Universität Potsdam - Institut für Informatik

Structural FSM Traversal - Theory and a Practical Algorithm

Dominik Stoffel and Wolfgang Kunz

Preprint 005/1997
November 1997

ISSN 0946-7580

Adresse: Universität Potsdam
Institut für Informatik
Postfach 601553
D-14415 Potsdam

Tel.: + 49 331 977 1173
Fax: + 49 331 977 1190

email: dominik@mpag-inf.uni-potsdam.de
kunz@mpag-inf.uni-potsdam.de

Zusammenfassung

Die vorliegende Arbeit beschreibt eine neue Methode zur Traversierung finiter Zustandsautomaten, die wir "*structural FSM traversal*" genannt haben. Der Ansatz beruht auf einer funktionalen Dekomposition zur Durchführung einer existentiellen Abstraktion der in jedem Taktzyklus erreichbaren Zustandsmenge. Wir beschreiben eine exakte Methode zur Berechnung aller erreichbaren Zustände des finiten Automaten und analysieren ihre theoretischen Eigenschaften. Um den Nutzen des beschriebenen Ansatzes für industrielle Anwendungen zu zeigen, betrachten wir ein Näherungsverfahren für das funktionale Dekompositionsproblem und wenden es darauf an, die Äquivalenz zweier sequentieller Schaltungen mit Anfangszustand nachzuweisen. Dazu verwenden wir ein iterative Schaltungsmodell, bei dem die sequentielle Schaltung iterativ in sog. "*time frames*" expandiert wird. Ausgehend vom Anfangszustand werden Schritt für Schritt in jedem *time frame* Operationen durchgeführt, die die Äquivalenz der primären Ausgänge überprüfen und die Schaltlogik der einen Schaltung in die Schaltlogik der anderen Schaltung überführen. Diese Operationen werden in einer "*instruction queue*" gespeichert. Die *instruction queue* wird iterativ in jedem *time frame* erweitert bis zu dem Punkt, wo beim Schritt in den nächsten *time frame* keine weiteren Veränderungen erforderlich sind. Die Äquivalenz der Schaltungen folgt dann durch Induktion. Experimentelle Ergebnisse belegen das große Potential dieser Methode für die Verifikation von Schaltungen nach der Synthese mit *retiming*.

Schlüsselwörter: formale Verifikation, Äquivalenztest, sequentielle Schaltung, retiming

Abstract

This paper proposes a new technique for the traversal of finite state machines (FSMs) called *structural FSM traversal*. The approach is based on a functional decomposition to perform existential abstraction on the set of reachable states at each clock cycle. We present an exact algorithm for structural FSM traversal and discuss its theoretical properties. To make our approach practical we consider an approximation to the decomposition problem and apply our method to sequential logic equivalence checking. For a practical implementation of the proposed FSM traversal the machine is expanded into an iterative circuit array and well-known combinational synthesis techniques are applied in each time frame. Starting from the initial state, for each time frame we check the equivalence of the outputs and 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 re-synthesis and retiming.

Keywords: formal verification, finite state machine traversal, reachability analysis, equivalence checking

Computing Reviews Classification: B 6.3

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

1 Introduction

This paper describes a new approach for the traversal of *finite state machines* (FSMs) and its application to the problem of equivalence checking for sequential circuits. In order to evaluate functional properties of a finite state machine it is necessary to (explicitly or implicitly) explore all states that can be reached in the machine by any valid input sequence. Such *reachability analysis* is a very complex problem as it requires the complete traversal of the machine's state space. However, once the set of all reachable states in a machine is known, many functional properties of a design can be proved in a fairly easy way.

In the past, essentially two approaches for FSM traversal have been used. Older methods are based on building the *state transition graph* (STG) for the machine. If the STG is available reachability analysis can be performed with polynomial complexity by well-known algorithms for cyclic graphs, see e.g. [HaSo96]. However, representing a finite state machine by an STG is tractable only for very small designs. Therefore, most modern tools take a different approach referred to as *symbolic FSM traversal.* First proposed by Coudert et al. [CoBe89] and McMillan [McMi93] this approach exploits the power of binary decision diagrams (BDDs) [Brya86] for representing Boolean functions. The problem of calculating all reachable states in a machine is mapped into a symbolic Boolean problem by introducing characteristic Boolean functions that describe what states are reachable in the machine until some clock cycle. Based on this Boolean state representation a fixed point iteration can be performed that calculates all reachable states in the machine. Symbolic FSM traversal has become an integral part of many tools for formal verification and has led to great improvements over previous methods. Many refinements of symbolic FSM traversal have been developed such as [CaCa96], [EiJe96], [Quer96], [RaSo95]. Yet, the performance of verification tools is far from satisfactory. Even for small and mid-size circuits there are many examples where symbolic FSM traversal fails.

In this paper we develop a new method for FSM traversal which we call *structural FSM traversal*. We demonstrate the feasibility and great potential of this approach in formal verification by applying it to *sequential logic equivalence checking*. It is beyond the scope of this paper, however, to investigate its usefulness in the important domain of *model checking* [McMi93].

Equivalence of two finite state machines with known initial state can be established by proving the equivalence of the initial states: 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 [PiSi94].

Since BDD-based FSM traversal often suffers from state explosion for large designs, Huang et al. [HuCh96], [HuGl96] 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 [HuCh96], [HuGl96] 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. As mentioned, our work is based on the observation that a fixed point iteration for FSM traversal can also be formulated based on structural concepts. The resulting structural FSM traversal leads to a natural way of exploiting similarities between designs but does not rely on the equivalence of state variables.

2 Preliminary Remarks

Our approach is based on modeling the finite state machine (FSM) by an iterative circuit array. The iterative circuit array for a synchronous sequential circuit is obtained by breaking up all feedback loops and by replicating the combinational logic for each clock cycle being considered. 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 behavior 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. This product machine is obtained by connecting the corresponding inputs to a common fanout stem and attaching an XOR tree to the outputs [HaSo96]. If the machines are equivalent the product machine produces a sequence of zeros for all valid input sequences. The iterative circuit array of the product machine is also called *miter array* and is shown in Figure 10.

Obviously, the two finite state machines A and B are equivalent if they produce a constant 0 at the output of the XOR tree for any input vector $\underline{x}_t$ in every time frame $t = 0, 1, 2, 3, \ldots \infty$. This can be checked by combinational techniques in each time frame. In order to anchor the combinational comparison on a common basis for both circuits we start the procedure at the initial state that we assume is known for both circuits. At time $t = 0$ the circuit array is initialized, i.e., we assign the initial state to the state variables and forward imply the constant values. The circuit array can then be simplified by removing all constant nodes. We can now traverse the product machine by combinational techniques that in each time frame check whether or not a constant 0 is obtained at the output of the XOR tree. Since the number of states in the machine is finite we can stop this procedure after a finite number of time frames when we are sure that all possible states have been considered. However, without any additional concepts we can stop only after we have traversed 2^n time frames where n is the number of state variables. This is impractical for all but the smallest designs. Fortunately, often, the sequential depth of finite state machines is much smaller and all legal states can be reached after a relatively small number of clock cycles for many practical circuits. Obviously, we can finish the verification process at a time t_{fix}, if we know at t_{fix} that no states are reachable in later time frames that have not been reached before, i.e., the set of reachable states has reached a fixed point. Standard verification techniques check this condition by explicitly or implicitly calculating the reachable state sets at each clock cycle starting from time $t = 0$. However, such *reachability analysis* is a major bottle neck in most verification methods and represents an active field of research.

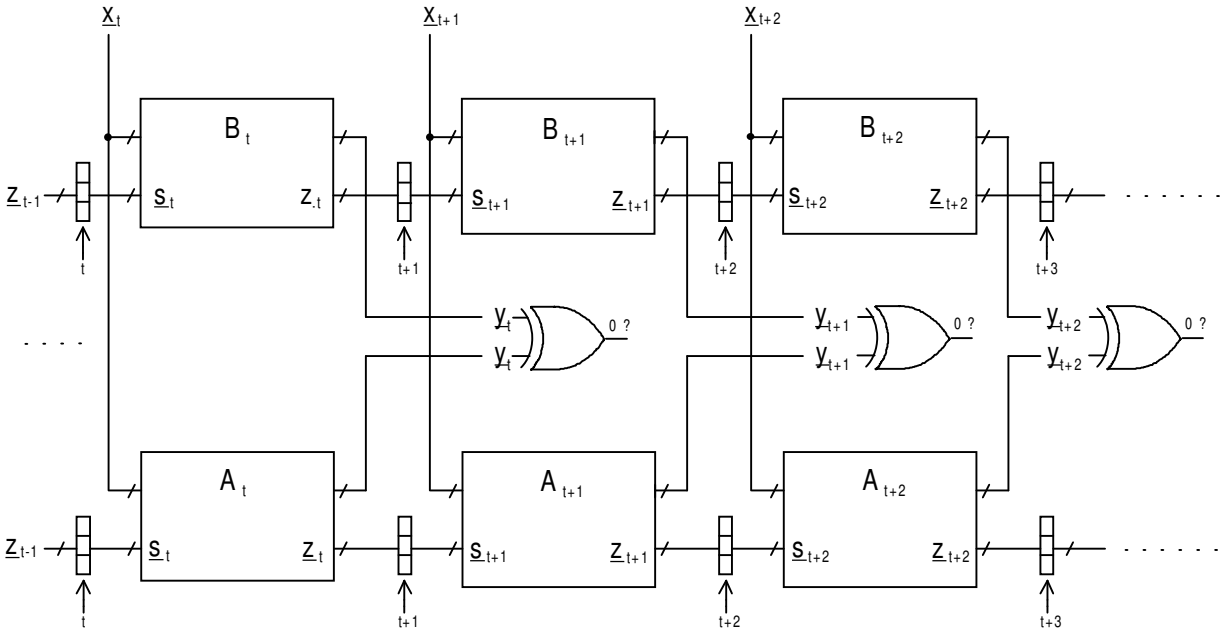

Figure 1: Iterative circuit array of product machine for equivalence checking ("miter array")

Also in our approach a lot of effort is dedicated to gain knowledge about the states reachable in each time frame. However, we do not represent these states explicitly by state transition graphs or by BDDs. Instead, the information about the reachable states is compressed into a general combinational network called *stub circuit*. In this way we avoid memory blow-up in the case of state explosion and also pave the way towards approximative methods that seem unavoidable to overcome the complexity barriers imposed by large designs. Since the information on reachable states is stored in a non-canonical way and the stub circuit is obtained by logic synthesis techniques exploiting the structural properties of the considered FSMs, we speak of *structural FSM traversal*.

3 Structural FSM Traversal

We consider an FSM $M = (I, S, \delta, S_0, O, \lambda)$ where I is the input alphabet, S is the set of states, $\delta : S \times I \rightarrow S$ is 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 now 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 structural FSM traversal 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 as shown in Figure 2. At time 0 the circuit array is initialized, i.e., the initial state is assigned to the state variables. The constant

values at the state variables are implied forward and all constant nodes are removed from the circuit array.

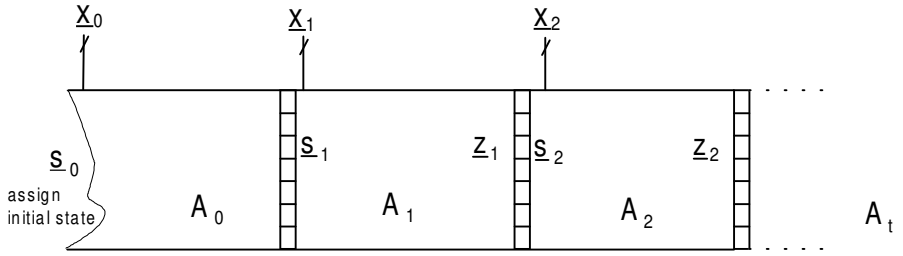

Figure 2: Iterative circuit array of finite state transition structure

We now describe a structural technique for FSM traversal. The FSM is traversed by a fixed point iteration that is based on a functional decomposition of the combinational circuitry in each time frame.

3.1 Existential Abstraction - from a Structural Point of View

Definition 1: The set of all states that can occur in an FSM at a specific time t after initialization is called *reachable state set at time t*, $R(t)$.

Note that $R(t)$ only contains the states that are reachable exactly at time t. States that can be reached before t but not exactly at t are *not* contained in $R(t)$. By way of contrast, in other literature, $R(t)$ often refers to all states that can be reached at *some* time t', with $t' \leq t$.

Figure 3 shows a functional decomposition for a given combinational block A_t. The upper part shows the circuit array of the FST at time t. We consider a decomposition of δ_t into functions δ'_t and σ_t. The combinational circuitry A_t' with output vector $\underline{c}_t$ of length k implements function $\delta'_t(\underline{s}_t, \underline{x}_t)$. The circuit labeled $STUB_t$ implements function $\sigma_t(\underline{c}_t)$ and is called *stub circuit*, $\sigma_t(\underline{c}_t)$ is called *stub function*.

Definition 2: Let A_t be a combinational block in an iterative circuit array. Let $R(t)$ denote the set of reachable states at the state variables $\underline{s}_t$ and let $R(t+1)$ denote the set of states reachable at the next state variables $\underline{z}_t$. Then, a function, $\sigma_t(\underline{c}_t)$, is called *stub function* if and only if it has the following properties:

1) the set of all combinations of value assignments at the variables $\underline{c}_t$ is mapped to exactly those combinations of value assignments at $\underline{z}_t$ that correspond to the reachable states $R(t + 1)$, i.e., it is σ_t: $\{0, 1\}^k \rightarrow R(t + 1)$.

2) it holds that $\sigma_t(\delta_t'(\underline{s}_t, \underline{x}_t)) = \delta(\underline{s}_t, \underline{x}_t)$.

A circuit implementing a stub function is called *stub circuit*.

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 time frames does not change. By the above decomposition in combination with cutting off all logic in front of the stub circuit we lose 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 lose 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.

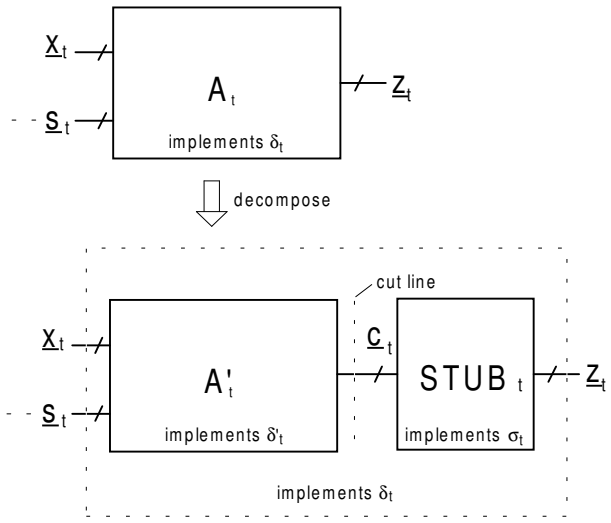

Figure 3: Existential abstraction by decomposition and cut

The question arises how this decomposition can be computed. There are many degrees of freedom for constructing the stub circuit and one possibility is to choose $\delta'_t = \delta_t$. 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 $\underline{z}_t$. Further, the combinations of value assignments that can not occur at $\underline{c}_t$ must be mapped to values at $\underline{z}_t$ that can occur in the circuit array.

Under the assumption that we choose $\delta_t' = \delta_t$ we will now describe a procedure for calculating the stub circuit. This procedure is based on the notion of *implicants* in multi-level circuits determined by AND/OR reasoning graphs as described in [StKu97], [KuSt97]. For reasons of self-containment, these notions are briefly repeated.

A *literal* is a variable in the *combinational network* or its complement. A *product term* is a literal or a conjunction of literals. For example a, $\overline{a}$, b, $\overline{b}$, c, $\overline{c}$.... are literals, abc, $a\overline{b}c$, $\overline{a}\overline{b}\overline{c}$ ... are product terms. A *1-implicant* (*0-implicant*) for a given function y in a combinational network A is a product term t such that y assumes the value 1 (0) for every set of value assignments at the primary inputs of A for which t assumes the value 1. An *implicant* for a node y is a product term which is either a 0-implicant or a 1-implicant for y. An *implicant* is called *prime* if the removal of any literal makes the implicant a product term that is not an implicant.

Note that the literals of an implicant for some node *y* in a *multi-level* combinational network can belong to arbitrary nodes in the network. Unlike in the case of a two-level circuit, they are not necessarily primary inputs and do not even have to be in the transitive fanin of *y*. Prime implicants at a node in the network may be composed out of arbitrary network variables.

In [StKu97] it has been described how prime implicants in multi-level combinational networks can be determined by *AND/OR reasoning graphs*. Given a 1-implicant for some node *y* in the circuit, we say that the implicant is "added to the cover of *y*" if we implement the implicant by an AND gate and connect this AND gate through an OR gate to signal *y*. Similarly, a 0-implicant can be "added to the cover of *y*" by connecting the literals of the implicant into a NAND gate which is connected to *y* through an AND gate. (Note that we defined also 0-implicants as a conjunction of literals although a disjunction may seem more natural. This has been done to have a uniform treatment of both types of implicants when extracting them from AND/OR graphs.)

By means of these concepts the stub circuit can be synthesized. Table 1 shows a procedure for calculating the stub circuit that fulfills the properties as described above.

```
/* routine takes as input an iterative circuit array H_t */
existential_abstraction(H_t)
{
  redundancy removal in H_t;

  L := {z_0,t, z_1,t, z_2,t, ... z_m,t}  /* list of state variables in last time frame */
  for (each state variable z_i,t ∈ L)
  {
      L : = L \ { z_i,t};
      determine all 1-implicants for z_i,t such that all literals
      belong to state variables of L;

      modify H_t by adding 1-implicants to cover at z_i,t;

      determine all 0-implicants for z_i,t such that all literals
      belong to state variables of L;

      modify H_t by adding 0-implicants to cover at z_i,t;
  }

  mark added implicants and fanout systems which feed them;
  cut off all unmarked logic in H_t;
  redundancy removal in H_t;

  return (STUB_t : = H_t);
}
```

Table 1: Routine for exactly calculating the stub circuit

Procedure *existential_abstraction()* is illustrated by means of the example in Figure 4. For both circuits the structural netlist representation and the transition graph is given. The state transition graph for the

upper circuit has been "unfolded" so that both states 0 and 1 appear twice. This makes it easier to see that the two machines are equivalent.

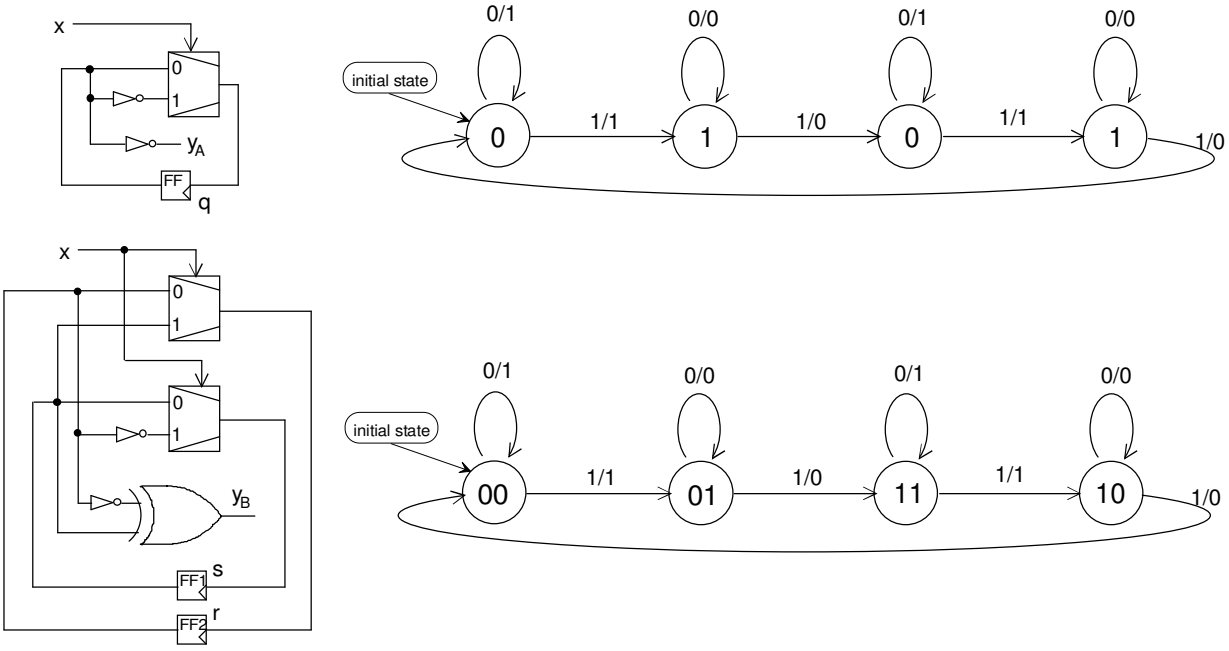

Figure 4: Example for two equivalent FSMs

For our example, the two circuits of Figure 4 are combined to form a product machine which is expanded into an iterative circuit array until time $t = 3$. The resulting circuit array is shown in Figure 5. Note that the circuit array has been simplified by assigning the initial state $\underline{0}$ to the state variables in the first time frame and by removing all constant nodes that result from the forward implications for the initial state. The state variables are called q, r and s. Not all states are reachable at a specific time. Table 2 lists the combinations of value assignments that are possible at the state variables for a specific t. The reader may verify that these sets are reachable by looking at the state transition graphs of Figure 4. Note that the state variables q_3, r_3 and s_3 in the circuit array of Figure 5 can only assume the values as listed in Table 2 for $t = 3$. Hence, the stub circuit $STUB_3$ that will now be constructed must exclude all other combinations of value assignments.

states (q, r, s) reachable at $t = 0$	states (q, r, s) reachable at $t = 1$	states (q, r, s) reachable at $t = 2$	states (q, r, s) reachable at $t = 3$
000	000	000	000
	101	101	101
		011	011
			110

Table 2: sets of reachable states at time t

The circuit array of Figure 5 is passed to the algorithm of Table 1 to calculate the stub circuit $STUB_3$. Routine *existential_abstraction()* is based on calculating implicants for the state variables whose literals belong to other state variables. As an example, in the circuit of Figure 5, assume that we assign a value 0 at state variable r_3 and value 1 at state variable s_3. The reader may verify that these values imply 1 at state

variable q_3. This implication also follows from the consideration that ($q_3 = 0$, $r_3 = 0$, $s_3 = 1$) is an unreachable state whereas ($q_3 = 1$, $r_3 = 0$, $s_3 = 1$) is reachable. Hence, if both assignments $r_3 = 0$ and $s_3 = 1$ occur in the circuit, q_3 must be 1 and, hence, $\overline{r_3} \cdot s_3$ is an implicant for q_3 . Such implicants can be determined by AND/OR reasoning graphs. Building the AND/OR graph for $q_3 = 0$ as described in [KuSt97] yields all 1-implicants for q_3 and building the AND/OR graph for $q_3 = 1$ yields all 0-implicants for q_3. Specifically, in this example, for q_3 we obtain the 1-implicants $\overline{r_3} \cdot s_3$ and $r_3 \cdot \overline{s_3}$ and the 0-implicants $r_3 \cdot s_3$ and $\overline{r_3} \cdot \overline{s_3}$. To form the *stub circuit* these implicants must be added to the cover of q_3. Note that the set of 1-implicants in this example can be represented by an XOR gate and the set of 0-implicants can be represented by an XNOR gate so that we obtain the circuit shown in Figure 6. After adding the implicants we obtain a new signal q' which is equivalent to q_3. We have only added redundancy to the circuit but have not changed its function.

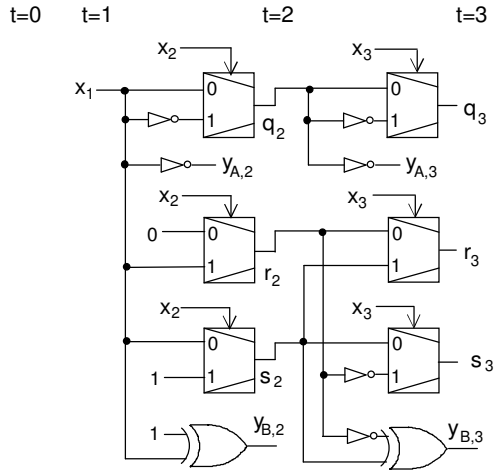

Figure 5: Iterative circuit array of product machine for circuits of Figure 1 until $t = 3$

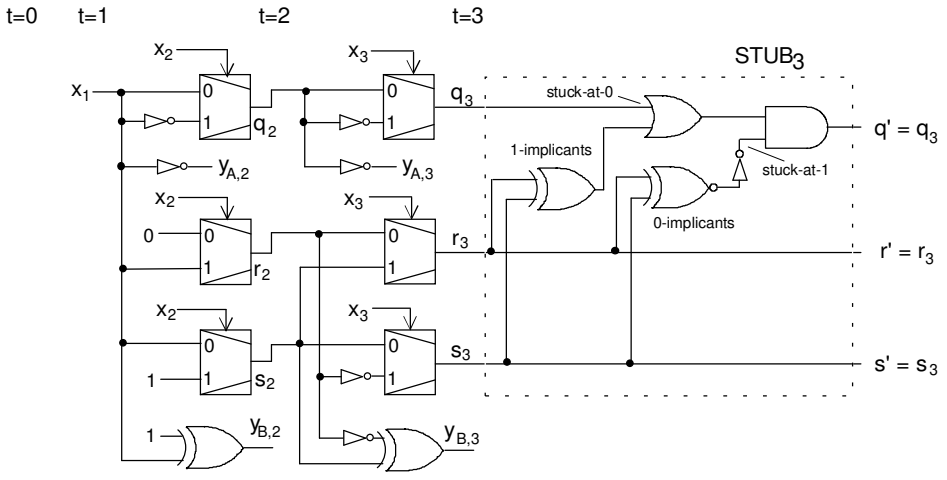

Figure 6: Adding implicants to create stub circuit

Similarly, the implicants are determined for all other state variables. It is sufficient to only consider those implicants whose literals belong to a set L. In each iteration L only contains those state variables for which no implicants have been determined in a prior iteration so that we avoid to add more implicants than are actually needed. In this example, no further implicants have to be considered and Figure 6 shows the stub circuit. The reader may verify that, if we cut off all logic in the transitive fanin of the added implicants, the function of q', r' and s' changes, however, the set of states reachable at these variables remains the same. This is guaranteed by the construction of the implicants that constitute the stub circuit.

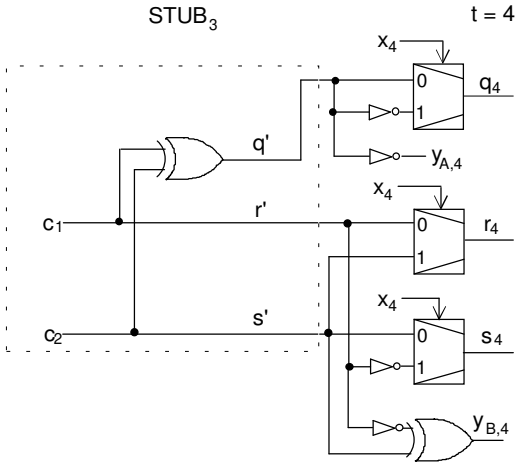

Figure 7: iterative circuit array until $t = 4$ after existential abstraction

After cutting off all logic in front of the sub circuit redundancy removal can be applied to further simplify the stub circuit. Figure 6 indicates the redundancies that can be removed after cutting. The resulting stub circuit $STUB_3$ is shown in Figure 7. We have lost functional information of previous time frames and only *abstracted out* the information *what* states are reachable at time $t = 3$. However this is sufficient for our further considerations, e.g., we can now easily determine that $y_{A,4} = y_{B,4}$.

Theorem 1: For any iterative circuit array H_t, the routine given in Table 1 determines a stub circuit $STUB_t$ according to Definition 2.

Proof:

The set of all unreachable states in a FSM can be represented by a set of prime cubes $P = \{\alpha_1, \alpha_2...\}$ where each literal l_j in a cube α_i corresponds to a state variable. A literal belongs to the complemented state variable if 0 at this state variable cannot be reached, it belongs to the uncomplemented state variable if 1 cannot be reached. For example, for a state vector $\underline{s} = (a, b, c, d)$ *abc* means that ($a = 1$, $b = 1$, $c = 1$, $d = $ X) is an unreachable state with X being the don't care value. The cubes are called prime in the usual sense, i.e., if any literal is removed from the cube, the cube covers at least one reachable state.

Without loss of generality we consider a ternary prime cube *abc* that represents a set of unreachable states in a given FSM as above. For any set of value assignments that can occur in the machine and

which assigns 1 to both state variables *a* and *b*, the machine must produce 0 at state variable *c*. This state is reachable, otherwise *abc* would not be prime. Hence, *ab* is a 0-implicant for *c*. Similarly, *bc* is a 0-implicant for *a* and *ac* is a 0-implicant for *b*. Suppose, we add the implicant *ab* to the cover of variable *c* using the usual circuit transformation rules as in [KuSt97]. This does not change the function of the circuit. Note, however, that at the presence of this implicant any combination of value assignments belonging to an unreachable state covered by the prime cube *abc* is in conflict with this implicant. Hence, if we removed all other logic and only maintained this implicant in the circuit, all unreachable states described by *abc* would still be unreachable. We say "the implicant covers" all unreachable states given by the cube *abc*. The same is true if we added the implicant *bc* to *a* or the implicant *ac* to *b*. The same argument applies for all cubes in *P* and for an arbitrary number of literals in each cube.

Hence, if we determine all 0-implicants for a state variable *s*, and add all these implicants to the cover of *s*, these implicants cover all unreachable states that contain $s = 1$. Similarly, the 1-implicants for *s* cover all unreachable states that contain $s = 0$. In order to also cover those unreachable states represented by prime cubes that do not contain *s* we need to identify the 0-implicants and 1-implicants for the other state variables as well. However, if we obtain, for some state variable *r* an implicant that contains a literal belonging to *s*, this implicant need not to be added, since it belongs to an unreachable state cube that contains *s*, and those have already been covered by the implicants for *s*. Hence, all implicants can be excluded from consideration that contain a literal belonging to a state variable for which the implicants have already been determined earlier. Hence, the set of implicants added to the cover of the state variables as given in *existential_abstraction()* covers all states that are unreachable in the circuit array H_t at time *t*. Further, since the implicants do not change the function of the circuit, the circuitry belonging to the inserted implicants represents a stub circuit according to Definition 2.

q.e.d.

3.2 A Structural Fixed Point Iteration

Based on our method to calculate the stub circuit we can formulate the fixed point iteration shown in Figure 8 that represents the core of the proposed FSM traversal. 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 $R(t)$, $R(t + 1)$, ... $R(t + T)$ of sets of states reachable at time t, $t + 1$, ... $t + T$ that repeats with a period of length *T*. This is detected by determining that the decomposition steps to produce the stub circuit are the same as the steps performed *T* time frames earlier. For a practical implementation an *instruction queue* can be used for this purpose as will be described in Section 4. 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 8 reaches a fixed point for any completely specified, deterministic FST after a finite number of steps.

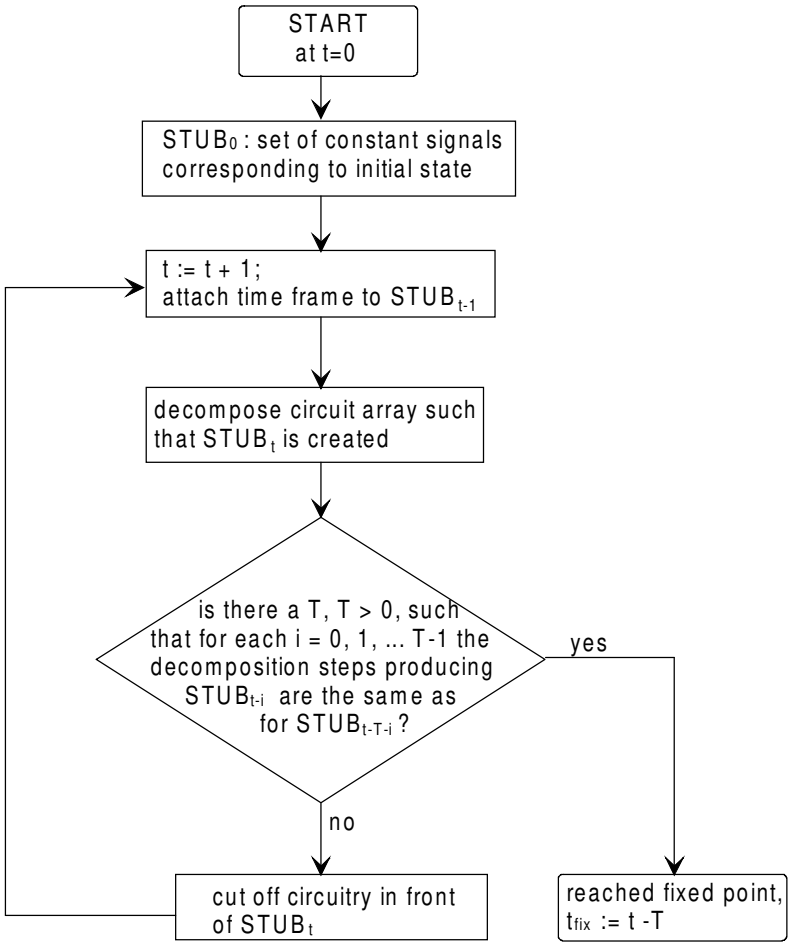

Figure 8: Structural fixed point iteration

For solving practical verification problems this fixed point iteration must be applied to an appropriate finite state machine. In the case of equivalence checking this FSM is obtained by forming the product machine [HaSo96] of the two machines to be checked for equivalence. Similarly, other verification tasks can be mapped to checking functional properties in appropriate FSMs. Using the above fixed point iteration the verification problem is solved by checking the validity of the considered functional property of the combinational circuitry in each iteration. If the property can be proved in each iteration until the fixed point is reached the correctness of the property follows by induction.

3.3 Theoretical Results

The structural fixed point iteration proposed in the previous section raises several theoretical questions. In particular, the structure of the stub circuit may "oscillate" with a period *T*. How long is this period? How many iterations are needed to reach the fixed point? This section is dedicated to answering these questions by relating the behavior of the proposed fixed point iteration to certain properties of the machine's *state transition graph* (STG).

Definition 3: A *cycle* in a state transition graph (STG) of a finite state machine is a directed path whose head node and tail node are identical and which does not contain any edge more than once.

Definition 4: A state which lies on a cycle of the state transition graph is called *cycle state*.

Definition 5: A *strongly connected component* (SCC) of an STG is a sub-graph of the STG such that every state of the SCC can be reached from every other state of the SCC.

Obviously, all states of an SCC are cycle states.

Definition 6: A *maximum SCC M* is an SCC such that all SCCs with nodes in M do not have any nodes outsides M.

Remember that the stub circuit $STUB_t$ represents all reachable states according to Definition 1. Since it does not contain the states that are reachable before t if they are not also reachable exactly at t, our procedure may reach a fixed point that consists of a *period* of different stub circuits. In the following we will examine how this period can occur and how it is determined.

Definition 7: A state that can occur in the reachable state set $R(t)$ only at a single time t and at no other time, is called a *transient* state.

Transient states can occur during the initialization of the machine before the machine has entered its "normal" mode of operation.

Definition 8: A state that can occur in the reachable state set at several different times is called *periodic state*.

Notice that a cycle state is always a periodic state. But not every periodic state is a cycle state. A non-cycle state can occur only once in a single run of the machine. Nevertheless it may be a periodic state if it can occur at different times for *different* runs of the machine. A non-cycle state is always periodic if it can be reached from a cycle state.

Definition 9: If a state S occurs in the reachable state set $R(t)$ repeatedly with a period of p starting at a certain time t', i.e., $S \in R(t' + k \cdot p)$, for all $k \in \{1, 2, ...\}$, we call p the *recurrence period of S* and t' its *birth time*.

Lemma 1: If a state S lies on a cycle of length l, it has a recurrence period $p = l$. The birth time of p is the time where S first occurs in $R(t)$.

Proof: obvious.

Note that a state S can lie on several cycles and, therefore, it can have several recurrence periods. Furthermore, if a state has the recurrence period p then all integer multiples of p are also recurrence periods of S.

Lemma 2: If a state Z can be reached from a periodic state S through a path of length l in the STG, state Z *inherits* all recurrence periods of S, i.e., for each recurrence period p and its birth time t' of a state S we obtain:

$$S \in R(t' + k \cdot p) \Rightarrow Z \in R((t' + l) + k \cdot p) \text{ for all } k \in \{0, 1, 2, ...\}$$

Proof:

Since state Z can be reached from state S through a path of length l, we obtain for every t: $S \in R(t) \Rightarrow Z \in R(t + l)$. Hence, this is also true for $t = t' + k \cdot p$.

q.e.d.

Lemma 3: Consider an arbitrary state S being on a cycle of the state transition graph with length p_1. Furthermore, let S have a recurrence period p_2 with birth time t'. Then, state S also has a recurrence period p_g with birthtime t'' such that

$$t'' \leq t' + t_{trans}, \text{ with } t_{trans} = \text{LCM}(p_1, p_2)$$

$$p_g = \text{GCD}(p_1, p_2).$$

(Notation: GCD: greatest common divisor, LCM: least common multiple)

Proof:

Since S is on a cycle of length p_1 it has the two recurrence periods p_1 and p_2 after t'. Let p denote the smaller one of the two recurrence periods, q the greater one and g their greatest common divisor. Hence, it is:

$$p = m \cdot g,\ q = n \cdot g,\ g = \text{GCD}(p, q),\ q > p.$$

Since m and n have no common divisor, $k = \text{LCM}(p, q) = m \cdot n \cdot g$ is the least common multiple of p and q. If state S is contained in $R(t')$ it will obviously re-occur after integer multiples of p and q, i.e., it is $S \in R(t' + t)$ with $t = i \cdot p + j \cdot q$. In other words, the state appears in S at time t because, for example, after t' it occurred j times after p_1 cycles and i times after p_2 cycles. The time intervals after which S can re-occur are integer multiples of $g = \text{GCD}(p, q)$, since it follows from $t = i \cdot p + j \cdot q$ that $t = i \cdot m \cdot g + j \cdot n \cdot g = (i \cdot m + j \cdot n) \cdot g$.

To prove Lemma 3 it must be shown that for any time $t = t_{trans} + k \cdot g$ there always exists a $i \geq 0$ and $j \geq 0$ such that $t = i \cdot p + j \cdot q$.

For such a pair (i, j) to exist it follows from $t - j \cdot q = i \cdot p$ that the following condition must be fulfilled:

$$(t - j \cdot q) \bmod p = 0, \text{ with } j \cdot q \leq t$$
$$\Leftrightarrow \quad t \bmod p = (j \cdot q) \bmod p.$$

This condition must be fulfilled for every $t = t_{trans} + k \cdot g$. First, we calculate all values that $r_t = t \bmod p$ can assume:

$$r_t = (t_{trans} + k \cdot g) \bmod p = (k \cdot g) \bmod (m \cdot g) = (k \bmod m) \cdot g$$

(Note that $t_{trans} \bmod p = 0$ since t_{trans} is a multiple of p). Obviously, the remainder can assume values $r_t = l \cdot g$ with $0 \le l \le m - 1$.

It needs to be shown that for each of the m possible remainders r_t there exists a j such that $r_t = r_j = (j \cdot q) \bmod p$. We now determine the first m values for r_j:

$$V = ? \{r_j\} \text{ with } r_j = (j \cdot q) \bmod p.$$

Since every r_j is an integer multiple of g we again obtain:

$$r_j = (j \cdot q) \bmod p = (j \cdot n \cdot g) \bmod (m \cdot g) = ((j \cdot n) \bmod m) \cdot g = l \cdot g.$$

Furthermore, it results from the modulo operation that $0 \le l \le (m - 1)$. Hence, if the set V contains exactly m different elements (as will shown below) then these m values can only be given by $l \cdot g$ for $0 \le l \le m - 1$. This means that for every remainder r_t there exists a j such that $(j \cdot q) \bmod p = r_t$ with $j \cdot q \le (m - 1) \cdot q < m \cdot q = m \cdot n \cdot g = \mathrm{LCM}(p, q) = t_{trans}$.

Hence, it only remains to be shown that all r_j in V are different. This can be proved by contradiction. Suppose there is a pair (j_1, j_2) with $0 \le j_1 < j_2 \le m - 1$, such that $r_{j1} = r_{j2}$. Then, we would obtain

$$\begin{aligned} &(j_1 \cdot q) \bmod p = (j_2 \cdot q) \bmod p \\ &\Rightarrow j_1 \cdot q - i_1 \cdot p = j_2 \cdot q - i_2 \cdot p \\ &\Rightarrow (i_2 - i_1) \cdot p = (j_2 - j_1) \cdot q \end{aligned}$$

This means there is a common multiple of p and q which is smaller than $m \cdot q = m \cdot n \cdot g$ because of $j_2 - j_1 < m$. However, this results in a contradiction, since $m \cdot n \cdot g$ is the LCM of p and q. Hence all r_j in V must be different.

q.e.d.

Lemma 3 describes the general rule to derive the recurrence periods with their birth times for the states in a SCC. Note that a state can have the recurrence period p_g also before time $t' + t_{trans}$ if it inherits this period from another state. Therefore, $t' + t'_{trans}$ is an upper bound for the birth time. Furthermore, we obtain the following:

Lemma 4: After a finite transition time t'_{SCC}, the smallest recurrence period p_{SCC} is the same for all states in an SCC. This period p_{SCC} is given by the GDC of all cycle lengths in the SCC and of all recurrence periods for states in the SCC that have been "inherited" from predecessor states outsides the SCC.

Proof:

Since all states in the SCC are reachable from all other states of the SCC, they inherit from each other all their recurrence periods after a finite number of steps. Each state has recurrence periods according to the cycles on which it is located. Furthermore, some states may have inherited recurrence periods from states outsides the SCC which they can pass on to the states insides the SCC. Therefore, after a finite transition time every state has all recurrence periods, including a smallest one. The recurrence periods are obtained by forming the GCD of all pairs of recurrence periods according to Lemma 3. Hence, the smallest recurrence period is the GCD of all recurrence periods of all states of the SCC.

q.e.d.

Lemma 5: The set of states of an SCC with recurrence period p_{SCC} can be partitioned into p_{SCC} disjoint subsets C_m, $m \in \{0, 1, 2... p_{SCC} - 1\}$, such that $R(t'_{SCC} + k \cdot p_{SCC} + m) = C_m$, for $k \in \{0, 1, 2...\}$.

Proof: obvious.

Lemma 5 states that all states of an SCC can be grouped into disjoint subsets C_m such that each subset corresponds to a set of reachable states $R(t)$ that repeats periodically every p_{SCC} cycles starting from a time $t'_{SCC} + m$.

Definition 10: An SCC which is entered by initialization or reached exclusively through transient states after initialization is called *entry-SCC*.

Most practical FSMs can be expected to have a unique entry-SCC. However, for sake of generality we assume for the following theorem that initialization may lead us to different entry-SCCs depending on what state we start from after power-up.

Theorem 2: Let P be the set of all periodic states of an FSM. There always exists a cover $\{C_0, C_1,... C_{T-1}\}$ of P with $C_k \in 2^P$ and there is a time $0 \leq t_{fix} < \infty$, such that

$$R(t_{fix} + m \cdot T + k) = C_k,\ m \in \{0, 1, 2, ...\},\ 0 \geq k \geq T - 1$$

and it is $T = \text{LCM}\ (p_1, p_2,..., p_n)$, where p_i are the periods of the entry-SCCs of the FSM. The period T is called *fixed point oscillation period*.

Proof:

Let p_{SCC} be the period of an entry-SCC. All SCCs succeeding this entry-SCC have periods that are divisors of p_{SCC}. Hence, p_{SCC} is the smallest recurrence period that is common to all states in the entry-SCC and its successor-SCCs. This means that the states in the entry-SCC and its successor-SCCs can be divided into p_{SCC} subsets K_i such that all states in a K_i occur at the same time in the reachable state set, and the subsets K_i are called "blocks" and represent a cover for all states in the entry-SCC and its successor-SCC.

Without loss of generality we consider an STG with two entry-SCCs with periods p_1 and p_2. Let S_1 be the set of all states in the first entry-SCC and its successor-SCCs and let S_2 be the set of all states

in the second entry-SCC and its successor-SCCs. Then, $P = S_1 \cup S_2$ is the set of all periodic states of the STG. We now consider a block K_1 of states of S_1 with recurrence period p_1 and a block K_2 of states of S_2 with recurrence period p_2. There always exists a finite t' such that $K_1 \subset R(t' + k \cdot p_1)$ and $K_2 \subset R(t' + l \cdot p_2)$. This means at time t' the states of K_1 and K_2 occur simultaneously in the reachable state set, after t' the states of K_1 occur every p_1 clock cycles and the states of K_2 occur every p_2 clock cycles. The times where both sets are contained in the reachable state set occur every T clock cycles where T is the least common multiple of p_1 and p_2.

Hence, there are T different combinations C_k in which the blocks of S_1 can occur together with the blocks of S_2 in the reachable state set. The states in S_1 occur in $(T/p_1) \geq 1$ combinations, the states of S_2 occur in $(T/p_2) \geq 1$ combinations. Hence, the C_k cover all periodic states P of the STG.

The generalization of this argument from two entry-SCCs to n entry-SCCs with recurrence periods $p_1, p_2, ..., p_n$ is straightforward.

q.e.d.

For illustration of Theorem 2 and its proof consider the STG in Figure 9. It has 8 states {A, B, C, D, E, F, G, H}. Since A is not located on a cycle and cannot be reached from any state of the SCC it is a transient state. The set of periodic states is P = {B, C, D, E, F, G, H}. There are two entry SCCs, S_1 = {B, C, D} and S_2 = {E, F, G, H} with periods $p_1 = 3$ and $p_2 = 2$. The fixed point oscillation period is T = LCM (p_1, p_2) = 6. Table 3 shows the reachable state sets for the first 10 cycles. The birth time for the recurrence period $p_1 = 3$ is $t_1' = 3$. Starting from this time all states in the SCC S_1 have this period. For the second SCC S_2 it is $p_2 = 2$ and $t_2' = 2$. The fixed point in this example is reached at $t_{fix} = 2$. After this time all state combinations repeat periodically.

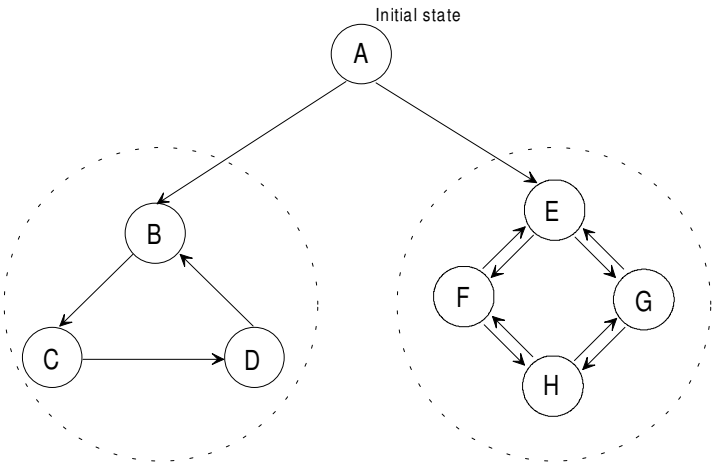

Figure 9: Example of an STG

t	$S(t)$	C_j
0	$\{A\}$	-
1	$\{B, E\}$	-
2	$\{C, F, G\}$	C_0
3	$\{D, E, H\}$	C_1
4	$\{B, F, G\}$	C_2
5	$\{C, E, H\}$	C_3
6	$\{D, F, G\}$	C_4
7	$\{B, E, H\}$	C_5
8	$\{C, F, G\}$	C_0
9	$\{D, E, H\}$	C_1
…	…	…

Table 3: reachable state sets for STG in Figure 9

Theorem 2 in combination with the previous lemmas is the theoretical basis for the structural fixed point iteration proposed in this paper. First of all, this theorem states that the proposed fixed point iteration will always reach a fixed point. This is not surprising since the stub circuits as proved in Theorem 1 exactly represent the sets of reachable states $R(t)$ and it follows immediately from the finiteness of the state space that these sets must repeat periodically after a finite number of cycles. Note that the time to reach the fixed point may be longer than for conventional FSM traversal. This is because the reachable state set only contains the states reachable exactly at time t. We could not determine a simple and accurate upper bound for the number of iterations until the fixed point, however, Lemma 3 presents an upper bound for the transition time needed when two recurrence periods are "merged" to form a new recurrence period.

Secondly, we have studied the periodicity of reachable state sets that may lead to complications when detecting the fixed point. Note in the procedure of Figure 8 that the fixed point is detected by examining a series of stub circuits of length T. In practice, a long period T can deteriorate the computational efficiency of our approach significantly. However, our theoretical results show that long periods can be expected very rarely. A complication in our theoretical analysis comes from the fact that we assume FSMs with several SCCs and in particular with several entry-SCCs. If we assume that the machine operates in a single SCC after initialization the period T does not exceed the greatest common divisor of all cycle lengths of this SCC. This is a very encouraging result because it means that for most practical cases we can expect a period of $T = 1$. For example, if a state transition graph contains two cycles whose length correspond to a prime number, we obtain $T = 1$. Note that most state transition graphs contain cycles of length 1 like the one shown in Figure 4. This also implies a period of $T = 1$. Consequently, for most practical cases we can further assume that $STUB_t$ does not only represent all states reachable exactly at time t, but all states that are reachable in the machine at any time until t. Below we will consider a special situation where this is guaranteed to always be the case.

Definition 11: [Koha78] A *synchronizing sequence* of a machine M is an input sequence that brings M to a known state S regardless of the initial state or the output sequence. The state S is called *synchronization state* of M.

Corollary 1: If a circuit has a synchronizing sequence of length l the fixed point oscillation period T of the machine is upper-bounded l.

Proof:

If a machine M has a synchronizing sequence the synchronization state S lies within one terminal strongly connected component (TSCC) [PoRe96]. Consider the initial state S. By definition of the synchronizing sequence S must be reachable from S in l clock cycles. Thus, S lies on a cycle of length l or less and hence the TSCC contains cycles of length l or less. The smallest recurrence period p_{TSCC} according to Lemma 4 is calculated as the GCD of all cycle lengths and represents the fixed point oscillation period of M since M can only have one entry-SCC. Therefore, it is $T \leq l$.

q.e.d.

Corollary 1 considers synchronizing sequences which bring the machine into a single synchronization state S. In [PoRe96] so called *weak-synchronizing* sequences have been considered which bring the machine into a set of equivalent states. Pomeranz and Reddy showed that in this case, the machine always has a weak-synchronizing sequence that brings the machine into strongly connected components which are all sinks in the cycle-free graph of all SCCs. If the length of this weak-synchronizing sequence is l, analogously to Corollary 1, it follows that the fixed point oscillation period of M is upper-bounded by $l + 1$.

Corollary 2: If a sequential circuit is implemented such that there exists a synchronous reset line for all state variables, the fixed point oscillation period T of the machine is exactly 1. The time needed to reach the fixed point is given by the *sequential depth* of the machine, i.e., it is given by the minimum number of cycles needed to reach any state in the machine starting from the initial state.

Proof:

The length of the synchronizing sequence for such a circuit is 1. There exists a unique synchronization state S in the STG. All states reachable from this synchronization state lie in one terminal strongly-connected component (TSCC). The state $\underline{s}_0$ is reachable by itself by activating the reset line, therefore, the initial state lies on a cycle of length 1. Hence, it is $T = 1$.

Every state in the STG inherits the recurrence period of 1 when this state appears in the reachable state set for the first time. The time needed to visit every node in the STG is equal to the sequential depth of the machine.

q.e.d.

In this special case, we always have the same situation as in the conventional symbolic FSM traversal. The reachable state set at any time t contains all states visited until t. The fixed point is reached when every state in the STG has been visited.

In a practical implementation of structural FSM traversal Corollary 2 could be used to get rid of the problem to detect an oscillation of the fixed point. If a design to be verified contains non-reset latches these can simply be replaced by reset latches. The reset inputs of the latches are connected to an additional primary input designated to set the machine to the initial state whenever activated. Using this construction no periodic behavior can develop and the fixed point is reached in minimum time. However, in our current implementation we do not make use of this extension. Even without this extension as discussed above, our theory shows that oscillation periods greater than 1 are quite unlikely and we have never observed any oscillation of the fixed point in the experiments conducted so far.

4 Implementation for Equivalence Checking - the Record & Play Procedure

This section presents how structural FSM traversal can be applied to equivalence checking. Although conventional state encoding techniques are used only rarely in today's design compilers, the state encoding of a given circuit is often modified through *structural* modifications of the design. For example, designers often improve the performance of their design by changing the pipelining at the RTL level. Similarly, at the gate level, *retiming*, i.e., the relocation of registers, becomes increasingly important in the industrial design process. However, there is great concern among designers because to date no techniques exist that can verify circuits of realistic size after such transformations. Therefore, in this work we particularly focus on the problem of verifying circuits after synthesis with retiming and show that the structural FSM traversal presented in the previous section offers a promising solution to this problem.

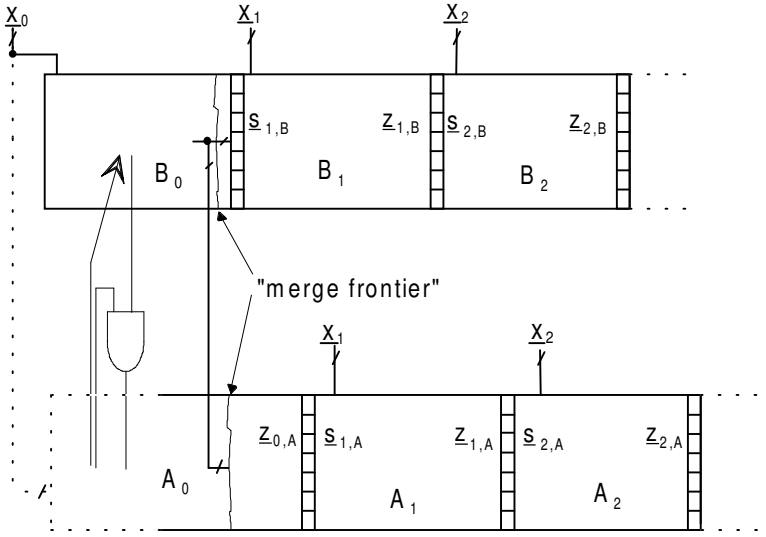

Figure 10: Time frame merging by sharing of logic

In our practical implementation we approximate the decomposition introduced in Section 3. Instead of calculating the stub circuit exactly by determining all implicants for state variables in terms of other state variables we perform implication-based transformations at all nodes in the circuit. By "merging" the logic

between different parts of the product machine we compress important information about the reachability of states into a relatively small area near the next state variables of the last time frame.

Consider Figure 10. The verification procedure is based on performing logic transformations as in [KuSt97]. 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 *A* and *B* and the separate circuitry for each machine. This is schematically shown in Figure 10. In our practical method, the circuitry around the merge frontier represents an approximation for the exact stub circuit defined in Section 3.1.

As described in Section 3.2 we detect a fixed point by noting that the decomposition steps to create a stub circuit repeat with a certain period. Motivated by the theoretical results of Section 3.3, for the time being, we may assume that this period is 1. In our practical procedure, the main goal is to identify in a given time frame a set of circuit transformations between the nodes of machine *A* and the nodes of machine *B* that is valid also in future 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 re-using 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 later time frames and which completes the task of proving the equivalence for the circuit outputs. Now it must be shown that this instruction queue is also valid in *all* future time frames. For this induction the *cutting* procedure is very important. In our practical implementation the merge frontier heuristically indicates the stub circuit that has been created by the circuit transformations. We may assume that the signals of the merge frontier correspond to the variables $\underline{c}_t$ of the exact solution in Figure 3. Since we have only created an approximate solution for the stub circuit some information about the reachability of states may be lost if we cut through the circuit array at these signals. As will be further discussed later the resulting *false negatives* can be avoided by keeping additional levels of logic, called *stub levels*, in front of the merge frontier. If an appropriate cut is found and the instruction queue can be played a sufficient number of times, the combinational structure in the circuit array repeats periodically. We say that our procedure has reached a "structural fixed point". The procedure will now be described in more detail.

Figure 11 shows the proposed algorithm for FSM equivalence checking that is a practical refinement of the general procedure described in Figure 8 of Section 3. Routine *record&play()* is given in Table 4. The different steps of the algorithm are illustrated by means of an example depicted in Figure 12 to Figure 18.

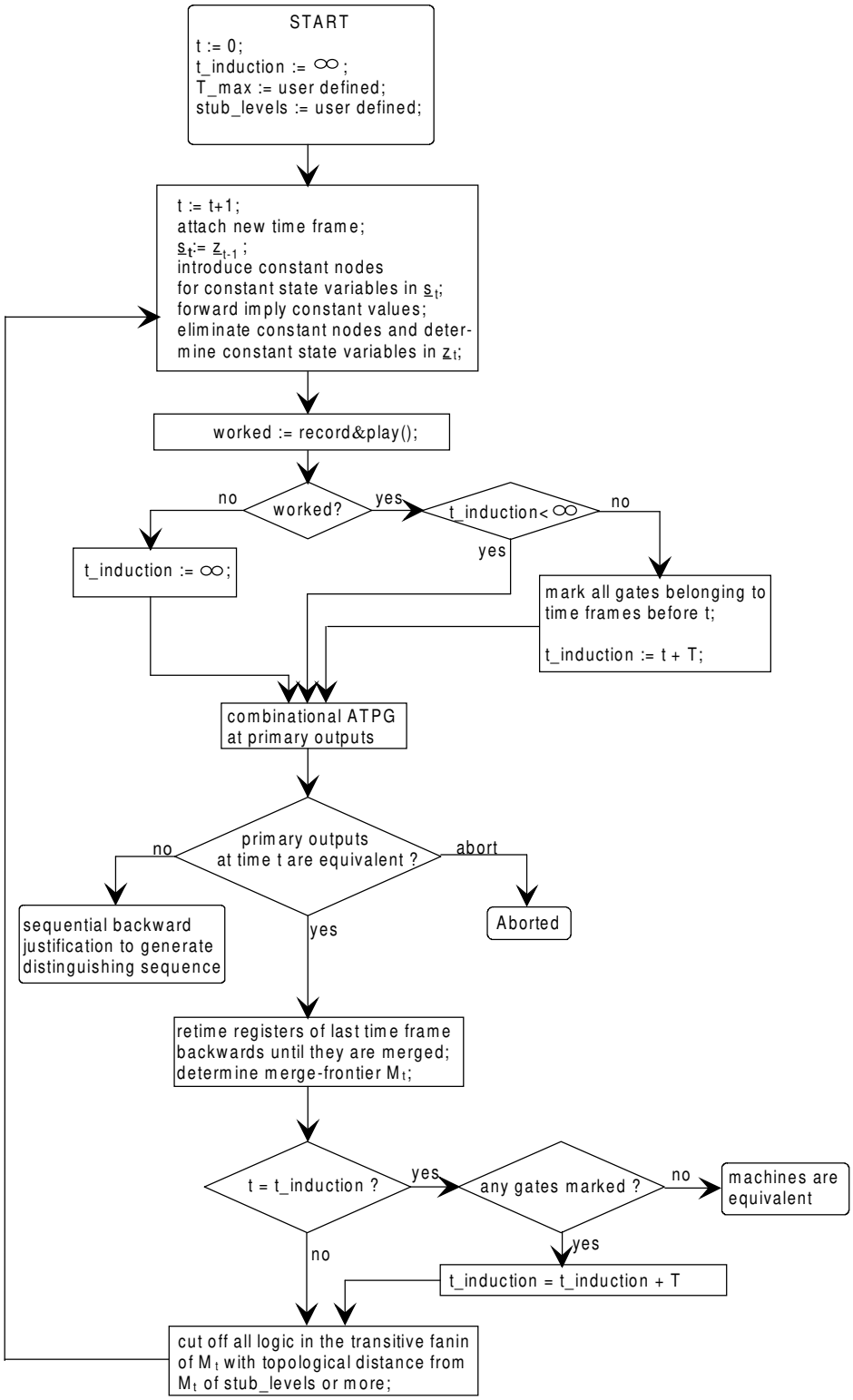

Figure 11: sequential equivalence checking algorithm

```
/* routine operates on a global data structure for the current miter array with present
state variables s and next state variables z. It has t, t_induction, T and T_max of
Figure 11 as global variables */

record&play()
{
    if (t_induction < ∞)  /* trying induction */
        PLAY := {Q_{t-T}};
    else  /* check old queues to find fixed point */
        PLAY := { Q_i | i ∈ {t-1, t-2, ..., t-T_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)
        {   T := 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 4: Routine *record&play()*

The variables *stub_levels* and *T_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 12. 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 13. 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 *T_max* is used to restrict the search to the last *T_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 13 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.

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 ATPG tools is invoked to calculate a distinguishing sequence.

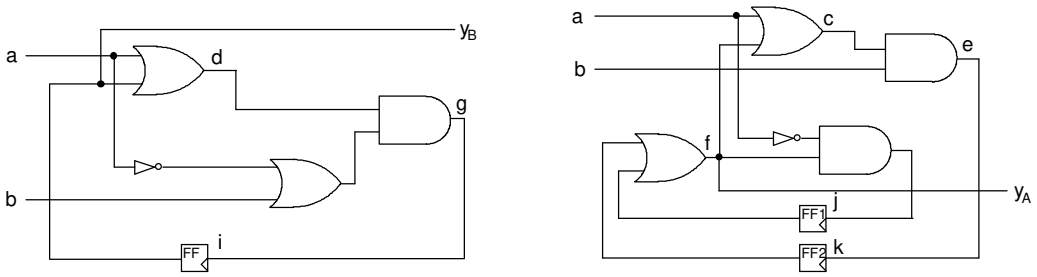

Figure 12: Circuit examples with initial states $S_{0,A} = S_{0,B} = \underline{0}$

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* [LeSa91] 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 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 11 and is a user-defined parameter. Typical values are between 0 and 5.

Consider again the right portion of Figure 13. 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 14. The newly introduced variable at the cut line is called *s*. Note that the fanout system of *s* and the constant signal j_1 constitute our approximation of the stub circuit $STUB_0$ in this example.

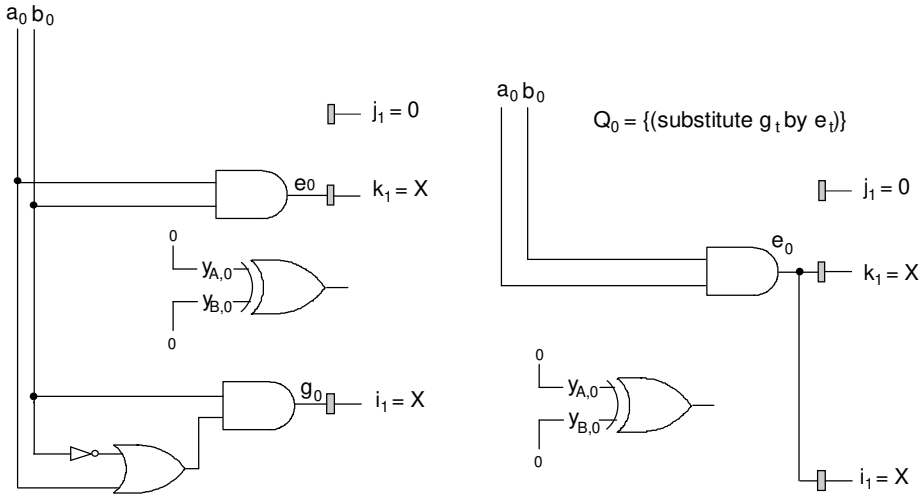

Figure 13: Circuit array in first iteration

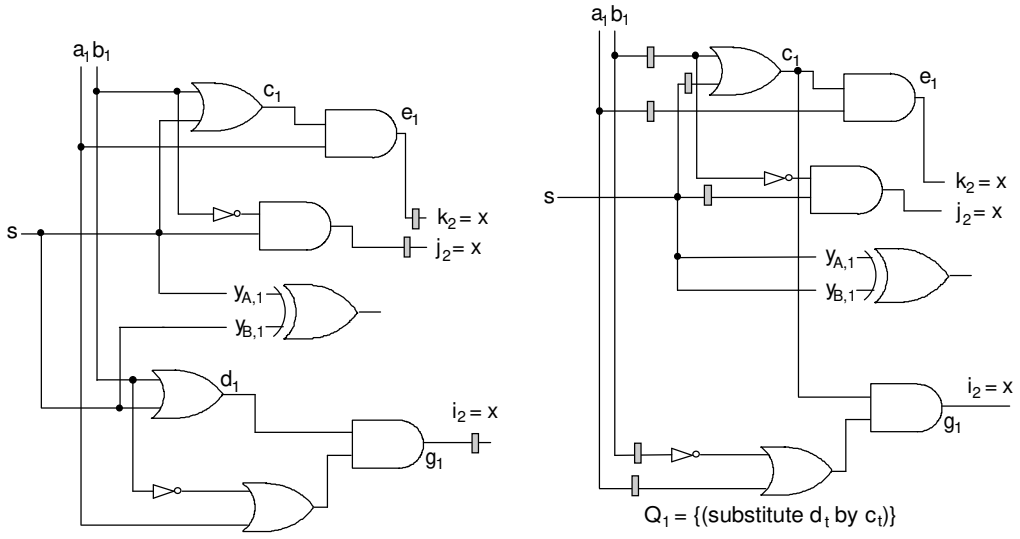

Figure 14: Circuit array in second iteration before (left) and after (right) merging

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 14. 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 14. 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.

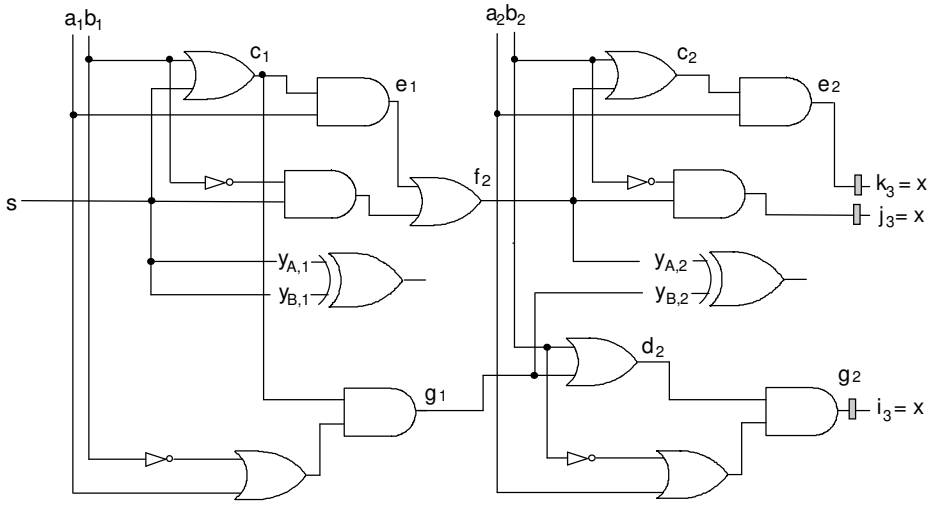

Figure 15: Circuit array in third iteration before merging

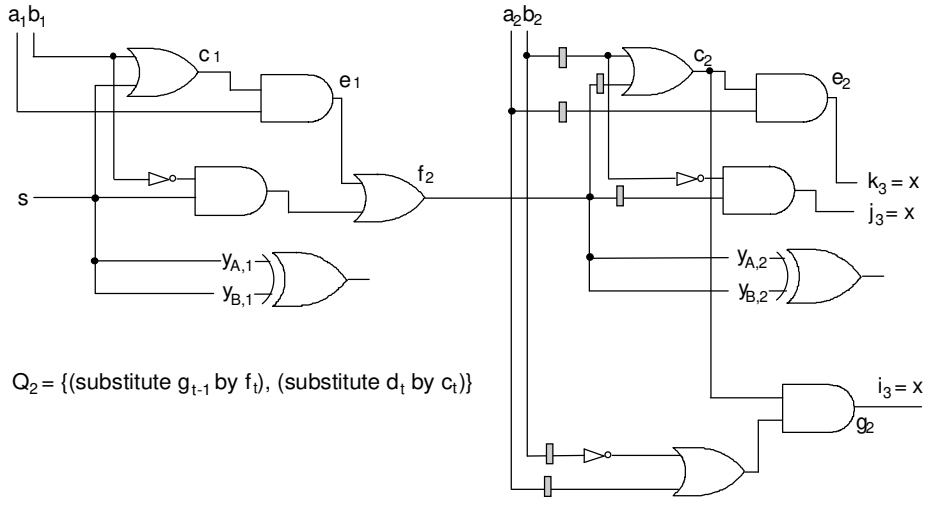

Figure 16: Circuit array in third iteration after merging

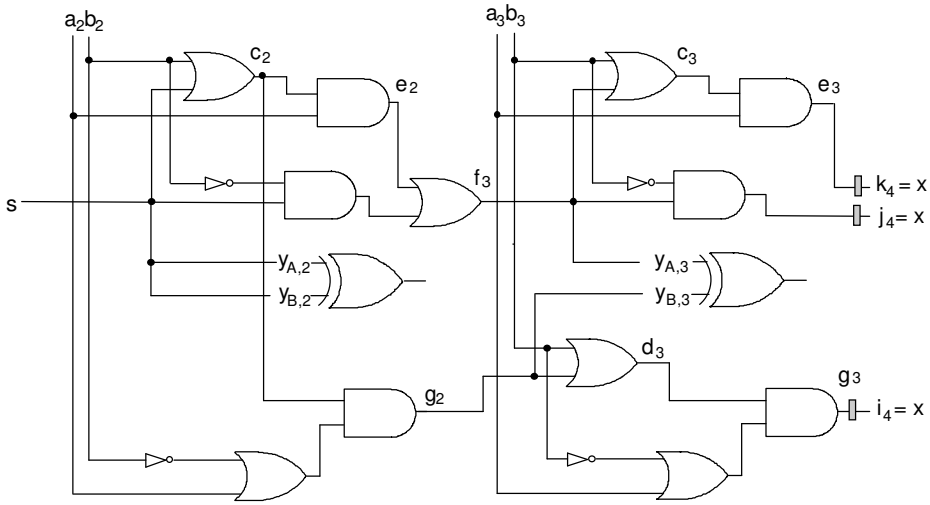

Figure 17: Circuit array in fourth iteration before merging

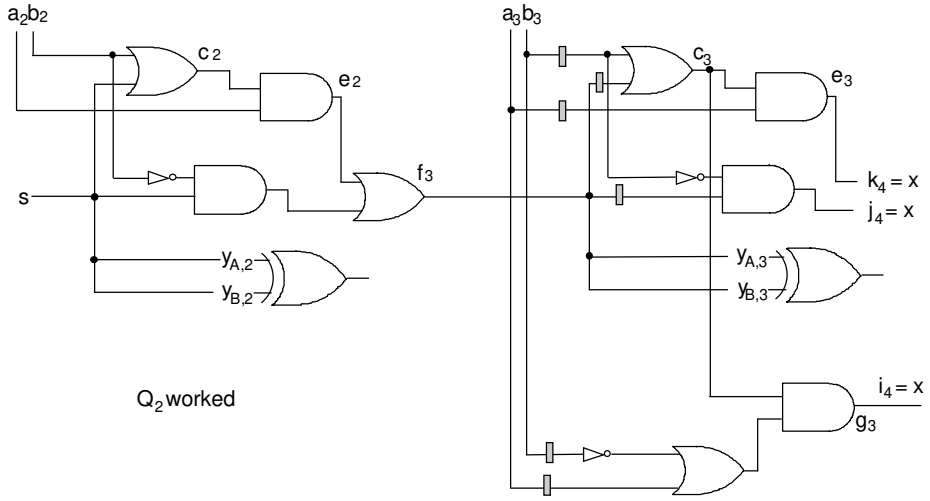

Figure 18: Circuit array in fourth iteration after merging

A new time frame is attached as shown in Figure 15 and a new instruction queue must be recorded. The transformations lead to the circuit array shown in Figure 16. We determine a new merge frontier suggesting a cut at signal f_2. The next iteration leads to the circuit array of Figure 17. 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 18 results.

If an instruction queue has been played successfully the algorithm enters the induction mode. This is done by setting variable *t_induction* to $t + T$ where T 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 $T = 1$. 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 $t - T$ 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, i.e., we need to create a stub circuit for the first time frame. 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 [PiSi94].

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 $R(t)$, i.e., we may consider more states than are actually reachable. As mentioned 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. Furthermore, 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 [HuCh96], [HuGl96] can be useful as a pre-processing 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. For example, in the special case where all state variables are known to be equivalent the problem is combinational and our technique would produce a "working" instruction queue after only one iteration.

5 Experimental Results

A prototype of the described approach has been incorporated into the HANNIBAL [KuSt97] package. We evaluated the techniques by verifying circuits of the ISCAS89 benchmark set against the optimized and retimed circuits. The circuits were optimized by kernel extraction (using *fx* in SIS). After optimization retiming is performed. The resulting circuits were verified against the original ones. For the original circuits we assumed an initial state of 0 for all registers.

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 developed 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. 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.

Circuit Name	# registers	***record&play*** (HANNIBAL)		***verify_fsm*** (SIS)
		# 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 5: Verification of optimized and retimed circuits

Table 5 shows our experimental results for a SUN Ultra I workstation. We compare our techniques with the conventional verification approach by symbolic FSM traversal. The results for *verify_fsm* in SIS are shown in the left column of Table 5. 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 $T_max = 1$.

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

[Bran93] Brand D.: "Verification of Large Synthesized Designs", *Proc. Int. Conf. on Computer-Aided Circuit Design* (ICCAD), Santa Clara, pp. 534-537, Nov. 1993.

[Brya86] Bryant R.: "Graph-based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, vol. 35, pp. 677-691, August 1986.

[CaCa96] 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.

[Cho94] 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.

[CoBe89] 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.

[EiJe96] 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.

[HaSo96] Hachtel G., and Somenzi F.: *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, Boston 1996.

[HuCh96] 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.

[HuGl96] 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.

[Koha78] Kohavi Z.: "Switching and Finite Automata Theory", McGraw-Hill, 1978.

[KuSt97] Kunz W., and Stoffel D.: "Reasoning in Boolean Networks - Logic Synthesis and Verification Using Testing Techniques" *Kluwer Academic Publishers*, Boston 1997.

[KuMe97] 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.

[LeSa91] Leiserson C.E., and Saxe J.B.: "Retiming Synchronous Circuitry", *Algorithmica*, vol. 6., pp. 5-35, 1991.

[McMi93] McMillan K.L.: "Symbolic Model Checking", *Kluwer Academic Publishers*, 1993.

[PiSi94] 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.

[PoRe96] Pomeranz I., and Reddy S.M.: "On Removing Redundancies from Synchronous Sequential Circuits with Synchronizing Sequences", *IEEE Transactions on Computers*, vol. 45, no. 1, January 96, pp.20 - 32.

[Quer96] Quer S. et al.: "Incremental Re-encoding for Symbolic Traversal of Product Machines", *Proc. European Design Automation Conf.*, 1996.

[RaSo95] Ravi K., and Somenzi F.: "High Density Reachability Analysis", *Proc. Int. Conf. on Computer-Aided Design* (ICCAD), pp. 154-158, 1995.

[StKu97] 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.