Detection of Equivalent State Variables in Finite State Machine Verification

C.A.J. van Eijk and J.A.G. Jess
Eindhoven University of Technology, Department of Electrical Engineering
Design Automation Section, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Email: C.A.J.v.Eijk@ele.tue.nl and J.A.G.Jess@ele.tue.nl

Abstract

This paper proposes a new technique to detect equivalent state variables for finite state machine verification. The technique can easily be integrated with existing verification methods and significantly extends the ability of these methods to handle finite state machines (FSMs) with similar state encodings. This is of practical importance as it enlarges the class of FSMs for which verification is feasible. The proposed technique can also be used to verify combinational equivalence of FSMs with unknown correspondence between the state variables. The effectiveness of the technique is shown by experimental results on well-known benchmarks with up to 1426 state variables.

1 Introduction

This paper addresses the problem of verifying the equivalence of finite state machines. Impressive progress has been made in this area by the introduction of so-called symbolic techniques, which are based on the application of binary decision diagrams (BDDs) to traverse the state space (see e.g. [2][3][5][14]). Although these methods can conceivably handle large FSMs with current BDD-based implementation techniques, the biggest problem faced is still that of scale. The symbolic methods that have been proposed are based on an implicit traversal of the state space. A major drawback is their inability to exploit any similarities of the FSMs under comparison. These similarities can for example consist of functionally equivalent state variables in the two FSMs. Because many design steps do not completely change the state encoding of a design, such similarities are likely to occur in practice.

This paper proposes a fully automatic technique to detect equivalent state variables before the reachable state space of the product machine is calculated, and discusses how it can be integrated with existing verification methods. The proposed technique does not require extra information from the designer and forms a robust extension of existing verification methods. As will be shown in section 2, the removal of equivalent state variables always results in a smaller BDD representation for the reachable state space which is also less sensitive to the selected variable order. Because the detection technique does not rely on names to identify equivalences between state variables, it also works in cases where different names are used in the specification and the implementation. This can for example be the case when the correctness of a transistor-level implementation has to be proven against a high-level design specification (see e.g. [7][8]). A similar problem may also occur when both descriptions have different hierarchical structures, because then equivalent state variables may have different hierarchical names. In some other cases the designer may not know which state variables have been modified. For these reasons it is important to have a verification method that can detect such equivalences automatically.

This paper is organized as follows. Section 2 first introduces some basic definitions and discusses symbolic verification methods for finite state machines. Then it explains how a technique that exploits equivalent state variables can be integrated with such methods. Section 3 discusses how equivalent variables can be detected automatically with a minimum of extra effort. Some experimental results are presented in section 4.

2 Finite State Machine Verification

Definition 1 introduces the concept of a finite state machine. It slightly differs from the usual definition in that it uses a set of state variables to define the notion of state instead of a set of states; this will enable a more compact presentation of the technique to detect equivalent state variables. The set of boolean values is denoted by $B = \{0, 1\}$.

Definition 1

A finite state machine M is a 6-tuple $(I, O, V, S_0, \Delta, \Lambda)$, where

- $I = (i_1, \ldots, i_m)$ is an ordered set of inputs;
- $O = (o_1, \ldots, o_p)$ is an ordered set of outputs;
- $V = (v_1, \ldots, v_n)$ is an ordered set of state variables;
- $S_0 \subseteq B^n$ is the non-empty set of initial states;
- $\Delta : B^n \times B^m \rightarrow B^n = (\delta_1, \ldots, \delta_n)$ is the next-state function;
- $\Lambda : B^n \times B^m \rightarrow B^p = (\lambda_1, \ldots, \lambda_p)$ is the output function.

A state of a finite state machine is a value assignment to its state variables. Because V is an ordered set, this assignment is simply represented by a boolean vector of length $|V|$, i.e., the state space of M is B^n. The reachable state space is the set of states which can be reached in zero or more steps from some initial state and is denoted $Reach(M)$.

If two finite state machines have identical sets of inputs and outputs, it is possible to compare their behaviour. Then two FSMs are called functionally equivalent iff they produce the same sequence of output vectors for any sequence of input vectors. To formalize this requirement, the so-called product machine of two FSMs is defined.

Definition 2

Given two FSMs $M_A = (I, O, V_A, S_{A,0}, \Delta_A, \Lambda_A)$ and $M_B = (I, O, V_B, S_{B,0}, \Delta_B, \Lambda_B)$. The product machine $M_A \times M_B = (I, O, V, S_0, \Delta, \Lambda)$ is defined by

- $V = (v_{A,1}, \ldots, v_{A,n_A}, v_{B,1}, \ldots, v_{B,n_B})$,
- $S_0 = \{(s_1, \ldots, s_{n_A+n_B}) \mid (s_1, \ldots, s_{n_A}) \in S_{A,0}, (s_{n_A+1}, \ldots, s_{n_A+n_B}) \in S_{B,0}\}$,
- $\Delta(\underline{s}_A, \underline{s}_B, \underline{i}) = (\delta_{A,1}(\underline{s}_A, \underline{i}), \ldots, \delta_{A,n_A}(\underline{s}_A, \underline{i}), \delta_{B,1}(\underline{s}_B, \underline{i}), \ldots, \delta_{B,n_B}(\underline{s}_B, \underline{i}))$,
- $\Lambda(\underline{s}_A, \underline{s}_B, \underline{i}) = (\lambda_{A,1}(\underline{s}_A, \underline{i}) \equiv \lambda_{B,1}(\underline{s}_B, \underline{i}), \ldots, \lambda_{A,p}(\underline{s}_A, \underline{i}) \equiv \lambda_{B,p}(\underline{s}_B, \underline{i}))$.

Equivalence of FSMs can now be formulated as follows: two FSMs are functionally equivalent iff every output of the corresponding product machine is assigned the value 1 in all its reachable states. In the sequel of this paper, $M_A = (I, O, V_A, S_{A,0}, \Delta_A, \Lambda_A)$ and $M_B = (I, O, V_B, S_{B,0}, \Delta_B, \Lambda_B)$ represent the two FSMs which are verified. The corresponding product machine is represented by $M = (I, O, V, S_0, \Delta, \Lambda)$.

Sequential verification methods typically proceed as follows. They start with two FSMs which e.g. have been extracted from circuit descriptions. The product machine is constructed and then its state space is traversed. Figure 1 shows an example of a symbolic algorithm which can be used for this traversal. The algorithm is based on a breadth-first calculation of the reachable state space. For every intermediate set of reached states it is tested whether both machines are still equivalent. In an implementation of algorithm 1, all sets and functions are represented by BDDs. The most important operation in the algorithm is the calculation of the *image* of a set of states; this is the set of states which can be reached in a single step from the given set of states. Several techniques have been proposed to implement this calculation efficiently (see e.g. [3][4][5][14]).

$reached := \emptyset$; $frontier := S_0$;
do {
  if ($\exists \underline{s} \in frontier, \underline{i} \in B^m : \Lambda(\underline{s}, \underline{i}) \neq (1, \ldots, 1)$)
    generate counter-example and stop;
  $reached := reached \cup frontier$;
  $frontier :=$ Image($\Delta, frontier$) $\setminus$ $reached$;
} while ($frontier \neq \emptyset$) ;

Fig. 1. Outline of a symbolic algorithm for state space traversal

Although symbolic algorithms can conceivably handle large FSMs with current BDD-based implementation techniques, sequential verification is still not feasible for many FSMs of practical size. The major limitations are the sizes of the BDD representations for the next-state function and the set of reached states, and the required number of iterations. Therefore new techniques are needed to increase the feasibility of sequential verification. This can be done by developing still more efficient state space calculation techniques. This paper follows a complementary approach which is based on the application of a reduction technique.

The objective of reduction techniques is to minimize the size of the verification problem without changing the outcome of the problem. This can be done in several ways. In [4], Cabodi et al. introduce the so-called general product machine; some explicitly known relation between the state encodings of both machines is exploited to construct a good encoding for the state space of this product machine; no methods are proposed to derive the required relation between the state encodings automatically. In this paper we propose a technique which exploits equivalent state variables. This idea has been applied independently in the industrial verification system CVE developed at Siemens [12], where it is called *state bit identification*, and in the sequential verification algorithm of the industrial synthesis system TIGER developed by Coudert, Madre and Touati [6]. In their algorithm, equivalent (and opposite) variables are detected on the fly during the state space traversal.

The next section describes a technique to detect equivalent state variables in the product machine before the state space is traversed; it can be used as a preprocessing step for a symbolic traversal algorithm. This results in a verification method as depicted in figure 2. The basic idea is to detect and prove the equivalence of state variables before the state space is traversed. Whenever it is proven that two state variables have equal values in all reachable states, both variables can be interchanged without affecting the behaviour of the product machine. Therefore, one of them can be removed from the machine by substitution. The proposed technique does not have the same computational limitations as a symbolic traversal algorithm, because it is not based on a state space traversal. It forms an effective preprocessing step for such an algorithm, because it can be implemented efficiently. It can however not be guaranteed that all equivalent variables are detected.

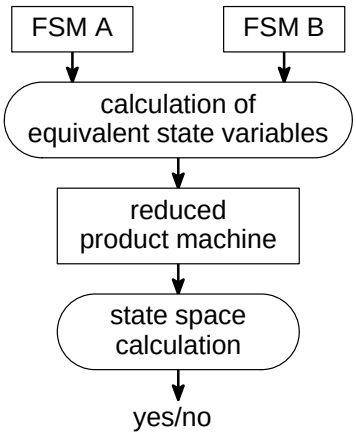

Fig. 2. Outline of the verification method

Although the proposed reduction technique is relatively simple, it can lead to significant improvements. The advantages of removing equivalent state variables are twofold. First of all, it may cause the outputs of both circuits to be expressed in the same state variables. This may provide enough information to already conclude the equivalence of some or all of the outputs. The second advantage is of course that every variable that is removed effectively reduces the size of the product machine. To further explain the corresponding advantages, let's consider the case where we have to verify two circuits with exactly the same state encoding. Then the state space of the product machine can be written as:

$$Reach(M) = Reach(M_A) \wedge (v_{A,1} \equiv v_{B,1} \wedge \ldots \wedge v_{A,n_A} \equiv v_{B,n_B}) .$$

A reduction of at least a factor two [4] can be achieved for the BDD representation of the reachable state space if the equivalent state variables are detected and removed before the state space is traversed. The condition which expresses the equivalence of the state variables is very sensitive to the variable order that is used; it only has a compact representation if equivalent variables are grouped together. Therefore the removal of equivalent variables also leads to a more robust representation which is less sensitive to the selected variable order. Of course, dynamic variable ordering can be used to automatically maintain the variable order. Especially the sifting algorithm has proven to provide a good balance between the quality of the variable order and the overhead in run-time [13]. Also when this algorithm is used, the removal of equivalent variables leads to a more robust representations, because the pairs of equivalent (or in other words symmetric) variables cause the sifting algorithm to easily get stuck in a local minimum; equivalent variables are typically grouped together but the optimum position of this group may not be found without sifting the equivalent variables together [11]. It is difficult for a variable ordering algorithm to exploit the dependencies between equivalent variables automatically, because equivalent state variables are only symmetric in the BDD for the reachable state space and not in the BDDs for the next-state and output functions.

3 Detection of Equivalent State Variables

This section describes the technique to detect functionally equivalent state variables in a finite state machine. It partitions the set of state variables into classes of functionally equivalent variables. More exactly, it calculates an equivalence relation on the set of state variables which induces this partition. This particular equivalence relation is called the variable correspondence relation. To guarantee its correctness, two conditions are imposed on it which together form an inductive argument for the equivalence of the variables in the same class of the partition. The first condition requires that equivalent variables always have the same initial value. The second condition requires that if equivalent variables have the same value in the current state, they necessarily have the same value in every next state. If these two conditions are satisfied, it can directly be concluded that all equivalent variables necessarily have the same value in every reachable state. To express the condition that a state conforms to a given relation, the variable correspondence condition is introduced.

Definition 3

Given an equivalence relation $R : V \times V \rightarrow B$. Then the variable correspondence condition $R_{VC} : B^n \rightarrow B$ is the predicate that defines whether a state conforms to R, i.e., whether equivalent variables are indeed assigned the same value in this state:

$$R_{VC}(s_1, \ldots, s_n) = (\forall\, v_j, v_k \in V : R(v_j, v_k) \rightarrow s_j \equiv s_k) .$$

The variable correspondence relation can now be defined as follows.

Definition 4

An equivalence relation $R : V \times V \rightarrow B$ is a variable correspondence relation iff it satisfies the following two conditions:

- it holds in every initial state: $\forall\, \underline{s} \in S_0 : R_{VC}(\underline{s})$,
- it is invariant under the next-state function: $\forall\, \underline{s} \in B^n, \underline{i} \in B^m : R_{VC}(\underline{s}) \rightarrow R_{VC}(\Delta(\underline{s}, \underline{i}))$.

Note that the second condition in definition 4 is sufficient but not necessary for the equivalence of two variables; it is chosen specifically because its evaluation does not require the reachable state space. As a consequence also non-reachable states may be taken into account by this condition, and therefore it is not guaranteed that all equivalent variables are indeed detected.

An important question for the applicability of the variable correspondence relation is whether there always exists a unique solution. There may exist several variable correspondence relations for a FSM. The following property shows that two such relations can always be combined to a single larger relation. Because there is only a finite number of state variables, this means that there always exists a unique maximum variable correspondence relation.

Property 5

If $R_A, R_B : V \times V \rightarrow B$ are both variable correspondence relations, then the relation $R_{A \vee B} : V \times V \rightarrow B$ defined by

$$R_{A \vee B}(v_j, v_k) = R_A(v_j, v_k) \vee R_B(v_j, v_k) .$$

is also a variable correspondence relation.

The method to calculate the maximum variable correspondence relation follows directly from definition 4. It consists of a greatest fixed point computation. The first approximation R_0 is based on the first condition of definition 4:

$$R_0(v_j, v_k) = (\forall (s_1, \ldots, s_n) \in S_0 : s_j \equiv s_k) .$$

Starting with R_0, a series of approximations R_h can be calculated by applying the second condition of definition 4:

$$R_{h+1}(v_j, v_k) = R_h(v_j, v_k) \wedge (\forall \underline{s} \in B^n, \underline{i} \in B^m : R_{h,VC}(\underline{s}) \rightarrow \delta_j(\underline{s}, \underline{i}) \equiv \delta_k(\underline{s}, \underline{i})) .$$

Since there is only a finite number of state variables, a fixed point is reached after a finite number of iterations, i.e., at some point $R_h = R_{h+1}$. Then this R_h is by construction the maximum variable correspondence relation. The maximum number of iterations is $|V| + 1$, because in every iteration, except the last one, at least one new class is created.

Every R_h is an equivalence relation. Therefore it can be represented by the partition it induces. This partition is stored explicitly. The refinement of a relation R_h to a relation R_{h+1} corresponds to splitting some classes of the partition. This is done by checking if the next-state functions of the variables in the same class are equivalent under the variable correspondence condition; this can be calculated with BDDs. The BDD representation of the condition $R_{h,VC}(\underline{s})$ is very sensitive to the variable order that is used, because its general form is $(t_1 \equiv u_1 \wedge t_2 \equiv u_2 \wedge t_3 \equiv u_3 \wedge \ldots)$ where the t's and u's are state variables. Therefore only the relevant part of this condition is constructed for every comparison of two variables; just the state variables in the support of the δ_j and δ_k concerned are taken into account.

The fixed point calculation can also be implemented as follows. From every class of equivalent variables, a single variable is selected as the unique representative of that class, i.e., a function $rep_h : V \rightarrow V$ is chosen which satisfies $R_h(u, v) \leftrightarrow (rep_h(u) = rep_h(v))$. Then the next-state functions are expressed in terms of the selected variables. This way the variable correspondence condition is satisfied by construction and the expression to refine an approximation R_h becomes:

$$R_{h+1}(v_j, v_k) = R_h(v_j, v_k) \wedge (\forall \underline{s} \in B^n, \underline{i} \in B^m : \delta_j(rep_h(\underline{s}), \underline{i}) \equiv \delta_k(rep_h(\underline{s}), \underline{i})) ,$$

where $rep_h(\underline{s}) = (rep_h(s_1), \ldots, rep_h(s_p))$. When BDDs are used to represent the next-state functions, the main advantages of this approach are that the BDD for the variable correspondence condition does not have to be

constructed and that less BDD variables are needed; the maximum number of variables needed to represent state variables equals the number of classes of the maximum variable correspondence relation. Of course it is necessary to recalculate the BDDs for the next-state functions after every iteration; this is not a significant drawback because BDD packages cache the results of previous computations and therefore these recalculations are typically performed very efficiently. Of course this technique is not restricted to BDD-based verification methods. After all state variables are assigned a representative variable, the comparison of the resulting next-state functions can be done with any combinational verification method. If the descriptions of both FSMs have similar structures, it could be useful to apply a verification method which exploits these similarities, such as e.g. presented in [1][9][10].

The efficiency of the fixed point computation can be improved with the following technique. It is based on testing the second condition of definition 4 for only a limited number of states and input vectors. A signature is calculated for every state variable by evaluating the next-state functions for some randomly chosen states and input vectors; of course it is necessary that the chosen states conform to R_h. Then every class of the partition is split into classes which only contain state variables with the same signature. This technique is very effective to obtain a more accurate initial approximation R_0 which reduces the number of iterations required to reach the fixed point.

The presented method can easily be extended to also detect state variables which have opposite values in all reachable states. This requires the following equivalence relation $T : V \times V \rightarrow B$:

$$T(v_j, v_k) = (\forall\ (s_1,\ldots,s_n) \in S_0 : s_j \equiv s_k) \vee (\forall\ (s_1,\ldots,s_n) \in S_0 : s_j \equiv \overline{s_k})\ .$$

All state variables equivalent under T can be given the same initial value by selectively complementing some variables. Of course, the next-state and output functions have to be changed accordingly. After this transformation, the presented fixed point computation can be applied to calculate the variable correspondence relation.

The method can also be extended to detect state variables which have a constant value in all reachable states. This requires the introduction of an extra function which defines if a variable always has the same value or not. Then the variable correspondence condition can be strengthened with this information, and the correctness of the function can be checked by testing whether the next-state function of a variable is constant under the variable correspondence condition.

When the state encoding of a design has not been modified, the full generality of sequential verification is not needed. If the correspondence between the state variables of both descriptions is known, the problem essentially becomes a combinational verification problem. This correspondence is however not always obvious. If for example the implementation has been extracted from a transistor level description (see e.g. [7][8]), it cannot be assumed that the correspondence can be derived from the names of the state variables. The maximum variable correspondence relation can be used to define this type of combinational equivalence without requiring that the correspondence between the state variables is given explicitly.

Definition 6

Two FSMs M_A and M_B are *combinationally equivalent* iff the corresponding product machine M satisfies the following condition:

$$\forall\ \underline{s} \in B^n, \underline{i} \in B^m : R_{VC}(\underline{s}) \rightarrow \Lambda(\underline{s}, \underline{i}) = (1,\ldots,1)\ ,$$

where R denotes the maximum variable correspondence relation of M.

As has been shown in this section, the maximum variable correspondence relation can be calculated by repeatedly testing the equivalence of boolean functions. Therefore, it should not be very difficult to extend existing combinational verification tools to also verify the combinational equivalence of FSMs as defined in definition 6.

4 Experimental Results

This section presents the results of some preliminary experiments which have been performed with the proposed technique to detect equivalent state variables. The method has been implemented in C++ using the BDD package developed in our department; the sifting algorithm [13] is used to dynamically control the variable order. All tests have been performed on a HP9000/735 workstation.

The verification method has been used to compare some circuits from the IWLS'91 benchmark set. Every circuit is compared against an equivalent circuit from the benchmark set; if only one implementation is available, a second implementation has been synthesized with the logic synthesis system *SIS* developed at the University of California, Berkeley. The applied transformations only modify the combinational part of the circuits; they include collapsing, factoring and technology mapping. The resulting circuits are identified by adding a 'c' to their name. To avoid any coincidental similarities between the two circuit descriptions, the order of the latches in every second description has been changed randomly. The results are shown in table 1.

Table 1. Experimental results for some IWLS'91 benchmarks

Circuits	Nr. of state vars	Extraction		Expl. Cond.		Impl. Cond.		Nr. of iter.
		time (s)	mem. (kb)	time (s)	mem. (kb)	time (s)	mem. (kb)	
s208.1 – s208.1c	8 + 8	0.1	288	0.2	288	0.1	281	1
s298 – s298c	14 + 14	0.1	291	0.2	291	0.1	279	1
s344 – s349	15 + 15	0.1	297	0.5	362	0.1	285	1
s382 – s400	21 + 21	0.1	301	0.5	366	0.2	285	1
s386 – s386c	6 + 6	0.1	281	0.2	345	0.1	276	1
s420.1 – s420.1c	16 + 16	0.5	371	0.8	371	0.2	293	1
s444 – s444c	21 + 21	0.1	302	0.7	366	0.2	285	1
s510 – s510c	6 + 6	0.2	291	0.5	355	0.2	285	1
s526 – s526n	21 + 21	0.1	302	0.4	366	0.2	285	1
s641 – s713	19 + 19	1.7	388	2.4	388	0.2	304	1
s820 – s832	5 + 5	0.2	289	0.4	353	0.2	284	1
s838.1 – s838.1c	32 + 32	6.2	775	9.4	775	3.1	401	14
s953 – s953c	29 + 29	0.2	389	0.7	389	0.5	377	3
s1196 – s1238	18 + 18	0.5	352	0.9	352	0.8	355	1
s1423 – s1423c	74 + 74	17.4	862	77.1	1075	5.0	402	1
s1488 – s1494	6 + 6	0.4	347	1.2	347	0.4	277	1
s5378 – s5378c	164 + 163	23.6	710	197.3	9954	9.3	481	4
s9234.1 – s9234.1c	211 + 145	18.3	885	137.9	885	9.4	469	9
s13207.1 – s13207.1c	638 + 474	44.4	2807	—	>50Mb	16.7	864	24
s15850.1 – s15850.1c	534 + 504	—	>50Mb	—	>50Mb	435.6	2745	17
s38417 – s38417c	1636 + 1463	—	>50Mb	—	>50Mb	—	>50Mb	—
s38584.1 – s38584.1c	1426 + 1260	98.8	20164	—	>50Mb	88.2	1383	6

The column ‘extraction’ shows the run times and memory usage for building the BDDs for both FSMs. These numbers are given for comparison. Memory usage only includes the memory used by the BDD package. The column ‘expl. cond.’ shows the results of the algorithm which uses the correspondence condition explicitly to calculate the maximum variable correspondence relation. The column ‘impl. cond.’ shows the results for the algorithm which selects unique representatives for the state variables and thus satisfies the correspondence condition implicitly. This requires that the BDDs for both FSMs are recalculated after every iteration of the fixed point calculation. The last column shows the number of iterations required to reach the fixed point. Both algorithms use signature calculations to obtain an initial approximation of the maximum variable correspondence relation. Signatures are calculated by evaluating the next-state functions for 32 vectors in parallel. This is repeated until the partition does not change during 16 successive runs.

The results clearly demonstrate that the maximum variable correspondence relation can be calculated very efficiently with the algorithm that satisfies the correspondence condition implicitly. Benchmark s38584.1 with 1426 state variables is for example completely verified within 90 seconds. The algorithm only fails to complete the calculations for the benchmark s38417, for which we have not been able to build BDDs within the memory limit of 50 Mb. The efficiency of the algorithm is even more evident when the results are compared to the run times and memory usage needed to build BDDs for both circuits separately. In almost all cases it requires less memory because it assigns the same BDD variable to equivalent state variables. Therefore more BDD nodes can be shared by both FSMs. Another important effect of this technique is that the variable ordering algorithm is better capable of finding a good variable order; this is especially clear for the benchmarks with more than 50 state variables. Of course, this also leads to shorter run times. The efficiency of the algorithm which calculates the correspondence condition explicitly is acceptable for the smaller benchmarks with up to about 32 state variables. It is however more sensitive to the variable order and therefore it is not sufficiently robust for the larger benchmarks.

Table 2 gives an overview of the benchmarks for which the verification method also detects equivalent state variables within a single circuit. The first two columns respectively show the name of the benchmark and the total number of state variables. The next columns show the number of state variables that are not directly or indirectly connected to an output, the number of state variables which have a constant value and the number of state variables which are removed because they are duplicates of other variables. The column ‘nr. unconn. vars after’ shows the number of state variables which initially seem to be directly or indirectly connected to an output, but which become unconnected after the removal of constant and duplicate variables. The last column gives the resulting number of state variables. Especially for the three largest benchmarks this number is considerably less than the total number of state variables.

Table 2. Classification of the state variables for some IWLS’91 benchmarks

Circuit	Total nr. vars	Nr. unconn. vars before	Nr. const. vars	Nr. dupl. vars	Nr. unconn. vars after	Nr. unique vars
s641	19	0	4	1	0	14
s1423	74	0	0	1	0	73
s5378	164	0	0	1	0	163
s9234.1	211	66	3	13	0	129
s13207.1	638	21	74	174	106	263
s15850.1	534	10	54	18	16	436
s38584.1	1426	0	37	107	141	1141

The experimental results clearly demonstrate that the proposed method forms a robust extension of any symbolic traversal algorithm. Of course, more experiments need to be performed to test the performance gain

when only some of the state variables are equivalent and therefore the state space of the reduced product machine has to traversed. It is however clear that the technique does not introduce a significant overhead and can have a very positive effect on the run time, the memory usage and the robustness of any BDD-based method for FSM verification.

5 Conclusions and Future Research

This paper has presented a fully automatic technique to detect equivalent state variables for FSM verification. It can easily be integrated with existing verification methods. The technique is based on the detection and removal of equivalent state variables from the product machine before the state space is traversed. It is very robust because the removal of equivalent state variables always simplifies the verification problem and the proposed detection algorithm is very efficient; furthermore it can easily be extended to also handle opposite and constant variables. Therefore it significantly extends the ability of existing verification methods to verify FSMs with similar state encodings. The experimental results show that the maximum variable correspondence relation can be calculated efficiently, especially when the variable correspondence condition is used implicitly in the calculations. The maximum variable correspondence relation can also be used to define combinational equivalence for FSMs with unknown correspondence between the state variables.

There are some aspects which require further research. More experiments need to be performed to test the performance gain of the method when two FSMs partly have equivalent state variables. The repeated calculation of BDDs for the next-state and output functions can probably be implemented more efficiently. It could also be very interesting to experiment with other combinational verification methods to implement the detection of equivalent state variables. A more challenging problem is the development of other efficient automatic reduction techniques. One approach would be to take all internal points into account during the detection of equivalences and not just state variables. This could lead to an efficient verification method for retimed circuits.

Acknowledgements

We would like to thank Geert Janssen for his support and for the use of his BDD package.

References

[1] D. Brand: Verification of Large Synthesized Designs. Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, pp. 534–537, 1993

[2] J.R. Burch, et al.: Sequential Circuit Verification Using Symbolic Model Checking, Proc. 27th ACM/IEEE Design Automation Conf., pp. 46–51, 1990

[3] J.R. Burch, et al.: Symbolic Model Checking for Sequential Circuit Verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 4, pp. 401–424, April 1994

[4] G. Cabodi, et al.: A New Model for Improving Symbolic Product Machine Traversal, Proc. 29th ACM/IEEE Design Automation Conf., pp. 614–619, 1992

[5] O. Coudert, C. Berthet, and J.C. Madre: Verification of Synchronous Sequential Machines based on Symbolic Execution, Proc. Workshop on Automatic Verification Methods for Finite State Machines, pp. 365–373, Lecture Notes in Computer Science vol. 407, 1989

[6] O. Coudert, private communication, March 1995

[7] P. Déverchère, et al.: Functional Abstraction and Formal Proof of Digital Circuits, Proc. European Conf. on Design Automation, pp. 458–462, 1992

[8] A. Kuehlmann, A. Srinivasan, and D.P. LaPotin: Verity – A Formal Verification Program for Custom CMOS Circuits, to be published in IBM Journal of Research and Development, 1995

[9] W. Kunz: HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning, Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, pp. 538–543, 1994

[10] R. Mukherjee, J. Jain, and M. Fujita: VERIFUL: VERIfication using FUnctional Learning, Proc. European Design and Test Conf., pp. 444–449, 1995

[11] S. Panda, F. Somenzi, and B. Plessier: Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams, Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, 1994

[12] M. Payer, private communication, March 1995

[13] R. Rudell: Dynamic Variable Ordering for Ordered Binary Decision Diagrams, Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, pp. 42–47, 1993

[14] H.J. Touati, et al.: Implicit State Enumeration of Finite State Machines using BDD's, Proc. IEEE Int. Conf. on Computer-Aided Design, pp. 130–133, 1990