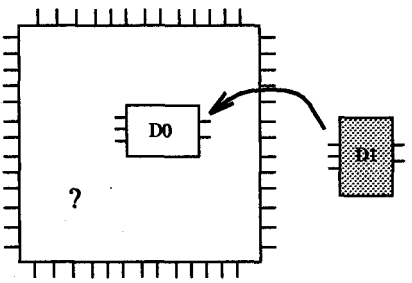

Fig. 1. Replacement of a sequential design

However, these methods only work for designs where all latches have a reset line which determines the designated initial state for the circuit. One key point where our work differs from previous work on sequential verification is that we do not assume any reset lines running to the latches. While it is well-known that data pipeline and memory designs frequently have latches with no reset lines, it is also true that many industry-level control designs have some latches or flip-flops without reset lines. An important reason for having latches without reset lines is the large saving in area by avoiding the routing of the reset lines all over the design. Also, latches without reset lines cost less (consume fewer transistors) than latches with reset lines. Since the latches can power-up in any state, we cannot assume a designated start state for the design. For such designs, the question that needs to be answered is "when can we replace a design with another, so that while the replacement design can power up in any state, there is no way the environment can detect the replacement"?

A notion of sequential hardware equivalence for designs which may not have a designated initial state is presented in [7]. Efficient BDD-based techniques are presented which verify this equivalence for two given designs. We will show that for sub-designs embedded in a large design (or the environment), this notion of equivalence is not always applicable.

In this paper, we will present our condition for safe design replacement. Although this condition is strong enough so that the interacting environment cannot detect the replacement, it does not require that every state in one design be equivalent to one in the other design (the classical notion of machine equivalence, as presented in [5]). Our condition also preserves possible interactions with the environment during initialization. We also explore the methods which can be used to verify the necessary and sufficient conditions for a new (replacement) design to be a safe replacement for an old (existing) design.

An orthogonal problem to the verification problem is the problem of using our replaceability notion to do sequential resynthesis on the existing design. We are working on this problem and, based on preliminary work, the notion of replaceability does seem to provide sufficient flexibility to achieve some optimizations.

2 Terminology and Background

Here we define some notation and a little background that we will need later in this paper.

Definition 1. A *deterministic Finite State Machine (DFSM)* M is a quintuple, $(Q, I, O, \lambda, \delta)$, where Q is the set of states, I is the set of input values, O is the set of output values, λ is the output function, and δ is the next state function. The output function λ is a completely-specified function from domain $(Q \times I)$ to range O. The next state function is a completely-specified function from domain $(Q \times I)$ to range Q. A hardware *design* D is a DFSM with n input wires, m output wires and t latches, Q has 2^t states, I has 2^n values and O has 2^m values.

We will also use λ and δ to denote the output and next state functions on sequences of inputs. So, if $\pi = a_1 \cdot a_2 \cdot a_3 \cdots a_p \in I^p$ is a sequence of p inputs, these functions are recursively defined as $\lambda(s, \pi) = \lambda(s, a_1) \cdot \lambda(\delta(s, a_1), \pi')$ and $\delta(s, \pi) = \delta(\delta(s, a_1), \pi')$, where $\pi' = a_2 \cdot a_3 \cdots a_p$. So, the range-domain relationships are $\lambda : Q \times I^p \rightarrow O^p$ and $\delta : Q \times I^p \rightarrow Q$.

Two designs are said to be *compatible* with each other if they have the same number of input and output wires. All notions of equivalence and replaceability developed in this paper are meaningful only for pairs of compatible designs. Henceforth, when talking about two different designs we will implicitly assume compatibility of the two. In this paper we will assume that designs D_0 and D_1 denote the quintuples $(Q_{D_0}, I, O, \lambda_{D_0}, \delta_{D_0})$ and $(Q_{D_1}, I, O, \lambda_{D_1}, \delta_{D_1})$, respectively.

Definition 2. A set of states, $S \subseteq Q$, is *closed (under all inputs)* if for each state $s \in S$, for each input $a \in I$, $\delta(s, a) \in S$.

Definition 3. A set of states, $S \subseteq Q$, is called a *closed strongly connected component (closed SCC)* if S is closed and for any two states $s_1, s_2 \in S$, there exists a finite input sequence $\pi \in I^*$ such that $\delta(s_1, \pi) = s_2$. It is easy to show that for any state s of design D, a closed SCC is reachable from s.

Definition 4. Given two states $s_0 \in Q_{D_0}$ and $s_1 \in Q_{D_1}$, state s_0 is *equivalent* to state s_1 ($s_0 \sim s_1$) if for any sequence of inputs $\pi \in I^*$, $\lambda_{D_0}(s_0, \pi) = \lambda_{D_1}(s_1, \pi)$. It can be easily shown that if $s_0 \sim s_1$, then for any input sequence $\pi \in I^*$, $\delta_{D_0}(s_0, \pi) \sim \delta_{D_1}(s_1, \pi)$. We say that π *distinguishes* s_0 and s_1 if $\lambda_{D_0}(s_0, \pi) \neq \lambda_{D_1}(s_1, \pi)$.

We now give a classical notion of equivalence between two DFSM's [5, page 23].

Definition 5. Two DFSM's M_1 and M_2 are *equivalent* to each other ($M_1 \equiv M_2$) if for each state s in M_1 there is a state t in M_2 such that $s \sim t$, and for each state t in M_2 there is a state s in M_1 such that $s \sim t$.

3 Sequential Hardware Equivalence (SHE)

In this section we will briefly review the work presented in [7] about the theory of sequential hardware equivalence for equivalence between two gate-level hardware designs without assuming any knowledge of initial state. When the design powers up, the state it powers up in cannot be predicted. The desired input/output behavior is achieved from the design by driving a fixed synchronizing sequence of input vectors through the design after the power-up.

Definition 6. A sequence of inputs $\pi \in I^*$ is called an *essential reset sequence* (or a *synchronizing sequence*) if for any pair of states $s_0, s_1 \in Q_{D_0}$, $\delta_{D_0}(s_0, \pi) \sim \delta_{D_0}(s_1, \pi)$. A state $s \in Q_{D_0}$ is called an *essential reset state* if there exists a state $s_0 \in Q_{D_0}$ and a synchronizing sequence π such that $\delta(s_0, \pi) \sim s$. A design which has an essential reset state is called *essentially resettable.*

Definition 7. A state pair $(s_0, s_1) \in Q_{D_0} \times Q_{D_1}$ is *alignable* if there is a sequence of inputs $\pi \in I^*$ such that $\delta_{D_0}(\pi, s_0) \sim \delta_{D_1}(\pi, s_1)$. The sequence π is called an *aligning sequence.*

Definition 8. Designs D_0 and D_1 are *equivalent* ($D_0 \approx D_1$) if all state pairs are alignable.

Definition 8 defines the notion of sequential hardware equivalence. The following results were shown in [7].

Theorem 9. *$D_0 \approx D_1$ if and only if there is a single (but not necessarily unique) aligning sequence that aligns all state pairs in $Q_{D_0} \times Q_{D_1}$.*

Theorem 10. *The relation $\approx$ is symmetric and transitive, but not reflexive.*

For the class of essentially resettable designs, the relation $\approx$ is an equivalence relation. The non-reflexivity of SHE comes from the fact that a non-essentially-resettable design does not have an aligning sequence with itself. Thus the design is not equivalent to itself. An example of such a design is shown in Figure[2] 2. In this design the state pair (10, 11) is not alignable.

4 Sequential Replaceability

Here we start by justifying why the notion of sequential hardware equivalence, discussed in Section 3, does not work for safe replacement of sequential designs. Then we present our new notion of sequential replaceability, followed by some properties of this new notion.

[2] We will frequently represent designs by state transition graphs (STG's). A label a/b on an edge denotes that under input a, the source state outputs b. The destination of the edge denotes the next state for that input. The t-bit binary-valued label on a state denotes that in the design the state is implemented by that assignment of the t latches. Notice that because a combinational function can be implemented in many different ways, the design-to-STG transformation is a many-to-one mapping.

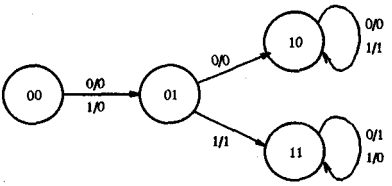

Fig. 2. Example of a design which does not have an essential reset sequence

4.1 Problems with Sequential Hardware Equivalence

From Theorem 9, two designs are considered equivalent if there exists a universal aligning sequence. This sequence is a synchronizing sequence for either design. However, in the design process, often the designer does not know the synchronizing sequence for his/her design $D0$. Even if they can determine such a sequence π for their design, it may not be possible for the environment to generate π. So, for a safe replacement of $D0$ we need to preserve all initializing sequences, and not just one.

The notion of SHE does not place any constraints on the outputs of the designs during the synchronization phase. However, we claim that this condition is too weak for a safe replacement[3]. *A priori*, we cannot assume that the external environment is not sensitive to the outputs during the synchronization phase. This is especially important because there may be another interacting design whose synchronizing sequence may be driven by an output of design $D0$, and affecting the outputs of $D0$ during synchronization may destroy that synchronizing sequence.

Finally, the notion of SHE does not work for designs which are not essentially resettable. As Theorem 10 states, such designs are not even equivalent to themselves. It will be presumptuous on our part to assume that such designs do not exist in real designs, and to present a theory which fails to replace such designs even by themselves. There can be two classes of real designs which are not essentially resettable. First, if the environment has some flexibility for the input/output behavior it can accept from the design, the design may have multiple non-equivalent closed SCC's (for example, Figure 2). In this example, the environment has a don't care condition so that the design is acceptable as long as it always toggles the input (state 11) or always outputs the input (state 10), after the synchronization phase. For the second class, consider the design in Figure 3. It can be seen that there is no synchronizing sequence for this design, and hence this design is not essentially resettable. However, once the design powers up, its state can be determined from its outputs, and based on the outputs the design can be driven to state 0. Thus, the behavior of this design can be controlled.

[3] We are indebted to Dr. Richard Rudell of Synopsys, Inc. for comments about sequential replacement.

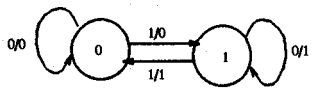

Fig. 3. Example of a design which does not have an synchronizing sequence

4.2 Conditions for Valid Replacement

Here we present our new condition for safe replacement of a sequential design. We assume that no latches have any reset lines[4]. Since it cannot be predicted which state the design powers up in, we can safely assume that no matter which state the original design powers up in, the subsequent input/output behavior of the design is acceptable to the environment. This motivates the following condition (the *safe replacement condition*):

Definition 11. Design D_1 is a *safe replacement* for design D_0 ($D_1 \preceq D_0$) if given any state $s_1 \in Q_{D_1}$ and any finite input sequence $\pi \in I^*$, there exists some state $s_0 \in Q_{D_0}$ such that the output behavior $\lambda_{D_1}(s_1, \pi) = \lambda_{D_0}(s_0, \pi)$.

For example, consider the design D_0 in Figure 4 consisting of 1 input wire, 1 output wire and 3 latches. Design D_1 in Figure 5 satisfies the safe replacement condition ($D_1 \preceq D_0$). States 00, 11 and 11 in D_1 behave like states 000, 011 and 101, respectively, in D_0 for all input sequences. The remaining state 10 in D_1 behaves like state 010 for all input sequences starting with 0, and like state 101 for all input sequences starting with 1.

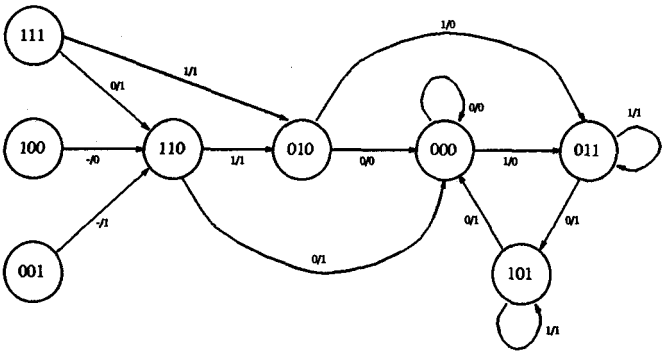

Fig. 4. Example design D_0

We would like to emphasize some interesting properties of the safe replacement condition:

[4] If some latches do have a reset line, they can be modeled by a latch without a reset line if we treat the reset line as another primary input.

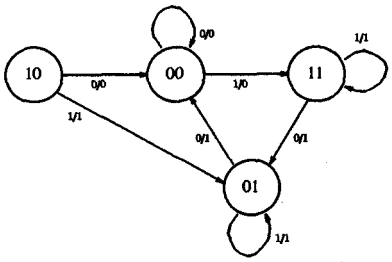

Fig. 5. Replacement design D_1

Remark 1: If $D_1 \preceq D_0$, then there may be states $s_0 \in Q_{D_0}$ and $t_0 \in Q_{D_1}$ such that for all states $s \in Q_{D_0}$ and $t \in Q_{D_1}$, $s_0 \not\sim t$ and $t_0 \not\sim s$. For example, state 111 in D_0 (Figure 4) is not equivalent to any state in D_1 (Figure 5); also, state 10 in D_1 is not equivalent to any state in D_0. Thus classical machine equivalence [5], that requires that every state in each design be equivalent to be some state in the other design, is not necessary for safe replacement, although it is sufficient.

Remark 2: As is obvious from Definition 11, the relation $\preceq$ is reflexive and transitive. However, the relation $\preceq$ is not symmetric. Although $D_1 \preceq D_0$, it is not true that $D_0 \preceq D_1$ because the state 110 in design D_0 produces an output sequence of $1 \cdot 0$ on the input sequence $1 \cdot 1$ and there is no state in D_1 which exhibits this behavior. However, an equivalence relation can easily be defined from a transitive and reflexive one.

Definition 12. Designs D_0 and D_1 are *replacement equivalent* if $D_0 \preceq D_1$ and $D_1 \preceq D_0$.

Consider the equivalence classes of designs modulo replacement equivalence— a design D belongs to an equivalence class $[D_0]$ if and only if it is replacement equivalent to D_0. We say that $[D_1] \preceq [D_0]$ if and only if $D_1 \preceq D_0$. Now, $\preceq$ is a partial ordering on these design equivalence classes (since it can be easily shown that it is reflexive, transitive and anti-symmetric).

Remark 3: Although the replacement design has to be compatible with the original design (same number of inputs and outputs), it does not have to the same number of latches (for example, D_0 in Figure 4 has 3 latches whereas the replacement design D_1 in Figure 5 has only 2 latches).

Remark 4: While the theory of sequential hardware equivalence Section 3 cannot be applied to designs which are not essentially resettable, our safe replacement conditions work for any designs. For essentially resettable designs, sequential replaceability is a stronger condition than SHE (sequential replaceability implies SHE, because as the following Theorem 13 shows, if $D_1 \preceq D_0$, any synchronizing sequence for D_0 can align all state pairs in $Q_{D_0} \times Q_{D_1}$.

Theorem 13. *If $D_1 \preceq D_0$ and ρ is a synchronizing sequence for D_0 then ρ is also a synchronizing sequence for D_1 and for any states $s_0 \in Q_{D_0}$ and $s_1 \in Q_{D_1}$, $\delta_{D_0}(s_0, \rho) \sim \delta_{D_1}(s_1, \rho)$.*

Proof. Pick a state $s' \in Q_{D_0}$. For the proof, we just need to show that for any state $t \in Q_{D_1}$, $\delta_{D_1}(t,\rho) \sim \delta_{D_0}(s',\rho)$. Suppose not. Then $\delta_{D_1}(t,\rho) \not\sim \delta_{D_0}(s',\rho)$. There exists a sequence π such that $\lambda_{D_1}(\delta_{D_1}(t,\rho),\pi) \neq \lambda_{D_0}(\delta_{D_0}(s',\rho),\pi)$. However, since $D_1 \preceq D_0$, there exists a state $s \in Q_{D_0}$ such that $\lambda_{D_1}(t,\rho\cdot\pi) = \lambda_{D_0}(s,\rho\cdot\pi)$. This also means that $\lambda_{D_1}(\delta_{D_1}(t,\rho),\pi) = \lambda_{D_0}(\delta_{D_0}(s,\rho),\pi)$. However, since ρ is a synchronizing sequence for D_0, $\delta_{D_0}(s,\rho) \sim \delta_{D_0}(s',\rho)$. Thus, $\lambda_{D_1}(\delta_{D_1}(t,\rho),\pi) = \lambda_{D_0}(\delta_{D_0}(s,\rho),\pi) = \lambda_{D_0}(\delta_{D_0}(s',\rho),\pi)$, which is a contradiction. ∎

Remark 5: The idea of safe replacement implicitly uses the fact that the original design D_0 can power up in any state. Power-up states are generally beyond the control of designers for physical reasons. It may be possible that, by design, D_0 cannot power up in some states or that the likelihood of powering up in some states is so remote that D_0 is never observed to do so. The notion of safe replacement still applies with replacing Q_{D_0} and Q_{D_1}, in Definition 11, by the power-up states of D_0 and D_1, respectively.

Necessary Conditions for a Safe Replacement Even though there is some flexibility for the implementation of the replacement design, it cannot have arbitrarily few states; in fact, as the following results show, each closed SCC in the replacement design must be equivalent (Definition 5) to some closed SCC in the original design.

Lemma 14 (Lemma 2 in [2]). *Suppose that DFSM's M_0 and M_1 have no equivalent states then there is an input sequence π such that for any states s_0 of M_0 and s_1 of M_1, $\lambda_{M_0}(s_0,\pi) \neq \lambda_{M_1}(s_1,\pi)$.*

Lemma 15. *If $D_1 \preceq D_0$, and $t \in Q_{D_1}$ lies in a closed SCC of D_1, then there exists state $s \in Q_{D_0}$ such that $s \sim t$.*

Proof. (by contradiction). Assume that $D_1 \preceq D_0$. Let M be a closed SCC of D_1. Then M is a DFSM. Suppose that no state of M is equivalent to any state of D_0. By Lemma 14, there is a sequence π that differentiates every state of M from every state of D_0. *A fortiori*, π differentiates a particular state, say s of M from every state of D_0. Therefore, the assumption that $D_1 \not\preceq D_0$ is false. ∎

Theorem 16. *If $D_1 \preceq D_0$, and M_1 is a closed SCC in design D_1, then there must be a closed SCC M_0 in design D_0 such that $M_0 \equiv M_1$.*

Proof. Consider a state t in M_1. From Lemma 15, we know that there exists a state $s \in Q_{D_0}$ such that $s \sim t$. We will show that if M_0 is any closed SCC reachable from s then M_0 is equivalent to M_1.

Consider any state s' in M_0. Let π be an input sequence such that $\delta_{D_0}(s,\pi) = s'$. Since M_1 is closed under all inputs, $t' = \delta_{D_1}(t,\pi)$ lies in M_1. Also, since $s \sim t$, we have $s' \sim t'$.

Similarly, consider any state t'' in M_1. Let ρ be an input sequence such that $\delta_{D_1}(t',\rho) = t''$. Again, $s'' = \delta_{D_0}(s',\rho)$ lies in M_0, and $s'' \sim t''$.

Thus, DFSM's M_0 and M_1 are equivalent. ∎

As remarked previously, the equivalence classes modulo replacement equivalence are partially ordered by safe replacement ($\preceq$). Theorem 16 shows that each closed SCC of design D_0 defines a minimal element that is $\preceq$ to the equivalence class $[D_0]$. Designs with unique minimal predecessors are therefore ones with unique closed SCC's, or ones where all SCC's are equivalent to each other.

Special Case - Known Initializing Sequence Set

Sometimes, the designer knows the initializing sequences for the design and does not need a replacement condition as strong as in Definition 11. The designer knows that whenever the design powers up, one of sequences from an initializing sequence set Π is applied and the design is reset to some desired behavior. The designer does not care about the outputs while an initializing sequence is applied[5], and knows that the design "works" as long as some sequence from Π is applied after power-up. As an example, consider a design with two input lines a and b. The designer knows that after power-up the input a will be set at 1 for the first 2 clock cycles to initialize the design. For this example, the initializing set $\Pi = \{(10 \cdot 10, 10 \cdot 11, 11 \cdot 10, 11 \cdot 11\}$, where a represents the first input and b the second input. For an initializing sequence set Π we can modify our safe replacement condition from Definition 11 to the following:

Definition 17. Design D_1 is a *safe replacement* for design D_0 under the initializing sequence set Π ($D_1 \overset{\Pi}{\preceq} D_0$) if given any state $s_1 \in Q_{D_1}$, an initializing sequence $\pi_1 \in \Pi$, and any finite input sequence $\rho \in I^*$, there exists some state $s_0 \in Q_{D_0}$ and an initializing sequence $\pi_0 \in \Pi$ such that the output behavior $\lambda_{D_1}(\delta_{D_1}(s_1, \pi_1), \rho) = \lambda_{D_0}(\delta_{D_0}(s_0, \pi_0), \rho)$.

We can also derive the following result (the proof is similar to that of Theorem 16 and is omitted for brevity).

Theorem 18. *If $D_1 \overset{\Pi}{\preceq} D_0$ and M_1 is a closed SCC in design D_1, then there must be a closed SCC M_0 in design D_0 such that $M_0 \equiv M_1$.*

It should be noted that for a single design a designer may have more than one set of initializing sequences. Each of these set might be used to initialize the design to a different behavior. For such a situation the designer would like to verify that the replacing design is a safe replacement under each initializing sequence set.

4.3 Verification for Sequential Replaceability

Although we have safe replacement conditions in Definitions 11 and 17, we still need a decision procedure to verify if a replacement design satisfies these conditions. In this section we develop two methods to answer this verification question.

[5] If the designer does care about the outputs during the initialization phase, it is easy to modify our safe replacement condition for such a case also.

Method I - Finding a Discriminating Sequence

The following procedure decides whether (new) design D_1 is a safe replacement for (original) design D_0, i.e, if $D_1 \preceq D_0$. If D_1 is not a safe replacement for D_0 then this algorithm finds a state s_1 of D_1 and an input sequence π that distinguishes s_1 from every state of D_0.

We construct a multiple rooted, acyclic directed graph whose nodes are labeled by pairs of the form (s, A) where s is a state of D_1 and A is a subset of states of D_0. Nodes are either marked or unmarked; the markings may be **FAIL**, **JUMP**(N) or **SUCCEED**, where N is another node. Edges of the graph are labeled by a single input $a \in I$. We presume that the equivalent state pairs of D_0 and D_1 have already been computed, see [7].

The roots of the digraph are pairs of the form (s_0, Q_{D_0}) where s_0 is a state of D_1, Q_{D_0} is the set of all states of D_0 and s_0 is not equivalent to any state in Q_{D_0}. If the set of roots is empty, then clearly $D_1 \preceq D_0$.

loop until some leaf is marked **FAIL** or all leaves are marked **JUMP** or **SUCCEED**:

Choose a node N labeled (s_1, A) of the existing tree that has no **JUMP** or **SUCCEED** mark and choose an input a such that no edge out of N has the label a.

Let $s_1' = \delta_{D_1}(s_1, a)$ and $A' = \{s' |$ for some s in A, $s' = \delta_{D_0}(s, a)$ and $\lambda_{D_1}(s_1, a) = \lambda_{D_0}(s, a)\}$. If a node labeled (s_1', A') already exists, say node N', create an edge labeled a from N to N', and goto the beginning of the loop. Else, create a new edge out of N labeled a and pointing to a new node N' labeled (s_1', A').

Mark the new node N' as follows:

1. If A' is empty, mark the new node N **FAIL** and exit the program.

2. If s_1' is state equivalent to any state in A', mark N **SUCCEED**, and go to the beginning of the loop.

3. If there exists a node N'' labeled (s_1', A'') such that $A'' \subset A'$, mark N' as **JUMP**(N''), and go to the beginning to the loop.

4. For each node N'' labeled (s_1', A'') such that $A'' \supset A'$, mark N'' as **JUMP**(N').

End **loop**

Proof. Termination: Each node must have a distinct label and there can only be finitely many labels. Furthermore each node has an upper bound on the number of edges emanating from it – the number of primary input combinations. Therefore the program must terminate.

If the program terminates because a **FAIL** node is created, any path from a root (s, Q_{D_0}) to the **FAIL** node gives a sequence that distinguishes s from any state of Q_{D_0}. If there is no **FAIL** node then all leaf nodes are marked **SUCCEED** or **JUMP**.

Claim: If all leaf nodes are marked **SUCCEED** or **JUMP** then no input sequence will distinguish a state of D_1 from all the states of D_0.

Observation: We first observe that there cannot be loop of **JUMP** nodes— N_1 -**JUMP**(N_2)→ N_2-**JUMP**(N_3) → $\cdots$ → N_k -**JUMP**(N_1)→ N_1 because

each **JUMP** reduces the cardinality of the second coordinate of the label of a node.

The proof of the claim is by contradiction. Suppose there were a state s_1 of D_1 and an input sequence $\pi = a_1 \cdot a_2 \cdots a_k$ that distinguishes s_1 from all states of D_0. Let $N_0 = (s_1, Q_{D_0})$. Notice that node N_0 cannot be marked **SUCCEED**, because then s_1 would be equivalent to a state in D_0, a contradiction. Construct the sequence of nodes $N_0, N_1, \ldots, N_k$, none of which is marked **SUCCEED**, by applying the following procedure recursively. If node N_i is unmarked then node N_{i+1} is the node reached by traversing the edge labeled a_{k+1} from N_i. Otherwise, N_i is marked **JUMP**(N); jump to N and keep jumping nodes until a node N' is reached which is not marked **JUMP** (see the observation above). This node cannot be marked **SUCCEED**. [This is because nodes marked **SUCCEED** have their left hand component equivalent to some state in their right hand component. But then N_i would have the same property and would have been marked **SUCCEED** rather than **JUMP** which is a contradiction.] Now, N_{i+1} is the node reached by traversing the edge labeled a_{k+1} from N'.

Node N_{i+1} labeled (s_{i+1}, A_{i+1}) cannot be marked **SUCCEED**, because s_{i+1} cannot be equivalent to any state in A_{i+1}. [If it was so, by backtracking the edges traversed, we can find a state in Q_{D_0} that cannot be distinguished from s_1.] Since there are no nodes marked **FAIL**, the last node N_k labeled (s_k, A_k) must have a non-empty set A_k. But then by choosing a state in A_k, and backtracking the edges traversed in constructing the sequence of nodes, one can find an element of Q_{D_0} that is not distinguished from s_1 by π. ∎

Before we execute the above algorithm, we could check to see if each closed SCC of D_1 has a state equivalent to some state of D_0. If not, then by Theorem 16, $D_1 \not\preceq D_0$. Otherwise, if there is no state outside the closed SCC's of D_1, then $D_1 \preceq D_0$ and we are done. If there is state outside the closed SCC's of D_1, we use the method outlined above knowing that each root (s_0, Q_{D_0}) of the digraph is such that s_0 is outside all closed SCC's of D_1. Thus, if the number of states outside closed SCC's is small compared to the the number of states which lie in closed SCC's, we can probably expect our algorithm to be efficient. Also, note that the algorithm would be correct if we did not mark any states **JUMP** (remove substeps 3 and 4 in the marking step). Marking states as **JUMP** is just a way to prune the search space, and make the algorithm more efficient.

Known Initializing Sequence Set First, based on Theorem 18, we check if each closed SCC of D_1 has a state equivalent to some state of D_0. If this check fails, we know that D_1 is not a safe replacement for D_0 under the initializing sequence set Π. Otherwise, we compute sets $Q_0 = \{s |$ there exists $s_0 \in Q_{D_0}$ and $\pi \in \Pi$ such that $\delta_{D_0}(s_0, \pi) = s\}$, and $Q_1 = \{s |$ there exists $s_1 \in Q_{D_1}$ and $\pi \in \Pi$ such that $\delta_{D_0}(s_1, \pi) = s\}$. Now, we can use the same digraph-construction method described above, except that the roots of the digraph are pairs of the form (s, Q_0) such that $s \in Q_1$ and s is not equivalent to any state in Q_0. The proof of correctness is similar to the one above, and is omitted for brevity.

Method II - Using Language Containment

Definition 19. A non-deterministic finite automaton (NFA) is a 5-tuple $A = (Q, \Sigma, \delta, I, F)$, where Q (the set of states) and Σ (the alphabet) are finite non-empty sets and $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition relation, $I \subseteq Q$ is the set of non-deterministic initial states, and $F \subseteq Q$ is the set of final states. The language of A, denoted by $\mathcal{L}(A)$, is a set of finite strings of the alphabet, and is defined as in [6].

Given the original design D_0 and the new design D_1, we will construct two NFA's $A_0 = (Q_{D_0}, \Sigma, \delta_{A_0}, I_{A_0}, F_{A_0})$ and $A_1 = (Q_{D_1}, \Sigma, \delta_{A_1}, I_{A_1}, F_{A_1})$ such that $D_1 \preceq D_0$ if and only if $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_0)$. Here $\Sigma = I \times O$. For $i \in \{0, 1\}$, $\delta_{A_i}(s, (a, b)) = t$ if and only if $\delta_{D_i}(s, a) = t$ and $\lambda_{D_i}(s, a) = b$. Also, $F_{A_i} = Q_{D_i}$ and $I_{A_i} = Q_{D_i}$.

However, since the problem of language containment between two NFA's is PSPACE-complete [4, page 265], this approach is not likely to be more efficient that Method I. Although Method I may also be inefficient in the worst case, we can probably hope to terminate with **SUCCESS** or **FAIL** without exploring the entire search space, especially if D_1 has been derived from D_0 using some synthesis algorithms.

5 Conclusions

We have defined a notion of safe replaceability ($\preceq$) that is independent of initial states of a design, is independent of the intended environment of a design, and applies to all sequential designs, resettable or not. This notion accomplishes for sequential designs what the notion of boolean equivalence accomplishes for combinational designs. We have shown by example that this notion is strictly weaker than the property that every state of the replacement design is equivalent to some state of the original design. We observed that safe replaceability is a reflexive and transitive relation (i.e., a partial ordering of designs). Finally, we gave two algorithms for deciding whether one design is a safe replacement for another.

6 Acknowledgements

We would like to thank the CAD research groups at the University of Colorado at Boulder, Motorola Austin and the University of California at Berkeley for many useful comments on this work.

References

1. H. Cho, G. D. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG Aspects of FSM Verification. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 134–137, 1990.

2. H. Cho, S.-W. Jeong, F. Somenzi, and C. Pixley. Synchronizing Sequences and Symbolic Traversal Techniques in Test Generation. *Journal of Electronic Testing: Theory and Applications*, 4(12):19–31, 1993.
3. O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, June 1989.
4. M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., 1979.
5. J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Intl. Series in Applied Mathematics. Prentice-Hall, Englewood Cliffs, N.J., 1966.
6. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
7. C. Pixley. Introduction to a Computational Theory and Implementation of Sequential Hardware Equivalence. In E. M. Clarke and R. P. Kurshan, editors, *Proc. of the Conf. on Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 54–64, June 1990.
8. H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 130–133, November 1990.