Available online at www.sciencedirect.com
ScienceDirect
Computers and Electrical Engineering 33 (2007) 58–69
Computers and Electrical Engineering
www.elsevier.com/locate/compeleceng

On state reduction of incompletely specified finite state machines

Sezer Gören [a,*], F. Joel Ferguson [b]

[a] Department of Computer Engineering, Bahçeşehir University, Beşiktaş 34349, Istanbul, Turkey
[b] Department of Computer Engineering, University of California, Santa Cruz, CA 95064, USA

Received 9 September 2005; received in revised form 7 January 2006; accepted 7 June 2006
Available online 29 September 2006

Abstract

State reduction of incompletely specified finite state machines (ISFSMs) is an important task in optimization of sequential circuit design and known as an NP-complete problem. Removal of redundant states reduces the logic, because of this, chip area decreases. In addition, test generation is easier when the sequential circuit is irredundant. In this paper, we present a heuristic for state reduction of ISFSMs. The proposed heuristic is based on a branch-and-bound search technique and identification of sets of compatible states of a given ISFSM specification. We have obtained results as good as the best exact method in the literature but with significantly better run-times.

1. Introduction

Finite state machines (FSMs) have been widely used to express algorithms, communication protocols, systems (at a high-level of abstraction), sequential logic circuits, and sequential logic cells. State reduction of FSMs is a well-known and important problem in sequential circuit synthesis. The state table of an FSM often contains redundant states that may have been introduced by the designer. Elimination of redundant states in an FSM reduces the logic needed to implement, synthesize, and verify it.

The state reduction problem for completely specified FSMs can be solved in polynomial time [1,2]. For ISFSMs, the problem is known to be NP-complete [3]. The standard approach for the reduction of ISFSMs is based on the enumeration of the set of compatibles and satisfaction of a set of constraints. Paull and Unger [4] developed a general framework and proposed methods for generating the maximal compatibles and obtaining the minimal closed cover. Luccio [5] proposed prime classes and formulated the minimization problem in the form of a binate covering problem. Based on this approach a number of researchers proposed techniques that use explicit [6] and implicit [8] enumeration of compatibles. Rho et al. [6] presented a program called *stamina* that runs in exact and heuristic modes using explicit enumeration for the state minimization problem. The exact mode of *stamina* is based on Grasselli and Luccio's binate covering approach. Ahmad and Das [7]

* Corresponding author. Tel.: +90 212 236 54 90; fax: +90 212 258 70 83.
E-mail address: sgoren@bahcesehir.edu.tr (S. Gören).

0045-7906/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compeleceng.2006.06.001

proposed a heuristic called *void* that follows a strategy of handling the closure condition first and satisfying the cover condition afterwards. Higuchi and Matsunaga [8] proposed a heuristic program called *slim* that is based on generating the sets of all maximal compatibles and reducing the size of the initial solution by iterative improvements. Kam et al. [9] proposed an exact state minimization technique using implicit enumeration of the compatibles. The implementation of their technique is called *ism*. Pena and Oliveira [10] presented an exact state minimization algorithm based on mapping ISFSMs to tree FSMs (TFSMs) which combined Bierman's search algorithm [11] with Angluin's L^* algorithm [12]. Pena and Oliveira compared their method, *bica*, with *ism* and *stamina* in the exact mode and showed that *bica* was more robust in most cases.

In this work, we propose a heuristic for state reduction of ISFSMs. This algorithm can reduce ISFSMs that have deadlock and unreachable states and is based on a checking sequence generation technique [13]. A checking sequence is an I/O sequence that distinguishes an FSM from all other FSMs. Checking sequences are functional tests [14] that provide a black-box testing method for sequential circuits. Our initial findings were presented in [15]. In this paper, we present an improved version of the algorithm with significantly better performance. We will present our heuristic, then compare its performance with the exact algorithm *bica* and the heuristics *stamina*, and *slim*. The effectiveness of our approach is shown by comparing our results using various ISFSM benchmarks.

The organization of this paper is as follows. We present the preliminaries in Section 2 and the outline of the algorithm with the help of an example in Section 3. We present results in Section 4. Section 5 concludes the paper. We present the pseudo code of the algorithm in the Appendix.

2. Preliminaries

This section describes the forms of inputs and outputs to our algorithm and defines their relationships.

Definition 2.1. An FSM is a tuple $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ where $\Sigma \neq \emptyset$ is a finite set of input symbols, $\Delta \neq \emptyset$ is a finite set of output symbols, $Q \neq \emptyset$ is a finite set of states, and $q_0 \in Q$ is the initial state. The transition function is defined as $\delta(q,a) : Q \times \Sigma \to Q \cup \{\phi\}$, and the extended transition function, $\hat{\delta} : Q \times \Sigma^* \to Q \cup \{\phi\}$ is defined as $\hat{\delta}(q,xa) = \delta(\hat{\delta}(q,x),a)$. The output function is defined as $\lambda(q,a) : Q \times \Sigma \to \Delta \cup \{\varepsilon\}$, and the extended output function $\hat{\lambda} Q \times \Sigma^* \to \Delta \cup \{\varepsilon\}$ is defined as $\hat{\lambda}(q,xa) = \lambda(\hat{\lambda}(q,x),a)$. We will use $a \in \Sigma$ to denote a particular input symbol, $x \in \Sigma^*$ to denote a string, $b \in \Delta$ a particular output symbol, ϕ to denote an unspecified state, and ε to denote an unspecified output.

Definition 2.2. An FSM, M, is completely specified if from each state it has a transition for each input symbol. M is incompletely specified (ISFSM) if it is not completely specified.

Definition 2.3. An output $b_I = \{b_{i1}, b_{i2}, b_{i3}, \ldots, b_{iz}\}$ is equal to an output $b_s = \{b_{s1}, b_{s2}, b_{s3}, \ldots, b_{sz}\}$, $b_i = b_s$ if and only if $b_{ik} = b_{sk}$ for all k, $1 \leqslant k \leqslant z$.

Definition 2.4. States s and s' of a completely specified FSM are distinguishable if there exists an input sequence for both states that causes non-equal output sequences from these states; otherwise they are equivalent states.

Definition 2.5. A completely specified FSM, M, is reduced if no two of its states are equivalent.

Definition 2.6. An input sequence for an FSM is an acceptable sequence for the FSM if and only if it never visits an unspecified next state.

Definition 2.7. An output symbol, $b_I = b_{I1} b_{I2} b_{I3}, \cdots, b_{Ik}$ is bitwise compatible with an output symbol $b_S = b_{S1} b_{S2} b_{S3}, \cdots, b_{Sk}$, $b_I \cong b_S$ if and only if $b_{Sx} = b_{Ix}$ or $b_{Sx} = \varepsilon$ $\forall x$, $1 \leqslant x \leqslant k$.

Definition 2.8. States s and s' of an incompletely specified FSM are distinguishable if there exists an acceptable input sequence for both states that causes incompatible output sequences from these states; otherwise they are compatible states.

Definition 2.9. An incompletely specified FSM, M, is reduced if no two of its states are compatible.

Definition 2.10. An FSM M_I is compliant with another FSM M_S for an acceptable input sequence if it produces a compatible output sequence with M_S.

Definition 2.11. An FSM M_I is compatible with another FSM, M_S if M_I is compliant with M_I for any acceptable input sequence of M_S.

Definition 2.12. Two states are output incompatible when, for some particular input, these states produce a different output. Two states are transitively incompatible if, on the same input, they lead to incompatible states. Incompatibility function I:

$$I(q_{s1}, q_{s2}) = \begin{cases} 1 & \text{if } \exists a \ \lambda(q_{s1}, a) \ncong \lambda(q_{s1}, a) \\ 1 & \text{if } \exists a \ \delta(q_{s1}, a) = q_{d1} \text{ and } \delta(q_{s2}, a) = q_{d2} \text{ and } I(q_{d1}, q_{d2}) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Definition 2.13. A set of states, C, is a compatible class iff $\forall q_i, q_j \in C$ are pairwise compatible, i.e., $I(q_i, q_j) = 0$.

Definition 2.14. A set of compatible states is maximal if it is not a proper subset of another set of compatibles.

Definition 2.15. Compatibility cover is a set of compatible classes $S = \{C_0, C_1, \ldots, C_k\}$ such that every state in Q belongs to one or more compatibles in S.

Definition 2.16. A set of states, denoted $IS(C, a)$, is an implied set of a compatible class C for input a that are the next states reachable from the states in C, $IS(C, a) = \{q_d | \delta(q_s, a) = q_d, \forall q_s \in C\}$. A set S of compatible classes is closed if for each $C \in S$, all of its implied sets $IS(C, a)$ are contained in some element of S for all inputs. (A minimum cardinality cover that is consistent with this covering and closure requirement is a solution for the exact state minimization problem of ISFSM [7,9,10].)

Definition 2.17. An output symbol b is compatible with a set of output symbols $B = \{b_0, b_1, \ldots b_k\}$ iff b is compatible with every element of B.

$$b \cong B \quad \text{iff } \forall b_i \in B, \ b \cong b_i. \tag{2.2}$$

Let $M' = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$ be an ISFSM. Let $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ be the reduced FSM. Let S' be a set of compatible classes and a closed cover, $C' \in S'$ a compatible class of M', and $x \in \Sigma^*$ be an acceptable input sequence for M'.

Definition 2.18. The compatible class output function $\lambda'(C', a)$ is defined as

$$\lambda'(C', a) = B' \quad \text{where } B' = \{b'_i : \lambda'(q', a) = b'_i \ \forall q' \in C'\}. \tag{2.3}$$

The compatible class extended output function is defined as:

$$\hat{\lambda}'(C', xa) = \lambda'(\hat{\lambda}'(C', x), a). \tag{2.4}$$

Definition 2.19. The compatible class transition function $\delta'(C', a)$ is defined as

$$\delta'(C', a) = IS(C', a). \tag{2.5}$$

The compatible class extended transition function is defined as

$$\hat{\delta}'(C', xa) = \delta'(\hat{\delta}'(C', x), a). \tag{2.6}$$

Definition 2.20. A function $F: Q \rightarrow S'$ is defined as a one-to-one mapping between the set of states of M and the closed cover S' of M' if it satisfies

$$\forall x\ \hat{\lambda}(q_0, x) \cong \hat{\lambda}'(F(q_0), x) \tag{2.7}$$

$$\forall x\ F(\hat{\delta}(q_0, x)) = \hat{\delta}'(F(q_0), x). \tag{2.8}$$

Using Definitions 2.7, 2.18, and 2.19, Eq. (2.7) states that function F maps each state $q \in Q$ to a compatible class C' in M' that satisfies the output of M' for all possible input sequence x. Eq. (2.8) states that the implied set of a compatible class is correctly mapped. The output and transition requirements for a valid mapping function F are depicted in Fig. 1. In this figure, it is shown that if C'_d is the implied set of a compatible class C'_s for input a, B is the corresponding set of outputs, and q_s is mapped to C'_s, then the implied set of C'_s, C'_d, has to be mapped to the next state of q_d and the output set B and output b have to be compatible. We will use the same example given in [9] and shown in Fig. 2a to demonstrate the mapping between FSMs M and M'. State q_0 of M is mapped to the compatible class $C'_0 = \{q'_0, q'_2\}$ of M', whereas state q_1 is mapped to the compatible $C'_1 = \{q'_1, q'_2\}$. The implied sets and the output sets of C'_0 and C'_1 in M' versus the outputs and the next states of q_0 and q_1 for the same input are as follows:

1. $IS(C'_0, 0) = \{q'_0, q'_2\} = C'_0$ and $\delta(q_0, 0) = q_0$.
2. $IS(C'_0, 1) = \{q'_1, q'_2\} = C'_1$ and $\delta(q_0, 1) = q_1$.
3. $IS(C'_1, 0) = \{q'_2\} \subset C'_0$ and $\delta(q_1, 0) = q_0$.
4. $IS(C'_1, 1) = \{q'_0, q'_2\} = C'_0$ and $\delta(q_1, 1) = q_0$.
5. $\lambda'(C'_0, 0) = \{0\}$ and $\lambda(q_0, 0) = 0$.
6. $\lambda'(C'_0, 1) = \{0, \varepsilon\}$ and $\lambda(q_0, 1) = 0$.
7. $\lambda'(C'_0, 0) = \{0\}$ and $\lambda(q_1, 0) = 0$.
8. $\lambda'(C'_1, 1) = \{1, \varepsilon\}$ and $\lambda(q_1, 1) = 1$.

Since the output and the transition requirements are satisfied, the function F is a valid mapping function for M and M' shown in Fig. 2a and b, respectively. Therefore, M is compatible with M'.

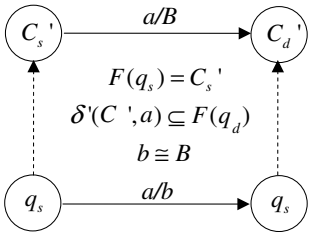

Fig. 1. Output and transition requirements for a valid mapping function F.

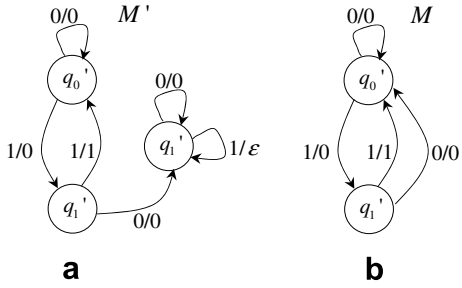

Fig. 2. (a) FSM specification M'. (b) FSM M compatible with M'.

Definition 2.21. The upper bound on the states of the reduced machine [6] is the cardinality of the set of maximal compatibles (Definition 2.14).

Definition 2.22. The lower bound on the states of the reduced machine [6] is the number of states contained in a maximal incompatible with maximum number of states.

3. Proposed algorithm

In this section, we will explain how the algorithm works by referring to the variables and procedures of the pseudo code given in Appendix.

1. Start with an ISFSM, M' which we want to reduce the number of states.
2. Start with an acceptable sequence, *seq,* for M' with a length of 1 which is just the reset input.
3. Set *upperBound* to the upper bound on the states of the reduced M' (Definition 2.21).
4. Set *lowerBound* to the lower bound on the states of the reduced M' (Definition 2.22).
5. Construct an FSM, M that is compliant with M' for the sequence *seq* and search for an FSM such that the number of states of M is less than the number of states of M'. Exit if time is out or no such FSM exists.
6. Build the product machine of M and M'. During this, whenever M and M' produce incompatible outputs, stop building the product machine, construct the corresponding contradicting sequence, seq_{contra}, and concatenate seq_{contra} to *seq* and go to step 5. If no contradicting sequence is found, display M, and set *upperBound* to one less than the number of states of M. Exit if *upperBound* is less than *lowerBound* (Definition 2.22). Go to step 5.

We introduced Steps 3 and 4 to the algorithm presented in [15] to increase the performance. Since the algorithm is based on a branch and bound technique, setting an upper and a lower bound on the states significantly narrows down the search space in Step 5. Step 5 of the algorithm corresponds to the search for compliant FSM M that gives compatible output with the specification FSM M' for an input sequence *seq*. This input sequence is incrementally constructed by concatenating the contradicting sequence, seq_{contra}, found when M and M' are incompatible. If M is found to be compatible with M', we output M. After this, search for a better solution continues with $upperBound = |Q| - 1$, where $|Q|$ is the number of states of M. Our state reduction algorithm for ISFSMs is based on a checking sequence generation technique [12]. We incrementally construct an I/O sequence similar to this technique [12] but without deriving entire checking sequence. Checking sequence generation requires that all states are reachable and connected; therefore, we augment an FSM to get out of deadlock states and to reach every state. FSM specification, M', is augmented by adding a reset state and transitions (AugmentResetState procedure shown in Appendix). We introduce this extra state and its transitions to take care of deadlock and unreachable states as the example in Fig. 2a illustrates. Next, we will explain how the algorithm works using the FSM shown in Fig. 2a.

Step 1: First we augment the ISFSM in Fig. 2a (flow-table is shown in Table 1) by introducing an extra state q'_r, a transition from every other state to this extra state, and transitions from the extra state to the initial state q'_0. If there is no initial state, one of the original states is picked as the initial state. Note that the new state q'_r should be incompatible with q'_0, q'_1, and q'_2. In order to make the newly added state incompatible, we introduce an extra input and an output. The flow-table representation of the augmented FSM is shown in Table 2. In Table 2, the leftmost input and output bits are added. The leftmost output bit of each of the original transi-

Table 1
Flow-table of FSM specification, M'

	0	1
q'_0	$q'_0, 0$	$q'_1, 0$
q'_1	$q'_2, 0$	$q'_0, 1$
q'_2	$q'_2, 0$	q'_2, ε

Table 2
Flow-table of M' after adding extra reset state and transitions

	00	01	10
q'_0	$q'_0, 00$	$q'_1, 00$	$q'_r, 10$
q'_1	$q'_2, 00$	$q'_0, 00$	$q'_r, 10$
q'_2	$q'_2, 00$	$q'_2, 0\varepsilon$	$q'_r, 10$
q'_r	$q'_0, 10$	$q'_0, 10$	$q'_r, 10$

tions is set to "0" whereas the leftmost input bit is set to "0". The output of each of the new transitions is set to "10". By introducing an extra state and extra transitions, every state of the augmented M' becomes reachable. We also introduce a reset state and transitions to the FSM under construction. The flow-table of the augmented FSM is shown in Table 3.

After adding a reset state and transitions, a search for FSM M that is compliant with M' for the input sequence ($seq = \langle 10/10 \rangle$) begins by the branch and bound procedure, *FindCompliantFSM* shown in Appendix. This procedure is recursive. Every recursion corresponds to a state transition and the depth of recursion is the length of the I/O sequence. The I/O sequence is incrementally constructed by using the procedure *FindContradictingSequence*. This procedure is based on breadth-first search and therefore it returns the shortest contradicting sequence for M and M'. Also note that the ordering of the vertices can be chosen randomly during the breadth first search.

In Table 3 there is an initial state q_0 and a reset state q_r. We set $F(q_r)$ to q'_r. We initially apply input sequence "10" at Step 1 in Table 3 and expect output sequence "10" from both M and M'. When this sequence ($seq = \langle 10/10 \rangle$) is, applied, M' is in state q'_r and M is in q_r.

Step 2: Then we call the FindContradictingSequence procedure to check whether M (Table 3) and M' are compatible. A contradicting sequence ($seq_{\text{contra}} = \langle 00/10, 00/00 \rangle$) is found since M' produces an output "10, 00", whereas M gives an undefined output and next-state. We concatenate seq_{contra} to seq ($seq = \langle 10/10,\ 00/10,\ 00/00 \rangle$). Then we fill the flow-table of M as shown in Table 4. We set $F(q_0)$ to $\{q'_0\}$.

Step 3: Next we call the FindContradictingSequence procedure. A contradicting sequence ($seq_{\text{contra}} = \langle 01/00 \rangle$) is found, since output and next-state of M are undefined for this input. If we pick q_0 as next-state, then we have to set $F(q_0)$ to $\{q'_0, q'_1\}$ but these states are incompatible by Definition 2.12 ($\lambda'(q'_0, 1) = 0$ and $\lambda'(q'_1, 1) = 1$). Therefore, we introduce a new state q_1 and set $F(q_0)$ to $\{q'_1\}$. We fill the flow-table for M. Then we check whether M and M' are compatible by calling *Find-ContradictingSequence* procedure. A contradicting sequence ($seq_{\text{contra}} = \langle 00/00 \rangle$) is found, since at input "00", output and next-state are undefined. We can choose q_0 as next state because it does not give any contradiction when we set $F(q_0)$ to $\{q'_0, q'_2\}$. We fill the flow-table of M as shown in Table 5. At this point seq is equal to $\langle 10/10, 00/10, 00/00, 01/00, 00/00 \rangle$.

Step 4: Then we call the *FindContradictingSequence* procedure to check whether M (Table 5) and M' are compatible. A contradicting sequence ($seq_{\text{contra}} = \langle 01/0\varepsilon,\ 01/0\varepsilon \rangle$) is found. The first "01" of the sequence requires $F(q_1)$ equal to $\{q'_1, q'_2\}$, and this will not contradict with our definition of a valid mapping function. At the second "01" input we can choose q_0 as next-state and then we fill the flow-table entry as shown in Table 6.

Table 3
Flow-table of M at Step 1

	00	01	10
q_0	$\phi, \varepsilon\varepsilon$	$\phi, \varepsilon\varepsilon$	$q_r, 10$
q_r	$q_0, 10$	$q_0, 10$	$q_r, 10$

Table 4
Flow-table of M at Step 2

	00	01	10
q_0	$q_0, 00$	$\phi, \varepsilon\varepsilon$	$q_r, 10$
q_r	$q_0, 10$	$q_0, 10$	$q_r, 10$

Table 5
Flow-table of M at Step 3

	00	01	10
q_0	$q_0, 00$	$q_1, 00$	$q_r, 10$
q_1	$q_0, 00$	$\phi, \varepsilon\varepsilon$	$q_r, 10$
q_r	$q_0, 10$	$q_0, 10$	$q_r, 10$

Table 6
Flow-table of M at Step 4

	00	01	10
q_0	$q_0, 00$	$q_1, 00$	$q_r, 10$
q_1	$q_0, 00$	$q_0, 0\varepsilon$	$q_r, 10$
q_r	$q_0, 10$	$q_0, 10$	$q_r, 10$

Table 7
Flow-table of M at Step 5

	00	01	10
q_0	$q_0, 00$	$q_1, 00$	$q_r, 10$
q_1	$q_0, 00$	$q_0, 01$	$q_r, 10$
q_r	$q_0, 10$	$q_0, 10$	$q_r, 10$

Table 8
Flow-table of M at Step 6

	0	1
q_0	$q_0, 0$	$q_1, 0$
q_1	$q_0, 0$	$q_0, 1$

Table 9
Constructed I/O sequence

#	0	1	2	3	4	5	6	7	8	9	10
I	10	00	00	01	00	01	01	10	00	01	01
O	10	10	00	00	00			10	10	00	01
$S_{M'}$	q'_r	q'_0	q'_0	q'_1	q'_2	q'_2	q'_2	q'_r	q'_0	q'_1	q'_0
S_M	q_r	q_0	q_0	q_1	q_0	q_1	q_0	q_r	q_0	q_1	q_0
	Step 1	Step 2		Step 3		Step 4			Step 5		

I, Input; O, Output; $S_{M'}$: Current state of M', S_M: Current state of M.

Step 5: We call the *FindContradictingSequence* procedure to check whether M (Table 6) and M' are compatible. It returns a contradicting sequence ($seq_{\text{contra}} = \langle 10/10, 00/10, 01/00, 01/01 \rangle$). We fill the flow-table of M as shown in Table 7.

Step 6: After calling the *FindContradictingSequence* procedure, M (Table 7) and M' are found to be compatible ($seq_{\text{contra}} = \langle\rangle$). Finally, the extra state q_r, the transitions to and from q_r, and the leftmost inputs and outputs are discarded. We obtain an FSM shown in Table 8 that is same as the FSM shown in Fig. 2b. The constructed I/O sequence, *seq*, and corresponding states that M and M' are in are given in Table 9.

4. Results and performance

To evaluate our algorithm we used the same set of ISFSMs used in previous work [8,10]. These ISFSMs come from a variety of sources such as standard benchmarks, asynchronous synthesis, learning problems, and synthesis of interacting FSMs.

Table 10
Experimental results

Benchmark			N_{final}				Run-time in seconds			
FSM	PI	N_{init}	*bica*	*stamina*	*slim*	*chesmin*	*bica*	*stamina*	*slim*	*chesmin*
alex_1	5	42	6	6	6	6	19.11	7.10	0.25	0.33
intel_edge	3	28	4	4	4	4	1.39	0.29	0.04	0.04
isend	7	40	4	4	4	4	10.01	0.80	0.07	0.31
rcv-ifc	8	46	2	2	2	2	9.08	0.15	0.47	1.0
rcv-ifc.m	8	27	2	2	2	2	4.71	0.05	0.03	0.8
send-ifc	8	70	2	2	3	2	34.83	0.63	0.09	1.7
send-ifc.m	8	26	2	2	2	2	12.07	0.04	0.03	1.1
vbe4a	6	58	3	3	3	3	27.29	99.14	1.41	0.8
ifsm0	7	38	3	3	3	3	2.92	0.08	0.05	1.98
th.20	2	21	4	6	6	4	0.46	0.07	0.07	0.03
th.30	2	31	5	9	7	6,5	0.98	0.37	0.04	0.12, 7.54
th.40	2	41	8	15	9	9,8	1.68	0.36	0.11	0.12, 4
th.55	*2*	*55*	*8*	*24*	*13*	*10,9*	*2088.65*	*1.84*	*2.91*	*0.46, 3.09*
fo.20	2	21	3	4	3	3	0.45	0.05	0.01	0.08
fo.30	2	31	3	5	4	3	0.72	0.31	0.05	0.34
fo.40	2	41	4	8	7	5,4	7.96	55.61	0.11	0.05, 7.05
fo.50	2	51	6	11	9	7,6	5.11	2.96	0.07	0.24, 26.19
fo.70	*2*	*71*	*FAILS*	*14*	*10*	*8,7*	*FAILS*	*7.01*	*0.19*	*0.37, 0.55*
e271	2	19	2	2	2	2	2.37	0.02	0.01	0.02
e285	2	19	2	2	2	2	0.68	0.02	0.01	0.02
e304	2	19	2	2	2	2	0.64	0.02	0.02	0.02
e423	2	19	2	3	3	2	0.42	0.31	0.03	0.31
e680	*2*	*19*	*2*	*2*	*2*	*2*	*0.78*	*0.02*	*0.02*	*0.02*
rubin18	1	18	3	3	3	3	0.12	0.02	0.11	0.02
rubin600	1	600	3	*FAILS*	3	3	29.47	*FAILS*	4.23	1.52
rubin1200	1	1200	3	*FAILS*	3	3	229.26	*FAILS*	17.96	15.30
rubin2250	1	2250	3	*FAILS*	3	3	138,498	*FAILS*	66.33	58.24

PI: Number of primary inputs. N_{init}: initial number of states. N_{final}: Final number of states.

The final number of states and run-times obtained from *bica*, heuristic mode of *stamina*, *slim*, and *chesmin* are given in Table 10. The run-times of *bica*, *stamina*, and *chesmin* are obtained by running on the same Pentium/133 MHz PC with Linux. Unfortunately, we were unable to obtain the source code of *slim* or its executable for our platform. Therefore, we could not run it on our platform. However, Higuchi et al. [8] presented run-times of *slim* and *stamina* on their platform. Since we have *stamina*'s run-times on our platform, we have scaled down *slim*'s run-times by a factor of "*stamina*-run-time-on-our-platform/*stamina*-run-time-on-[8]". In Table 10 for some cases two numbers are given for the number of states for *chesmin*. This is because *chesmin* is an incremental algorithm and hence continues to search for solutions with fewer states when it finds a solution.

We demonstrate the final number of states and the run-time comparions of *bica*, heuristic mode of *stamina*, *slim*, and *chesmin* in Table 11. In Table 11 we labeled N_{final}/Run-time using the following rules:

- + when *chesmin*'s N_{final} is fewer.
- − when *chesmin*'s N_{final} is more.
- Blank when *chesmin*'s N_{final} is equal.
- Compare run-times only when *chesmin*'s N_{final} is equal. + when *chesmin*'s run-time is less. − when *chesmin*'s run-time is more.
- We highlighted the rows to point out cases: (th.55: 2088s vs. 3.09s) where *chesmin* ran remarkably faster than *bica* but had one additional state, and (fo.70) where *bica* fails, *stamina*'s N_{final} (14) and *slim*'s N_{final} (10) is more than *chesmin*'s N_{final} (8, 7).
- We used the term "FAILS" when we waited for several hours but no solution was found.

Table 11
Comparison of *chesmin* versus *bica, stamina*, and *slim*

Benchmark	Comparison N_{final}/run-time		
	vs. *bica*	vs. *stamina*	vs. *slim*
alex_1	=/+	=/+	=/−
intel_edge	=/+	=/+	=/=
isend	=/+	=/+	=/−
rcv-ifc	=/+	=/−	=/−
rcv-ifc.m	=/+	=/−	=/−
send-ifc	=/+	=/−	+/+
send-ifc.m	=/+	=/−	=/−
vbe4a	−/=	−/=	−/=
ifsm0	=/+	=/−	=/−
th.20	=/+	+/+	+/+
th.30	=/−	+/+	+/+
th.40	=/−	+/+	+/+
th.55	−/=	+/+	+/+
fo.20	=/+	+/+	=/−
fo.30	=/+	+/+	+/+
fo.40	=/+	+/+	+/+
fo.50	=/−	+/+	+/+
fo.70	+/+	+/+	+/+
e271	=/+	=/=	=/−
e285	=/+	=/=	=/−
e304	=/+	=/=	=/=
e423	=/+	+/=	+/=
e680	=/+	=/=	=/=
rubin18	=/+	=/=	=/+
rubin600	=/+	+/+	=/+
rubin1200	=/+	+/+	=/+
rubin2250	=/+	+/+	=/+

Comparison rules (N_{final}/*run-time*): (1) Compare run-times only if N_{final} is equal. (2) Blank N_{final} if N_{final} is equal. (3) + if N_{final} is fewer. (4) − if N_{final} is more. (5) Blank run-time if N_{final} is fewer. (6) + if run-time is less. (7) − if run-time is more.

Fewer states is our key criterion in comparing the methods. We believe run-time is only a factor if two methods are comparable in their ability to reduce the number of states. Having said that *chesmin* is much superior than *stamina* and *slim*. *Chesmin* surpasses both *stamina* and *slim* in 10 cases, loses only in 1, and ties it in 13. Since *bica* is an exact method, *chesmin* is not expected to find the minimum number of states. (However, note that *bica* fails in one benchmark.) Hence, *chesmin*'s advantage is in its faster run-times. Here is how *chesmin* compares to *bica*:

- *Chesmin* tied *bica* in 24 benchmarks.
- Out of 24, *chesmin* had faster run-times in 21 of them. In three of them, it ran slower.
- In two cases, *chesmin*'s solutions had one more state (vbe4a, th.55).
- However, one of the above cases (th.55) *chesmin* ran 675× faster than *bica*.
- In one case (fo.70), *bica* failed and *chesmin* found a solution.
- *Chesmin* on the average ran 53× faster than *bica*.

From our empirical results, we have observed that *chesmin* runs in polynomial between $O(N_{init}^2)$ and $O(N_{init}^2)$ However, the worst-case run-time can be exponential as this is an NP-complete problem [3].

5. Conclusion

We have proposed a heuristic algorithm for the state reduction problem of incompletely specified FSMs. We have obtained fewer or equal number of states and better run-time than the previous work in the literature.

In some cases we have found a solution where other algorithms could not. *Chesmin* performed as good as the exact algorithm and the run-time is much better with almost no compromise in the final number of states. *Chesmin* is a branch-and-bound technique where an I/O sequence is incrementally built and the search space becomes narrower as the number of ISFSMs that comply with this I/O sequence becomes less and less. Due to this, it is more efficient than the other techniques. *Chesmin* has the additional advantage that, in addition to finding a reduced compatible FSM, it generates an I/O sequence that can be used as test vectors to verify the implementation FSM.

Appendix

```
void main begin
    seq[0] = ⟨Reset⟩;
    M' = ReadSpecificationFSM();
    upperBound = UpperBound(M'); // Definition 2.21
    lowerBound = LowerBound(M'); // Definition 2.22
    M = NewFSM();
    // add extra reset state and transitions
    M'.AugmentResetState(); M.AugmentResetState();
    FindCompliantFSM(q_r, q_0, q'_r, q'_0, upperBound, seq, 1);
end //main
void FindCompliantFSM(q_s, q_d, q'_s, q'_d, upperBound, seq, index)begin
    if (TimeOut()) exit (0);
    if (|Q| > upperBound) return;
    if (index == |seq|) begin
        seq_contra = FindContradictingSequence(q_s, q'_s, seq[index]);
    if (seq_contra = ⟨⟩) begin
        Display(M,Run-Time()); //found a compatible FSM with M'
        upperBound = |Q| − 1;
        if (upperBound < lowerBound) exit (0);
        return;
        end // if (seq_contra = ⟨⟩) begin
    else seq = seq · seq_contra; //concatenate
    end // if (index = = |seq|);
    input = seq[index];
    if (∃q' such that q' ∈ F(q_d) and I(q'_d, q) == 1) return; // Definition 2.13
    output = λ'(q'_s, input);
    if (δ(q_s, input) ≠ φ and δ(q_s, input) ≠ q_d) return;
    if (λ(q_s, input) ≱ output) return;
    λ(q_s, input) = output ⊗ λ(q_s, input); // Table 12 for ⊗.
    δ(q_s, input) = q_d;
    F(q_s) = F(q_s) ∪ {q'_s}; // Definition 2.20
    index + +;
```

Table 12
⊗ operation

⊗	0	1	ε
0	0	N/A	0
1	N/A	1	1
ε	0	1	ε

N/A, not applicable.

```
    if (|Q| < upperBound)Q = Q ∪ {q_new};
    input = seq[index];
    q' = δ'(q'_d, input);
    output = λ(q_d, input);
    q_dd = δ(q_d, input);
    foreach q ∈ Q begin
        F_qd = F(q_d);
        FindCompliantFSM(q_d, q, q'_d, q', upperBound, seq, index);
        // Undo after recursion
        λ(q_d, input) = output;
        δ(q_d, input) = q_dd;
        F(q_d) = F_qd;
    end //foreach
    index–;
end//FindCompliantFSM
```

References

[1] Huffmann DA. The synthesis of sequential switching circuits. J Franklin Institute 1954;257:161–90.
[2] Hopcroft JE. NlogN algorithm for minimizing states in finite automata. Tech Rep, Stanford University 1971; CS 71/190.
[3] Pfleeger CF. State reduction in incompletely specified finite state machines. IEEE Trans Comput 1973;22:1099–102.
[4] Paull MC, Unger SH. Minimizing the number of states in incompletely specified sequential switching functions. IRE Trans Electron Comput 1959;8:356–67.
[5] Luccio F. Extending the definition of prime compatibility classes of states in incompletely specified flow tables with the help of prime closed sets. IEEE Trans Electron Comput 1969:953–6.
[6] Rho J-K, Hachtel G, Somenzi F, Jacoby R. Exact and heuristic algorithms for the minimization of incompletely specified finite state machines. IEEE Trans Comput Aided Des 1994;13:167–77.
[7] Ahmad I, Das AS. A heuristic algorithm for minimization of incompletely specified finite state machines. Comput Electric Eng 2001;27:159–72.
[8] Higuchi H, Matsunaga Y. A fast state reduction algorithm for incompletely specified finite state machines. Des Automat Conf 1996:463–6.
[9] Kam T, Villa T, Brayton R, Sangiovanni-Vincentelli A. Synthesis of FSMs: functional optimization. Kluwer Academic Publishers; 1997.
[10] Pena JG, Oliveira AL. A new algorithm for exact reduction of incompletely specified finite state machines. IEEE Trans Comput Aided Des 1999;18:619–32.
[11] Biermann AW, Feldman JA. On the synthesis of finite state machines from samples of their behavior. IEEE Trans Comput 1972;21:592–7.
[12] Angluin D. Learning regular sets from queries and counter examples. Inform Comput 1987;75:87–106.
[13] Gören S, Ferguson FJ. Checking sequence generation for asynchronous sequential elements. Int Test Conference 1999:406–13.
[14] Hennie FC. Fault detecting experiments for sequential circuits. Int Symp Swit Circuit Theory Logic Des 1964:95–110.
[15] Gören S, Ferguson FJ. CHESMIN: a heuristic for state reduction in incompletely specified finite state machines. IEEE/ACM Des Automat Test Confer Europe 2002:248–54.

Sezer Gören received her B.Sc. and M.Sc. degrees in Electrical and Electronic Engineering from Boğaziçi University, Istanbul, Turkey, and a Ph.D. degree in Computer Engineering from University of California, Santa Cruz, USA. She worked as a senior design verification engineer in Silicon Valley, CA, USA from 1998 until 2004. Since December 2004, she has been with the Department of Computer Engineering at Bahçeşehir University, Istanbul, Turkey, where she is currently an assistant professor. Her research interests include design automation of digital systems, design verification, and VLSI test.

F. Joel Ferguson received a B.S.E in Engineering Analysis and Design from the University of North Carolina at Charlotte, USA. He received an M.S.E.E. and a Ph.D. in Computer Engineering from Carnegie-Mellon University, USA. Currently, he has been with the Department of Computer Engineering at University of California, Santa Cruz, USA, where he is a professor. His research interests include fault modeling, test generation, and design-for-test of digital circuits and systems, fault-tolerant computing, VLSI design, and computer-aided manufacturing for VLSI.