CHESMIN: A Heuristic For State Reduction In Incompletely Specified Finite State Machines

Sezer Gören F. Joel Ferguson
Department of Computer Engineering
University of California
Santa Cruz

Abstract

A heuristic is proposed for state reduction in incompletely specified finite state machines (ISFSMs). The algorithm is based on checking sequence generation and identification of sets of compatible states. We have obtained results as good as the best exact method in the literature but with significantly better run-times. In addition to finding a reduced FSM, our algorithm also generates an I/O sequence that can be used as test vectors to verify the FSM´s implementation.

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 flow-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.

In this paper we propose a heuristic algorithm, *chesmin* (**che**cking **s**equence based state **min**imization) that minimizes large incompletely specified FSMs. The effectiveness of our approach is shown with a number of benchmark examples.

The paper is organized as follows. In the next section, we review the previous work in the literature. We present the preliminaries and an outline of the algorithm with an example in Section 3. The algorithm's performance is shown in Section 4. Section 5 concludes the work in this paper. Detailed pseudocode of the algorithm is given in the Appendix.

2: Previous Work

The state reduction problem for completely specified FSMs can be solved in polynomial time [1][2]. For *incompletely* specified FSMs (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. Paul and Unger [4] developed a general framework and proposed methods for generating the maximal compatibles and obtaining the minimal closed cover. Grasselli and 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 the 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. Higuchi and Matsunaga [7] 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 and Villa [8] proposed an exact state minimization technique using implicit enumeration of the compatibles. The implementation of their technique is called *ism*.

Pena and Oliveira [9] presented an exact state minimization algorithm based on mapping ISFSMs to tree FSMs (TFSMs) which combined Bierman's search algorithm [10] with Angluin's L* algorithm [11]. 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 [12]. A checking sequence is an I/O sequence which distinguishes an FSM from all other FSMs. Checking sequences are functional tests [13] that provide a fault and realization independent testing method for sequential circuits. In the next sections we will present our heuristic and then compare its performance with the exact algorithm *bica* and the heuristics *stamina* and *slim*.

3: Proposed Algorithm

3.1: Definitions

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

Definition 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, $q_0 \in Q$ is the *initial* state, $\delta(q,a)$: $Q \times \Sigma \rightarrow Q \cup \{\phi\}$ is the transition function, $\lambda(q,a)$: $Q \times \Sigma \rightarrow \Delta \cup \{\varepsilon\}$ is the output function. We will use $a \in \Sigma$ to denote a particular input symbol, $b \in \Delta$ a particular output symbol, ϕ to denote an unspecified transition, and ε to denote an unspecified output.

Definition 2: An output b_i is compatible with an output b_j ($b_i \overset{\bowtie}{=} b_j$) iff $b_i = b_j$ or $b_i = \varepsilon$ or $b_j = \varepsilon$.

Definition 3: Two states are *output incompatible* when, for some particular input, the two states produce a different output. Two states are *transitively incompatible* if, on the same input, they lead to incompatible states. Incompatibility function I:

$$I(q_{s_1}, q_{s_2}) = \begin{cases} 1 & \text{if } \exists a \;\; \lambda(q_{s_1}, a) \overset{\bowtie}{\neq} \lambda(q_{s_2}, a) \\ 1 & \text{if } \exists a \;\; \delta(q_{s_1}, a) = q_{d_1} \text{ and} \\ & \quad \delta(q_{s_2}, a) = q_{d_2} \text{ and} \\ & \quad I(q_{d_1}, q_{d_2}) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Definition 4: A set of states, C, is a compatible iff $\forall q_i, q_j \in C$ are pairwise compatible, i.e., $I(q_i, q_j) = 0$.

Definition 5: *Compatibility cover* is a set of compatibles $S = \{C_0, C_1, ..., C_k\}$ such that every state in Q belongs to <u>one or more</u> compatibles in S.

Definition 6: A set of states, denoted $IS(C, a)$, is an *implied set* of a compatible C for input a that are the next states reachable from the states in C, $IS(C,a) = \{q_d \mid \delta(q_s, a) = q_d, \forall q_s \in C\}$. A set S of compatibles is *closed* in M if, for each $C \in S$, all of its implied sets $IS(C, a)$ are contained in some element of S for each input a. (A minimum cardinality cover that is consistent with this covering and closure requirement is a solution for the exact state minimization problem of ISFSM [8][9].)

Definition 7: An output b is compatible with a set $B = \{b_0, b_1, ..., b_k\}$ of outputs iff b is output compatible with every element of B.

$$b \overset{\bowtie}{=} B \text{ iff } \;\; \forall b_i \in B \;\; b \overset{\bowtie}{=} b_i \tag{2}$$

Let $M' = (\Sigma, \Delta, Q', q_0', \delta', \lambda')$ be an incompletely specified FSM. Let $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ be the minimized FSM. Let S' be a set of compatibles and a closed cover in M'.

Definition 8: We overload the interpretation of the output function λ' such that $\lambda'(C',a)$: $S' \times \Sigma \rightarrow B$ where $B = \{b_i \mid b_i \in \Delta \cup \{\varepsilon\}\}$ is a set of outputs and C' is a compatible in M' and $C' \in S'$:

$$\lambda'(C', a) = B \text{ where } B = \{b_i \mid \lambda'(q', a) = b_i \; \forall q' \in C'\} \tag{3}$$

Definition 9: Similarly, we also overload the interpretation of the transition function δ' such that $\delta'(C',a) : S' \times \Sigma \rightarrow S'$ where C' is a compatible:

$$\delta'(C', a) = IS(C', a) \tag{4}$$

Definition 10: A function F: $Q \rightarrow S'$ is a valid one-to-one mapping function between the set of states of M and a closed set S' of compatibles in M' iff it satisfies:

$$\forall s \;\; \lambda(q_0, s) \overset{\bowtie}{=} \lambda'(F(q_0), s) \tag{5}$$

$$\forall s \;\; F(\delta(q_0, s)) = \delta'(F(q_0), s) \tag{6}$$

The equation (5) states that function F maps each state q to a compatible C' in M' that satisfies the output of M' for every possible input sequence s. The next equation (6) states that the implied set of a compatible is correctly mapped. These output and transition requirements for a valid mapping function F are depicted in Figure 1. In this figure, it is shown that if C_d' is the implied set of a compatible 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_s and the output set B and output b have to be compatible.

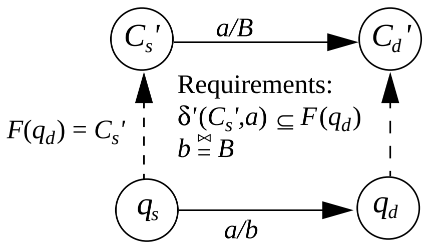
C_s'
a/B
C_d'
Requirements:
$\delta'(C_s',a) \subseteq F(q_d)$
$b \overset{\bowtie}{=} B$
$F(q_d) = C_s'$
q_s
q_d
a/b

Figure 1: Output and transition requirements for a valid mapping function *F*.

We will use the same example given in [9] and shown in Figure 3a to demonstrate the mapping between FSMs M' and M. State q_0 of M is mapped to the compatible $C_0' = \{q_0', q_2'\}$ 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. $\lambda'(C_1',0) = \{0\}$ 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, therefore the function F is a valid mapping function for M and M', and as a result M is compatible with M'.

3.2: How The Algorithm Works

In this section we will explain how the algorithm works by referring to the variables and procedures of the pseudocode given in the Appendix section (Figure 4.) Our state reduction algorithm constructs a *consistent* FSM, M, with no more than *upperBound* number of states by traversing the specification FSM, M', and fitting specification transitions which are obtained from the constructed input sequence, *seq*, into the FSM under construction, M. We define “consistent” in terms of an FSM generating an output sequence that is compatible with the output sequence generated by M' on the same input sequence. The *upperBound* is initially set to one less than the number of states of M'.

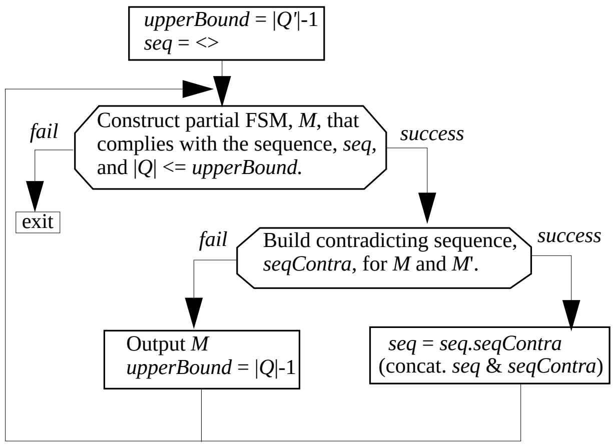
upperBound = |Q'|-1
seq = <>
Construct partial FSM, M, that complies with the sequence, seq, and |Q| <= upperBound.
fail
success
exit
Build contradicting sequence, seqContra, for M and M'.
fail
success
Output M
upperBound = |Q|-1
seq = seq.seqContra
(concat. seq & seqContra)

Figure 2: Flow diagram of the algorithm. $|Q'|$: number of states of M'. $|Q|$: number of states of M.

The flow diagram of the algorithm is shown in Figure 2. The loop in the diagram corresponds to the search for consistent FSM M that gives compatible output with the FSM M' for an input sequence. This input sequence is incrementally constructed by concatenating the contradicting sequence, *seqContra*, found when M and M' are incompatible. If M is found to be compatible with M', we output M. After outputting M, the search for a better solution continues with a *upperBound* equal to $|Q|$ - 1, where $|Q|$ is the number of states of M.

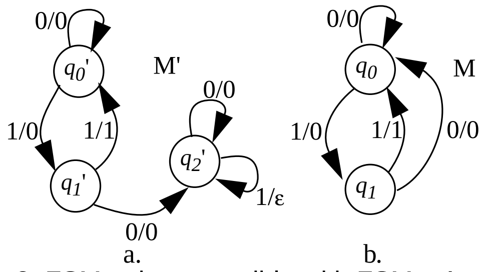

Figure 3: FSM M is compatible with FSM M'. $F(q_0) = \{q_0', q_2'\}$ and $F(q_1) = \{q_1', q_2'\}$.

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 the deriving entire checking sequence. Checking sequence generation requires that all states are reachable and connected, therefore we augment the FSM to get out of deadlock states and to reach every state. In the main program of the pseudocode, the specification FSM, M', is augmented by adding a reset state and transitions (*AugmentResetState* procedure in Figure 4.) We introduce this extra state and its transitions to take care of deadlock and unreachable states as the example below illustrates.

Next, we will explain how the algorithm works step by step with an **example** by using the FSM shown in Figure 3a:

	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 1: Original flow-table of the specification M'.

	00	01	10
q_0'	q_0', 00	q_1', 00	q_r', 10
q_1'	q_2', 00	q_0', 01	q_r', 10
q_2'	q_2', 00	q_2', 0ε	q_r', 10
q_r'	q_0', 10	q_0', 10	q_r', 10

Table 2: Flow-table of the augmented specification M'.

Step 1: First we augment the ISFSM in Figure 3a (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 newly added state q_r' should be incompatible with every state q_0', q_1', q_2'. In order to make the newly added state incompatible, we introduce an extra input and an output. flow-table representation of the augmented FSM is shown in Table 2. In Table 2, the leftmost input and output bits are newly added. The leftmost output bit of each of the original transitions 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 representation of the augmented FSM under construction is shown in Table 3.

	00	01	10
q_0	ϕ, εε	ϕ, εε	q_r, 10
q_r	q_0, 10	q_0, 10	q_r, 10

Table 3: Step 1.

	00	01	10
q_0	q_0, 00	ϕ, εε	q_r, 10
q_r	q_0, 10	q_0, 10	q_r, 10

Table 4: Step 2.

	00	01	10
q_0	q_0, 00	q_1, 00	q_r, 10
q_1	q_0, 00	ϕ, εε	q_r, 10
q_r	q_0, 10	q_0, 10	q_r, 10

Table 5: Step 3.

	00	01	10
q_0	q_0, 00	q_1, 00	q_r, 10
q_1	q_0, 00	q_0, 0ε	q_r, 10
q_r	q_0, 10	q_0, 10	q_r, 10

Table 6: Step 4.

	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 7: Step 5.

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

Table 8: Step 6.

After adding a reset state and transitions, a search for FSM *M* that is consistent with the input sequence (*seq*=<>) is started by means of a branch and bound procedure, *FindConsistenFSM* shown in Figure 4. This procedure is recursive. Every recursion corresponds to a state transition and the depth of recursion is the length of I/O sequence. The I/O sequence is incrementally constructed by using the procedure *FindContradictingSequence* shown in Figure 4. 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 randomly chosen during the breadth-first search.

#	0	1	2	3	4	5	6	7	8	9
I	10	00	01	00	01	01	10	00	01	01
O	10	00	00	00	0ε	0ε	10	10	00	01
$S_{M'}$	q_r'	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_1	q_0	q_1	q_0	q_r	q_0	q_1	q_0
	Step1	Step2	Step3		Step4		Step5			

Table 9: Incrementally constructed I/O sequence. #: Sequence number, I: Input, O: Output:, $S_{M'}$: Current state in *M'*, S_M: Current state in *M*.

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* nd *M'*. When this sequence (*seq*=<10/10>) 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 (*seqContra*=<00/00>) is found since *M'* gives an output “00,” whereas *M* gives an undefined output and next-state. We concatenate *seqContra* to *seq* (*seq*=<10/10, 00/00>.) 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 (*seqContra*=<01/00>) 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 as per Def. 3 ($\lambda'(q_0',1)=0$ and $\lambda'(q_1',1)=1$.) Therefore we introduce a new state q_1 and set $F(q_1)$ to $\{q_1'\}$. We fill the flow-table for *M*. Then we check whether *M* and *M'* are compatible by calling the *FindContradictingSequence*. A contradicting sequence (*seqContra*=<00/00>) 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 <10/10,

BENCHMARKS			N_{final}				RUN-TIME in seconds (s.)				COMPARISON N_{final} / Run-time		
FSM	*PI*	N_{init}	*bica*	*stamina*	*slim*	*chesmin*	*bica*	*stamina*	*slim*	*chesmin*	*vs. bica*	*vs. stamina*	*vs. slim*
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	4	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	12.52	/ +	+ /	/ –
rubin1200	1	1200	3	FAILS	3	3	229.26	FAILS	17.96	95.30	/ +	+ /	/ –
rubin2250	1	2250	3	FAILS	3	3	1384.98	FAILS	66.33	616.04	/ +	+ /	/ –

Table 10: Experimental results. *PI*: Number of primary inputs. N_{init}: Initial number of states. N_{final}: Final number of states. 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.

00/00, 01/00, 00/00>.

Step 4: Then we call the *FindContradictingSequence* procedure to check whether *M* (Table 5) and *M'* are compatible. A contradicting sequence (*seqContra*=<01/0ε, 01/0ε>) 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.

Step 5: We call the *FindContradictingSequence* procedure to check whether *M* (Table 6) and *M'* are compatible. It returns a contradicting sequence (*seqContra*=<10/10, 00/10, 01/00, 01/01>.) 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 (*seqContra* = <>.) 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 Figure 3b. The constructed I/O sequence, *seq*, and corresponding states that *M* and *M'* are in are given in Table 9. Note that this sequence does not have to be a complete checking sequence.

4: Results and Performance

To evaluate our algorithm we used the same set of problems used in previous work [7][9]. These problems come from a variety of sources: Standard benchmarks, asynchronous synthesis, learning problems, and synthesis of interacting FSMs.

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/133MHz 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* [7] presented run-times of *slim* and *stamina* on their platform. Since we have *stamina*'s run-times on our platform, we have scaled *slim*'s run-times by a factor of "*stamina*-run-time-on-our-platform / *stamina*-run-time-on-[7]." 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.

In Table 10 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.

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 3 of them, it ran slower.
- In 2 cases, *chesmin*'s solutions had one more state (vbe4a, th.55.)
- However, one of the above cases (th.55) *chesmin* ran 675X faster than *bica*.
- In one case (fo.70), *bica* **failed** and *chesmin* found a solution.
- *Chesmin* on the average ran 46X faster than *bica.*

From our empirical results, we have observed that chesmin runs in polynomial between $O(N_{init}^2)$ and $O(N_{init}^3.)$ 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* 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 FSM's implementation.

Appendix

```
main begin
    seq = <initialInput>;
    upperBound = |Q'| - 1;
    M' = ReadKiss();
    M = NewFSM();
    //add extra reset and transitions
    M'.AugmentResetState(); M.AugmentResetState();
    $F(q_r) = \{q_r'\}$;
    FindConsistentFSM($q_r$, $q_0$, $q_r'$, $q_0'$, upperBound, seq, 0);
end

FindContradictingSequence($q_s$, $q_s'$, input) begin
    seqContra = <>;
    ($q'$, $i$) = GetContradictingTransition(($q_s'$, input), ($q_s$, input));
    while (($q'$, $i$) $\neq$ ($q_s'$, input)) begin
        seqContra = seqContra.i;
        ($q'$, $i$) = Parent($q'$, $i$);
    end
    return seqContra;
end
```

Figure 4: Pseudocode of *chesmin.*

```
**GetContradictingTransition**($(q_s', input), (q_s, input)$) **begin**
    $queue_{M'} = \{(q_s', input)\}$;
    $queue_M = \{(q_s, input)\}$;
    **while** ($queue_{M'} \neq EMPTY$) **begin**
        $(q', i_{parent}) = PopHead(queue_{M'})$;
        $(q, i_{parent}) = PopHead(queue_M)$;
        $MarkAsVisited(q', q, i_{parent})$;
        $q_d' = \delta'(q', i_{parent})$; $q_d = \delta(q, i_{parent})$;
        **foreach** $i_{child} \in Adjacent(i_{parent})$ **begin**
            **if** ($\delta'(q_d', i_{child}) == \phi$) **continue;**
            **if** ($Visited(q_d', q_d, i_{child}) ==$ TRUE) **continue;**
            $output = \lambda(q_d, i_{child})$;
            $output' = \lambda'(q_d', i_{child})$;
            **foreach** bit $i$ **begin**
                **if** (($output'[i] \neq \varepsilon$) **and** ($output[i] \neq output'[i]$)) **begin**
                    $Parent(q_d', i_{child}) = (q', i_{parent})$;
                    **return** $(q_d', i_{child})$;
                **end**
            **end**
            $Push((q_d', i_{child}), queue_{M'})$; $Push((q_d, i_{child}), queue_M)$;
            $MarkAsVisited(q_d', q_d, i_{child})$;
        **end**
    **end**
**end**

**FindConsistentFSM**($q_s, q_d, q_s', q_d', upperBound, seq, index$) **begin**
    **if** ($TimeOut()$) **exit;**
    **if** ($|Q| > upperBound$) **return;**
    **if** ($index == |seq|$) **begin**
        $seqContra = FindContradictingSequence(q_s, q_s', seq[index-1])$;
        **if** ($seqContra == <>$) **begin**
            //found a compatible FSM with $M'$
            $Output(M, RunTime())$;
            $upperBound = |Q| - 1$;
            **return;**
        **end**
    **else** $seq = seq.seqContra$; //concatenate
    **end**
    $input = seq[index]$;
    **if** ($\exists q'$ such that $q' \in F(q_d)$, $I(q_d', q') == 1$) **return**;
    $output = \lambda'(q_s', input)$;
    **if** ($\delta(q_s', input) \neq \phi$) and $\delta(q_s, input) \neq q_d$) **return**;
    **if** ($\lambda(q_s, input) \overset{\bowtie}{\neq} output$)) **return;**
    //See Table 11 for definition of $\otimes$.
    $\lambda(q_s, input) = \lambda(q_s, input) \otimes output$;
    $\delta(q_s, input) = q_d$;
    $F(q_s) = F(q_s) \cup \{q_s'\}$;
    $index{+}{+}$;
    **if** ($|Q| < upperBound$) $Q = Q \cup q_{new}$;
    $input = seq[index]$;
    $q' = \lambda'(q_d', input)$;
    $output = \lambda(q_d, input)$;
    $q_{dd} = \delta(q_d, input)$;
    **foreach** $q \in Q$ **begin**
        $F_{qd} = F(q_d)$;
        $FindConsistentFSM(q_d, q, q_d', q', upperBound, seq, index)$;
        $\lambda(q_d, input) = output$; //Undo
        $\delta(q_d, input) = q_{dd}$; //Undo
        $F(q_d) = F_{qd}$; //Undo
    **end**
    $index{-}{-}$;
**end**
```

Figure 4: Pseudocode of *chesmin* continues.

$\otimes$	0	1	ε
0	0	N/A	0
1	N/A	1	1
ε	0	1	ε

Table 11: $\otimes$ operation. N/A: Not Applicable.

References

[1] D.A. Huffmann, “The Synthesis of Sequential Switching Circuits,” *Journal of the Franklin Institute*, vol. 257, no. 3, pp. 161-190, 1954.

[2] J.E. Hopcroft, “NlogN Algorithm for Minimizing States in Finite Automata,” Tech. Rep. CS 71/190, Stanford University, 1971.

[3] C.F. Pfleeger, “State Reduction in Incompletely Specified Finite State Machines,” *IEEE Trans. on Computers*, vol. C-22, pp. 1099-1102, 1973.

[4] M.C. Paull and S.H. Unger, “Minimizing the Number of States in Incompletely Specified Sequential Switching Functions,” *IRE Trans. on Electronic Computers*, vol. EC-8, pp. 356-367, 1959.

[5] F. Luccio, “Extending the Definition of Prime Compatibility Classes of States in Incompletely Specified Flow Tables with the Help of Prime Closed Sets,” *IEEE Trans. on Electronic Computers*, pp. 953-956, 1969.

[6] J.-K. Rho, G. Hachtel, F. Somenzi, and R. Jacoby, “Exact and Heuristic Algorithms for the Minimization of Incompletely Specified Finite State Machines,” *IEEE Trans. on Computer-Aided Design*, vol.13, no.2, pp. 167-177, 1994.

[7] H. Higuchi, Y. Matsunaga, “A Fast State Reduction Algorithm for Incompletely Specified Finite State Machines,” *Design Automation Conference*, pp. 463-466, 1996.

[8] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, “Synthesis of FSMs: Functional Optimization,” *Kluwer Academic Publishers*, 1997.

[9] J.M. Pena, A.L. Oliveira, “A New Algorithm for Exact Reduction of Incompletely Specified Finite State Machines,” *International Conference on Computer Aided Design*, pp. 482-489, 1998.

[10] A.W. Biermann and J.A. Feldman, “On the Synthesis of Finite State Machines from Samples of Their Behavior,” *IEEE Trans. on Computers*, vol. 21, pp. 592-597, 1972.F

[11] D. Angluin, “Learning Regular Sets from Queries and Counter Examples,” *Inform. Comput.*, vol. 75, no. 2, pp. 87-106, 1987.

[12] S. Gören, F.J. Ferguson, “Checking Sequence Generation for Asynchronous Sequential Elements,” *Int. Test Conference*, pp. 406-413, 1999.

[13] F.C. Hennie, “Fault Detecting Experiments for Sequential Circuits,” *Int. Symp. on Swit. Circuit Theory and Logic Design*, pp. 95-110, 1964.