Structural FSM Traversal

Dominik Stoffel, *Member, IEEE*, Markus Wedler, Peter Warkentin, and Wolfgang Kunz, *Member, IEEE*

***Abstract*—This paper discusses a "structural" technique for traversing the state space of a finite state machine (FSM) and its application to equivalence checking of sequential circuits. The key ingredient to a state-space traversal is a data structure to represent state sets. In structural FSM, traversal-state sets are represented noncanonically and implicitly as gate netlists. First, we present an exact algorithm, which is based on an iterative expansion of the FSM into time frames and a network-decomposition procedure serving the same purpose as an existential quantification operation. Then, we discuss approximative algorithms for the application of structural FSM traversal to sequential equivalence checking. We theoretically analyze the properties of the exact as well as the approximative algorithms. Finally, we give details on the implementation of a sequential equivalence checker and present experimental results that demonstrate the effectiveness of the proposed approach for equivalence checking of optimized and retimed circuits.**

***Index Terms*—Equivalence checking, finite state machine (FSM), formal hardware verification, reachability analysis, retiming, sequential circuits, structural techniques.**

I. Introduction

ONE OF THE central problems in synthesis and verification of sequential circuits is reachability analysis. For example, the properties to be checked by an automatic verification tool are required to hold in those states that the system can assume after starting in a designated start state. *Reachability analysis* is the task of finding this set. Sequential circuits are usually modeled as finite state machines (FSMs). Reachability analysis relies on a traversal of the state transition graph (STG) of an FSM, often called FSM traversal.

Standard FSM traversal algorithms [1], [2] are based on implicit representations of state sets using binary decision diagrams (BDDs) [3]. Large sets of states can be represented by constructing the BDD of the characteristic function of a state set. BDD-based reachability analysis is also called symbolic FSM traversal. An important property of BDDs is that they are a canonical representation of a Boolean function. This property makes them especially attractive for use in formal verification. For example, in FSM traversal this property is used to check that two sets of states are identical, by proving the isomorphism of their BDDs. Despite its usefulness, the canonicity property often leads to exponential growth of data structures for the state sets, even though the BDD representation is an implicit one.

Manuscript received April 15, 2003; revised August 22, 2003. This paper was recommended by Associate Editor J. H. Kukula.

D. Stoffel, M. Wedler, and W. Kunz are with the Department of Electrical Engineering and Information Technology, University of Kaiserslautern, 67653 Kaiserslautern, Germany (e-mail: stoffel@ieee.org).

P. Warkentin is with the Siemens AG, Munich, Germany.

Digital Object Identifier 10.1109/TCAD.2004.826552

For this reason, research efforts have been made to develop formal verification methods that can operate without the use of canonical representations of Boolean functions. In the domain of combinational equivalence checking, there has been great success with methods operating directly on the structural gate netlist of the circuit [4]–[10] or related data structures like *signed AND graphs* (SAGs) [9]. Since they are capable of making efficient use of structural design properties, they are often referred to as structural techniques. They have made combinational equivalence checking feasible for circuits with millions of gates.

Unlike in combinational verification, it is not straightforward to exploit structural similarity in the verification of sequential circuits. This explains why only few structural approaches exist ([11]–[14]). Note that the titles of papers [13] and [14] suggest that the proposed methods work *without* a state-space traversal. This is true in the sense that they are not based on a standard (symbolic) FSM traversal. However, just like [11], they actually perform an approximative *structural* FSM traversal. The sets of states visited in the traversal are an over-approximation of the states reachable from the initial state of the FSM. The representation of the state sets is derived by exploiting the internal structure of the sequential circuit. It is usually a combination of a gate netlist together with a set of logical relations between the signals in the netlist. This set of relations can be represented in various ways, e.g., an instruction queue for node substitutions [11], a partition of the set of circuit functions into equivalence classes [13] or sets of implications between circuit signals [14].

In this paper, we study the general principles of structural FSM traversal and its application to equivalence checking of sequential circuits. The basis of this work is the practical algorithm for sequential equivalence checking that has been introduced in [11]. In this paper, we analyze in more detail how this algorithm can be viewed as an *approximative* structural FSM traversal and how it relates to an exact algorithm. In Section II, we introduce the general concept of structural FSM traversal, how state sets are represented, and how an algorithm based on a "structural fixed point" iteration can be formulated. We analyze commonalities and differences to symbolic FSM traversal algorithms. In Section III, we review the technique presented in [11], relating it to the exact algorithm of Section II. Section IV discusses and compares with closely related algorithms [13] and [14]. The theoretical properties of exact and approximative structural FSM traversal are analyzed in Section V. Section VI discusses implementation details of a practical implementation of the sequential equivalence checker, emphasizing on improvements made in the structural representation of the state sets, and presenting experimental results for circuits that have been heavily optimized and retimed.

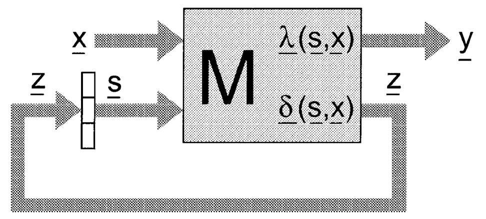


Fig. 1. Sequential circuit implementing an FSM.

TABLE I
(FORWARD) FSM TRAVERSAL

```
FSM_traversal($\delta$, $S_0$)
{
  $t$ := 0;
  $R(t)$ := $S_0$;   /* initial state set */
  repeat
    $t$ := $t$ + 1;
    $R(t)$ := new_states($\delta$, $R(t-1)$);
  until finished({$R(0)$, $R(1)$, ..., $R(t)$}, $t$);
}
```

II. STRUCTURAL FSM TRAVERSAL

FSMs are common models of sequential digital circuits. An FSM M is a 6-tuple $M = (I, S, \delta, S_0, O, \lambda)$, 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 a set of initial states, 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 $S_0 = \{s_0\}$. However, a set S_0 containing several initial states can be treated in a similar way. A sequential circuit (Fig. 1) implementing such an FSM has a set of *(primary) inputs* $\underline{x}$, a set of *(present) state variables* $\underline{s}$, which are fed by registers, a set of *next-state variables* $\underline{z}$ providing the input of the registers, and a set of *(primary) output variables* $\underline{y}$.

Many applications in formal verification or logic synthesis use FSM traversal to obtain information about the reachable state set of the machine. Table I outlines a typical algorithm for forward traversal of the state space of an FSM, starting with the initial states. In each iteration of the repeat-loop, a new set of states $R(t)$ is determined in function new_states() which includes the immediate successors of the states, $R(t-1)$, of the previous iteration. The loop is iterated until we can be sure that all states in the FSM have been visited. This is determined by function finished. One possibility of implementing this algorithm is to select function new_states() in the following way:

$$\text{new\_states}(\delta, S) := S \cup \text{img}(\delta, S).$$

In this case, function new_states() computes the union of the set of states reached so far $R(t-1)$ with the image of $R(t-1)$ under the transition function. As a consequence, once a state s has been visited $s \in R(t)$, it will remain in the reachable state set in all future iterations. It is, therefore, easy to determine when all reachable state sets have been visited, since we need only detect that a fixed point of the image computation under the transition function is reached

$$\text{finished}() := \big(R(t) = R(t-1)\big).$$

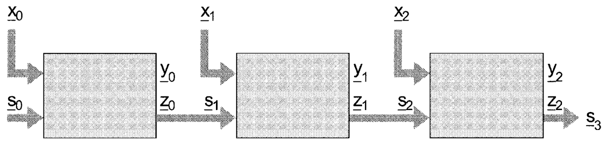


Fig. 2. Time-frame expansion of an FSM.

A crucial ingredient in an FSM traversal algorithm is the efficient representation of sets of states. In symbolic FSM traversal, BDDs are used for representing characteristic functions of state sets, allowing for a compact representation of very large state sets. However, in many cases, BDD-based FSM traversal still fails because of the state-explosion problem.

A. FSM Traversal by Time-Frame Expansion

Looking at the success of structural techniques in combinational verification and synthesis, the question comes to mind whether and how structural techniques can be useful for sequential circuits. Specifically, we are interested in some way for representing sets of states using (and exploiting) the structure of the circuits being modeled. In fact, this is possible and has been used for many years in the field of sequential ATPG and, more recently, also in bounded model checking [15]. A simple structural representation of a set of states can be obtained by an expansion of the FSM into *time frames* by replicating the combinational logic of a circuit for each clock cycle being considered. Each time frame is a copy of the combinational logic implementing the transition function $\delta(\underline{s}, \underline{x})$ and output function $\lambda(\underline{s}, \underline{x})$ of the FSM. The circuit structure obtained by this time-frame expansion is a purely combinational structure called *iterative circuit array*. There are no storage elements. Instead, the values of the state variables of the product machine at different points in time are associated with signal values in different time frames of the iterative circuit array.

As an example, Fig. 2 shows the expansion of an FSM into three time frames. The initial state s_0 is injected at the present state variables $\underline{s}_0$ of the first time frame. This circuit array is a combinational network, which calculates for a given input sequence $(\underline{x}_0, \underline{x}_1, \underline{x}_2)$ the output sequence and next-state response, $\underline{s}_3$, of the FSM. By applying all input sequences of length 3 to this circuit, we obtain, at the state variables $\underline{s}_3$, all possible states the machine can assume at $t = 3$. Obviously, an expansion of an FSM into t time frames is an implicit representation of the set of states, $R(t)$, the FSM can assume after exactly t clock ticks.

Definition 1 (Reachable State Set at Time t): The set of all states, $R(t)$ being possible in an FSM at a specific time t after initialization is called *reachable state set at time t*.

Note an important subtlety in this definition. In conventional terminology, the reachable state set of a machine at a time t refers to the set of all states that are reachable at any time between 0 and t. In the context of structural FSM traversal, however, we consider sets of states that are reachable *exactly* at time t.

Because the state set of an FSM is finite, obviously, there is a time t_s for every state s of the FSM reachable from an initial state such that $s \in R(t_s)$. The time series $R(t)$ traverses the STG of the FSM and "visits" every state. We see that it is rather natural to use a time frame expansion for the FSM traversal algorithm of Table I. However, the functions new_states() and functions() are slightly different from the standard (BDD-based) algorithm described above. In each iteration, the new set of states is obtained by appending a new time frame to the iterative circuit array. Since the next-state logic of the time frame computes for each state vector and for each input vector the successor state, the iterative circuit array implicitly represents the set of states reachable at a specific time t. In other words, function new_states() computes the image of the set $R(t-1)$ under the transition function δ of the FSM

$$\text{new\_states}(\delta, S) := \operatorname{img}(\delta, S).$$

Note that this is different from the standard FSM traversal discussed above, where new_states() computes a union of the image with the set of states reached so far.

Simply using time-frame expansion for traversing the state space of an FSM is not practical. First, the iterative circuit array grows linear with the number of iterations, leading to a constantly growing state-set representation. For large designs, this quickly becomes infeasible. Second, it is not clear when all states of the FSM have been visited so that the iteration can be terminated. There exist approaches to determine the sequential depth or the diameter of an FSM [15]–[17]. These are complex problems by themselves and approximative solutions have to be attempted. However, they often result in bounds that are not practical for large designs.

B. "Existential Quantification" in Structural FSM Traversal

The question of when the iteration in the traversal of Table I can be finished is closely related to the convergence behavior of the set of reachable states $R(t)$. In order to determine a point in time t_{fix} for which it is guaranteed that all states reachable from the initial state have been visited we need to know how the series $R(t)$ develops over time. This is analyzed in detail in Section V. For every FSM, there is a time t_{fix} and a period T such that

$$R(t) = R(t-T) \qquad \text{for } t \geq t_{\text{fix}}$$

and all states of the FSM have been visited. In other words, in the general case the time series $R(t)$ enters an oscillation behavior starting at some time t_{fix}. The oscillation period T depends on the structure of the STG of the FSM. Interestingly, as will be shown in Section V, for most circuits encountered in practice, the oscillation period T can be expected to be small. For most circuits, it is even $T = 1$, i.e., there is no oscillation at all. For an FSM traversal by time-frame expansion, the exit condition for the loop is, therefore,

$$\text{finished}() := \big(\text{exists } T \text{ such that } R(t) = R(t-T)\big).$$

However, how can this exit condition be determined? Note that for symbolic FSM traversal, it is easy to recognize the fixed point because sets of states and their images under the transition function are stored *canonically* as BDDs. In function finished(), the BDDs representing $R(t)$ and $R(t-1)$ of two successive iterations in Table I simply have to be checked for isomorphism to detect the fixed point. The iterative circuit array, however, represents state sets *noncanonically* as the *range* of a multioutput Boolean function implemented as a combinational circuit. In

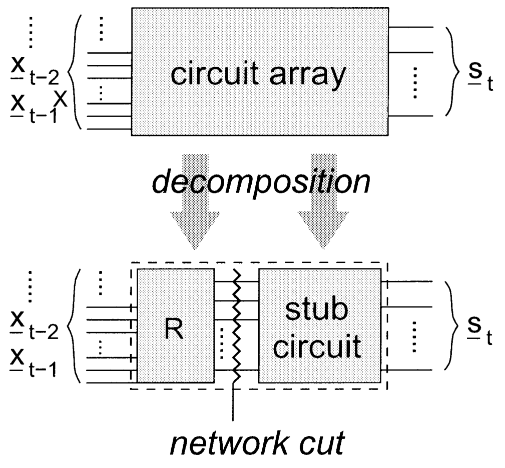


Fig. 3. Structural form of existential quantification.

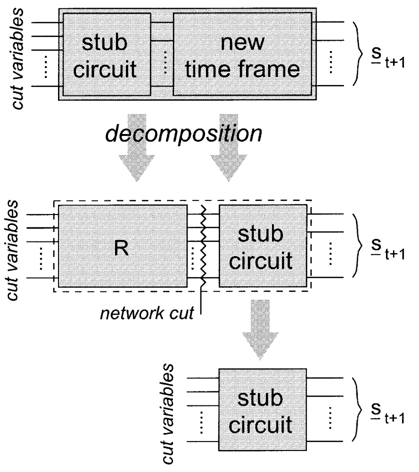


Fig. 4. Repeated quantifications.

order to detect the fixed point, it is necessary to check that the ranges of the Boolean functions of two time-frame expansions of lengths $t+T$ and t, respectively, are equal (where T is the fixed-point oscillation period).

To accomplish this, we introduce a structural form of existential quantification, which is based on a functional decomposition of the iterative circuit array and a network cut as shown in Fig. 3. The upper part of Fig. 3 shows the iterative circuit array representing the set of reachable states, $R(t)$, at the state variables $\underline{s}_t$ at time t. This combinational circuit comprises the complete functional information relating input sequences of length t to the resulting states of the FSM (and can be quite large for large values of t). The information we are interested in, however, is only *what* states are possible, not *how* they can be produced in the machine. We are only interested in the *range* of the multioutput function given by the state variables $\underline{s}_t$. Therefore, the circuit array is decomposed into two parts: a range-equivalent circuit (called "stub circuit" [11], [18]) and the remaining circuitry (denoted R in Fig. 3). Two circuits are called range-equivalent if their underlying Boolean functions have the same range. This means, both circuits produce the same set of output values, although they have different domains and functions. The range-equivalent stub circuit produces exactly the same output values as the original circuit array. By cutting the remaining circuitry

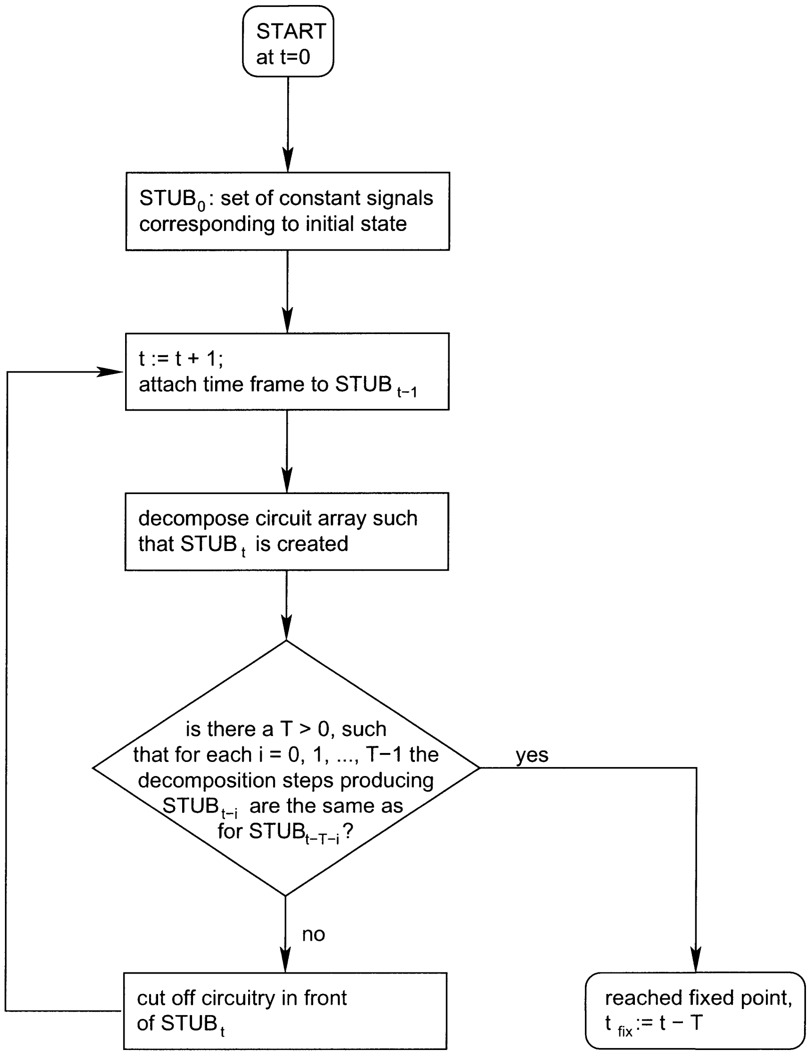

START at t=0
STUB$_0$: set of constant signals corresponding to initial state
t := t + 1; attach time frame to STUB$_{t-1}$
decompose circuit array such that STUB$_t$ is created
is there a T > 0, such that for each i = 0, 1, ..., T−1 the decomposition steps producing STUB$_{t-i}$ are the same as for STUB$_{t-T-i}$?
yes
no
cut off circuitry in front of STUB$_t$
reached fixed point, t$_{fix}$:= t − T

Fig. 5. Flowchart of Structural FSM Traversal.

off along the indicated cut line, the functional dependency of the state variables $\underline{s}_t$ on the input variables $\underline{x}_t, \underline{x}_{t-1}, \ldots$ is removed, but the information about which states are reachable is maintained. Cutting at the cut line, in some sense, existentially quantifies the structural set representation given by the circuit array with respect to these input variables. This greatly simplifies the data structure because it removes a lot of unnecessary information.

The decomposition and cutting step can be performed in every iteration of structural FSM traversal [just like existential quantification is performed in every image computation during symbolic FSM traversal (see Fig. 4)]. Instead of appending a time frame to the end of the iterative circuit array, we append it to its range-equivalent circuit. For this new circuit, we can again perform a decomposition as in Fig. 3 so that we obtain a new range-equivalent circuit representing the state vectors for the next iteration.

Besides significantly compacting the size of the set representations, this decomposition serves a second important purpose: it allows us to detect the fixed point. If $R(t)$ and $R(t+T)$ are equal, it must be possible to decompose the circuit arrays of length t and of length $(t+T)$ such that identical range-equivalent circuits are produced in both cases. This means that the fixed point check amounts to checking the *identity of the decomposition steps* used in different iterations.

Fig. 5 shows a flowchart of the overall algorithm of structural FSM traversal. It starts by constructing a stub circuit representing the initial state. In each iteration of the loop that follows, a new time frame is attached to the current circuitry representing the state set $R(t-1)$, and the decomposition leading to the new stub circuit is performed. This step together with the network cut corresponds to procedure new_states() of Table I. Function finished() in the *until* condition corresponds to checking whether previous decompositions could be successfully repeated, so that the same range-equivalent circuits are produced again.

The most important question, however, is still open: How can a range-equivalent circuit be synthesized? This is a difficult problem and a crucial point in structural FSM traversal. It is possible to formulate an exact algorithm that produces an exact range-equivalent circuit. Such an (academic) algorithm is described and illustrated in [18]. For practical circuits, however, an exact stub calculation is prohibitively complex. Instead, approximations to range-equivalent circuits must be sought.

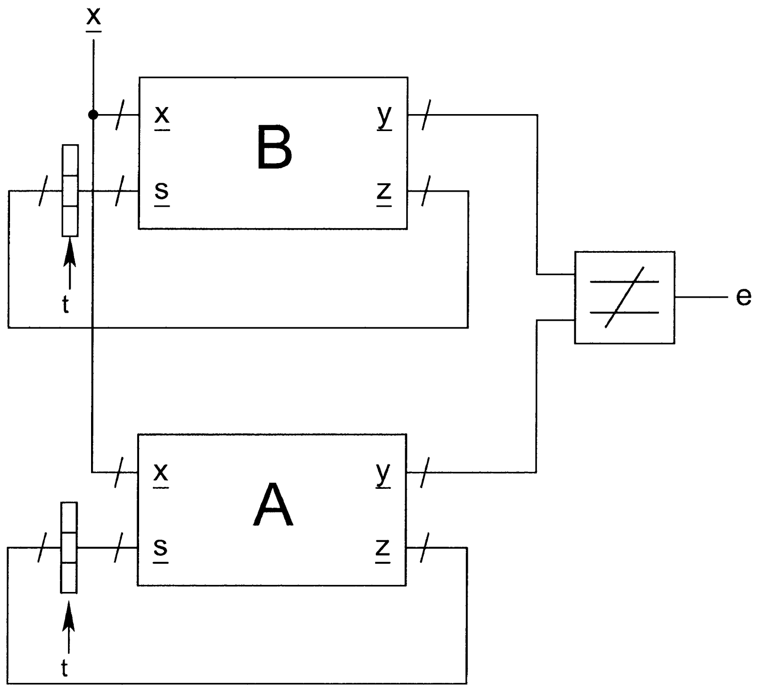


Fig. 6. Product machine of designs A and B (sequential "miter").

These approximations have to be application-specific. For property checking they may have to look different than for sequential equivalence checking. In fact, it turns out that for sequential equivalence checking, the stub can be approximated by the same circuit transformations that are common practice in today's combinational equivalence checking techniques. Simply by determining functional equivalences between internal signals of the two designs, we are able to find good decompositions to approximate the set of reachable states by a stub circuit. This is the subject of the following section.

III. Sequential-Equivalence Checking

In this section, we consider the application of structural FSM traversal to checking functional equivalence of sequential circuits. In particular, we review and discuss an algorithm called Record&Play initially proposed in [11] for sequential equivalence checking. Note that there are several notions of equivalence proposed in the literature. This paper is based on the most widely used notion called *reset equivalence*. It is assumed that designated initial states (or sets of initial states) for the two circuits under comparison exist together with a method (e.g., an external reset line) to bring each machine into a designated initial state. Checking design equivalence then amounts to checking the equivalence of the initial states. A similar notion proposed by Pixley is *sequential hardware equivalence* [19], which models the initialization process within the FSM descriptions of the designs. For each circuit, an input sequence (or a set of input sequences) exists which drives the circuit into a desired start state (which may be one in a set of desired start states). The output behavior of the circuits during initialization is disregarded when checking equivalence. Only the postsynchronization behavior is considered. There are also other notions of equivalence which are not considered in this work as, e.g., *safe replaceability* [20] and *3-valued equivalence* [21].

Sequential equivalence checking is a classical application of FSM traversal. The sequential equivalence checking problem is typically modeled by building a product machine (Fig. 6) of the two designs A and B being compared (also called *sequential miter*). The two components A and B of the product machine have the same set of primary inputs (PIs). Their corresponding outputs feed a comparator producing a single output e. The combined states of the FSM's A and B form the product states of the product machine. The initial state of the product machine is given by the concatenation of the individual initial states of A and B. The designs are equivalent if the product machine produces $e = 0$ for all inputs and all its reachable states. If there is a reachable state of the product machine for which $e = 1$ is possible, the sequence of input patterns driving the product machine into this state is called a *distinguishing sequence* or *(sequential) counter example* of the designs A and B.

When expanding the product machine into time frames, the iterative circuit array is an array of combinational logic blocks of the miter which are connected at the state variables. Note that the computation of an exact range-equivalent circuit as discussed in Section II is not feasible but for very small designs. Clearly, an approximative technique is needed. The question arises, however, how an approximation to a range-equivalent circuit can be determined. As mentioned, the same kind of transformations that have proven successful in combinational equivalence checking are also very useful in this application. A typical combinational equivalence checking tool determines internal signals of the combinational miter which are functionally equivalent (sometimes called "cut-points" [9]). These cut-points are merged in order to simplify the verification problem, making it easier to prove the equivalence of other cut-points and, eventually, of the circuit outputs.

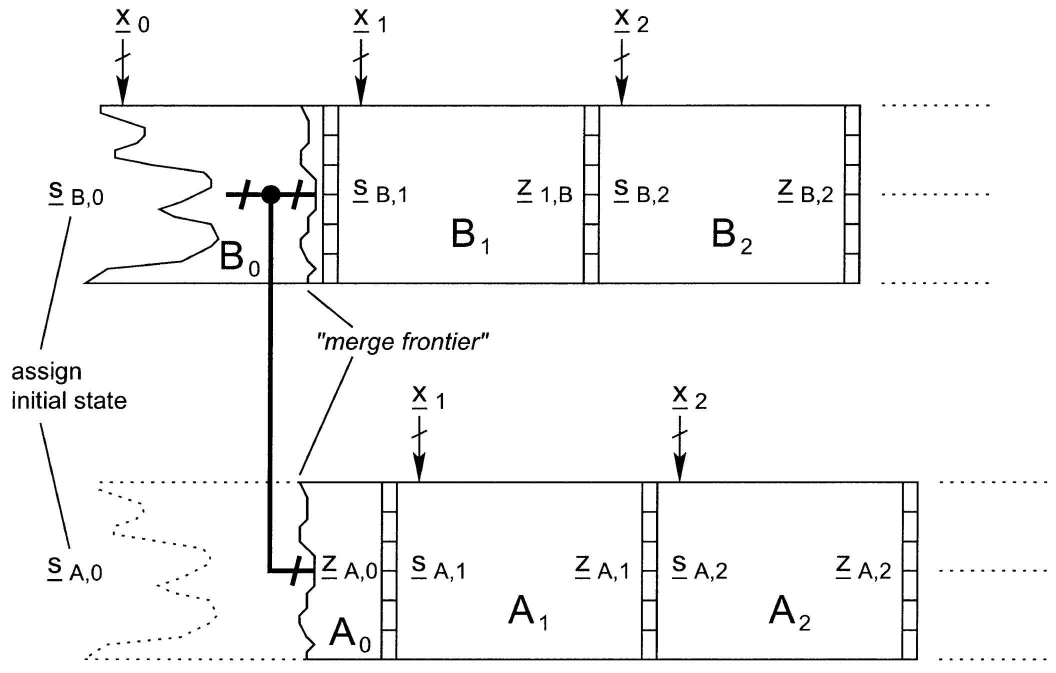


Fig. 7. Time-frame merging by sharing of logic.

In sequential equivalence checking, these functional equivalences can be used to *approximate the state set* in a structural FSM traversal.

Fig. 7 illustrates approximative structural FSM traversal based on merging of cut-points. The iterative circuit array obtained from a time frame expansion of the product machine consists of two components resulting from the two designs, A and B, being compared for equivalence. In Fig. 7 these two components are drawn separately. This iterative circuit array is also called *miter array* in the sequel.

At the beginning of the traversal the first time frame of the product machine is constructed. At the state variables $\underline{s}_{A,0}$ and $\underline{s}_{B,0}$ of the first time frame the initial state is injected as a set of constant signals. These constants simplify the logic and may propagate into the iterative circuit array. In the combinational logic of the time frame expansion, we can now identify functionally equivalent internal signals. These cut-points are merged such that logic between the two components of the miter array is shared and subsequent steps of cut-point identification and merging are simplified. 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.

The circuitry behind the merge frontier constitutes an approximation to a range-equivalent circuit. The inputs to this circuit are the merged signals of the merge frontier and the outputs are the state variables $\underline{s}_{A,t}$ and $\underline{s}_{B,t}$ of the iterative circuit array. By cutting through the signals of the merge frontier, we perform the existential quantification step of Fig. 3. Note that the merging and cutting process yields a very compact representation of a state set as a circuit which in the extreme case may consist only of a set of wires. As an example, consider Fig. 8. It shows a stub circuit derived from a time frame expansion of a product machine for equivalence checking of two designs with three state

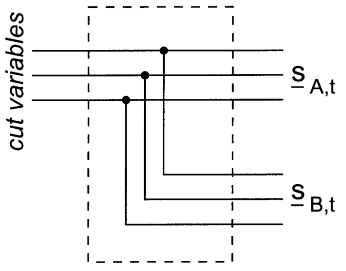


Fig. 8. Simple-stub circuit.

variables each. Since there are six state variables, the state space of the product machine comprises $2^6 = 64$ states. The stub circuit very compactly represents $2^3 = 8$ of these.

The process of merging and cutting produces a stub circuit which is not exactly range-equivalent but an approximation. The deviation from the exact image is in fact introduced by the cutting procedure and not by merging of cut-points, since the latter does not alter the logic functions of the iterative miter array and therefore maintains the image of the miter array.

However, the approximation is always a conservative *over-approximation* of the reachable state set. The range of the circuitry after the cut is a superset of the range of the original network. This becomes intuitively clear by reconsidering the decomposition of Fig. 3. The remainder circuit denoted "R" imposes constraints on the inputs of the stub circuit. Before cutting, the ranges of the decomposed circuit and the original circuit are identical because they are functionally equivalent. By cutting, the constraints on the inputs to the stub circuit are removed. For a certain set of input vectors, the stub circuit will still produce the same output vectors as the original circuit. However, additional output vectors may be possible. These additional vectors corresponding to additional states can be modeled by a set, $A(t)$ being added to the set of reachable states, $R(t)$, in each iteration of the traversal of Table II.

TABLE II
APPROXIMATIVE STRUCTURAL FSM TRAVERSAL

```
approx_structural_FSM_traversal($\delta$, $S_0$)
{
  $t := 0$;
  $R(t) := S_0$;   /* initial state set */
  repeat
  {
    $t := t + 1$;
    $R(t) := img(\delta, R(t-1)) \cup A(t)$ ;
  }
  until (finished());
}
```

As a consequence, not only the states reachable from the initial state will be visited during traversal, but also all states "injected" by the approximation and all states reachable from these. In the application of structural FSM traversal in sequential equivalence checking, it is necessary that in the additional, possibly unreachable, states output equivalence holds. Otherwise, false negatives may occur. Fortunately, it is in the nature of the described approach of identifying equivalent signals and cutting at the merge frontier that this property is very often fulfilled.

On the other hand, the over-approximation of the reachable state set has also a very beneficial effect. If the additional states, $A(t)$, are states of the reachable state set that exact FSM traversal visits in a later iteration then the fixed-point iteration is accelerated, sometimes greatly. Refer to [18] for an example illustrating this effect.

As discussed in Section II, the fixed point is detected by determining that the same stub circuit can be obtained over and over again. This can be done by recording for each stub circuit the set of circuit transformations that yields this circuit. If the same transformations can be repeated, the same stub circuit is produced and the fixed point is reached.

A key aspect of structural FSM traversal is that the same circuit transformations and reasoning steps performed in a miter for combinational equivalence checking can be used for approximating the stub in the sequential case. The order in which these transformations and reasoning steps are performed is important. For this reason, a general data structure called *instruction queue* was introduced in [11]. It is a *queue* of instructions for circuit transformations being used to merge the logic of a certain time frame t in the miter array. 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 operation is called record in [11]. 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 every time frame. Finally, an instruction queue is obtained that remains valid also in later time frames and which fulfills the task of proving the equivalence of the circuit outputs. The procedure of [11] is now described in more detail. We restrict the description of the algorithm to the case that the oscillation period is $T = 1$. Section V will justify that this is sufficient for virtually all practical designs. For $T = 1$, only a single instruction queue needs to be processed. The general case for $T > 1$ handling multiple instruction queues has already been described in our original paper [11] and in [18].

Fig. 9 shows the proposed algorithm for FSM equivalence checking, based on structural FSM traversal. The pseudocode for the algorithm record_and_play() is given in Table III. The various steps of the algorithm are explained in detail using the following example.

The variable *stub_levels* 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. These constant values may propagate to the next state variables. Consider Fig. 10. 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 Fig. 11. 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, routine record_and_play() (Table III) is called. This algorithm identifies network transformations to merge the logic circuitry and to facilitate equivalence checking at the outputs of this time frame. In our example, only node substitutions of functionally equivalent signals (cut points) are considered as network transformations, however, more general transformations can be conceived. These network transformations are performed in a controlled way by the instruction queue in order to detect whether the fixed point is reached. For each time frame, we store one set of instructions Q_t that keeps exact records of all transformations performed in that time frame. Routine record_and_play() consists of two steps. In the first step, the previous instruction queue Q_t is analyzed to determine whether or not the recorded circuit transformations are still valid in the current time frame. In the second step, additional network transformations are determined. Every transformation performed is put into the new instruction queue Q_{t-1}. The fixed point check consists of determining whether all previous instructions could be repeated and no additional transformations became possible. In the first step of our example, no previous instruction queues exist. The logic in the circuitry is merged as shown in the right part of Fig. 11 and the performed transformations are stored in Q_0.

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 or an SAT solver can be invoked to calculate a distinguishing sequence.

The algorithm now determines the merge frontier to prepare for the network cut. This merge frontier is found using a backward trace that starts at the state variables of each component A and B of the miter array. The backward trace finds those signals

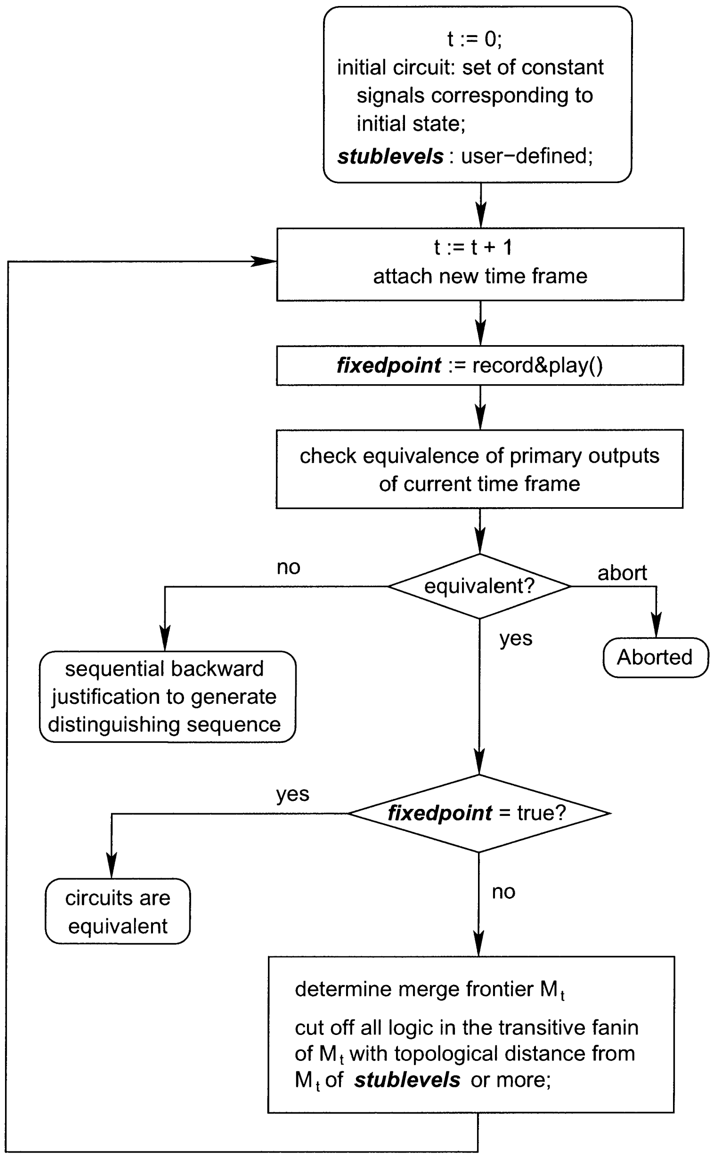

t := 0;
initial circuit: set of constant signals corresponding to initial state;
stublevels : user−defined;
t := t + 1
attach new time frame
fixedpoint := record&play()
check equivalence of primary outputs of current time frame
equivalent?
no
abort
yes
sequential backward justification to generate distinguishing sequence
Aborted
fixedpoint = true?
yes
no
circuits are equivalent
determine merge frontier M_t
cut off all logic in the transitive fanin of M_t with topological distance from M_t of stublevels or more;

Fig. 9. Sequential equivalence-checking algorithm.

which fanout into both components. These merge points constitute the cut frontier. The circuitry behind the merge points is the approximated stub circuit. In principle, *false negatives* can occur as a result of cutting at the merge points. In practice, this can often be avoided by leaving a sufficient number of additional logic levels in the fanin of the merge frontier. This number of logic levels is called *stub_levels* in Fig. 9 and is a user-defined parameter. Typical values are between 0 and 5.

Consider again the right portion of Fig. 11 and consider the state variables that are not assigned a constant value. These are the state variables 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. At this point, no other nodes belong to the merge frontier. In the figure, the merge point is denoted by a black dot. Assuming the parameter *stub_levels* equal to 0 the circuit array can be cut at signal e_0. A new time frame is attached. The result is shown in the left portion of Fig. 12. The newly introduced variable at the cut line is denoted s. The fanout system of s and the constant signal j_1 constitute our approximation of the stub circuit for the reachable state set $R(0)$ as indicated in Fig. 11.

Again, algorithm record_and_play() is invoked in the iterative circuit array. At this point, the previous instruction queue does not contain any instructions which are valid also in the current time frame. New transformations are recorded in Q_1 as shown in the right portion of Fig. 12. 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 by black dots in the right portion of Fig. 12. Assuming *stublevels* = 0, we cut the circuit array at these fanout stems. However, in this case, this does not result in any removal of logic.

A new time frame is attached as shown in Fig. 13 and a new instruction queue must be recorded. The transformations lead to the circuit array as shown in Fig. 14. We determine a new merge frontier suggesting a cut at signal f_2. The next iteration leads to the circuit array of Fig. 15. Just like in the previous time frame, no constant values exist at the state variables and it is determined in record_and_play() that the instruction queue

TABLE III
PROCEDURE record_and_play()

```
/* Routine operates on a global data structure for the current miter array with present
state variables $\underline{s}$ and next state variables $\underline{z}$. It has the time parameter $t$ and the set of
instruction queues $\{Q_i\}$ as global variables */

record_and_play()
{
  $Q_t := \emptyset$;
  fixedpoint := true;
  for (each instruction $\alpha \in Q_{t-1}$) {
    verify whether or not circuit transformation $\alpha$ is
      valid in current time frame;
    if (valid) {
      execute $\alpha$;   /* perform circuit transformation */
      $Q_t := Q_t \cup \{\alpha\}$;   /* put in queue */
    } else {
      fixedpoint := false;
    }
  }

  /* find new circuit transformations, if possible */
  for (each node in circuit array) {
    identify circuit transformation $\alpha$;
    if ($\alpha$ reduces literal count of circuit array) {
      perform transformation $\alpha$;
      $Q_t := Q_t \cup \{\alpha\}$;   /* put in queue */
      fixedpoint := false;
    }
  }
  return fixedpoint;
}
```

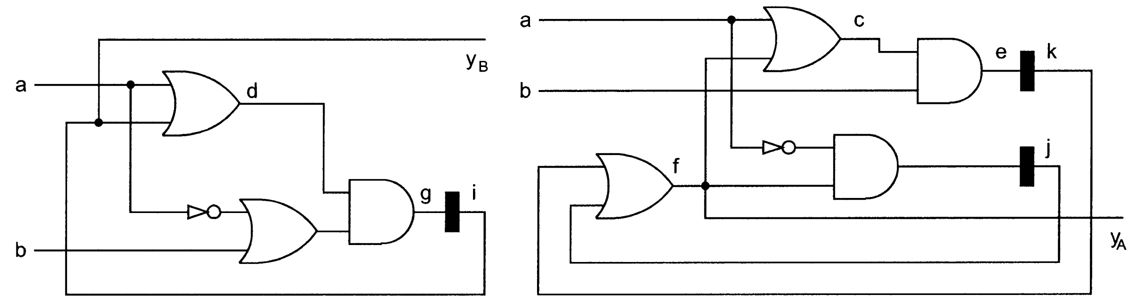


Fig. 10. Circuit examples with initial states $S_{0,A} = S_{0,B} = \underline{0}$.

Q_2 can be played. Actually, all recorded transformations turn out to be valid in the current time frame so that the circuit of Fig. 16 results. The algorithm record_and_play() returns true to indicate that a fixed point is reached, and the equivalence check terminates successfully.

Note that the described method is not restricted to a single initial state. If a set of initial states is used, additional circuitry must be attached in front of the first time frame that represents the given set of initial states. In other words, a stub circuit must be created, representing the set of initial states that feeds the first time frame. If an initializing sequence is given, the above iteration has to be modified slightly. Instead of assigning an initial state at the state variables, the values of the initializing sequence are assigned to the PIs 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 initialization process [20].

IV. RELATED WORK

In the field of sequential circuit verification, to-date there has not yet been an industrial breakthrough for structural techniques as has been the case for combinational equivalence checking. With the advent of bounded model checking [15], the concept of time-frame expansions is, however, now being successfully applied in property checking of sequential circuits.

For sequential equivalence checking, only a few approaches have been proposed using structural techniques. Huang *et al.* [12], [21], [22] explored the use of sequential ATPG algorithms. Their approach to equivalence checking is based on the assumption that practical designs under comparison contain a lot of equivalent state variables. The procedures in [12] start with a set of candidate pairs for equivalent state variables and perform an induction-based filtering process to eliminate the wrong candidates. It is assumed that simple relationships exist

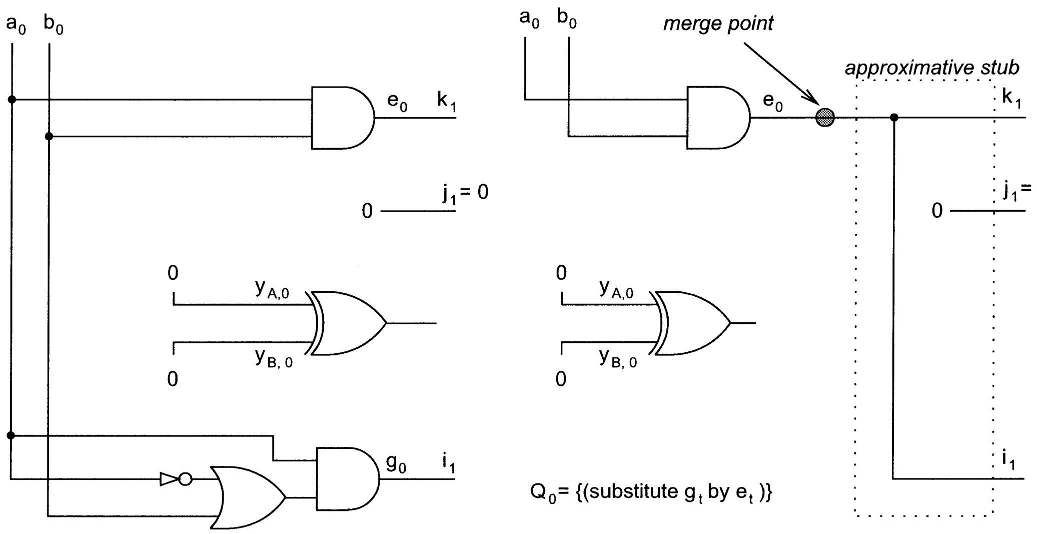


Fig. 11. Circuit array in first iteration.

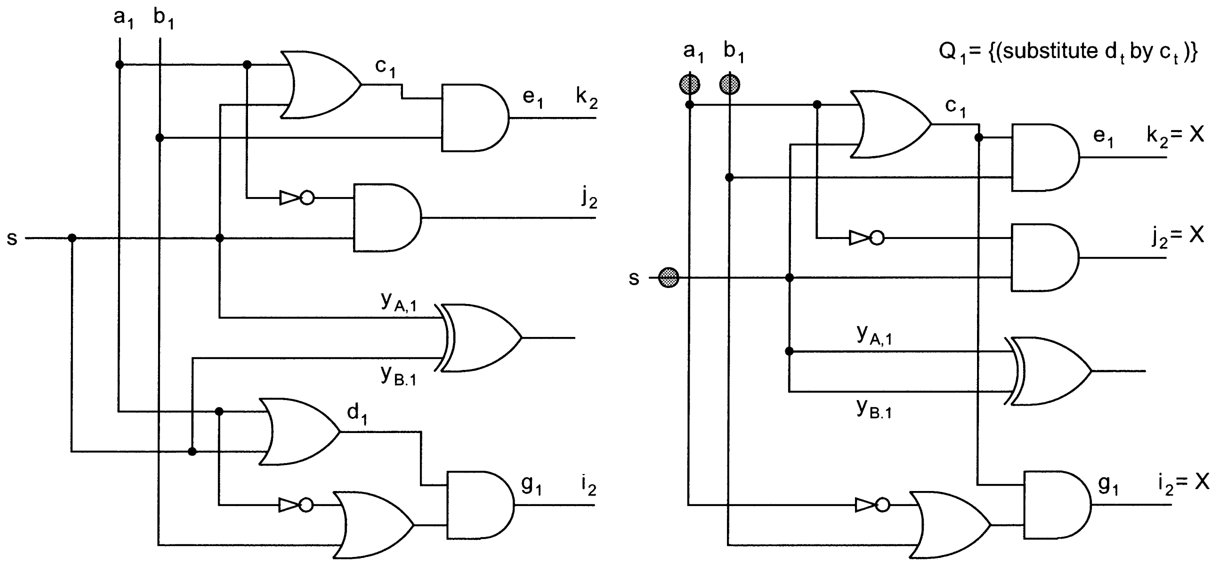


Fig. 12. Circuit array in second iteration before and after merging.

between state variables that can be obtained by simulation. This is promising for circuits with similar encodings but may fail in other cases. Reference [21] describes a simulation-based preprocessing technique to reverse the register movements introduced by retiming so that more equivalent latches can be found. Note, however, that there are several obstacles to this approach. In many cases, simulation yields large equivalence classes, hiding the correct reverse retiming. If additional logic optimizations have been performed, it may be impossible to find a corresponding register position. Also, when checking reset equivalence, reverse retiming requires us to recompute the initial state, which can be computationally costly. The work of [22] combines the previous approaches [12], [21] with local BDD-based verification techniques and extensions to backward justification to improve the overall framework.

An approach which uses a structural FSM traversal in a way very similar to the Record&Play algorithm [11] reviewed in Section III was proposed by van Eijk [13]. The algorithm performs a fixed-point iteration that iteratively refines an equivalence partition of the set of signals in the circuit. In each step of the iteration, two consecutive time frames of the sequential miter are considered. The first time frame is used to model the functional equivalences between signals according to the current equivalence partition. The second time frame is used to derive those functional equivalences being valid under the assumption that the equivalences in the first time frame hold, producing a new equivalence partition. Starting at the initial state, this is iterated until a fixed point is reached. The functionally equivalent signals are determined using BDDs, which are expressed in terms of state variables (time frame boundaries) and PIs. The equivalences found in each iteration are put into conjunction to form a characteristic function called *signal correspondence relation*. This function evaluates to 1 for every state for which the signals comply with the equivalence relation. The signal correspondence relation is a representation of the states considered in a certain iteration of the algorithm. The complement of the signal correspondence relation is used as a don't-care set when

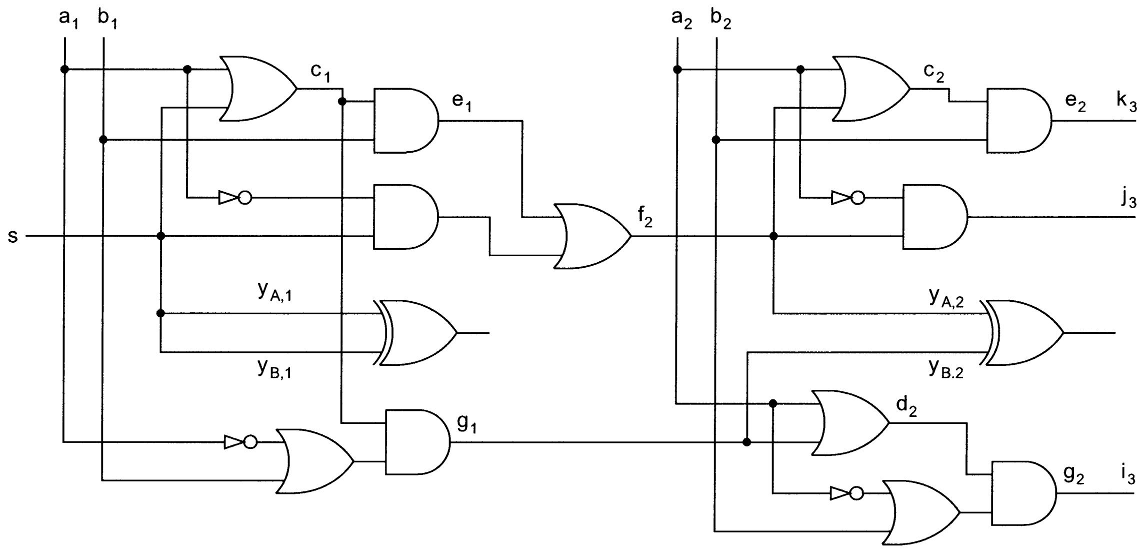


Fig. 13. Circuit array in third iteration before merging.

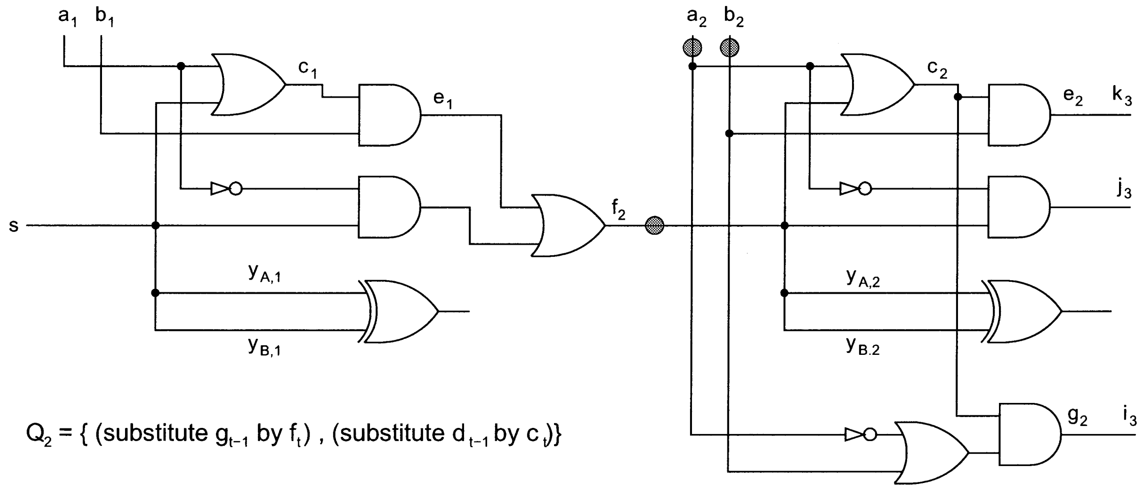


Fig. 14. Circuit array in third iteration after merging.

computing the equivalences among the functions of the next time frame.

At a first look, the similarities between the algorithms of [11] and [13] are not apparent, since the articles are written using different notations and emphasizing different aspects. A closer analysis, however, reveals that the method of [13] is very similar to [11] in various aspects. Both algorithms derive state-set representations from the structure of the circuit, i.e., they use relationships between signals in the circuit for representing sets of states. In [11], logic implications between signals are used for implicant-based circuit transformations [23]. Note that transformations based on functional *equivalences* between signals ("two-way" implications) become simple signal substitutions. In this special case, the instruction queue of [11] simplifies to a list of node substitutions. This has been illustrated already in the example of [11], repeated here in Section III (Figs. 10–16).

The algorithm of [13] is based on this restriction to functional equivalences instead of more general signal relationships. Instead of performing node substitutions, the equivalences are stored as a partition of the set of nodes in the circuit. (However, in the BDD computations, deriving new signal relationships in each iteration, state variables, which are functionally equivalent with other signals in the time frame, are indeed substituted by the functions at these signals. This is the equivalent to merging the corresponding nodes in the circuit.)

Note that there are several possibilities to represent and make use of equivalent internal node functions in a structural FSM traversal. Besides making node substitutions as in [11] or storing functional equivalences as a characteristic function [13], one can also store the functional equivalences as logic constraints imposed on Boolean reasoning engines such as implication engines [4] or SAT solvers. In Section VI, we discuss an implementation using this approach.

Also, although not obvious at first glance, the algorithm of [13] includes an over-approximation of the set of represented states similar to the network cut in [11]. In each iteration, the current equivalence partition is "injected" in the first of the two considered time frames. However, there is no logic feeding the first time frame. This has the same effect as cutting away the

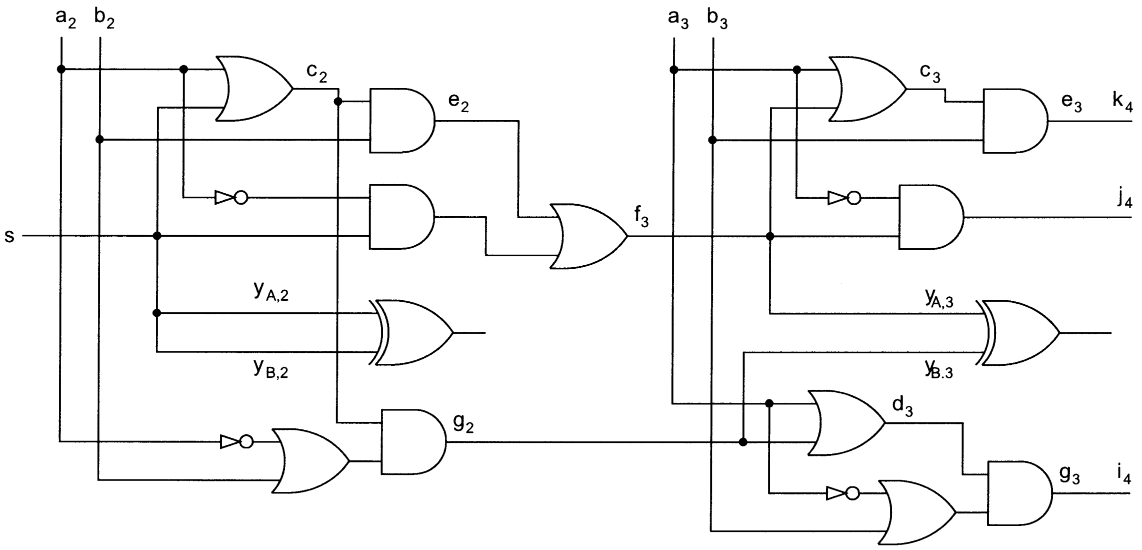


Fig. 15. Circuit array in fourth iteration before merging.

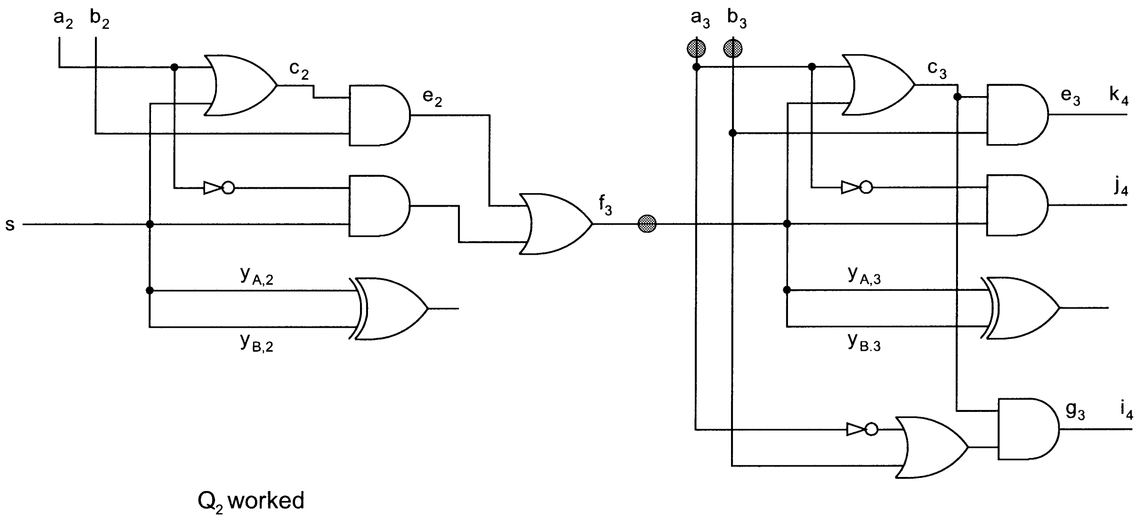


Fig. 16. Circuit array in fourth iteration after merging.

logic preceding the last two time frames in the iterative circuit array of [11], with a cut frontier always at the state variables.

In this sense, van Eijk's algorithm is a structural FSM traversal with specializations given by the restriction to functional equivalences and the implicit network cut at the state variables. These specializations simplify the framework and allow for a specialized formulation of the algorithm as a greatest fixed point computation on a series of equivalence partitions of the nodes in the miter. Besides the benefits given by these simplifications, there are also some drawbacks. First, in the formulation of [13], the algorithm provides no means for dealing with the false negative problem arising from the over-approximation by cutting the network at the state variables. In [11], the false negative problem is dealt with by leaving additional levels of logic in front of the merge frontier. Second, if the circuits have been retimed, van Eijk's algorithm needs to be extended by adding enough logic to capture the delays between equivalent retimed signals. If the circuits have been retimed heavily, this may result in appending a large number of additional time frames to the end of the two basic time frames. Note that in our approach, retiming is handled implicitly. Equivalent retimed signals are not merged until a sufficient number of time frames have been added. However, since the network cut is based on merging of equivalent nodes, the cut frontier usually also runs across several time frames cutting away the merged circuitry and maintaining the unmerged circuitry of these time frames. In this way, only the logic circuitry that is necessary to continue with the iteration is kept in the representation. This is not the case for the static cut frontier of [13] at the state variables.

The use of monolithic BDDs in the implementation of [13] makes it possible to identify all equivalences that exist between signals in the circuit, since all internal logic functions are represented by BDDs. However, such an implementation is likely to fail for larger circuit examples. This is especially true if the circuits have been retimed so that logic circuitry of more than two time frames has to be represented in a BDD. For this reason, researchers explored SAT-based versions of the specialized algorithm of [13]. Bjesse and Claessen [14] proposed a SAT-based structural FSM traversal for use in property checking that calculates an over-approximation of the reachable state set. In property checking, the considered product machine is composed of logic representing the design and logic representing the property. Note that one cannot generally expect to find many functionally equivalent signals within this product machine. In contrast to equivalence checking of circuits which exhibit structural similarity, it is only in special cases that the product machine in property checking contains many functional equivalences.

As a result, the approximation of the reachable state set obtained from the functional equivalences found may be very coarse and likely to yield false negatives. Therefore, [14] discusses several enhancements. One suggestion is to also consider "one-way" logic implications between signals in the circuit, i.e., dropping the restriction of using only functional equivalences. A second enhancement is to avoid false negatives by improving the approximation of the reached state set. In each iteration of the algorithm, when computing the new signal relationships, not only one time frame is considered but k time frames in which the old set of signal relationships is assumed to hold. Note that in our approach, this corresponds to leaving additional levels of logic in front of the merge frontier. Additionally, the proof method in [14] is strengthened by enforcing "unique states induction." Special formulas are added to the SAT instance that require the system to assume different states in every trace applied to the k time frames. By adding this constraint, the algorithm can be made complete because, for every system, there exists a k large enough to avoid any false negative. Note that this approach could also be taken in our framework by adding a unique states induction formula. On the other hand, our algorithm can also be made complete by computing an exact stub circuit by using more general circuit transformations. The reader may refer to [18] for a detailed discussion of this subject.

V. Theoretical Issues

In this section, we discuss some theoretical properties of structural FSM traversal. First of all, we are interested in the convergence behavior of the set of reachable states $R(t)$ in exact structural FSM traversal, since this has an influence on how many iterations are needed to determine that all states of the FSM have been visited. This is discussed in the following Section V-A.

In the succeeding sections, we consider the practical sequential equivalence checking algorithm *Record & Play* with regard to completeness. Certainly, *Record & Play* in general is not a complete algorithm, i.e., there may be functionally equivalent designs which have been retimed and optimized for which equivalence cannot be proven. In Section V-B, we analyze for which classes of verification problems *Record & Play* is complete.

A. Exact Structural FSM Traversal

For analyzing the evolution of the reachable state set $R(t)$ over time, it is helpful to imagine $R(t)$ as a set of states in the STG with a special mark that is passed on to successor states in the next time step. Consider a set of marked states, $R(t)$, at time t. One time step later, at $t+1$, these states pass their marks on to all of their immediate successor states. Their own mark is erased, unless they receive a new one from an immediate predecessor state. Now, all marked states form the reachable state set $R(t+1)$. As we proceed in time, the states in the STG become repeatedly marked and unmarked as described. Intuitively, since the FSM has a finite set of states, and its next-state behavior is deterministic, the time series $R(t)$ must at some point in time enter a stationary behavior. Either there will be a single final set R_∞ or $R(t)$ will periodically cycle through a set of state sets. If $R(t)$ converges to a final set R_∞, this set corresponds to a pattern of marks which does not change any more after a certain time t_{fix}, i.e., it becomes a static pattern. If $R(t)$ exhibits a periodic long-term behavior, this corresponds to periodically repeating patterns of marks in the STG, beginning at a certain time t_{fix}. As we will see, whether we have an aperiodic or a periodic behavior is determined by certain structural properties of the STG.

For the following analysis we need the following recursive definition.

Definition 2 (Recurrent State): A state in the STG, G is called *recurrent state* if it lies on a cycle in G, or if it has a predecessor which is a recurrent state.

Recurrent states are contained in the reachable state set $R(t)$ infinitely often, and they appear periodically. This is easy to see for recurrent states which are lying on a cycle (called *cycle states* in the sequel), by observing the $R(t)$–marks they are sending and receiving. The length of the cycle determines how many time steps it takes until a mark that has been sent out by a state returns back to it and is sent again. This time is called the state's *period of recurrence* and it is equal to the length of the cycle. However, there are also states which are not involved in a cycle, but are nevertheless recurrent. Such states are reachable from cycle states and, therefore, they receive, with some delay, all $R(t)$–marks which the cycle states send out. We say a state *inherits* the recurrence periods of the states from which it can be reached. Note that in a particular run of the machine, a recurrent state that is not a cycle state can occur only once. However, for different runs of the machine, it may occur at different times. The reachable state set $R(t)$ contains information about all possible runs of the machine. Therefore, a recurrent but noncycle state is an element of $R(t)$ periodically just like a "true" cycle state.

Definition 3 (Transient State): A state which is not recurrent is a *transient state*.

Transient states are not reachable by recurrent states. They occur only once in the reachable state set $R(t)$. Transient states are only possible if the initial state of the FSM is a transient state. They are usually part of the initialization process for the machine and do not belong to the normal mode of operation.

Note that our definitions of recurrent and transient states differ from literature concerned with a probabilistic analysis of FSMs, such as [24]. The objective there is to determine the long-run probability for an FSM to be in a certain state. In that context, for example, a state is defined transient if there is a nonzero probability that the FSM will not return to it. Our definition requires that it is *impossible* to return to a transient state. In case there is a *possibility* that the FSM returns to a state, we call it a recurrent state, because it will be an element of the reachable state set, even if the *probability* for this to happen may be zero.

Since transient states occur only once, they cannot be part of the reachable state set in the fixed point we are seeking. So, we can focus the analysis of our FSM traversal solely on the recurrent states.

As discussed above, a recurrent state may inherit recurrence periods from its predecessor states. It also has periods of recurrence associated with the cycles on which it is located. The following lemma tells us how these different recurrences interact.

Lemma 1: Consider an arbitrary state s lying on a cycle of the STG of length q. Furthermore, let s have a recurrence period p. Then, after a finite number of time steps, state s also has a recurrence period p_g which is the greatest common divisor of p and q.

Proof: Let t_0 be the first time that s is in the reachable state set $R(t)$. Then, $s \in R(t_0 + m \cdot p + n \cdot q)$, for all integers $m \geq 0$ because of given periodicity p, and for all integers $n \geq 0$ because of cycle length q. Let T be the set of integers $k, k = t - t_0 = m \cdot p + n \cdot q$. Set T is (obviously) closed under addition, i.e., for all $k_1, k_2 \in T$ it is $(k_1 + k_2) \in T$. For $T \setminus \{0\}$, the following number-theoretic result holds (see, e.g., [25], Theorem 1.4.1): *A set of positive integers that is closed under addition contains all but a finite number of multiples of its greatest common divisor.* Note that T contains only multiples of the greatest common divisor p_g of p and q, because all elements of T can be written as $(m \cdot p + n \cdot q) = (m \cdot i \cdot p_g + n \cdot j \cdot p_g)$, where i and j have no common divisor. Specifically, T contains the elements $i \cdot p_g$ and $j \cdot p_g$. Since i and j have no common divisors, p_g is the greatest common divisor of the whole set. Because there is only a limited number of multiples of p_g which are not in T, there is a $l_{\text{trans}} > 0$, such that $k = l \cdot p_g \in T$ for all $l \geq l_{\text{trans}}$. Hence, $s \in R(t_0 + l \cdot p_g)$ for all $l \geq l_{\text{trans}}$. ∎

The STG of an FSM can be decomposed into its "cyclic" parts, the strongly connected components (SCCs) [26]. All states in an SCC are reachable from each other, i.e., they are lying on cycles. The FSM can be in these states arbitrarily often, therefore, the SCCs determine the "long-run" or "steady-state" behavior of the machine. By collapsing the SCCs into single vertices, we obtain a direct acyclic graph called the *SCC graph.*

The recurrence period p in the lemma may be inherited from a predecessor state. Or it may be due to another cycle of length p that state s is lying on. We can apply Lemma 1 successively to all pairs of periods p and q that a state has due to period inheritance and the cycles in which it is involved. This leads us to an interesting lemma for the states of an SCC:

Lemma 2: After a finite transition time, the smallest recurrence period p_{SCC} is the same for all states in an SCC. This period p_{SCC} is given by the greatest common divisor of all cycle lengths in the SCC and of all recurrence periods for states in the SCC that have been inherited from predecessor states outside the SCC.

Proof: All states in an SCC are reachable from all other states in the SCC. Hence, every state inherits all recurrence periods that develop according to Lemma 1, including a smallest one which is the greatest common divisor of all of them. ∎

If we view again the set of reachable states $R(t)$ as a set of states in the STG carrying a mark, then this lemma says that after a sufficient amount of time, each state in an SCC will be marked periodically. If the period is, for example, $p_{\text{SCC}} = 5$, then a state will be marked in every fifth time step and will be unmarked during the remaining four time steps. Since at every time step at least one state of the SCC is marked, we can partition the states in the SCC into equivalence classes of simultaneously marked states.

Lemma 3: The set of states of an SCC with a recurrence period p_{SCC} can be partitioned into p_{SCC} disjoint subsets C_m with $m \in \{0, 1, \ldots, (p_{\text{SCC}} - 1)\}$, such that $C_m = R(t_{\text{SCC}} + k \cdot p_{\text{SCC}} + m)$, for all integers $k \geq 0$.

Proof: Follows immediately from Lemma 2. ∎

This is an interesting first result. Let us consider the special case of an FSM that has all its states including the initial state within one strongly connected component. Such systems are sometimes called *nondecomposable* sequential systems, because the SCC graph of their STG consists of only a single vertex. In this case, the set of reachable states, $R(t)$, converges to a series of final sets C_m that oscillate with a period p_{SCC} related to the structure of the STG.

It is possible, however, that p_{SCC} is equal to 1, yielding a single final set $R_\infty = C_0$ to which $R(t)$ converges. This happens if there is a state in the SCC which has a self-edge or if there are cycles with lengths whose greatest common divisor is 1. In fact, such an aperiodic behavior is very typical for FSMs implemented by practical systems. This is also confirmed by our experiments (Section VII).

One way of obtaining a nondecomposable system is by modeling the process of initialization within the FSM description itself. The FSM can then be put into its initial state by applying a special input sequence called initializing, or synchronizing sequence.

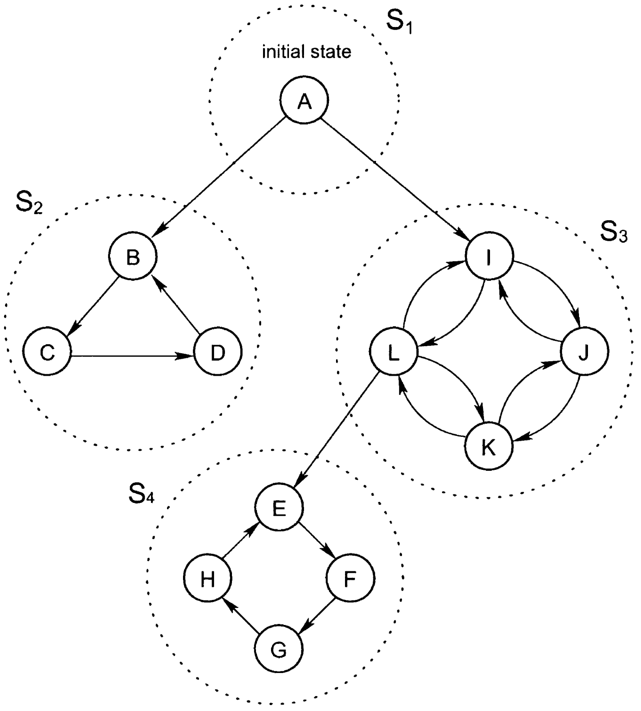
Definition 4: A *synchronizing sequence* of an FSM M is an input sequence that brings M to a known state s_0 regardless of the initial state or the output sequence. The state s_0 is called *synchronization state* of M.

If an FSM has a synchronizing sequence, then the synchronization state s_0 and all states reachable from it must be located within one terminal strongly connected component (TSCC). The reason is that from all states reachable from s_0 the machine can be put back into s_0 by applying the synchronizing sequence. In other words, s_0 is reachable from all states which are reachable from s_0. Hence, machines with synchronizing sequences are nondecomposable.

In the special case of a nondecomposable system, the basic definitions of transient and recurrent states of an FSM and those of a Markov chain are equivalent. Therefore, the following lemma which was originally derived in [24] for homogeneous discrete-parameter Markov chains with a finite state space can also be formulated in this context.

Lemma 4: If an FSM has a synchronizing sequence, then the fixed point recurrence period of all its states is 1.

Proof: Let l be the length of the synchronizing sequence. If the FSM is synchronizable, there is a walk of length l from each state s to a given state s_0. In particular, there is a closed walk $(s_0, s_1, \ldots, s_{l-1}, s_0)$ of length l. If $l = 1$ the greatest



t	$R(t)$	C_j
0	$\{A\}$	—
1	$\{B, I\}$	—
2	$\{C, J, L\}$	—
3	$\{D, E, I, K\}$	—
4	$\{B, F, J, L\}$	—
5	$\{C, E, G, I, K\}$	C_0
6	$\{D, F, H, J, L\}$	C_1
7	$\{B, E, G, I, K\}$	C_2
8	$\{C, F, H, J, L\}$	C_3
9	$\{D, E, G, I, K\}$	C_4
10	$\{B, F, H, J, L\}$	C_5
11	$\{C, E, G, I, K\}$	C_0
12	$\{D, F, H, J, L\}$	C_1
13	$\{B, E, G, I, K\}$	C_2
...	...	...

Fig. 17. STG and the first values of its reachable state set $R(t)$.

common divisor of all cycles in the SCC is 1. If $l > 1$, there is also a cycle of length $l + 1$: Consider state s_1 reachable from s_0 in one step. Since there exists a synchronizing sequence of length l there is a walk from s_1 to s_0 of length l. Appending this walk with one step from s_0 to $_1$ closes the cycle of length $l + 1$. The greatest common divisor of l and $l + 1$ is 1. Hence, by Lemma 2 the fixed point recurrence period of all states is 1, also. ■

Lemma 5: If an FSM has a synchronizing sequence of length l and if d is the sequential depth of the machine, then it takes at most $l + d$ time steps until all states of the machine are in $R(t)$ and have a recurrence period of 1.

Proof: The synchronization state s_0 is reachable in l time steps from every state of the FSM, including itself. Hence, state s_0 is an element of $R(t)$ for every $t \geq l$, i.e., from this time on it has a recurrence period of 1. Every other state in the STG is reachable from s_0 in at most d time steps. Therefore, it takes at most d time steps until all states have inherited the recurrence period 1 from s_0. ■

Note that the *sequential depth* of an FSM is given by the longest path among all shortest paths from the initial state to all nodes in the STG of the FSM. In conventional symbolic FSM traversal, the sequential depth is equal to the number of iterations needed to reach the fixed point.

Although most FSMs encountered in practice actually fall into the category of nondecomposable systems [19], it is generally possible that the SCC graph contains more than one SCC. Also, the initial state does not have to be a recurrent state. We therefore need to discuss the general case of an arbitrarily structured SCC graph.

If the SCC graph of a STG has more than one SCC vertex, the periods are inherited along the directed edges between the SCCs. According to Lemma 2, the recurrence period of an SCC is a proper divisor of all its predecessor SCCs. Therefore, the largest periods of recurrence are found in the "earliest" SCCs after initialization of the FSM. We call them entry SCCs (ESCCs).

Definition 5 (Entry SCC): An SCC in the STG which is entered by initialization or reached exclusively via transient states after initialization is called *entry SCC (ESCC)*.

Note that an entry SCC need not necessarily be a source of the SCC graph. Only if the initial state is a recurrent state there is a unique entry SCC which is also the source of the acyclic SCC graph. If, however, the initial state is a transient state, then there can be several entry SCCs.

Fig. 17 shows an example of such a STG.

This STG is composed of four SCCs: $S_1 = \{A\}, S_2 = \{B, C, D\}, S_3 = \{E, F, G, H\}$, and $S_4 = \{I, J, K, L\}$. S_1 is an SCC without cycles. It contains only the initial state A which is a transient state. S_2 and S_3 are entry SCC's according to Definition 5. S_4 is a terminal SCC.

Also shown in Fig. 17 are the first values of the time series of the reachable state set, $R(t)$. The initial state is contained in $R(t)$ only once for $t = 0$. The remaining states are recurrent, and we can easily verify that Lemmas 1 and 2 are correct. In SCC S_2 there is a unique cycle of length 3. Therefore, the states B, C, and D are contained in $R(t)$ alternately every three time steps. In SCC S_3 there are several cycles whose lengths all are multiples of 2. Hence, the states in S_3 recur in $R(t)$ in two alternating sets, $\{I, K\}$ and $\{J, L\}$. The SCC S_4 contains only one cycle of length 4. However, the recurrence period of its states is 2. The reason for this is that whenever state L (of SCC S_3) is in $R(t)$, state E will be in $R(t + 1)$, one time step later. State E inherits state L's recurrence period. The greatest common divisor of the inherited period of 2 and the cycle length of 4 is two.

Obviously, after a sufficient amount of time ($t = 5$), all recurrence "interferences" have taken place and a stationary oscillation has evolved. In this example, there are six different values for the set of reachable states, $R(t)$, which are repeated in the same order with a period of six time steps. These six sets together form a cover of the set of all recurrent states.

Theorem 6: Let P be the set of all recurrent states of an FSM. There always exists a cover $\{C_0, C_1, \ldots, C_{T-1}\}$ of P with $C_k \in 2^P$, and there is a time $0 \leq t_{\text{fix}} \leq \infty$, such that

$$R(t_{\text{fix}} + n \cdot T + k) = C_k, \quad 0 \leq n, \quad 0 \leq k < T.$$

T is equal to the least common multiple of the recurrence periods of the entry SCCs of the FSM. T is called fixed point oscillation period.

Proof: Let p be the recurrence period of an entry SCC E. Let S be the set of states of the entry SCC and all succeeding SCCs. All states in S inherit p, so the recurrence period r of any state in S must be a proper divisor of p. According to Lemma 3, the set of ESCC states E can be partitioned into p disjoint subsets. Let $K_i \in S$ be the union of such a subset with states from succeeding SCCs such that K_i contains all states of S occurring simultaneously in the reachable state set $R(t)$. Obviously, states with a period $r < p$ will be in $m = p/r$ distinct subsets K_i. This means that the K_i represent a cover of S: $S = \bigcup_i K_i$. The subsets K_i of the cover describe all possible combinations of states in S which can be in $R(t)$ simultaneously.

Without loss of generality, we consider an STG with two entry SCCs E_1 and E_2, having recurrence periods p_1 and p_2, respectively. Let S_1 be the set of states in E_1 and all its successor SCCs. Let S_2 be the set of states in E_2 and all its successor SCCs.

There exists a cover of S_1: $S_1 = \bigcup_i K_{1,i}$ consisting of p_1 subsets of S_1, and a cover of S_2: $S_2 = \bigcup_j K_{2,j}$ consisting of p_2 subsets of S_2. The subsets $K_{1,i}$ of the cover describe all possible combinations of states in S_1 which occur in $R(t)$ simultaneously. The subsets $K_{2,j}$ of the cover describe all possible combinations of states in S_2 which occur simultaneously in $R(t)$. Let us pick one set $K_{1,i}$ and one set $K_{2,j}$ such that both sets occur simultaneously in $R(t)$. $K_{1,i}$ will reoccur after p_1 time steps, $K_{2,j}$ will reoccur after p_2 time steps. The number of time steps it takes until both reoccur simultaneously is equal to T, the least common multiple of p_1 and p_2. Therefore, there are T different combinations $C_k = K_{1,i} \cup K_{2,j}, 0 \leq k < T$ in which subsets of the cover of S_1 and subsets of the cover of S_2 occur simultaneously in $R(t)$. The states in S_1 occur in $(T/p_1) \geq 1$ combinations, the states in S_2 occur in $(T/p_2) \geq 1$ combinations. Hence, the C_k cover all recurrent states of the STG.

The generalization of this argument from two ESCCs to n ESCCs with recurrence periods $p_1, p_2, \ldots, p_n$ is straightforward. ■

In our example of Fig. 17, there are two entry SCCs, S_2 and S_3, with periods $p_2 = 3$ and $p_3 = 2$. The least common multiple of these two numbers is $T = 6$. This is the fixed point oscillation period of the reachable state set that we have found also empirically for this STG.

It is interesting to note that for the fixed point oscillation period, only the ESCCs of the STG are relevant. All other SCCs, including the TSCCs (unless they are, at the same time, ESCCs), do not influence the oscillation period T. (It should be noted, however, that the cycle lengths of nonentry SCCs determine how long it takes until the fixed point is reached.) This observation again points out the difference between a possibilistic state space analysis such as the characterization of the time series $R(t)$, and a probabilistic state space analysis [24], where the terminal SCCs play the important role in the analysis.

Theorem 6 justifies the formulation of a structural FSM traversal based on a time-frame expansion of an FSM. By considering the states that can be produced by the state vectors $\underline{s}_t$ of the iterative circuit array we are able to traverse the STG of the machine, visiting all states reachable from the set of initial states. For most practical systems (e.g., systems with synchronous resets or initializing sequences), the set of reachable states grows monotonically from time frame to time frame and the fixed point of the iteration consists of a single set, R_∞. For these systems, the number of iterations needed to reach the fixed point is only slightly larger (by the length of the initializing sequence) than in conventional FSM traversal. Note that in cases where this desirable property is not given by the considered FSMs, a fixed point oscillation period of 1 can be artificially enforced. By adding an auxiliary input that allows to reset the FSM to its initial state (or, alternatively, to let it hold its current state), the fixed point oscillation period is always 1.

B. Approximative Structural FSM Traversal

We now look at properties of approximative structural FSM traversal. Of course, the approximative algorithms are not complete, i.e., they will not give an answer for every problem instance. To obtain a better intuition of the circumstances under which a sequential equivalence checking algorithm based on approximative structural FSM traversal will terminate, we take a look at two corner cases. In the first case, the two circuits being compared only differ in the combinational logic, e.g., one has been derived from the other by logic optimization. The state encoding of both circuits is identical. In the second case, no logic optimizations have been performed. Instead, the circuits have been retimed. Interestingly, in both cases a sequential equivalence checking algorithm like record_and_play() will terminate.

In the first case, this is easy to see. Since only combinational optimizations have been performed, in each iteration there exists an equivalence partition of the set of registers, which is refined as the algorithm proceeds. Given enough reasoning power, we can always merge the logic feeding equivalent state variables. The stub circuit in each iteration consists of a set of fanout systems as in Fig. 8, which feeds equivalent state variables of the attached new time frame. After the fixed point is reached, all sequentially equivalent latches are "mapped" by the algorithm. So, in the case that only combinational optimizations have been performed, the algorithm implicitly performs a latch mapping while proving the equivalence of the circuit outputs.

In the second case, when no logic optimizations have been made and only retiming has been performed, the situation is a bit more complicated. For this case, we can state the following theorem.

Theorem 7: Let M be the product machine of two FSMs $M_{\text{impl}}, M_{\text{spec}}$, where M_{impl} is derived from M_{spec} by performing retiming.

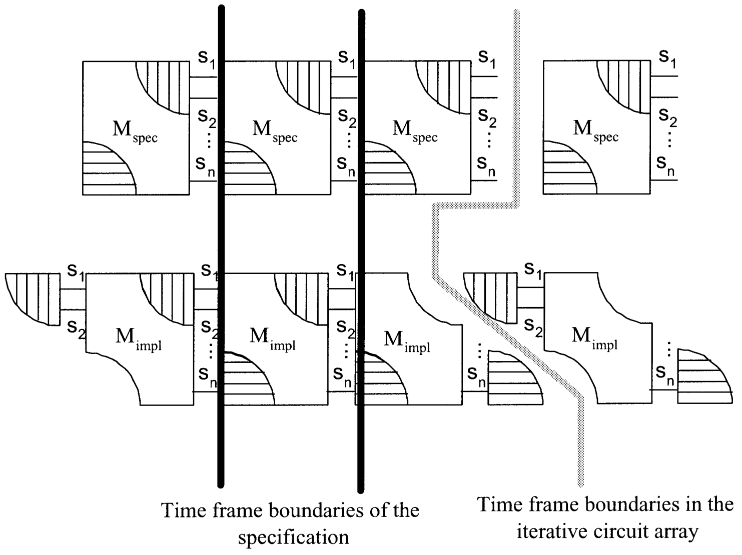

M spec
M impl
s1
s2
sn
Time frame boundaries of the specification
Time frame boundaries in the iterative circuit array

Fig. 18. Time-frame expansion of a product machine.

There is a $k > 0$ such that a fixed-point iteration on a set of equivalences on the time-frame expansion of M with length $k+1$ will end up with a set of equivalences that implies the equivalence of the outputs.

Proof: Let k be the maximum time difference between two signals in the implementation and let M_{k+1} be the time-frame expansion of length $k+1$.

It is clear that both, the implementation and the specification part of the time-frame expansion contain a full time-frame of the original machine M_{spec}. Fig. 18 shows the situation for $k = 2$.

The signals $s_i^{\text{spec}}, s_i^{\text{impl}}$ feeding this time frame are pairwise equivalent if M_{k+1} is constrained by the initial state. This is guaranteed by the initial state calculation of the retiming algorithm. Note that some of these pairs might even be constant.

The equivalences $s_i^{\text{spec}} \Leftrightarrow s_i^{\text{impl}}$ are invariant under the transition relation of M because attaching a new time frame to M_{k+1} will complete the next time frame of M_{spec} in both implementation and specification part.

Therefore, $s_i^{\text{spec}} \Leftrightarrow s_i^{\text{impl}}$ are in the fixed point of the iteration. Thus, the outputs rely on the same inputs and the same state variables $s_i^{\text{spec}} \Leftrightarrow s_i^{\text{impl}}$. As the logic for both is identical they are equivalent. ■

In the general case, both, combinational optimizations as well as retiming may have been applied. In this case, the signals that feed registers may disappear after retiming and further combinational optimizations (across the former register boundaries). In this case, in principle, false negatives may occur because there are not enough internal equivalences. Nevertheless, in practice most often there still exist pairs of equivalent signals such that the equivalence of the outputs can be proven.

It should be mentioned that verification of retiming without any Boolean optimizations is a much simpler problem than the general case and methods like [27] are applicable.

VI. Implementation Issues

When implementing structural FSM traversal, there are many possibilities regarding the representation of the stub circuit, the choice of transformations allowed and the way the network cut is chosen. In this section, we present a technical enhancement of the basic technique, discussed in Section III, that helped us considerably to improve the robustness of our method with respect to false negatives.

A. State-Set Representation

In the algorithm *Record* and *Play* presented in Section III, the decomposition and network cut of Fig. 3 are performed such that the set of states represented by the stub circuit after the cut is a conservative over-approximation of the set of reachable states. Such an approximation is possible if the two designs to be compared are structurally similar, i.e., it is possible to identify many logic implications between the internal logic functions of the designs. There are several possibilities to make use of the structural similarity of the designs being compared. One possibility is to perform general network transformations as they are also used for combinational logic optimization (e.g., [23]). Note that, in principle, any logic optimization technique can be used for this purpose. In practice, the most common technique is indeed to perform node substitutions based on functionally equivalent cut points as shown in the example in Section III. The instruction queue is a general data structure that records all transformations made. The network-cut frontier is selected such that in the transitive fanin of the nodes on the frontier all logic is merged.

A different possibility is to leave the circuitry unchanged. Instead of making node substitutions, equivalences between nodes of the two subcircuits of the circuit array are simply stored, in the same way as it is done in the combinational equivalence checker [4]. The equivalences can be stored directly at the nodes involved or in a special data structure. Note that for sequential equivalence checking also *constants* need to be stored. The reason is that the initial state is "injected" as an assignment of constant values to the state variables of the first time frame and then propagated by implication. In many cases the constants are not confined to the first time frame but may be propagated deep into the iterative circuit array.

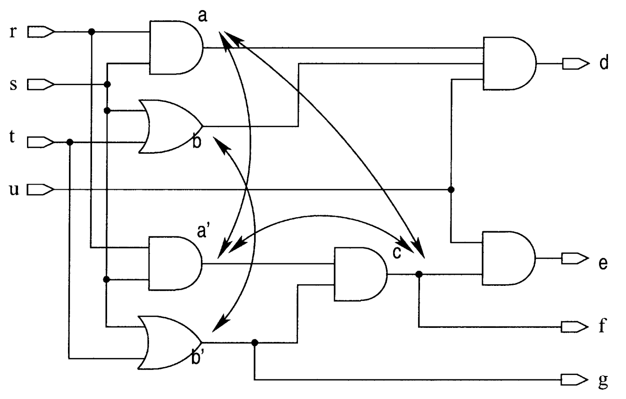


Fig. 19. Circuit with internal equivalences.

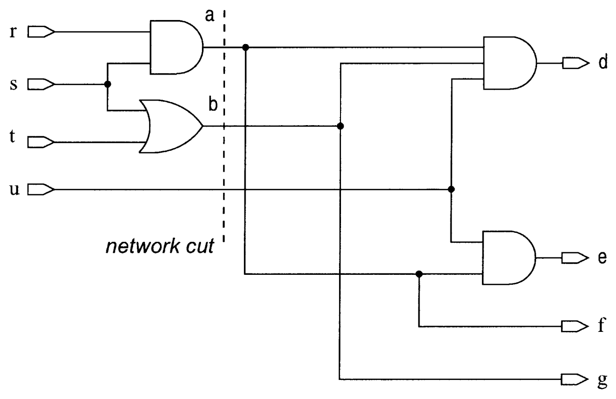


Fig. 20. After merging of logic: network cut not possible.

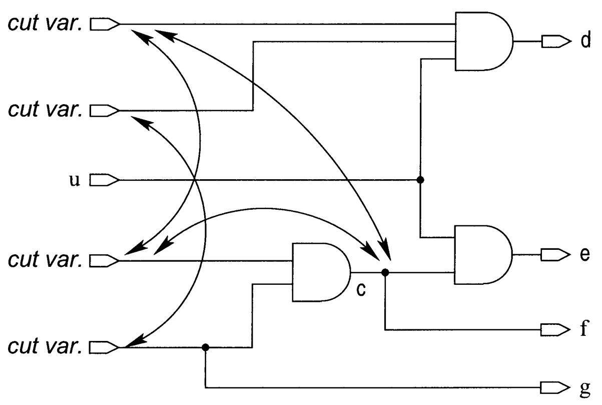


Fig. 21. Network with stored equivalences after cut.

When storing equivalences instead of making node substitutions the stub circuit of Fig. 3 consists of a piece of circuitry in which the logic functions are constrained by stored implications and constants. Note that the network cut associated with the decomposition of Fig. 3 can be carried out much in the same way as in the case for node substitutions. Instead of seeking a merge frontier, we determine by backward tracing a frontier of functionally equivalent points–cut points.

The stub circuit containing logic constraints "filters" input vectors that lead to conflicts according to the stored implication pointers. The implication techniques operating on this structure implicitly take the constraints into account by not only traversing the gate netlist but also the stored implication pointers when making logic value assignments. The stub circuit with stored implications (and constants) characterizes the set of reachable states in the same way as a stub circuit derived using synthesis transformations. The set of states is given by the state vectors that the stub circuit produces if all input vectors satisfying the stored constraints are applied.

Storing of implications has some advantages over making netlist transformations when using them in *Record & Play*. One advantage is that implications are valid independently of the order in which they are derived. In contrast, the validity of network transformations is dependent on the order in which they are carried out. Therefore, the instruction queue must reflect this order. When storing implications, the instruction *queue* can be replaced by order-independent data structures. Equivalences can, for example, be stored in a partition [13].

Another advantage of storing implications rather than using them for network transformations is that the stub circuit with stored constraints seems to be a more accurate representation of the set of reachable states than a merged circuit. In our experience, the likelihood of false negatives is significantly reduced. A simple example is shown in Figs. 19–21.

Fig. 19 shows some circuitry from an iterative circuit array with some internal equivalences indicated by arrows: signals $\{a, a', c\}$ and signals $\{b, b'\}$ are functionally equivalent, respectively. In addition, signals d and e are also equivalent. Let us suppose the task is to prove this equivalence after the network cut of the existential quantification step of Fig. 3.

If the equivalent logic is merged, i.e., signals a' and c are replaced by signal a and signal b' is replaced by b, we obtain the circuit of Fig. 20. Note that if a network cut was made as indicated by the dashed line, signals d and e can no longer be proven to be equivalent. A false negative would occur.

If, however, the equivalences shown in the original circuit are stored as constraints and the network cut is performed such that all equivalent signals are kept, we obtain the stub circuit of Fig. 21. Note that now the signals d and e can still be determined to be equivalent functions.

B. Constraint-Satisfying Logic Simulation

When implementing *Record & Play* with a stub circuit based on stored equivalence constraints instead of network transformations, we encounter an important problem. In a practical implementation, it is crucial for the performance of the algorithm that the detection of the individual equivalence relationships of the internal logic functions in the iterative circuit array is carried out fast. If using recursive learning, direct, and indirect implications are derived as in [4] in order to find all equivalent signals of a node, large recursion depths may be needed, especially if the designs have been heavily optimized. A significant speed-up can be obtained if we are given potential equivalent candidate signals that only need to be checked for equivalence. In this approach, pairs of contradictory values are assigned to both circuit nodes and their logic inconsistency is then proven using recursive learning or any SAT solver. For detecting this logic inconsistency fewer recursion levels are needed, significantly reducing CPU time.

Usually, equivalence candidates are determined in a preprocessing step by random pattern simulation. In our case, where we have a stub circuit with additional constraints given as stored equivalences between nodes of the circuit, a standard simulation approach is not possible, since most random vectors, when simulated, will violate the constraints and can therefore not be used to distinguish inequivalent signals. In this section, we describe a method to find equivalence candidates in a circuit with stored constraints using so-called *constraint-satisfying logic simulation* [28].

There are two possible straightforward approaches.

1) The first is to generate random vectors and check for each of them if it satisfies the constraints. However, in this approach the probability to find a random vector that satisfies all the constraints is very low, so that the simulation times become prohibitively long.
2) The second approach assigns the values at the PIs one after another. It backtracks when necessary because of the constraints. We have implemented this approach and found that it is effective. But it is quite slow because of the large number of backtracks needed. Also, the resulting classes are not small enough. This is a result of the fact that the backtracking often leads to similar assignments.

Note that the first approach suffers from the problem that the constraints in the stub circuit guide the backtracking procedure such that input patterns are generated that comply with the constraints, however, often they do not split the equivalence classes. Therefore, additional concepts are needed. Consider the class of constant nodes, for example. This class can be very large, since the initial state is injected as a set of constant values to the state variables of the first time frame. These constants can propagate several time frames, and the corresponding equivalence class obtained from simulation can become extremely large, because many signals in the transitive fanout of the constant nodes have a high probability of "sticking" to one of the two logic values. We can split the class if we generate patterns that justify the opposite value of one of the nodes in the class. Such an approach is proposed in [29].

A more general heuristic is to inject a value v at internal nodes n and justify this from the PIs. In the case of the constant class, we take the opposite value and a random value otherwise. The resulting input pattern is simulated and the results of the simulation are used to split the equivalence classes. We have implemented this heuristic in procedure generate_vectors() that is called by contraint_satisfying_vectors() as shown in Tables IV and V.

Since the constraints are not equally distributed in the circuit, but are located near the PIs, it is not necessary to probe values at every node. In order to generate a diversity of signal values within the circuit such that the equivalence classes can be split, we choose a cut through the circuit (see Fig. 22).

We inject a random value at every node of this cut. Generally, after this first phase, still a number of large classes remain. Especially the constant class still contains many false candidates. Therefore, in the second phase, we take the large classes in the partition and justify a random value at every node in every large class. In the constant class, we try to justify the opposite value for every node. Fig. 22 shows the gates selected by the algorithm.

TABLE IV
CONSTRAINT-SATISFYING SIMULATION ALGORITHM

```
constraint_satisfying_simulation( cir,constraints) {
  eq := {{ n | n is node of cir }}
   /*First phase*/
  cut =find_cut( cir,CUT_LEVEL)
  generate_vectors(cir,constraints,eq,cut)
   /*Second phase*/
  for each(large class class ∈ eq)
    generate_vectors(cir,constraints,eq,class)
  return eq;
}
```

TABLE V
ALGORITHM FOR GENERATING AND CLASS-REFINING INPUT VECTORS

```
generate_vectors( cir,constraints,eq,node_set) {
  for each( node ∈ node_set) {
    value:=random();
    objectives:=
      create_init_objectives(node,value);
    pi_objectives:=
      constraint_aware_multiple_backtrace
        (cir,constraint,objectives);
    values:= set_pi_values
        (cir,constraints,pi_objectives);
    refine(eq,values);
  }}
```

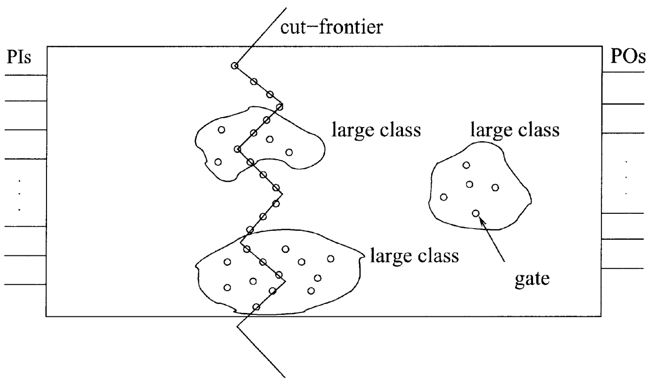


Fig. 22. Gates selected to justify a random value.

To avoid backtracks during the justification of a random value, we perform a multiple backtrace routine [30] that has been adapted to take into account logic constraints. This will be described in more detail below.

The result of the multiple backtrace procedure is a set of objectives [30] at the PIs, called PI objectives. They suggest value assignments at the PIs called *PI values* in the following.

To find an input vector that justifies the value at the selected node we use a multiple backtrace procedure given in Table VI. This procedure is adopted from the well-known ATPG algorithm FAN [30]. FAN tries to leave the conflicting area of the search space as soon as possible. This is done by tracing possible value assignments called objectives back to fanout stems. An objective is defined as follows.

Definition 6 (Objective): An *objective* is a triplet (s, z, o) where s is a signal, z is the number of times the value 0 is required at s and o is the number of times the value 1 is required at s.

TABLE VI
CONSTRAINT-AWARE MULTIPLE BACKTRACE

```
constraint_aware_multiple_backtrace
( cir, constraints, objectives) {
  while (objectives $\neq \emptyset$) {
    $(n, z, o)$ := choose_objective (objectives);
    objectives := objectives - $\{(n, z, o)\}$;
    for each ($f \in$ fanin($n$)) {
      $(z_f, o_f)$ := propagate($f, z, o$);
      if (!is_pi ($f$)) {
        objectives :=
        objectives $\cup \{(f, z_f, o_f)\}$;
      }
      else {
        pi_objectives :=
        pi_objectives $\cup \{(f, z_f, o_f)\}$;
    } }
    for each ($f \in$ equivalences($n$)) {
      $(z_f, o_f)$ := get_objective($f$);
      if (are_compatible($(n, z, o), (f, z_f, o_f)$)) {
        synchronize($(n, z, o), (f, z_f, o_f)$);
        if (!is_pi ($f$)) {
          objectives :=
          objectives $\cup \{(f, z_f, o_f)\}$;
        }
        else {
          pi_objectives :=
          pi_objectives $\cup \{(f, z_f, o_f)\}$;
  } } } }
  return (pi_objectives);
}
```

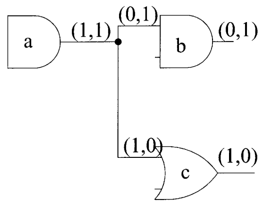


Fig. 23. Example for conflict detection in FAN.

For simplicity, we identify gates with their fanout lines. In Fig. 23, for example, we visualize the objective $(b, 0, 1)$ by annotating $(0, 1)$ on the output line of the gate b.

FAN traces objectives simultaneously through different branches of a fanout system. In this way, FAN is able to detect possible conflicts early. For example in the situation of Fig. 23, FAN would attempt to justify a value at the fanout node a.

Unlike FAN, we do not stop at internal fanout points but backtrace the objectives to the PIs. For every stored constant, we have an initial objective $(c, 0, 1)$ or $(c, 1, 0)$, depending on whether it is a constant 0 or 1. Justifying the value 1 at a signal n in the circuit generates the initial objective $(n, 0, 1)$, for example. The set of initial objectives is calculated in the procedure create_init_objectives() that is called in routine generate_vectors() of Table V.

In the backtrace procedure (Table VI), we use both, the signals and the stored equivalences to find new objectives. If the node of the current objective has any stored equivalences the objectives of the equivalent nodes should have the same value. As can be seen in Fig. 24, this can lead to cyclic dependencies.

It is also clear that overwriting the objective of the equivalent node leads to cyclic traces. These traces must be detected

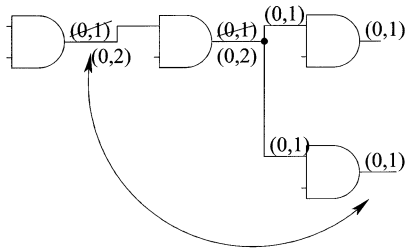


Fig. 24. Example for cyclic dependencies of objectives.

to make the algorithm terminate. Therefore, we introduce the following notion.

Definition 7 (Compatible Objectives): Let s, t be nodes of a given circuit. Two objectives $\alpha = (s, z_s, o_s), \beta = (t, z_t, o_t)$ for the nodes s and t are called *compatible*, iff $z_s = 0 \Leftrightarrow z_t = 0$ and $o_s = 0 \Leftrightarrow o_t = 0$.

Using this notion, the problem of cyclic traces can be resolved in the following way. For a given node s we only change the objective of an equivalent node t if s and t are not compatible. This ensures that a cycle caused by equivalences is traversed no more than twice so that the backtrace routine is guaranteed to terminate. Note that still all conflicting requirements are detected.

For backtracking algorithms, it has been shown to be a good heuristic to assign variables with conflicting requirements as early as possible. In our case, we start with assigning values to those PIs that have an objective (p, z_p, o_p) with both $z_p > 0$ and $o_p > 0$. We choose 0 if $z_p > o_p$ and 1 otherwise. Next, the unassigned PIs with $z_p > 0$ or $z_p > 0$ are assigned a value 0 if $z_p > 0$ and 1 otherwise. Finally, the remaining PIs obtain a random value. If inconsistencies are detected by the implication engine, we have to backtrack accordingly.

VII. EXPERIMENTAL RESULTS

We implemented *Record & Play*, including the constraint-satisfying simulation approach described in Section VI-B and integrated it into the HANNIBAL equivalence checking environment [4]. In our implementation, first, constraint-satisfying simulation is used to identify candidates for internal equivalences. Then, recursive learning is used to prove or disprove the equivalence of the candidates.

Experiments were run on the ISCAS'89 benchmark set. All circuits were optimized using SIS with Boolean optimization by *script.rugged*. Sequential transformations were done using retiming. Unfortunately, we could not generate optimized and retimed versions of the larger ISCAS'89 benchmarks since we were unable to run SIS *script.rugged* on these circuits. Table VII shows the results for verifying each circuit against its original version. The second column shows the needed maximum level of recursion for recursive learning. The next column shows the number of time frames needed to reach the fixed point. The last two columns give the total CPU time used and the amount of time spent in simulation, respectively. The experiments were carried out on a Linux PC with an AMD Athlon processor clocked at 1.4 GHz.

We note that our approach is also successful in cases like S1423, where an exact reachability analysis would require a huge number of iterations to reach the fixed point. Remember

TABLE VII
RESULTS FOR ISCAS'89 BENCHMARKS OPTIMIZED WITH SIS (*script.rugged*)

circuit	rec. depth	# time frames	CPU time (hh::mm:ss) total	simulation
S208	4	20	00:00:01	00:00:00
S298	4	15	00:00:02	00:00:01
S344	6	10	00:00:01	00:00:00
S349	6	10	00:00:02	00:00:01
S382	4	24	00:00:05	00:00:03
S386	4	11	00:00:02	00:00:01
S420	5	38	00:00:06	00:00:04
S444	4	25	00:00:05	00:00:03
S499	4	26	00:00:06	00:00:04
S510	5	22	00:05:15	00:02:30
S526	4	28	00:00:12	00:00:05
S635	4	70	00:00:20	00:00:12
S641	4	9	00:00:01	00:00:00
S713	4	9	00:00:01	00:00:01
S820	6	10	00:00:27	00:00:23
S832	4	13	00:03:14	00:01:12
S838	4	74	00:00:58	00:00:40
S938	4	74	00:00:58	00:00:40
S953	5	14	00:02:47	00:01:05
S967	4	10	00:00:26	00:00:22
S1196	4	7	00:00:39	00:00:02
S1238	6	7	00:00:04	00:00:01
S1423	4	18	00:00:22	00:00:13
S1488	6	9	00:00:51	00:00:28
S1494	6	9	00:00:55	00:00:32
S1512	6	17	00:00:41	00:00:28
S3271	3	20	03:23:54	00:37:12
S3330	4	11	00:03:32	00:03:14
S4863	3	6	00:02:23	00:02:10
S5378	3	36	11:28:51	02:19:38
S6669	3	12	00:42:03	00:21:55

TABLE VIII
RESULTS FROM AN INDUSTRIAL IMPLEMENTATION OF STRUCTURAL FSM TRAVERSAL

circuit	CPU time (mm:ss)
S208	00:01
S298	00:01
S344	00:02
S349	00:02
S382	00:02
S386	00:01
S420	00:01
S444	00:02
S499	00:08
S510	00:08
S526	00:10
S635	00:01
S641	00:01
S713	00:01
S820	00:17
S832	00:16
S838	00:01
S938	00:01
S953	00:22
S967	00:31
S1196	00:01
S1238	00:01
S1423	00:15
S1488	00:31
S1494	00:30
S1512	00:40
S3271	01:32
S3330	00:36
S3384	03:27
S4863	00:07
S5378	02:38
S6669	00:32
S9234	00:56

that our approach over-approximates the state space, thereby drastically improving the convergence behavior in cases like S1423.

In our approach we can effectively exploit structural similarities in the same way as in combinational equivalence checking. This is not possible in techniques based on symbolic FSM traversal, or *Record* & *Play*-like techniques that prove equivalences between cut points constructing monolithic BDDs. For example, the method in [13] is not able to verify some circuits of this benchmark set because of BDD explosion.

The proposed approach has also been adapted for integration into the Siemens/Infineon CVE environment. The industrial implementation makes use of a more sophisticated flow of powerful Boolean engines and heavily exploits constraint-satisfying simulation. Table VIII shows the results for the CVE implementation. The same benchmarks as in Table VII were used, i.e., the shown circuits were checked for sequential equivalence with their optimized (*script.rugged*) and retimed versions. It can be seen that the CVE implementation scales very well with growing design sizes. All equivalence checks required less than 4 min of CPU time.

VIII. CONCLUSION

In this paper, we discussed a sequential verification technique called *structural FSM traversal*. We have shown that time-frame expansion, together with a decomposition procedure, is an efficient representation of the state sets reachable in a forward FSM traversal. The key to make this approach effective is to use similarities between the designs under verification. This can be done by storing internal equivalences and performing network cuts. The presented theory analyzes the properties of exact as well as approximative structural FSM-traversal. Our experimental results clearly show that structural FSM traversal is a robust and highly performant approach to sequential equivalence checking of optimized and retimed circuits, and is ready for industrial application.

REFERENCES

[1] O. Coudert, C. Berthet, and J.-C. Madre, "Verification of synchronous sequential machines based on symbolic execution," *Lecture Notes Comput. Sci.*, vol. 407, pp. 365–373, June 1989.
[2] K. McMillan, *Symbolic Model Checking*. Norwell, MA: Kluwer, 1993.
[3] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C–35, pp. 677–691, Aug. 1986.
[4] W. Kunz, "An efficient tool for logic verification based on recursive learning," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 538–543.
[5] D. Brand, "Verification of large synthesized designs," in *Proc. Int. Conf. Computer-Aided Design*, 1993, pp. 534–537.
[6] S. Reddy, W. Kunz, and D. Pradhan, "A novel verification framework combining structural and OBDD methods in a synthesis environment," in *Proc. Design Automation Conf.*, June 1995, pp. 414–419.
[7] J. Jain, R. Mukherjee, and M. Fujita, "Advanced verification techniques based on learning," in *Proc. 32nd ACM/IEEE Design Automation Conf.*, June 1995, pp. 420–426.

[8] Y. Matsunaga, "An efficient equivalence checker for combinational circuits," in *Proc. Design Automation Conf.*, June 1996, pp. 629–634.

[9] A. Kühlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc. Design Automation Conf.*, Nov. 1997, pp. 263–268.

[10] D. Paul, M. Chatterjee, and D. K. Pradhan, "VERELAT: Verification using logic augmentation and transformation," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1041–1051, Sept. 2000.

[11] D. Stoffel and W. Kunz, "Record & play: A structural fixed point iteration for sequential circuit verification," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1997, pp. 394–399.

[12] S. Huang, K. Cheng, K. Chen, and U. Gläser, "An ATPG-based framework for verifying sequential equivalence," in *Proc. Intl. Test Conf.*, 1996.

[13] C. van Eijk, "Sequential equivalence checking without state space traversal," in *Proc. Conf. Design, Automation Test Eur.*, Paris, France, Mar. 1998, pp. 618–623.

[14] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Proc. Int. Conf. Formal Methods Computer-Aided Design*, Nov. 2000, pp. 372–389.

[15] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. (1999) Symbolic model checking without BDDs. *Lecture Notes Comput. Sci.* [Online], pp. 193–207. Available: http://citeseer.nj.nec.com/article/biere99symbolic.html

[16] M. Mneimneh and K. Sakallah, "SAT-based sequential depth computation," presented at the 1st Int. Workshop Constraints Formal Verification, Ithaca, NY, Sept. 2002.

[17] J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property checking via structural analysis," in *Proc. 14th Int. Conf. Computer-Aided Verification*, vol. 2404. Copenhagen, Denmark, May 2002, pp. 151–165.

[18] D. Stoffel, "Formal verification of sequential circuits using reasoning techniques," Ph.D. dissertation, Johann Wolfgang Goethe-Universität, Frankfurt am Main, Germany, 1999, http://www.eis.eit.uni-kl.de, submitted for publication.

[19] C. Pixley, "A theory and implementation of sequential hardware equivalence," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 1469–1478, Dec. 1992.

[20] C. Pixley, V. Singhal, A. Aziz, and R. K. Brayton, "Multi-level synthesis for safe-replaceability," in *Proc. Int. Conf. Computer-Aided Design*, 1994, pp. 442–449.

[21] S. Huang, K. Cheng, K. Chen, and U. Gläser, "On verifying the correctness of retimed circuits," in *Proc. Great Lakes Symp. VLSI*, 1996, pp. 277–280.

[22] S. Huang, K. Cheng, K. Chen, C. Huang, and F. Brewer, "AQUILA: An equivalence checking system for large sequential designs," in *IEEE Trans. Comput.*, May 2000, vol. 49, pp. 443–464.

[23] W. Kunz and F. Somenzi, *Reasoning in Boolean Networks—Logic Synthesis and Verification Using Testing Techniques*. Norwell, MA: Kluwer, 1997.

[24] G. D. Hachtel, D. Stoffel, A. Pardo, and F. Somenzi, "Markovian analysis of large finite state machines," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1479–1493, Dec. 1996.

[25] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*. New York: Springer-Verlag, 1976.

[26] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Norwell, MA: Kluwer, 1996.

[27] M. Mneimneh and K. Sakallah, "REVERSE: Efficient sequential verification for retiming," in *Proc. IEEE/ACM Int. Workshop Logic Synthesis*, Laguna Beach, CA, May 2003, pp. 133–139.

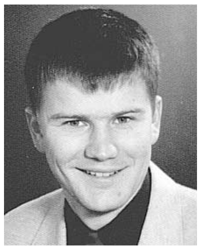
[28] M. Wedler, D. Stoffel, and W. Kunz, "Improving structural FSM traversal by constraint-satisfying logic simulation," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, Pittsburgh, PA, Apr. 2002, pp. 151–158.

[29] M. K. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing simulation with BDDs and ATPG," in *Proc. Design Automation Conf.*, 1999, pp. 385–390.

[30] H. Fujiwara and T. Shimono, "FAN: A fanout-oriented test pattern generation algorithm," in *Proc. IEEE Int. Symp. Circuits Syst.*, June 1985, pp. 671–674.

Dominik Stoffel (M'95) obtained the Diplom-Ingenieur degree in electrical engineering from the University of Karlsruhe, Karlsruhe, Germany, in 1992 and the Ph.D. degree in computer science from the University of Frankfurt, Frankfurt, Germany, in 1999.

From 1994 to 1998, he was with the Max-Planck Fault-Tolerant Computing Group, Potsdam, Germany. From 1998 to 2001, he was with the Electronic Design Automation Group, University of Frankfurt. Since 2001, he has been a Postdoctoral Researcher in the Electronic Design Automation Group, University of Kaiserslautern, Kaiserslautern, Germany. His research interests are in the field of formal hardware verification and logic synthesis.



Markus Wedler received the Dipl.-Math. degree in mathematics from the University of Hagen, Hagen, Germany, in 1999. He is currently working toward a Ph.D. in electrical and computer engineering at the University of Kaiserslautern, Kaiserslautern, Germany.

He conducts research in formal hardware verification, especially formal property verification for datapath and sequential circuits.

Peter Warkentin received the Ph.D. degree in mathematics from the University of Freiburg, Freiburg, Germany, in 1982.

Since 1984, he has been employed with Siemens AG, Munich, Germany. After joining the Research and Development Department, the focus of his work became logic programming. Since 1990, he has been working in formal verification. His major contributions are in the development of the BDD-based model checker SVE and the verification environment CVE of Siemens/Infineon.

Wolfgang Kunz (S'90–M'91) obtained the Dipl.Ing. degree of electrical engineering from University of Karlsruhe, Karlsruhe, Germany, in 1989 and the Ph.D. degree from the University of Hannover, Hannover, Germany, in 1992.

From 1993 to 1998, he was with Max Planck Fault-Tolerant Computing Group, University of Potsdam, Potsdam, Germany. From 1998 to 2001, he was a Professor in the Computer Science Department, University of Frankfurt, Frankfurt, Germany. Since 2001, he has been with the Electrical Engineering Department, University of Kaiserslautern, Kaiserslautern, Germany. He conducts research in the areas of logic and layout synthesis, equivalence checking, and ATPG.

Prof. Kunz has received several awards including the IEEE Transactions on Computer-Aided Design Best Paper Award.