Cent. Eur. J. Comp. Sci. • 3(2) • 2013 • 91-103
DOI: 10.2478/s13537-013-0106-0

VERSITA

Central European Journal of **Computer Science**

Iterative minimization of partial Finite State Machines

Research Article

Alex D.B. Alberto*, Adenilso Simao†

Instituto de Ciências Matemáticas e Computação,
Universidade de São Paulo, São Carlos, Brazil

Received 16 May 2011; accepted 9 June 2013

Abstract: Finite state machines have been used in many contexts, e.g., test generation, system modeling, language processing. In these applications, the size of the machine has a great impact on the complexity and quality of the results. Thus, it is desirable to obtain a machine without redundant states, i.e., to minimize the machine. However, the minimization of partially specified finite state machines is a known NP-complete problem, and many heuristics to obtain feasible solutions have been proposed. In this paper, we propose an approach to minimize partial finite state machines based on the selection of compatible states using maximum cliques on the distinction graph. We experimented with our method comparing it with previous methods and the data suggest that our approach is more efficient, generating good results in less time.

Keywords: minimization • heuristic • incompletely specified • partial • finite state machines

1. Introduction

Finite State Machines (FSMs) are often used in systems engineering practices, like design modeling and test generation. Such formal models are conceptually simple, but yet expressive enough to be of pragmatical usefulness. The number of states are usually a dominating parameter when evaluating the cost of using FSMs. Frequently, a model in use is unreduced, i.e, it contains states which can be merged without changing the behavior of the machine. These unreduced FSMs are result of faulty designs: for instance, it may be automatically generated from another model, and states of identical behavior may be introduced. Moreover, even when the FSM is designed by hand, the modeler may overlook some redundancy, yielding unnecessary states. In these situations, a reduced form of the FSM must be obtained.

Although the problem of minimizing a complete FSM can be solved in polynomial time [9], the minimization of partial FSMs, on the other hand, is known to be NP-complete [12]. The search for feasible solutions has received great attention, and several works on the subject were published over the last decades [2, 4–7, 11]. In order to mitigate the combinatorial explosion and provide good empirical results, many heuristic approaches were introduced.

In this paper, we propose a method for minimizing partial FSMs. Based on a distinction graph, we search for groups of pairwise adjacent nodes and use such information to identify pairs of compatible states, which may be merged without

* *E-mail: alex@icmc.usp.br*
† *E-mail: adenilso@icmc.usp.br*

affecting the input/output behavior of the FSM. In an iterative process, states are merged until a complete distinction graph is built or no information is left to go any further. We implemented the method and compared the results obtained by running a set of well-known FSM examples in our implementation and implementations of two other methods. We also experimented with the methods working on randomly generated FSMs. The results suggest that our method runs faster and generate results as good as the others in most cases. Thus, the experimental data demonstrate that, although there are many methods for tackling the problem of partially specified FSM minimization, the investigation of new heuristics, as those used by our method, is worthy.

This paper is organized as follows. In Section 2, we present previous work and discuss how it relates to the proposed method. In Section 3 there are some definitions of concepts used in this paper. In Section 4, we present the method. In Section 5, we present an example of the application of the method. In Section 6, the results of an empirical experiment are shown. Finally, concluding remarks and future work are discussed in Section 7.

2. Related work

The problem of minimizing arbitrary Partial FSMs (PFSM) is difficult. Pfleeger [12] shows how the graph coloring problem, a known polynomial complete (NP-complete) problem, is reducible to the problem of minimizing incompletely specified finite state automaton.

Paull and Unger [10] propose a method based on the selection of compatibility classes of states, known as the classic approach to minimize PFSMs. From their definition, states which are in a particular compatibility class can be merged without affecting the behavior of the FSM.

Grasselli and Lucio [2] improve the classic approach by observing that only a subset of the classes should be considered, resulting in noticeable optimization on the search for the minimum cardinality set. Hatchel et al. [5] discuss the feasibility of solving real world instances of PFSM minimization problem, providing an approach to Grasselli and Lucio's method based on an explicit enumeration of compatibility classes. Their method, known as `stamina`, exhibits good performance on processing hand-designed PFSM. Kam et al. [7] propose an approach based on implicit enumeration. The main advantage of their method, called `ism`, from implicit state minimizer, appears when minimizing FSMs generated by processes of logic synthesis of real design applications, which uses large number of states. In such instances, space limitations of a explicit enumeration strategy can be exceeded.

Higuchi and Matsunaga [6] propose a new heuristic state reduction algorithm for PFSM based on iterative improvements. Techniques were mixed, using explicit enumeration to maximize performance when the number of compatibility classes is likely to be under a given threshold. Over the threshold, binary decisions diagram are used, avoiding combinational explosion.

A different approach is proposed by Pena and Oliveira [11]. Not based on the enumeration of compatibility classes, the method takes advantages from well-known techniques for FSM identification. Therefore its performance is not directly related to the number of the compatibility classes on the problem instance. The comparisons presented on that paper show that `bica`, an implementation of the method, behaves more robustly than the others based on enumeration for most of the evaluated problems, solving instances for which `ism` and `stamina` fails.

Gören and Ferguson [4] propose a heuristic based on checking sequence generation and identification of sets of compatible states. They present a comparison showing that their implementation of the heuristic, named `chesmin`, exhibited good results when compared with previous exact methods for many problems.

For the approach proposed in this paper, we define a heuristic based on information yielded by cliques on distinction graphs. None of the previous works uses such information on similar way. The unique reference in the use of distinction graphs can be found on Pena and Oliveira [11] work, where the authors use the maximum cardinality clique size to estimate the number of states on a reduced FSM.

3. Finite State Machines

This section contains formal definitions of the FSM model along with concepts which are used in this paper.

Definition 1.
An FSM M is a 6-tuple $(\Sigma, \Delta, Q, q_0, \delta, \lambda)$ where

- Σ: the finite, non-empty set of input symbols,
- Δ: the finite, non-empty set of output symbols,
- Q: the finite, non-empty set of states,
- $q_0 \in Q$: the initial state,
- ϕ denotes an unspecified state,
- ϵ denotes an unspecified output,
- $\delta(q, a) : Q \times \Sigma \rightarrow Q \cup \{\phi\}$: the transition function,
- $\lambda(q, a) : Q \times \Sigma \rightarrow \Delta \cup \{\epsilon\}$: the output function.

A completely specified FSM has no transitions leading to unspecified states. On the other hand, an FSM which has one or more undefined transitions is said to be an incompletely specified FSM, or partial FSM.
We use $x \in \Sigma$ and $\alpha \in \Sigma^*$ to denote a particular input symbol and a sequence of input symbols, respectively. In a similar way, $y \in \Delta$ and $\beta \in \Delta^*$ are used to denote a particular output symbol and a sequence of output symbols, respectively. We refer to sequences of input symbols and sequences of output symbols as input sequences and output sequences. We also denote a particular state as $q \in Q$.
A transition is defined in an FSM M for a state q and an input symbol a if $\delta(q, a) \neq \phi$. An input sequence $\alpha = x_1 ... x_k, x_i \in \Sigma$ is defined for a state q if exists a sequence $q_1, ..., q_k$ of states where $q = q_1$, $q_{i+1} \neq \phi$ and $\delta(q_i, x_i) = q_{i+1}$, for all $1 < i < k$.
It is convenient to extend the definition of transition and output functions in order to allow them to be applied from a sequence of inputs. Therefore, we use $Q \times \Sigma^*$ as the domain of both functions. Given an input sequence α and a state q, $\delta(q, \alpha)$ results in the final state reached after sequentially applying all input symbols in α into an FSM currently in state q, and $\lambda(q, \alpha)$ results in the output sequence β exhibited by such action.
Given a state $q \in Q$, $\Omega(q)$ denotes the set of all defined input sequences for a state q. Ω_M is an alternative form of representing all the input sequences defined for an FSM M, or $\Omega(q_0)$.
Two states q_i and q_j are distinguishable if there exists $\alpha \in \Omega(q_i) \cap \Omega(q_j)$, such that $\lambda(q_i, \alpha) \neq \lambda(q_j, \alpha)$, i.e., there is at least one input sequence, defined for both states, that produces distinct output sequences. We use the notation $q_i \nsim q_j$ to denote that q_i and q_j are distinguishable states. Two states q_i and q_j are redundant if they show the same input/output behavior, i.e., they are not distinguishable. We use the notation $q_i \sim q_j$ to denote the states are redundant.
An FSM M is *reduced* if for all $q_i, q_j \in Q$, $q_i \neq q_j$ implies $q_i \nsim q_j$, i.e., all the states are pairwise distinguishable. An FSM $M' = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$ is *quasi-equivalent* to an FSM M if for all $\alpha \in \Omega_{M'}$, we have that $\lambda'(q'_0, \alpha) = \lambda(q_0, \alpha)$, i.e., for every input sequence that is accepted by M', M and M' produce the same output sequence. It implies that $\Omega_M \subseteq \Omega_{M'}$.
An FSM M is *initially connected* if for all $q \in Q$, there exists $\alpha \in \Sigma^*$, such that $\delta(q_0, \alpha) = q$, i.e., every single defined state is reachable through the initial state q_0. In this paper, only initially connected FSMs are considered, since any state which cannot be reached from the initial state can be removed without changing the machine's behavior.

4. Minimization method

In this section, we present the main steps of the proposed method (Algorithm 1). The FSM in Figure 1 is used to illustrate the main features of the method. A characteristic of the method is that instead of finding all sets of compatible states at once, subsets of compatible states are used to iteratively reduce the size of the PFSM. In this way, the PFSM gets smaller after each step, and the task of finding compatible states are simpler.
Our approach to eliminate redundant states from PFSM models consists in the following steps: (*i*) construction of the distinction graph, (*ii*) selection of compatible states and (*iii*) merge of selected states. The steps are applied iteratively, until the selection does not result in any more compatible states. Such procedure is represented by the main loop, in line 4 of Algorithm 1.
First, we describe the structure named *distinction graph*. A distinction graph for a given FSM is a conventional graph model, where the node set is the state set and the edge set contains each pair of distinguishable states of that FSM.

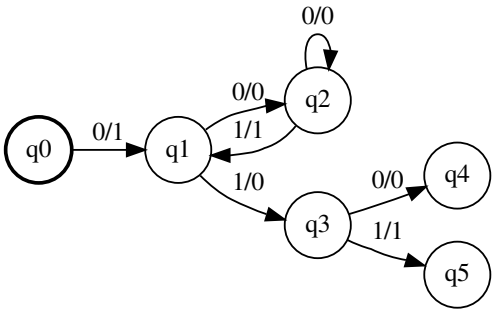

Figure 1. A partially specified FSM.

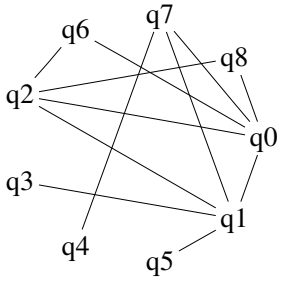

Figure 2. Distinction graph for the FSM in Figure 1.

Definition 2.

The graph $G = (V, E)$ is a *distinction graph* for M if, and only if, $V = Q$, $E \subseteq Q \times Q$ and for all $q_i, q_j \in Q, q_i \nsim q_j \Leftrightarrow (q_i, q_j) \in E$.

Figure 2 shows the distinction graph for the FSM of Figure 1. The construction of a distinction graph for an FSM is accomplished by iterating through all pairs of states and checking the existence of a sequence to distinguish them. Whenever such sequence is found, the pair of states is added to the edge set. The distinction graph building process is embedded in the procedure *BuildDistinctionGraph*, in Line 3 of Algorithm 1.

A graph G is *complete* if all its vertices are pairwise adjacent, i.e., for all $q_i, q_j \in V$, $q_i \neq q_j$ implies that $(q_i, q_j) \in E$. A complete distinction graph will always refer to a reduced FSM, since all states must be pairwise distinguishable.

Given a subset $S \subseteq V$, $G(S) = (S, E \cap S \times S)$ is the *subgraph* induced by S. The subset S is said to be a *clique* if the induced graph $G(S)$ is complete. The number of vertices in the subset S is the *cardinality* of the clique, denoted as $|S|$. Cliques in a distinction graph corresponds to subsets of pairwise distinguishable states in the FSM. Let $n = |S|$ be the cardinality of a clique S in a distinction graph of an FSM M. The existence of such a clique is enough to ensure that the reduced form of M must use at least n states. Considering this, estimating the number of states in the reduced form of an FSM M corresponds to finding out the largest clique cardinality in the distinction graph.

Our method uses cliques of maximum cardinality to select potential compatible states. Searching for such cliques is a well-known NP-Hard problem. However, the use of heuristics and hybrid algorithms showed overall success on solving instances of real-world inputs [1]. In the Algorithm 1, the search for maximal cliques appears in Lines 7 and 13.

Given a maximal clique $S \subset V$ in a distinction graph $G = (V, E)$, any node $q \in B \backslash S$, is related to a potential compatible state. For a node $q_a \in S$, let $C_{q_a} = \{q \in V \mid (q_a, q) \notin E\}$, i.e., a set of nodes not adjacent to q_a, consequently, not in the clique. Potentially, the nodes in C_{q_a} can be combined to the node q_a, resulting in a quasi-equivalent FSM with less states. We now introduce the operation used to merge states. The C_{q_a} set building process is represented in Algorithm 1 by calls to *GetCompatibleStates* procedure, as shown in Lines 8 and 16.

Given an FSM $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ and a pair of states $q_i, q_j \in Q, q_i \sim q_j$, let $J = \{(q_a, q_b) \in Q \times Q \mid \exists \alpha \in \Omega(q_i) \cap \Omega(q_j), \delta(q_i, \alpha) = q_a \wedge \delta(q_j, \alpha) = q_b \wedge q_a \neq q_b\}$, i.e., the set of all pairs of states reached through the common

defined input sequences from q_i and q_j. We define an equivalence relation on J, such that $[q_a] = \{q_b \in Q \mid (q_a, q_b) \in J\}$.

Definition 3.
To *merge* the pair (q_i, q_j) is to produce a new FSM $M' = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$, quasi-equivalent to M, where the input/output behavior of all states in each equivalence class $[q_a]$ is achieved by just one state $q'_a \in Q'$. We denote such operation as $M' = M/(q_i, q_j)$.

This merge operation is used in Lines 36 and 39 of the Algorithm 1. The quasi-equivalence of M' is guaranteed if two conditions are observed: (i) each single state of M is in one, and just one, equivalence class; (ii) given a defined equivalence class $[q_a]$ and an input symbol $x \in \Sigma$, for all $q_b \in [q_a]$, $\delta(q_b, x) = q$, $q \in [q_c]$, i.e., any defined transition for an input symbol from any state in the equivalence class should lead to states which are in a unique class $[q_c]$. These restrictions are derived from the classic approach to minimize PFSM [10]. Since the classes defined by the merging operation do not exclude states, the restriction (i) will be always observed. However, the restriction (ii) needs to be checked. When (ii) is satisfied, the operation is said to be successful and the new generated FSM takes place to continue the process. Otherwise, the operation is aborted and another pair, if available, is selected and used. In Algorithm 1, the restriction check occurs in Line 35.
The classic approach [10] uses exhaustive search to build compatibility classes, checking for the two restrictions. On the other hand, we use a heuristic based on the analysis of adjacent nodes in the distinction graph G. Let $N_G(q_a) = \{q \in V \mid (q, q_a) \in E\}$, i.e., all nodes adjacent to q_a in G. Let $K_m(S)$ be a set of all subsets of S with size m ($1 \leq m < n$, $m \in \mathbb{Z}$, $n = |S|$). For all $k \in K_m(S)$, we define the potential compatible state set $C_k = \{q \in V \mid N_G(q) \cap S = S \backslash k\}$, i.e., the nodes in V which are adjacent to all nodes in S, except those in the subset $k \subset V$. We refer to m as the *depth* of the state selection procedure. Starting with $m = 1$, each $k_i \in K_1(S)$ corresponds to a subset containing the node $q_i \in S$. Through the process, whenever selection results in empty C_k sets, m is increased until the maximum value of $m = n - 1$ is reached. The subsets in $K_m(S)$ are m-combinations of the elements in S.
The set $P = \{(q_i, q_j) \in k \times C_k \mid k \in K_m(S)\}$ contains all pairs which are candidates to trigger a successful merging operation. The potential of such pairs to lead to equivalence classes which do not violate the restriction (ii) decreases as the depth m used on the selection increases. The procedure used to build the P set is represented in Algorithm 1 by the loop starting in Line 26
In particular, when $m = 1$, given $k = \{q_a\}$, for all states $q \in C_k$, the merge operation (see Definition 3) $M' = M/(q_a, q)$ is always successful and we say that the state selection is in *exact mode*. While in this mode, all state q is selected to be merged with q_a due to a node on the graph which is adjacent to all nodes on the clique, except q_a, it means that such state q is incompatible to any other state, except q_a. When $m > 1$, the restriction (ii) may be violated for some pairs (q_a, q), so it needs to be checked in the resulting equivalence classes of each merge attempt. In exact mode, all selected states $q_a \in C_k$ are merged in the same iteration. When $m > 1$, just the first successful merge is accomplished, and the next iteration begins.
Due to the iterative nature of the method, it will only advance until there are compatible states to be merged. When no further minimization can be applied, it terminates and produces the best result found. If there are no compatible states on a given input, the output will be the exact same machine provided as the input.

5. Example

In this section, we illustrate the application of the proposed method. Given a PFSM, as shown in Figure 1, the first step is to build the distinction graph, which is shown in Figure 2.
The graph is not complete, allowing the beginning of the reduction process. At first, we need to search for maximum cliques. There are four of them: $S_1 = \{q_0, q_1, q_2\}$, $S_2 = \{q_0, q_1, q_7\}$, $S_3 = \{q_0, q_2, q_6\}$ and $S_4 = \{q_0, q_2, q_8\}$. Using the first found clique, we begin the selection. Starting with $m = 1$, $K_1(S_1) = \{\{q_0\}, \{q_1\}, \{q_2\}\}$, and for all $k \in K_1(S_1)$, the C_k sets are: $C_{\{q_0\}} = \{\emptyset\}$, $C_{\{q_1\}} = \{q_6, q_8\}$ and $C_{\{q_2\}} = \{q_7\}$. Since the selection is in exact mode, all pairs from the selected states in C_k, $P = \{(q_1, q_6), (q_1, q_8), (q_2, q_7)\}$, can be merged without checking for any restriction.
The FSM generated after merging the states is shown in Figure 3. All the states which are in the C_k sets where replaced by the state in k. For instance, the q_1 state now implements all the defined sequences of states q_1, q_6 and q_8 from the FSM in Figure 1. The new FSM has four less states, nevertheless it is quasi-equivalent to the

Algorithm 1 Minimization Procedure Algorithm

Require: an FSM M to be minimized
Ensure: an FSM M', quasi-equivalent to M, with less states

```
1:  function MINIMIZE(M)
2:      M' ← M
3:      G ← BuildDistinctionGraph(M')
4:      while not Complete(G) do
5:          S ← ∅;                                              ▷ Start exact mode search
6:          repeat
7:              S ← NextMaximalClique(G, S)
8:              C ← CompatibleStates(G, S, 1)
9:          until S = ∅ or (C_k ≠ ∅, k ∈ K_1(S))
10:         if (C_k = ∅, k ∈ K_1(S)) then                       ▷ Check resulting compatible states from exact mode
11:             S ← ∅;                                          ▷ If none found, start to search in approximate mode
12:             repeat
13:                 S ← NextMaximalClique(G, S)
14:                 m ← 2
15:                 repeat
16:                     C ← CompatibleStates(G, S, m)
17:                     m ← m + 1
18:                 until m > |S| or (C_k ≠ ∅, k ∈ K_m(S))
19:                 m ← m − 1
20:             until S = ∅ or (C_k ≠ ∅, k ∈ K_m(S))
21:             if then(C_k = ∅, k ∈ K_m(S))                    ▷ If there are no compatible state yet
22:                 return M'                                   ▷ Nothing else can be done
23:             end if
24:         end if
25:         P ← ∅
26:         for all C_k, k ∈ K_m(S) do
27:             for all q_i ∈ k, q_j ∈ C_k do
28:                 if (q_i, q_j) ∉ P then
29:                     P ← P ∪ {(q_i, q_j)}
30:                 end if
31:             end for
32:         end for
33:         for all (q_i, q_j) ∈ P do
34:             J ← GetEquivalenceClasses(M', q_i, q_j)
35:             if m > 1 and CheckRestrictions(J, M') then
36:                 M' ← M'/(q_i, q_j)                          ▷ Applying merge operation (see Definition 3)
37:                 break                                       ▷ Not in exact mode, merge should be done just for the first pair
38:             end if
39:             M' ← M'/(q_i, q_j)
40:         end for
41:         G ← BuildDistinctionGraph(M')
42:     end while
43:     return M'
44: end function
```

original FSM. We rebuild the graph to match the new machine, as also shown in Figure 3. The graph rebuilding process does not require the full analysis of the modified FSM, the nodes of the merged states may be merged as well. Since the rebuilt graph is not complete, a second iteration begins. We should restart the state selection procedure with a search for maximum cliques. However, we first attempt to reuse the clique found in the previous iteration: $S_1 = \{q_0, q_1, q_2\}$, which, in this case, is the only valid one. Starting with $m = 1$, that results in the same $k \in K_1(S_1)$ sets, we build the new C_k sets, but they're all empty: there is no node in S_1 which is adjacent to all but one node in S_1. Thus, we increase the depth to $m = 2$: $K_2(S_1) = \{\{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}\}$. For all $k \in K_2(S_1)$ we build

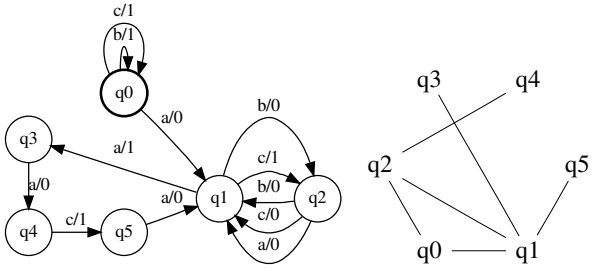

Figure 3. Reduced FSM after first iteration and its distinction graph.

the C_k sets: $C_{\{q_0,q_1\}} = \{q_4\}$, $C_{\{q_0,q_2\}} = \{q_3, q_5\}$ and $C_{\{q_1,q_2\}} = \{\emptyset\}$. We have the following potential compatible pairs: $P = \{(q_0, q_3), (q_0, q_4), (q_0, q_5), (q_1, q_4), (q_2, q_3), (q_2, q_5)\}$.

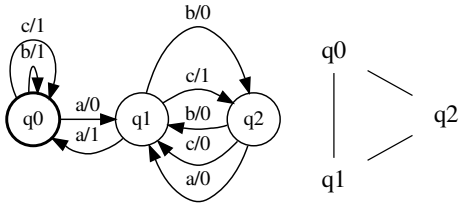

Figure 4. Reduced FSM after second iteration and its distinction graph.

Since $m > 1$, the restriction (ii) needs to be checked in the resulting equivalence classes from the merge operation. When not in exact mode, the iteration ends when the first successful merge occurs. The first merge attempts are done in order, so, the first one uses the pair (q_0, q_3). The operation $M' = M/(q_0, q_3)$ results the following propagation set: $J = \{(q_0, q_3), (q_1, q_4), (q_2, q_5)\}$, the equivalence classes on Q are: $\{[q_0, q_3], [q_1, q_4], [q_2, q_5]\}$. With the input symbol a, the defined transitions for the first class are $\delta(q_0, a) = q_1$ and $\delta(q_3, a) = q_4$. Since q_1 and q_4 are in the same class, the restriction is not violated. By the same way, after verifying all input symbols for all classes, we can conclude that the restriction (ii) is respected, so the merge is successful. The iteration ends, and the new FSM, with one state for each class, is shown in Figure 4. Rebuilding the distinction graph for this machine we can conclude that it is complete, as also shown in Figure 4. Since all remaining states are pairwise distinguishable, there are no possible reduction left.

6. Experimental results

The empirical performance of our approach was measured against two well-known methods: `bica` [11] and `stamina` [5]. The `bica` implementation was obtained directly from one of the authors. We used a `stamina` implementation from the SIS toolkit [13]. We named our implementation as `cosme`, which stands for Compatible State Merging.
Although we are aware of other methods such as `slim` [6] and `chesmin` [4], we are unable to get a working implementation for the benchmarks. As the authors of those works also compared their obtained results with `stamina`, we consider that comparing with `bica` and `stamina` is sufficient to show some evidence of the contribution of our method.

6.1. Empirical comparison

The FSMs used in the first experiment were generated as intermediary steps of an asynchronous synthesis procedure [8]. The set was originally used by Kam and Villa [7] to evaluate their minimization algorithm. Later work on this research subject as [4, 6, 11] also adopted such set for benchmarking. The evaluation procedure was done by using each method to minimize all the FSMs in the set. The measured values are available on Table 1, where the first column is the name of the FSM; the columns 2, 3, 4 and 5 are, respectively, the initial number of states (N_i) followed by the number of remaining states on the minimized machine after applying our method (C), and the methods `bica` (B) and `stamina` (S); the last three columns contains the total running time of each implementation. A "–" indicates where methods failed. All times were measured in milliseconds, on a Intel Pentium 4, 3.4 GHz, 6800 bogomips. Swap memory was not used by any processes under evaluation.
We observe that our method runs faster than `bica` in all but one FSM (`e423`), which is between those currently not reducible by `cosme`. Usually, the time advantage is very large (see, e.g., `ifsm2` and `th.55`). Compared with `stamina`, our method runs slower in only four out of the 29 samples. Regarding the obtained minimization, our method generated bigger FSMs, with a maximum of two extra states, only in `intel-edge`, `fo.40`, `th.30` and `th.55`. The last three rows contains the FSMs `e304`, `e423` and `e608`. Those FSMs has a very particular structure (most of the outputs are not defined) and our method is not able to find a compatible pair of states whose merging would not violate the condition to guarantee quasi-equivalence.

Table 1. Values measured in benchmark.

FSM	number of states				runtime		
name	N_i	C	B	S	cosme	bica	stamina
alex1	42	6	6	6	21.7	982.8	208.7
intel-edge	28	5	4	5	13.2	94.8	23.0
isend	40	4	4	4	22.3	552.4	33.4
rcv-ifc	46	2	2	2	25.8	489.1	20.5
rcv-ifc.m	27	2	2	2	12.9	188.0	18.7
send-ifc	70	2	2	2	75.1	1776.8	29.9
send-ifc.m	26	2	2	2	14.1	611.7	18.9
vbe4a	58	3	3	3	52.6	1459.4	2058.2
vmebus	32	2	2	2	41.8	49.7e3	30.0
th.20	21	4	4	6	7.8	42.2	11.9e3
th.25	26	4	4	-	8.4	47.3	-
th.30	31	6	5	10	10.5	67.9	83.9e3
th.35	36	7	7	-	11.9	87.5	-
th.40	41	8	8	-	14.5	106.6	-
th.55	55	10	8	-	22.0	12.8e3	-
fo.16	17	3	3	3	6.5	33.3	8456.0
fo.20	21	3	3	4	7.0	38.2	36.8e3
fo.30	31	3	3	-	9.3	57.4	-
fo.40	41	6	4	-	13.8	388.3	-
fo.50	51	6	6	-	17.1	265.9	-
fo.70	71	9	-	-	35.7	-	-
ifsm0	38	3	3	3	32.1	194.2	20.2
ifsm1	74	14	14	-	56.1	569.6	-
ifsm2	48	9	9	9	89.1	11.0e3	10.7e3
e271	19	2	2	2	9.3	135.3	17.8
e285	19	2	2	2	9.0	52.5	17.8
e304*	19	19	2	2	9.8	49.3	17.9
e423*	19	19	2	-	335.0	74.1	-
e680*	19	19	2	2	15.1	56.0	18.0

6.2. FSM parameters

Another experiment was accomplished by minimizing randomly generated FSMs. The goal was to observe how the performance of the method gets affected by varying specific FSM parameters. The random generator consists of a procedure requiring four parameters: number of states, transitions, inputs and outputs. Generated machines are initially connected through transitions with random input/output symbols. Further transitions are randomly inserted to match the count specified in the parameter setting. Random seeding is provided by system time.

The method `bica` was also included into the evaluations, providing a reference to the comparison. The results from method `stamina` are omitted, since its implementation exhibited a very large amount of timeout failures when minimizing the generated FSM samples. The failure percentage along with its distribution over the FSM sizes are shown in Figure 5.

We measured the time taken by the methods to minimize 30 random FSMs for each distinct parameter setup in defined intervals. Performance shown is the average of these 30 runs. The experimented parameters are: number of states (n), number of inputs (i), number of outputs (o) and number of transitions (t).

Regarding the number of states, the defined interval was from 3 to 100. In order to keep a intermediate number of transitions, the parameter t was defined in relation to n, as $n \times 2$. The number of inputs i and outputs o were fixed at 4. The results observed are shown in Figure 6.

The performance advantage of `cosme` implementation over `bica`. Over all the 2910 MEFs, `bica` implementation exhibited 1474 timeout failures. This is more than half of the count of the generated FSM samples, and can be explained by the

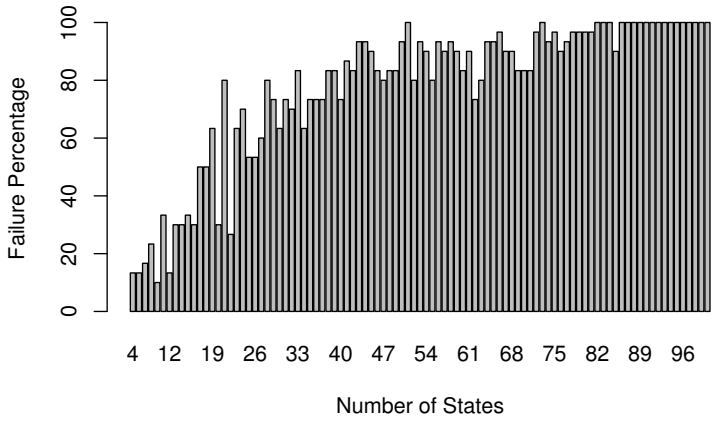

Figure 5. Percentage of failures exhibited by stamina for each value of n.

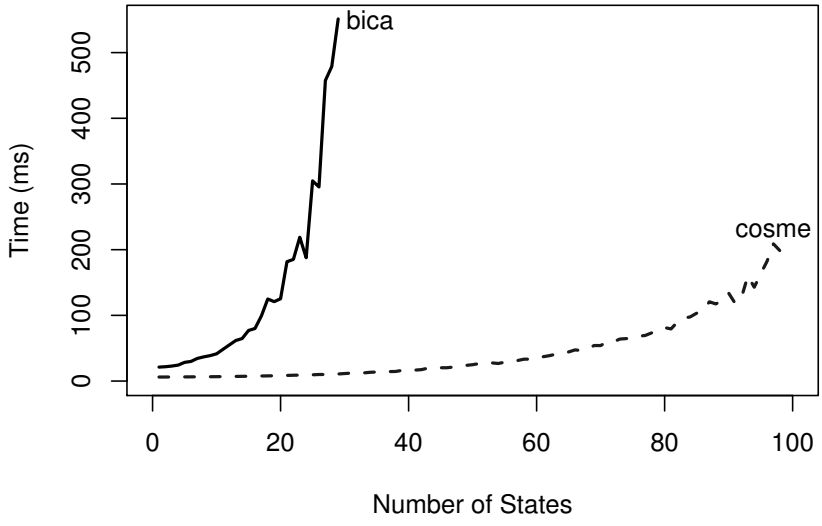

Figure 6. Performance of the methods regarding the number of states.

exponential complexity of the approach used in `bica` method. As shown in Figure 7, the runtime of `bica` grows quickly. The failures starts to appear when minimizing FSMs with more than 30 states. No failures where shown by `cosme` implementation, with a worst case of $208ms$ runtime.

Regarding the number of transitions, FSMs were generated using $n = 30$ states. The number of inputs and outputs was fixed to 4. The interval for the number of transitions t was between 29 and 120, i.e, starting at $n - 1$ and ending up to $n \times i$, the limits between an FSM with minimal number of transitions and a completely specified FSM. The minimization time is shown in Figure 8.

The performance of `bica` is affected by the number of transitions in the FSM. The biggest times shows up when minimizing FSMs with an intermediate number of transitions. Few transitions means little available information, therefore, less effort to identify distinguishable states. The closer the transitions gets to $n \times t$, the more the FSM behaves similarly to a completely specified one.

The `cosme` method performance is influenced by this parameter in a similar way, but in a smaller scale. As shown in Figure 8, `cosme` implementation shows an almost linear behavior in relation to `bica`. But when using the correct time scale, it is possible to notice the variation, as shown in Figure 9.

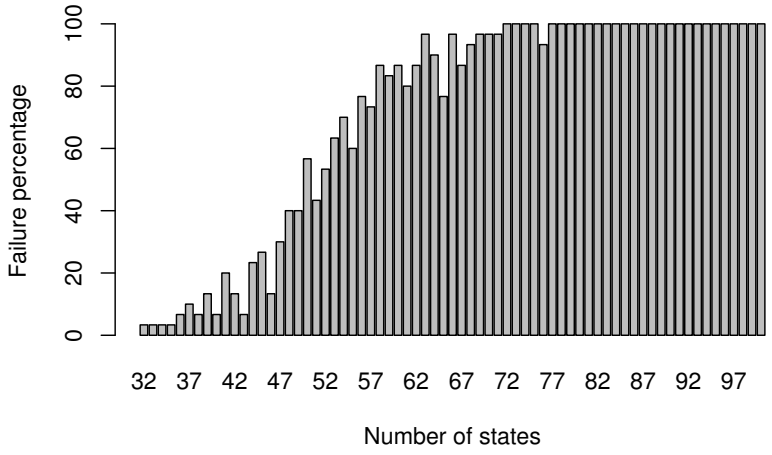

Figure 7. Failure percentage exhibited by bica implementation, related to the increasing number of states.

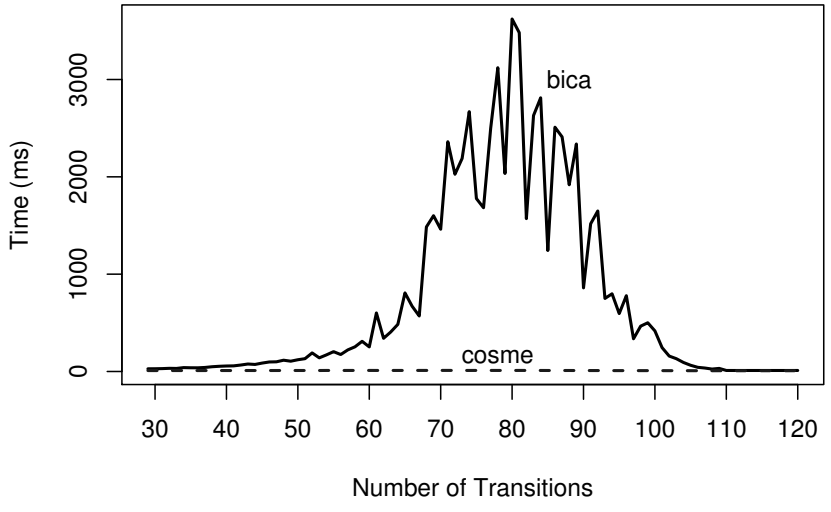

Figure 8. Methods performance regarding the number of transitions.

Regarding the number of inputs, FSMs were also generated with $n = 30$ states. With the output number fixed as $o = 4$, the number of inputs was evaluated between 2 and 20. To keep an intermediate number of transitions, t was defined related to i as $(((i + 1) \times 29)/2) - (10 \times i)$. The minimization times are shown in Figure 8.
The influence of the number of inputs on `bica` performance is close to exponential. To `cosme` implementation, once again the difference is small enough to be ignored in `bica` implementation timescale.

6.3. Number of outputs

To evaluate the influence of the output count, once again, FSMs with $n = 30$ states were generated. The number of inputs was defined at $i = 4$ and the output count was evaluated between 2 and 20. The number of transitions was stated as $t = 60$, i.e., an average of two transitions per state.
According to the observed data, the performances of both methods were not noticeably affected by the number of outputs. Since our method takes advantage of a heuristic to optimize the minimization time, it is expected that the produced results are not exact for some FSMs. It is relevant to evaluate how good are these results. The way we used to perform such evaluation was through the empirical comparison of the minimization power, between our method and the method

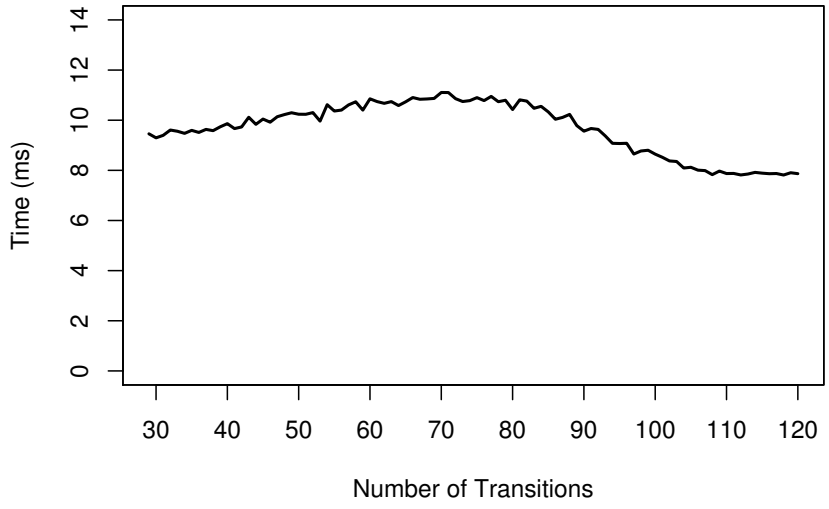

Figure 9. Influence of the number of transitions in the performance of cosme implementation.

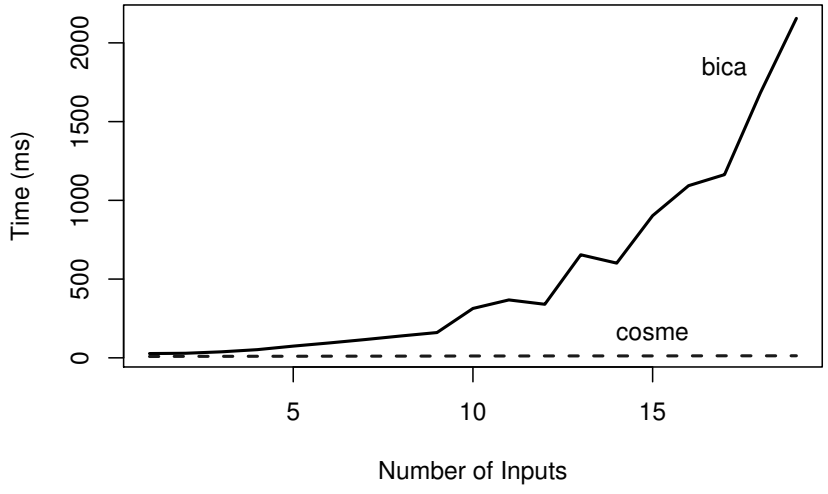

Figure 10. Methods performance regarding the input count.

`bica`, which produce the best results when subjected to benchmarks on previous works [3, 11].
The experiment was organized as follows: considering the information yielded from previous experiments, we defined the parameters in a way that an intermediate amount of effort is required from the methods. The values are: $n = 30$, $i = 4$, $o = 4$ e $t = 60$. With these parameters, 1000 FSMs were randomly generated and minimized by the implementations of `cosme` and `bica`. We compared the number of resulting states on both reduced versions of the each FSM, and the proportion of the differences are shown in Figure 11.
The values are the difference between the number of states exhibited by `cosme`, minus the number of states exhibited by `bica`. In more than a half of the FSMs, a total of 574, there was no difference between the number of states. In over a third of the samples, in 339 FSMs, the minimization from `cosme` produced only one extra state. Two and three extra states was generated, respectively, in only 77 and 10 FSMs. The FSMs reduced by `bica` resulted in an average of 15.22 states, while `cosme` did in 15.75 states. A comparison weighted by the count of results reveals an average difference of 0.523 extra states in minimizations produced by `cosme`.

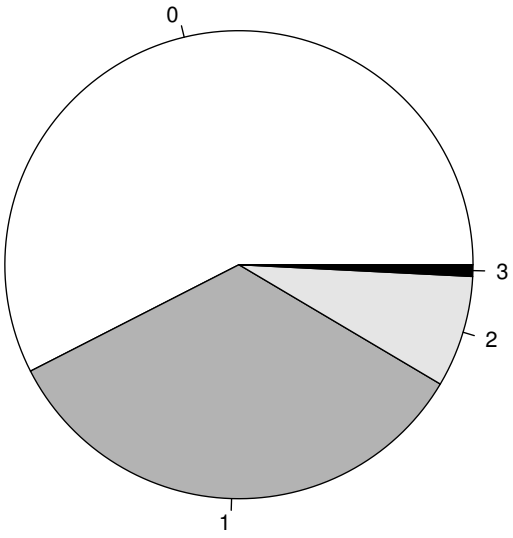

Figure 11. Minimization power comparison between cosme and bica.

7. Conclusions and future work

In this paper, we proposed a method for minimizing PFSMs, a known NP-Complete problem. The main idea is to use the information about the distinction of states to merge states. The resulting PFSMs has less states and are, thus, easier to analyse and handle. Further merging are applied until an FSM whose states cannot be merged without resulting in inconsistency is obtained. Using an empirical evaluation, we showed that our method can minimize PFSMs with better performance when compared with previous methods.

There is room for improvement. For instance, we aim to tackle the problem on the minimization of ISFSMs with few defined outputs, in which our method was not applicable at this time. Some further optimizations may be added to the procedure, improving performance and resulting in smaller FSMs. Moreover, experiments with FSMs from software industry may shed some light of how applicable the method is in real models.

References

[1] Bomze I., Budinich M., Pardalos P., Pelillo M., The maximum clique problem, In: Du D.-Z., Pardalos P. (Eds.), Handbook of Combinatorial Optimization, Kluwer Academic Publishers, 1999

[2] Grasselli A., Luccio F., A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks, IEEE T COMPUT, 1965, EC-14, 350-359

[3] Gören, S., Ferguson, F.J., CHESMIN: A Heuristic for State Reduction in Incompletely Specified Finite State Machines, DATE '02: Proceedings of the conference on Design, automation and test in Europe (Paris, France), 2002, 248-254

[4] Gören, S., Ferguson, F.J., On state reduction of incompletely specified finite state machines, COMPUT ELECTR ENG, 2007, 33, 58-69

[5] Hachtel G.D., Rho J.K., Somenzi F., Jacoby R., Exact and heuristic algorithms for the minimization of incompletely specified state machines, EURO-DAC '91: Proceedings of the conference on European design automation (Amsterdam, The Netherlands), IEEE Computer Society Press, 1991, 184-191

[6] Higuchi H., Matsunaga Y., A fast state reduction algorithm for incompletely specified finite state machine, DAC '96: Proceedings of the 33rd annual conference on Design automation (Las Vegas, Nevada, United States), ACM, 1996, 463-466

[7] Kam T., Villa T., Brayton R., Sangiovanni-Vincentelli A., A fully implicit algorithm for exact state minimization, DAC '94: Proceedings of the 31st annual conference on Design automation (San Diego, California, United States), ACM, 1994, 684-690

[8] Lavagno L., Moon C.W., Brayton R.K., Sangiovanni-Vincentelli A.L., Solving the state assignment problem for signal transition graphs, DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation (Anaheim, California, United States), IEEE Computer Society Press, 1992, 568-572

[9] Moore E.F., Gedanken-Experiments on Sequential Machines, In: Shannon C., McCarthy J. (Eds.), Automata Studies, Princeton University Press, Princeton, NJ, 1956

[10] Paull M.C., Unger S.H., Minimizing the number of states in incompletely specified sequential switching functions, IRE Trans. Electron. Comput., 1959, EC-8, 356-367

[11] Pena J.M., Oliveira A.L., A new algorithm for the reduction of incompletely specified finite state machines, ICCAD '98 Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design (San Jose, California, United States), ACM, 1998, 482-489

[12] Pfleeger C.P., State Reduction in Incompletely Specified Finite-State Machines, IEEE T COMPUT, 1973, 22, 1099-1102

[13] Sentovich E.M., et al., SIS: A System for Sequential Circuit Synthesis, Technical Report, University of California, Berkeley, CA, 1992