Record & Play: A Structural Fixed Point Iteration for Sequential Circuit Verification*

Dominik Stoffel *Wolfgang Kunz*
Institute of Computer Science III
University of Potsdam
14415 Potsdam, Germany

Abstract

This paper proposes a technique for sequential logic equivalence checking by a structural fixed point iteration. Verification is performed by expanding the circuit into an iterative circuit array and by proving equivalence of each time frame by well-known combinational verification techniques. These exploit structural similarity between designs by local circuit transformations. Starting from the initial state, for each time frame the performed circuit transformations are stored ("recorded") in an instruction queue. In subsequent time frames the instruction queue is re-used ("played") and updated when necessary. At some point the instruction queue does not need to be modified any more and is valid in all subsequent time frames. Thus, a fixed point is reached and machine equivalence is proved by induction. Experimental results show the great promise of this approach to verify circuits after resynthesis and retiming.

1 Introduction

This paper addresses the problem of logic equivalence checking for synchronous sequential circuits. Only completely specified machines are considered. Equivalence of two finite state machines with known initial state can be established by proving the equivalence of the initial state: let A and B denote two completely specified finite state machines with initial states $S_{0,A}$ and $S_{0,B}$, respectively. The finite state machines A and B are called equivalent if and only if for every possible input sequence the same output sequence is produced if both machines are in their respective initial state. In this paper we assume that either the initial state (or a set of initial states) is known for each machine, or an initializing sequence is given to bring the circuit into a well-defined state after power-up. We do not consider the problem of *safe replaceability* as raised by [14].

Conventionally the problem of logic equivalence checking for sequential circuits has been approached by a fixed point iteration based on BDD-based image computations [5]. This is referred to as "symbolic" FSM traversal and has been developped further, e.g., in [3, 4, 6, 15, 16]. In particular, van Eijk [6] describes a BDD-based method exploiting functional dependencies between state variables which is related to the method presented here.

Since BDD-based FSM traversal often suffers from state explosion for large designs, Huang et al. [8, 9] explored structural techniques based on sequential ATPG. Their approach to equivalence checking is based on the assumption that practical designs under comparison often contain a lot of equivalent state variables. The procedure in [8, 9] starts with a set of candidate pairs for equivalent state variables and performs an induction-based filtering process to eliminate the wrong candidates. It is assumed that simple relationships exist between state variables that can be obtained by simulation. This is promising for circuits with very similar encodings but may fail in other cases. Therefore, we take a different approach. Our work is based on the observation that a fixed point iteration for FSM traversal can also be formulated based on structural concepts. This results in a "structural" FSM traversal that leads to a natural way of exploiting similarities between designs but does not rely on the equivalence of state variables.

2 General Idea

Our approach is based on modelling the finite state machine (FSM) by an iterative circuit array. Each combinational block implements the transition function $\delta(\underline{s}, \underline{x})$ and output function $\lambda(\underline{s}, \underline{x})$ of the FSM. The vector $\underline{s}$ denotes the present state variables and $\underline{x}$ denotes the primary inputs of the circuit. The next state variables are denoted $\underline{z}$. Each block in the circuit array describes the behaviour of the machine for a certain time interval $[t, t+1]$ and is simply referred to as *time frame*.

Most techniques for sequential equivalence checking consider a specific product machine [7]. This product machine is obtained by connecting the corresponding inputs to a common fanout stem and attaching an XOR gate to the outputs. If the machines are equivalent the product machine produces a sequence of zeros for all valid input sequences. In our approach we consider the iterative circuit array of the product machine shown in Figure 1. The XOR tree for the outputs is like in a combinational *miter* [1] and is omitted in Figure 1.

In order to anchor the combinational comparison on a common basis for both circuits without making any assumptions on the equivalence of state variables as in [8, 9], we start our procedure at the initial state that we assume is known for both circuits.

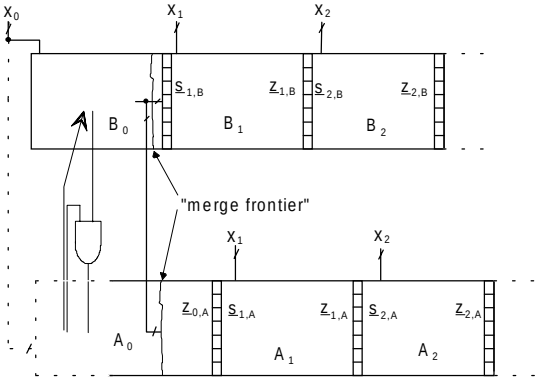

Figure 1: Time frame merging by sharing of logic

Consider Figure 1. The verification procedure is based on performing logic transformations identified by implications as described in [10, 11]. As we proceed time frame by time frame, the combinational logic of one machine is "merged" with the combinational logic of the other machine. We speak of the "merge frontier" denoting a set of gates that identifies a border line between the circuitry that is shared between the two machines and the separate circuitry for each machine. This is schematically shown in Figure 1.

Note that the iterative circuit array is a concatenation of identical combinational blocks. Therefore, many transformations that allow us to merge the logic of some time frame of

*Part of this research was performed while both authors were on sabbatical at the Mentor Graphics Boston Research Laboratories

0-89791-993-9/97 $10.00

machine *A* with the logic of some time frame of machine *B* remain valid also in the next time frame. The only reason why some transformation at time *t* might no longer be valid at time *t*+1 is that certain *controllability don't care conditions* [7] have been used at time *t* that are not valid at time *t*+1. However, similarly as the image computations of the conventional verification approach [5] identify all reachable states after a finite number of iterations, all different don't care sets at the nodes in the circuit array have been examined after a finite number of time frames.

Therefore, it is our goal to identify a set of circuit transformations between the nodes of machine *A* and machine *B* that is valid in all time frames. The basic instrument to find such a set is an *instruction queue* Q_t that contains a set of instructions for circuit transformations at some time *t*. The instruction queue is processed in a *first-in-first-out* manner. Circuit transformations at time *t* are stored in Q_t in the order in which they have been performed. This is operation is called "*record*". In the next time frame at *t*+1, we try to make maximum use of the instructions recorded previously and for each recorded circuit transformation we check whether or not it is still valid at time *t*+1. Only if it is invalid it is removed from the instruction queue, otherwise the transformation is performed also at *t*+1. This process of reusing the stored instructions is referred to as "*play*". If the circuit transformations in the instruction queue are not sufficient to establish the equivalence of the output signals at time *t*+1 additional transformations are identified and recorded.

By *recording* and *playing* the instruction queue is improved in each time frame. Finally, an instruction queue is created that remains valid also in the next time frame and which completes the task of proving the equivalence for the circuit outputs. Now it remains to be shown that this instruction queue is also valid in all future time frames. To permit this induction a *cutting* procedure is applied to the circuit array. The cut must be selected such that the don't care conditions needed for the validity of the circuit transformations are still present in the selected part of the circuit array. Most don't care conditions that allow us to merge logic between the two machines by local transformations are induced by the previous merging steps. Therefore, it is usually sufficient to keep only a few levels of logic in front of the merge frontier and to cut off all circuitry starting from the beginning of the circuit array up to a few levels before the merge frontier. If an instruction queue can be played a sufficient number of times, it is guaranteed that the combinational structure in the circuit array repeats periodically. Then we say that our procedure has reached a "structural fixed point". This is a simplified summary of the proposed method. It is described in more detail in the following section.

3 The *record & play* procedure

Figure 2 shows the proposed algorithm for FSM equivalence checking. Routine *record&play()* is given in Table 1. The different steps of the algorithm are illustrated by means of an example depicted in Figure 3 to Figure 9.

The variables *stub_levels* and *p_max* must be defined by the user as will be discussed later. At the beginning of each iteration a new time frame is attached to the current circuit array. Initially, the circuit array is empty. Whenever a new time frame is attached we assign constant values of the state variables to the corresponding nodes in the circuit array and simplify the logic accordingly. Initially, the constant values are given by the initial state. Note that these constant values may propagate to the next state variables. Consider the example in Figure 3. For both machines we are given an initial state of 0 for all registers. This leads to the situation shown in the left part of Figure 4. A constant value of 0 has propagated to the next state vector and it is $j_1 = 0$. This value will be propagated further when the next time frame is attached.

After the time frame has been attached the algorithm optimizes the logic to facilitate equivalence checking at the outputs in this and subsequent time frames. This optimization is performed in a controlled way by an instruction queue in order to detect a fixed point. For each time frame we store a set of instructions Q_t that keeps exact records of all transformations performed in that time frame. Routine *record&play()* has the task to select one of the previous instruction queues and to determine for the selected queue whether or not the recorded circuit transformations are still valid in the current time frame. If this is not the case another instruction queue is tried. Trying a large number of instruction queues can be time consuming, therefore the user-defined parameter *p_max* is used to restrict the search to the last *p_max* instruction queues. If no instruction queue is found that can be played successfully, new circuit transformations are identified and stored in the instruction queue. In our example, no previous instruction queue exists. The circuit is optimized as shown in the right part of Figure 4 and the performed transformations are stored in Q_0. For reasons of simplicity, in our example, node substitutions are considered as the only possible type of circuit transformations.

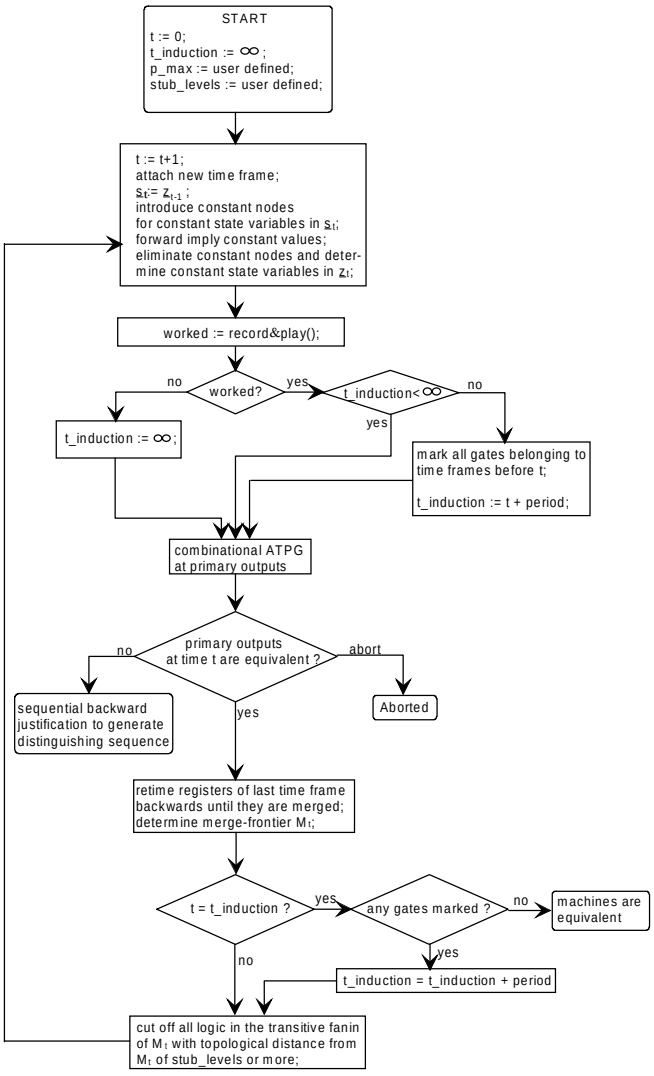

Figure 2: sequential equivalence checking algorithm

Next, it is checked whether the primary outputs in the current time frame are equivalent. If this is not the case, the circuits are not equivalent and a backward justification process like in

conventional sequential ATPG tools is invoked to calculate a distinguishing sequence.

```
/* routine operates on a global data structure for the current miter array
   with present state variables s and next state variables z and has t,
   t_induction, period and p_max of Figure 2 as global variables */

record&play()
{
  if (t_induction < ∞)  /* trying induction */
      PLAY := {Q_t-period};
  else  /* check old queues to find fixed point */
      PLAY := { Q_i | i ∈ {t-1, t-2, ..., t-p_max} and s_i = s_t};

  worked := NO;
  for (each Q_i ∈ PLAY)
  {   worked := YES;
      Q_t := ∅;
      for (each instruction α_j ∈ Q_i)
      {   verify whether or not circuit transformation α_j is valid
          in current time frame;
          if (valid)
          {   execute α_j (perform circuit transformation);
              Q_t := Q_t ∪ {α_j}; /* put in queue */
          }
          else  worked := NO;
      }
      if (worked = YES)
      {   period := t - i;
          break;
      }
      else reverse all transformations made for Q_i;
  }
  if (worked = NO)
  {   Q_t := ∅;
      for (each node in circuit array)
      {   identify implication based circuit transformation, α;
          if (α reduces literal count of circuit array)
          {   perform transformation α;
              Q_t := Q_t ∪ {α}; /* put in queue */
          }
      }
  }
  return (worked);
}
```

Table 1: Routine *record&play()*

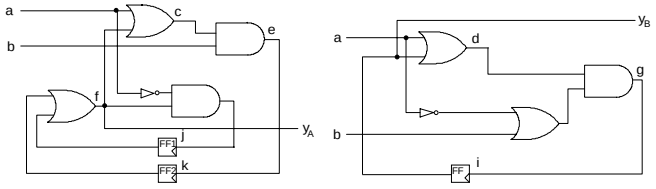

Figure 3: Circuit examples with initial states $S_{0,A} = S_{0,B} = \underline{0}$

The algorithm now determines whether previously processed portions of the circuit array can be cut off. Note that our algorithm only performs local transformations in the circuit array. For this reason it does usually not affect the quality of the performed circuit transformations if we cut off previously processed circuitry in a sufficiently large distance from the currently active area. A heuristic procedure based on *retiming* [12] determines the "merge frontier". This is accomplished by moving the registers at the end of the last time frame backwards until they are located in fanout branches such that different branches of the same fanout stem feed registers of different machines. The corresponding fanout stems represent the nodes of the merge frontier. The detailed description of this procedure must be omitted here for reasons of brevity. The cut through the circuit array is located in the transitive fanin of the merge frontier. In principle, *false negatives* can occur as a result of this cutting process. However, in practice we can always avoid false negatives by leaving a sufficient number of logic levels in front of the merge frontier. This number of logic levels is called *stub_levels* in Figure 2 and is a user-defined parameter. Typical values are between 0 and 5.

Consider again the right portion of Figure 4. We only consider the registers that are not assigned a constant value and are still physically present in the circuit array. These are the registers at k_1 and i_1. Note that k_1 stems from machine A and i_1 from machine B. They are located in fanout branches of the same fanout stem, hence e_0 belongs to the merge frontier, which here does not have any other nodes. In the example, we assume *stub_levels* = 0. Hence the circuit array can be cut at signal e_0 and a new time frame is attached. The result is shown in the left portion of Figure 5. The newly introduced variable at the cut line is called *s*.

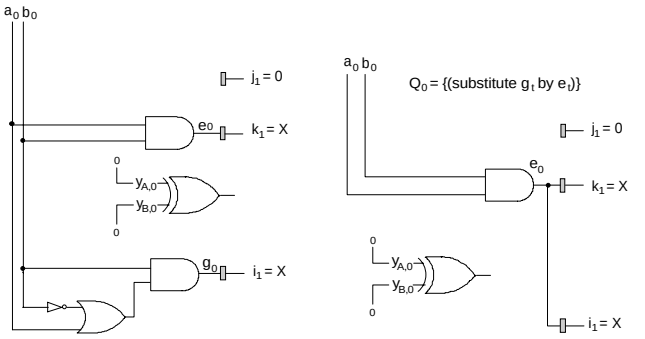

Figure 4: Circuit array in first iteration

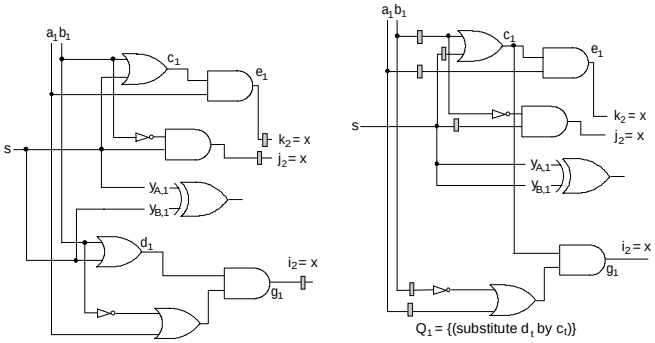

Figure 5: Circuit array in second iteration before (left) and after (right) merging

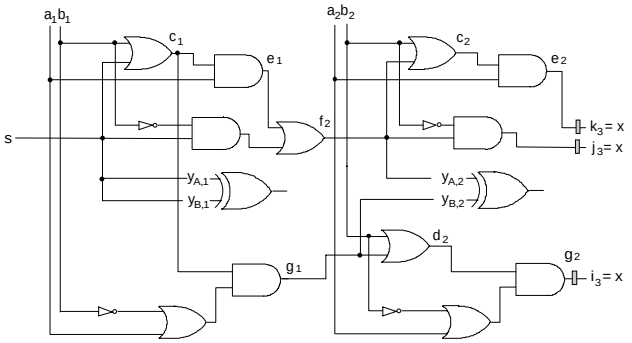

Figure 6: Circuit array in third iteration before merging

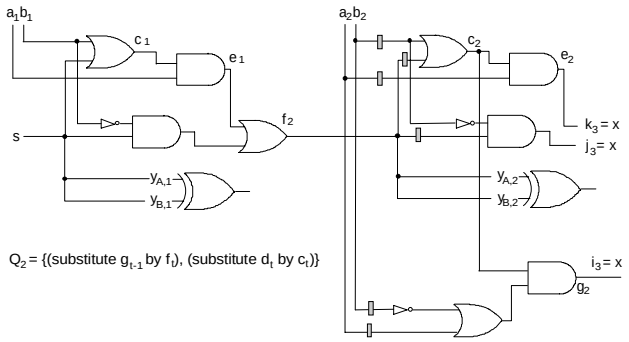

Figure 7: Circuit array in third iteration after merging

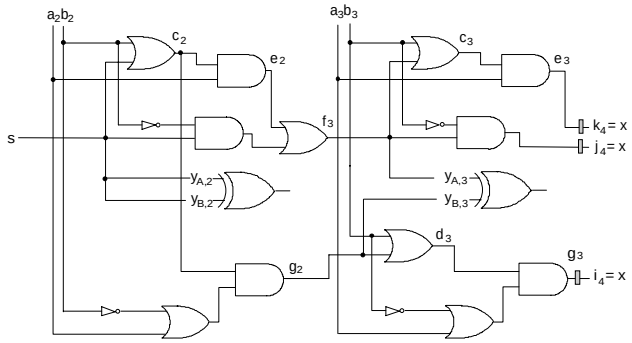

Figure 8: Circuit array in fourth iteration before merging

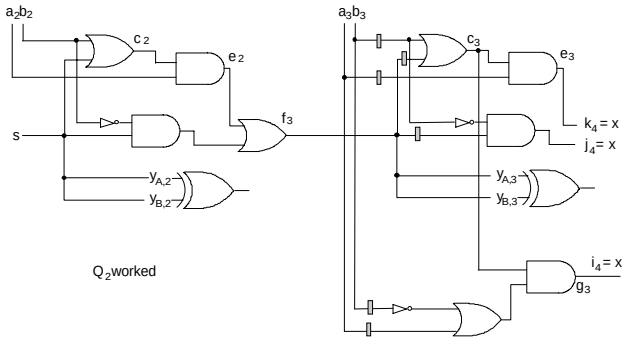

Figure 9: Circuit array in fourth iteration after merging

After a new time frame has been appended it is always checked whether a previous instruction queue can be played. As given by the definition of set *PLAY* in routine *record&play()* an instruction queue can only be played if it was recorded with the same constant values at the state variables as are given in the current time frame. In our example, no instruction queue can be played. New transformations are recorded in Q_1 as shown in the right portion of Figure 5. As a result of the optimization it is trivial to determine the equivalence of the primary outputs. Next, a merge frontier is determined as shown in the right portion of Figure 5. Assuming *stub_levels* = 0 we can cut the circuit array at the stems of these fanout systems. Here, this does not result in any removal of logic.

A new time frame is attached as shown in Figure 6 and a new instruction queue must be recorded. The transformations lead to the circuit array shown in Figure 7. We determine a new merge frontier suggesting a cut at signal f_2. The next iteration leads to the circuit array of Figure 8. Just like in the previous time frame no constant values exist at the state variables and it is determined in *record&play()* that the instruction queue Q_2 can be played. Actually, all recorded transformations turn out to be valid in the current time frame so that the circuit of Figure 9 results.

If an instruction queue has been played successfully the algorithm enters the induction mode. This is done by setting variable *t_induction* to *t* + *period* where *period* is the number of cycles since the successful instruction queue has been recorded. In most practical cases, this is the most recently recorded instruction queue so that *period* = 1. Furthermore, to ensure the correctness of the induction all gates belonging to time frames prior to the current time frame are marked. The algorithm continues the iteration and in each new time frame the instruction queue recorded at time *t - period* is played. This is done until all queues of a period have been played. Remember that we have marked all gates of previous time frames when we started the induction mode. We continue to play the instruction queues until all marked gates have disappeared as a result of the cutting procedure. At this point, it is guaranteed that the combinational structures generated in the circuit array will repeat periodically and hence, a *structural fixed point* of the iteration is reached.

Note that our method is not restricted to a unique initial state. If a set of initial states is used additional circuitry must be attached in front of the first time frame that encodes the given set of initial states. This is similar to the notion of the *stub circuit* to be described in the next section. If an initializing sequence is given the above iteration has to be slightly modified. Instead of assigning an initial state at the state variables, the values of the initializing sequence are assigned to the primary inputs for each iteration. During the application of the initializing sequence the equivalence check at the outputs is switched off unless the designer wants to check the equivalence of the machines also during the initializing process [14].

4 Discussion of Theoretical Issues

For a better understanding of the *record&play* procedure we now consider a more general formulation of a structural fixed point iteration. We consider a FSM $M = (I, S, \delta, S_0, O, \lambda)$ where I is the input alphabet, S is the set of states, $\delta : S \times I \rightarrow S$ where the next-state function, S_0 is the initial state, O is the output alphabet and $\lambda : S \times I \rightarrow O$ is the output function. For simplicity we restrict our discussion to a single initial state, however, a set of initial states can be treated in a similar way. To examine the nature of the proposed fixed point iteration we can ignore the output behavior of the machine, i.e., we only consider the corresponding *finite state transition structure* FST $H = (I, S, \delta, S_0)$.

We consider the iterative circuit array of the FST expanded from time 0 to time *t*. At time 0 the circuit array is initialized, i.e., the initial state is assigned to the state variables. The upper part of Figure 10 shows the time frame for time *t*. The transition function $\delta(\underline{s}, \underline{x})$ is implemented by combinational circuitry denoted *A*.

Let $S(t)$ denote the set of states reachable at state variables $\underline{s}_t$ and let $S(t+1)$ denote the set of states reachable at the next state variables $\underline{z}_t$. Figure 10 shows a decomposition of δ into functions δ'_t and σ_t. The combinational circuitry A' with output vector $\underline{c}_t$ of length k implements function $\delta'_t(\underline{s}, \underline{x})$. The circuit labelled $STUB_t$ implements function $\sigma_t(\underline{c}_t)$ and is called *stub circuit*. The *stub function* σ_t is defined to be a function that maps the set of all combinations of value assignments at the variables $\underline{c}_t$ to exactly those combinations of value assignments at $\underline{z}_t$ that correspond to the reachable states $S(t+1)$, i.e., we define $\sigma_t : \{0, 1\}^k \rightarrow S(t+1)$. Furthermore, for a valid decomposition it must hold that $\sigma_t(\delta'_t(\underline{s}, \underline{x})) = \delta(\underline{s}, \underline{x})$.

This decomposition has an important property: if we cut through the circuit array at the variables $\underline{c}_t$ and only maintain the circuit array starting from $STUB_t$, the set of reachable states at the variables $\underline{z}_t$ or at any state variables of later times does not change. By the above decomposition in combination with cutting off all logic in front of the stub circuit we loose functional information telling us under what conditions at the inputs

in previous time frames we can reach certain states at time t+1. However, by definition of this decomposition we do not loose the information what states are reachable at time t+1 or later. Therefore, this decomposition and cut can be considered as structural analogy to the *existential abstraction* operation used in the image computations for the conventional BDD-based fixed point iteration.

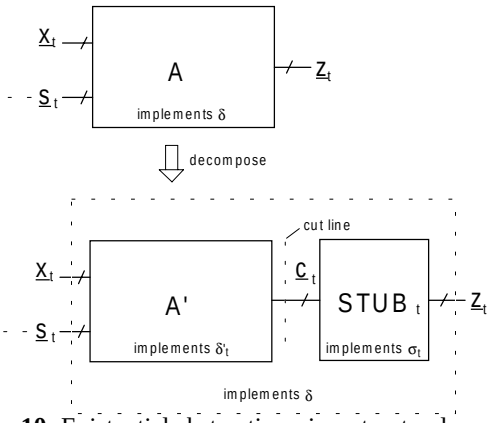

Figure 10: Existential abstraction - in a structural way

The question arises how this decomposition can be computed. One possibility is to choose $\delta' = \delta$. In this case the stub function σ_t has to be determined such that all combinations of values that can occur in the circuit array at $\underline{c_t}$ are mapped to identical values at $\underline{z_t}$, and the combinations of value assignments that can not occur at $\underline{c_t}$ are mapped to values at $\underline{z_t}$ that can occur in the circuit array. A synthesis procedure for the a stub circuit can be formulated based on the AND/OR reasoning graphs of [10, 17]. Assume that the unreachable states at a time t are represented by a list of cubes. As an example for such a cube, assume that (z_1 = 1, z_2 = 1, z_3 = 0, z_4 = X) denotes some unreachable states for the next state variables at time t. Also assume that the cube list is prime. Therefore, if value assignments z_1 = 1 and z_2 = 1 are made it can be *implied* that z_3 must be 1 to obtain a reachable state. Hence, in the given circuit array, z_1z_2 must be an implicant for z_3. Implicants that consist of literals belonging to arbitrary nodes in a Boolean network can be calculated using the method described in [10, 17]. Therefore, the stub circuit can be constructed as follows. For each next state variable at time t calculate all prime implicants that exist in terms of the other next state variables at time t and add them to the cover of the current next state variable. This introduces redundancy in the circuit but does not change its function. In fact, the combinational circuit implementing the added implicants represents a stub circuit according to the above definitions. We can now cut off all other logic without affecting the number of reachable states at $\underline{z_t}$.

In practice, approximate solutions must be considered. Instead of only calculating implicants for the state variables we perform implication-based transformations at *all* nodes in the network. Our heuristic to "merge" as much logic as possible between the different parts of the product machine is the attempt to compress all information about the reachability of states (and internal don't cares) into a relatively small area of the circuit array near the next state variable of the last time frame. If we did not put any restrictions on the recursion depth and computed all implicants at the next state variables we would always succeed in finding a decomposition with the above properties.

Based on this combinational decomposition we can formulate the fixed point iteration shown in Figure 11. In each iteration the above described decomposition is performed to create the stub circuit. Then all logic in front of the stub circuit is cut off. As explained, this does not affect the set of reachable states at the state variables of all future time frames. Note an important difference to the conventional FSM traversal. In each iteration we only take into account the number of states reachable exactly at time t but not the set of all states reachable at any time 0, 1, ...t. This must be considered when detecting the fixed point. The fixed point is reached when there is a sequence $S(t)$, $S(t + 1)$, ... $S(t + p)$ of sets of states reachable at time t, $t + 1$, ... $t + p$ that repeats with a period of length p. This is detected by determining that the decomposition steps to produce the stub circuit are the same as the steps performed p time frames earlier. In our practical implementation an instruction queue is used for this purpose. Since we are dealing with a *finite* state machine the sets of states reachable at a given time must repeat with a finite period and it follows that the fixed point iteration of Figure 11 reaches a fixed point for any completely specified, deterministic FST after a finite number of steps.

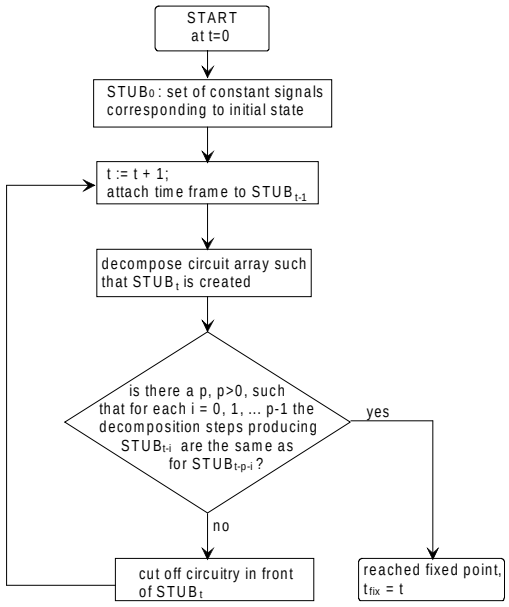

Figure 11: Structural fixed point iteration

Fortunately, p is very small for most practical circuits. This is confirmed by our experimental results as well as a theoretical analysis. This analysis must be omitted for reasons of brevity but can be summarized as follows: suppose we are given a machine M that is strongly connected [7]. We consider all q cycles in the state transition graph of M. Let $P = \{p_1, p_2, ...p_q\}$ be a set of integer numbers such that each number corresponds to the length of a cycle in the state transition graph of M. Then, the period p in our procedure is given by the *greatest common divisor* for all numbers in P. This explains why a period of 1 is sufficient for most practical cases. If the machine is not strongly connected the same result applies for the *strongly connected component* (SCC) of the machine.

Since the *record&play* procedure of the previous section only calculates an approximate solution to the above decomposition problem the resulting stub circuit represents a superset of $S(t)$, i.e., we may consider more states than are actually reachable. As discussed earlier, this can lead to false negatives. However, since a complete symbolic state traversal is impossible for most large designs usually the same information about unreachable states and local don't cares that has been used by the synthesis tool can also be compressed into the stub circuit using local transformations.

On the other hand, the fact that we may consider more states than are actually reachable can have a very beneficial effect.

With an approximate decomposition we may reach a fixed point much faster than with the exact solution.

Note that the methods [8, 9] can be useful as a preprocessing phase to our approach. If equivalent state variables can be determined then the corresponding substitutions can be added to the instruction queue for *every* time frame so that a fixed point is reached much faster.

5 Experimental Results

A prototype of the described approach has been incorporated into the HANNIBAL [10] package. We evaluated the techniques by verifying circuits of the ISCAS89 benchmark set against the optimized and retimed circuits. The circuits were optimized by kerneling (using *fx* in SIS). After optimization retiming is performed (using *retime* in SIS). The resulting circuits were verified against the original ones. For the original circuits we assumed an initial state of 0 for all registers.

Circuit		***record&play*** (HANNIBAL)		***verify_fsm*** (SIS)
Name	# registers	# iterations til fixed point	CPU-time (h:min:sec)	CPU-time (h:min:sec)
s208	8	15	0: 00: 08	0: 00: 03
s298	14	10	0: 00: 09	0: 00: 03
s344	15	9	0: 00: 11	0: 00 06
s349	15	9	0: 00: 11	0: 00: 06
s382	21	16	0: 00: 17	0: 00: 38
s386	6	9	0: 00: 48	0: 00: 01
s420	16	27	0: 00: 43	0: 27: 19
s444	21	16	0: 00: 18	0: 00: 28
s510	6	12	0: 00: 35	0: 00: 02
s526	21	21	0: 00: 35	0: 00: 12
s635	32	37	0: 01: 32	unable
s641	19	9	0: 00: 12	0: 00: 08
s713	19	9	0: 00: 12	0: 00: 09
s820	5	17	0: 36: 50	0: 00: 04
s832	5	16	0: 26: 37	0: 00: 04
s838	32	51	0: 08: 13	unable
s953	29	11	0: 01: 09	0: 00: 12
s1196	18	6	0: 00: 40	0: 00: 08
s1238	18	6	0: 00: 46	0: 00: 08
s1423	74	14	0: 03: 31	unable
s1512	57	16	0: 04: 09	> 30 h
s3271	116	19	0: 21: 17	unable
s3330	132	9	0: 11: 33	unable
s3384	183	17	0: 31: 24	unable
s4863	104	8	0: 36: 52	unable
s5378	179	36	0: 55: 23	unable
s6669	239	11	0: 47: 15	unable

Table 2: Verification of optimized and retimed circuits

A difficulty in our experimental evaluation comes from the fact that for larger circuits SIS cannot compute the initial state after retiming, because it uses symbolic FSM traversal for this task. Therefore, we developped a simple retiming algorithm that moves all registers as far as possible into forward direction. The new initial state can simply be calculated by forward implication of the old initial state. We applied our own retiming method to those circuits where SIS did not move any registers. In this way, we ensured that the encoding for all optimized circuits differs drastically from the encoding of the unoptimized original circuits and no simple relationships exist between state variables.

Table 2 shows our experimental results for a SUN Ultra I workstation. In [8, 9] a different notion of equivalence is used and no results are shown for circuits that are both optimized and retimed. Therefore we compare our techniques only with the conventional verification approach by symbolic FSM traversal. The results for *verify_fsm* in SIS are shown in the left column of Table 2. The results show the feasibility and great potential of this approach to verify circuits after synthesis and retiming. With our technique the verification could be completed within acceptable CPU-time for several cases where the conventional approach fails. In all cases the fixed point was reached with $p\_max = 1$.

Conclusion

Based on a combinational decomposition procedure we have introduced a method for structural FSM traversal. Our approach is promising in applications where the designs under comparison have been modified by a sequence of local transformations. This is the case after most industrial synthesis procedures and retiming. Future work will examine whether the proposed fixed point iteration is useful to verify typical engineering changes at the RTL level, like modifying the pipelining of the circuit.

References

[1] Brand D.: “Verification of Large Synthesized Designs”, *Proc. Int. Conf. on Computer-Aided Circuit Design* (ICCAD), Santa Clara, pp. 534-537, Nov. 1993.

[2] Bryant R.: “Graph-based algorithms for Boolean function manipulation“, *IEEE Transactions on Computers*, vol. 35, pp. 677-691, August 1986.

[3] Cabodi G., Camurati P., Quer S.: "Improved Reachability Analysis of Large Finite State Machines*", Proc. Int. Conf. on Computer-Aided Design* (ICCAD), pp. 354-360, 1996.

[4] Cho H. et al.: "A Structural Approach for State Space Decomposition for Approximate Reachability Analysis", *Proc. IEEE Int. Conf. on Computer Design*, pp. 236-239, 1994.

[5] Coudert O., Berthet C., and Madre J.Ch.: "Verification of Synchronous Sequential Machines Based on Symbolic Execution", in *Lecture Notes in Computer Science*, vol. 407, (Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France), Springer-Verlag, June 1989.

[6] van Eijk C.A.J., and Jess J.A.G.: "Exploiting Functional Dependencies in Finite State Machine Verification", *Proc. European Design & Test Conf.*, pp. 9-14, 1996.

[7] Hachtel G., and Somenzi F.: *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, Boston 1996.

[8] Huang S.Y., Cheng K.T., Chen K.C., and Gläser U.: "An ATPG-based Framework for Verifying Sequential Equivalence" *Proc. Int. Test Conference*, 1996.

[9] Huang S.Y., Cheng K.T., Chen K.C., and Gläser U.: "On Verifying the Correctness of Retimed Circuits" *Proc. Great Lakes Symposium on VLSI*, 1996.

[10] Kunz W., and Stoffel D.: "Reasoning in Boolean Networks - Logic Synthesis and Verification Using Testing Techniques" *Kluwer Academic Publishers*, Boston 1997.

[11] Kunz W., Stoffel D., and Menon P.: “Multi-Level Logic Optimization and Equivalence Checking by Implication Analysis", *IEEE Transaction on Computer-Aided Design*, Vol.16, no.3, pp. 266-281, March 1997.

[12] Leiserson C.E., and Saxe J.B.: "Retiming Synchronous Circuitry", *Algorithmica*, vol. 6., pp. 5-35, 1991.

[13] McMillan K.L.: "Symbolic Model Checking", *Kluwer Academic Publishers*, 1993.

[14] Pixley C., Singhal V., Aziz A., and Brayton R.K.: "Multi-level Synthesis for Replaceability", *Proc. Int. Conf. on Computer-Aided Design* (ICCAD), pp. 442-449, 1994.

[15] Quer S. et al.: "Incremental Re-encoding for Symbolic Traversal of Product Machines", *Proc. European Design Automation Conf.*, 1996.

[16] Ravi K., and Somenzi F.: "High Density Reachability Analysis", *Proc. Int. Conf. on Computer-Aided Design* (ICCAD), pp. 154-158, 1995.

[17] Stoffel D., and Kunz W.: "AND/OR Reasoning Graphs for Determining Prime Implicants in Multi-Level Combinational Networks", *Proc. Asia and South Pacific Design Automation Conference* (ASP-DAC), pp. 529 - 538, Japan, 1997.