© © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
DOI: 10.1109/ICCAD.2015.7372555

MeMin: SAT-based Exact Minimization of Incompletely Specified Mealy Machines

Andreas Abel and Jan Reineke
Department of Computer Science
Saarland University
Saarbrücken, Germany
Email: {abel, reineke}@cs.uni-saarland.de

***Abstract*—In this paper, we take a fresh look at a well-known NP-complete problem—the exact minimization of incompletely specified Mealy machines. Most existing exact techniques in this area are based on the enumeration of sets of compatible states, and the solution of a covering problem. We propose a different approach. In our approach, first, a polynomial-time algorithm is used to compute a partial solution. This partial solution is then extended to a minimum-size complete solution by solving a series of boolean satisfiability (SAT) problems.**

We evaluate our implementation on the same set of benchmarks used previously in the literature. On a number of hard benchmarks, our approach outperforms existing exact minimization techniques by several orders of magnitude; it is even competitive with state-of-the-art heuristic approaches on most benchmarks.

I. Introduction

The minimization of Mealy machines is a fundamental problem with applications in many different areas. It is, for instance, an important part of many approaches for logic synthesis, as it can reduce the complexity of the resulting circuits. State reduction also plays an important role in fault-tolerant design of sequential machines [1]. Another application is in the area of Model-Based Testing (MBT), where an abstract model of a system can be used to automatically derive test cases. Most approaches in this area require the models to be minimized [2]. State minimization also has applications in compiler design and language processing. Recently, it has been used as part of a watermarking technique for copyright protection of IP cores [3], as it can reduce the threat of losing states from the signature during an attack.

While the problem of minimizing Mealy machines is efficiently solvable for fully specified machines [4], it is NP-complete for incompletely specified machines, i.e., machines where one or more outputs or next states might not be specified [5]. Minimization of a machine M in this context means finding a machine M' with the minimal number of states that has the same input/output behavior on all input sequences, on which the behavior of M is defined (but M' might be defined on additional input sequences on which M is not defined). Unlike for fully specified machines, there is no canonical minimal machine.

The problem has been extensively studied before, and a number of exact and heuristic approaches have been proposed. The *standard*, or *classic*, technique is a two-step approach, that was originally proposed by Paull and Unger [6], and improved by Grasselli and Luccio [7]. With this approach, in a first step, a number of sets of compatible states (i.e., states for which no input sequence exists, such that their outputs are different) are determined. In the second step, a subset of these sets is selected such that certain closure and covering criteria are fulfilled.

Almost all exact methods and also many heuristic methods follow this two step approach. The disadvantage of all of these approaches is that in particular the enumeration in the first step can be computationally expensive, as there can be an exponential number of compatible classes.

In this paper, we propose a new approach, that is not based on the standard approach and does not require the enumeration of a large number of sets of compatible states. Instead, we propose a formulation of the problem as a boolean satisfiability (SAT) problem. Even though the search space of our method is larger compared to methods based on the standard approach, we show that current SAT solvers are powerful enough to efficiently solve these types of problems.

More specifically, in a precomputation step, we determine a set of pairwise incompatible states that are part of any solution, i.e., we compute some form of a "partial solution". The size of this partial solution also constitutes a lower bound on the number of states of the minimal machine. We then iteratively call a boolean satisfiability solver to check whether the partial solution can be extended to a complete solution of the size of the lower bound. If this is not the case, we increase the lower bound by one. This is repeated until a solution is found. This solution is then guaranteed to correspond to a minimal machine that covers the original machine.

We compare our method to several other approaches on two sets of standard benchmarks: the ISM benchmarks, used by, e.g., [8], [9], [10], [11], [12], and the MCNC benchmarks, used by, e.g., [13], [14], [15], [16], [12], [17], [18], [19]. These benchmarks come from several sources, e.g., logic synthesis, learning problems, and networks of interacting FSMs.

Our approach outperforms the other exact approaches significantly, in particular on a number of hard benchmarks. In some cases, it is faster than existing approaches by several orders of magnitude.

On most benchmarks, our approach is also competitive with state-of-the-art heuristic methods. There are only two benchmarks on which a heuristic approach is significantly faster. However, in these two cases, this heuristic approach is not able to find the minimal result.

A. Outline

In Section II, we introduce basic definitions used throughout the paper, and we formally define the problem addressed in this paper. Section III introduces the most important related work. In Section IV, we describe our new approach in detail, and in Section V, we describe details of our implementation. In Section VI, we evaluate our approach on a set of standard benchmarks. Finally, Section VII concludes.

II. Definitions

In this section, we formally define several concepts used throughout this paper.

A. Basic definitions

Definition 1 (Mealy Machine)**.** *A Mealy machine is a tuple* $M = (I, O, Q, q_r, \delta, \lambda)$*, where* $I \neq \emptyset$ *is a finite set of input symbols,* $O \neq \emptyset$ *is a finite set of output symbols,* $Q \neq \emptyset$ *is a finite set of states,* $q_r \in Q \cup \{\bot\}$ *is the initial (reset) state,* $\delta : (Q, I) \to Q \cup \{\phi\}$ *is the transition function, and* $\lambda : (Q, I) \to O \cup \{\epsilon\}$ *is the output function.* ϕ *denotes an unspecified state,* ϵ *denotes an unspecified output, and* $\bot$ *denotes an unspecified initial state.*

Regarding the initial state, previous definitions in the literature have not been consistent. Some approaches assume that there is always a designated initial state, while others assume that any state can be the initial state. With respect to minimization this can make a difference if states are not reachable from the initial state. Our definition is a generalization of the previous definitions.

A machine M is called **completely specified** if for all states, all next states and outputs are defined, i.e., for all $q \in Q$ and $a \in I$, $\delta(q, a) \neq \phi$ and $\lambda(q, a) \neq \epsilon$. M is called **incompletely specified** if one or more next states or outputs may be unspecified.

With this definition, a completely specified machine is a special case of an incompletely specified machine. Previous approaches often defined an incompletely specified machine as a machine where at least one transition is unspecified. However, since our approach can also be used to minimize completely specified machines, we choose this definition.

In the following, we will without loss of generality only consider machines M that have no transition for which only the output is specified, i.e., $(\delta(q, a) = \phi) \implies (\lambda(q, a) = \epsilon)$ for all $q \in Q, a \in I$. Note that any machine M can be transformed to such a machine M' by adding an additional target state with no outgoing transitions.

We extend δ and λ to sequences in the usual way (where $\delta(\phi, i) = \phi$ and $\lambda(\phi, i) = \epsilon$).

An input sequence $a_0 \dots a_n \in I^*$ is called **applicable for a state** q if there is a sequence of states $q_0 \dots q_n$ with $q_0 = q$ s.t. $\delta(q_i, a_i) = q_{i+1}$ and $q_{i+1} \neq \phi$ for all $0 \leq i < n$. An input sequence is called **applicable for a machine** M if it is applicable for its initial state. If the machine has no initial state ($q_r = \bot$), the sequence is applicable if it is applicable for at least one of the states of the machine.

Two **outputs** $o_1, o_2 \in O \cup \{\epsilon\}$ are **compatible** iff $o_1 = \epsilon$, or $o_2 = \epsilon$, or $o_1 = o_2$. Two output sequences $o = o_0 \dots o_n$ and $o' = o'_0 \dots o'_n$ are compatible if o_i and o'_i are compatible for all $0 \leq i \leq n$. An output sequence $o = o_0 \dots o_n$ **subsumes** a sequence $o' = o'_0 \dots o'_n$ if for all $0 \leq i \leq n$, $o'_i = \epsilon \vee o_i = o'_i$.

Two **states** are **compatible**, if for all applicable input sequences for both states, the corresponding output sequences are compatible. Two states are **distinguishable** if they are not compatible. In this case, there is a distinguishing input sequence that is applicable for both states such that the corresponding output sequences differ.

A Mealy machine $M' = (I, O, Q', q'_r, \delta', \lambda')$ **covers** a Mealy machine $M = (I, O, Q, q_r, \delta, \lambda)$, iff for all applicable input sequences $s \in I^*$ for M, $\lambda'(q'_r, s)$ subsumes $\lambda(q_r, s)$. If the machine has no reset state, we require that for all states $q \in Q$ there is a state $q' \in Q'$ such that for all applicable input sequences $s \in I^*$ for M, $\lambda'(q', s)$ subsumes $\lambda(q, s)$.

We are now ready to formally define the problem addressed in this paper.

B. Problem statement

Given a Mealy Machine M, our goal is to find a Mealy machine M' with the minimum number of states, such that M' covers M.

C. General approach

For a Mealy machine $M = (I, O, Q, q_r, \delta, \lambda)$, a **compatibility class** (also called a **compatible**) is a set $C \subseteq Q$, such that all elements of C are pairwise compatible.

For a compatibility class $C = \{q_0, \dots, q_n\}$ and an input a, we define a **successor** function

$$Succ(C, a) = \bigcup_{0 \leq j \leq n} \{\delta(q_j, a) \mid \delta(q_j, a) \neq \phi\}.$$

In previous work, the set $Succ(C, a)$ was often called the **implied set** of C under input a.

A set $S = \{C_1, \dots, C_n\}$ of compatibility classes is **closed** if for all $C_j \in S$ and all inputs $a \in I$ there exists a $C_k \in S$, such that $Succ(C_j) \subseteq C_k$.

A closed set of compatibility classes **covers** a Mealy machine M if every state of the machine is contained in at least one class of the set.

The following theorem is based on a theorem that was first proposed and proven by Paull and Unger [6].

Theorem 1. *From a closed set* $S = \{C_1, \dots, C_n\}$ *of compatibility classes with the minimum number of classes that covers Mealy machine* $M = (I, O, Q, q_r, \delta, \lambda)$*, one can derive a Mealy machine* $M' = (I, O, Q', q'_0, \delta', \lambda')$ *with the minimal number of states that covers* M *as follows:*

1) $Q' := S$
2) $q'_r := C_j$ *for some* j *s.t.* $q_r \in C_j$ *if* $q_r \neq \bot$*, and* $q'_r := \bot$ *otherwise*
3) $\delta'(C_j, a) = \phi$ *if* $Succ(C_j, a) = \emptyset$*, and otherwise* $\delta'(C_j, a) = C_k$ *for some* k *s.t.* $Succ(C_j, a) \subseteq C_k$
4) $\lambda'(C_j, a) = o$ *if* $\lambda(q, a) = o$ *for some* $q \in C_j$ *and* $\lambda'(C_j, a) = \epsilon$ *otherwise*

It is important to note that the elements of S can overlap, i.e., S is not necessarily a partition of Q.

III. Related Work

The concept of Mealy machines was introduced by G. H. Mealy in 1955 [20]. While it was initially believed that incompletely specified machines could be minimized by similar techniques as completely specified machines [21], Ginsburg [22] discovered that there are examples for which this is not possible. Pfleeger [5] proved in 1973 that the problem of minimizing incompletely specified machines is NP-complete.

Paull and Unger [6] proposed an approach for exact state minimization based on their general theory that was introduced in the previous section. The approach consists of two steps:

1) Enumeration of all compatibility classes.
2) Solution of a covering problem, i.e., choosing a set of compatibility classes (from step 1) of minimum size that satisfies the closure and covering conditions.

Grasselli and Luccio [7] discovered that only some compatibility classes (prime compatibles) need to be considered. Prime compatibles are those compatibles that are not included in any larger compatible with the same or fewer closure constraints. They proved that that for any Mealy machine, there is at least one covering with the minimal size that consists only of prime compatibles.

This approach is often called the "standard" [10] or "classic" [9] approach, and has also been described in textbooks [23]. Almost all subsequently proposed exact minimization techniques are based on various improvements of this standard method. Rho et. al. [14] presented a tool called STAMINA which implements Grasselli and Lucio's method, as well as an extension of their method based on compatible pairs that was first proposed by Sarkar et. al. [24]. Kam et. al. developed an approach that represents prime compatibles implicitly as Binary Decision Diagrams (BDDs), and is thus able to handle significantly larger sets of prime compatibles. Several approaches proposed improvements of either the identification of compatibility classes [25], [26], [1], or for solving the corresponding covering problem [18], [27], [28].

The downside of all of these approaches is that the generation of all compatibility classes can be computationally very expensive, as the number of compatibility classes can be exponential in the number of states [29]. This is also an issue for the implicit approaches, as not all sets of compatibility classes can be efficiently represented as BDDs.

Pena and Oliveira [10] presented the tool BICA which implements a method that is not based on the classic approach. Instead, it is based on a modification of Angluin's algorithm [30] for learning Mealy machines. They generate a sequence of tree-FSMs (i.e., an FSM for which the corresponding graph is a tree with the initial state as the root). Each TFSM is reduced to a minimal consistent Mealy machine using a variant of Bierman's search algorithm [31] until a solution that covers the original machine is found. Grinchtein and Leucker [32] later modified BICA by replacing Bierman's algorithm with a SAT-based approach. This lead to significantly better results on two benchmarks, for which the TFSM reduction time was the dominating factor before.

In addition to the exact approaches, a number of heuristic methods have been proposed. Some of these are also based on the classic two-step approach. Rho et. al. [14] described a method that uses a restricted subset of the prime compatibles, but solves the covering step exactly. On the other hand, Ahmad and Das [16] proposed a technique that requires the enumeration of all prime compatibles, but uses a heuristic for the second step. Several approaches use both a subset of the prime compatibles (typically the maximal compatibles), and heuristics for the covering step [33], [19], [17], [12], [15].

A few heuristic techniques are not based on the classic approach. Avedillo et. al. [34] described a method that reduces the number of states of a machine through a sequence of transformations on a symbolic description of the machine. Gören and Ferguson [11] presented an approach that is based on a checking sequence generation technique. Klimowicz and Solov'ev [13] proposed a method based on merging pairs of compatible states. Alberto and Simao [9] described a technique that selects compatible states using maximum cliques on the distinction graph.

IV. Approach

Unlike almost all previous exact minimization techniques, our technique does not require the enumeration of compatibility classes. Instead, we first compute a partial solution that is part of any solution. The size of this partial solution also corresponds to a lower bound on the number of states of the minimized machine. Then, we use a SAT solver to determine if there is a machine that covers the original machine with the size of the lower bound. If the SAT solver cannot find a satisfying assignment, we increase the lower bound by one.

So, at a high level, our algorithm works as follows.

Algorithm 1: Main Algorithm

```
Input: M = (I, O, Q, q_r, δ, λ)
begin
    bool[][] m ← computeIncompatibilityMatrix(M)
    set P ← computePartialSolution(m)
    lowerBound ← |P|
    for nClasses ← lowerBound to |Q| do
        list clauses ← createSATProblem(nClasses, M, m, P)
        (satisfiable, model) ← runSATSolver(clauses)
        if satisfiable then
            return buildMachine(model, M)
```

In the following subsections, we describe our approach in more detail. We assume that we want to minimize a machine with $|Q|$ states, and that the states are numbered from 0 to $|Q| - 1$.

A. Incompatibility matrix

The first step of our algorithm is to determine which pairs of states are compatible, and which pairs are incompatible. Similar to previous approaches, we store this information in a matrix m, such that $m[i][j] = 1$ if states i and j are incompatible, and $m[i][j] = 0$ if they are compatible.

However, to compute this matrix, we use a slightly different approach from the classic approach [7]. Our algorithm works as follows:

- Initialize all entries of the matrix with 0.
- Iterate over all state pairs (i, j). If $m[i][j]$ is not already set to 1, check whether there is an input symbol a such that the outputs of i and j differ. If this is the case, set this entry of the matrix to 1.
- Whenever an entry (i, j) of the matrix changes its value from 0 to 1, we set the value of all predecessor pairs under the same inputs to 1 as well.

The last step is executed at most $|Q|^2$ times. To determine the predecessor pairs efficiently, we can compute a list of predecessor pairs for each state pair by iterating over all state pairs and inputs once. Thus, these $|Q|^2$ lists have at most $|Q|^2 \cdot |I|$ elements in total. Since the last step iterates over each of the lists at most once, the overall complexity of our algorithm is in $O(|Q|^2 \cdot |I|)$.

The classic algorithm does not perform the update upon a change. Instead, it repeatedly iterates over all state pairs (i, j) and sets $m[i][j]$ to 1 if one of their successor pairs is set to 1. The algorithm terminates upon reaching a fixed point. Thus, its complexity is in $O(|Q|^3 \cdot |I|)$.

B. Encoding as a SAT problem

In this section, we describe, how we encode the problem "Is there a closed set of compatibility classes of size n that covers the machine?" as a boolean satisfiability problem. It is straightforward to show that if the problem is unsatisfiable for n classes, but satisfiable for $n+1$ classes, a satisfying assignment for $n+1$ classes fulfills the conditions of Theorem 1.

SAT solvers typically require the problem to be in conjunctive normal form (CNF). We will first give a high level-description of each subproblem, and then show how to translate it to CNF.

In the following, we will use literals of the form $s_{x,i} \in \mathbb{B}$ to denote that state x is in compatibility class i.

a) Covering Condition: All states of the original machine must be in at least one compatibility class. We therefore add, for all states x, a clause of the form

$$s_{x,0} \vee s_{x,1} \vee \cdots \vee s_{x,n-1}.$$

b) Compatibility: All states that are in the same class must be pairwise compatible. For a state x, let $Inc(x) \subseteq Q$ be the set of states that are incompatible to x. To ensure that no incompatible states are in the same class, we add for each state x and each class i the implication

$$s_{x,i} \implies \bigwedge_{\substack{y \in Inc(x) \\ y > x}} \neg s_{y,i}.$$

Incompatibility is symmetric. However, it suffices to have one constraint for each pair of states. This is ensured by $y > x$.

In CNF, the implication corresponds to the following set of clauses:

$$\bigwedge_{\substack{y \in Inc(x) \\ y > x}} (\neg s_{x,i} \vee \neg s_{y,i}).$$

c) Closure: For all states that are in the same class, there must be another class that contains all of their successor states. Thus, we have for each input symbol $a \in I$ and each class i a clause of the form

$$\exists j : \forall x : (s_{x,i} \implies s_{x',j}),$$

where x' is used to denote the successor state of x under input a. If the successor state is undefined, the corresponding implication is omitted.

Note that to fulfill the closure property it is not sufficient to require that for all pairs of states that are in the same class, their successor pairs must be in the same class. This is because the classes are not disjoint. So it is possible that for three states s_1, s_2, s_3, there is a class i that contains the successors of s_1 and s_2, a class j that contains the successors of s_2 and s_3, and a class k that contains the successors of s_1 and s_3, but there might not be a single class that contains the successors of all three states.

In the above formula, we can represent the existential quantifier as a disjunction, and the universal quantifier as a conjunction. The formula is thus equivalent to

$$\bigvee_{0 \le j < n} (\bigwedge_{x \in Q} (\neg s_{x,i} \vee s_{x',j})).$$

A direct conversion of this formula to CNF would lead to an exponential increase in its size. To obtain a more compact representation, we introduce an auxiliary literal for each input symbol and class of the form $Z_j^{a,i}$. The following formula is then equisatisfiable to the formula above:

$$(\bigvee_{0 \le j < n} Z_j^{a,i}) \wedge \bigwedge_{0 \le j < n} \bigwedge_{x \in Q} (\neg Z_j^{a,i} \vee \neg s_{x,i} \vee s_{x',j}).$$

C. Computing a partial solution

In this section, we present an approach to reduce the number of symmetrical cases the SAT solver has to consider by precomputing a "partial solution". More specifically, we first compute a set $S = \{x_1, \ldots, x_k\}$ of pairwise incompatible states. For all solutions of the SAT problem, each of these states must be in at least one class, but no pair of these states may be in the same class. However, since we are only interested in sets of compatibility classes, the ordering of the classes does not matter. Thus, we can obtain an equisatisfiable formula by just assigning all elements of S to arbitrary, different classes. To this end, we add a clause of the form

$$s_{x_1,1} \wedge s_{x_2,2} \wedge \cdots \wedge s_{x_k,k}.$$

Moreover, we can use the cardinality of S as a lower bound for the required number of classes.

If we represent the incompatibility matrix as a graph (such that there is an edge in the graph whenever two states are compatible), the problem of finding such a set S corresponds to finding an independent set in the graph. While the problem of finding a maximum independent set is NP-hard, a heuristic is sufficient for our purposes, since a non-maximal set would just lead to a smaller reduction of symmetries, and to a smaller lower bound, but it would still lead to a correct solution.

We use the following sequential greedy heuristic to find a set of pairwise incompatible states. We first create a list of all states that is sorted in reverse order based on the number of incompatible states for each state (or equivalently, the degree of the states on the incompatibility graph). The algorithm maintains a set $S \subseteq Q$ of pairwise independent states (S is initially empty). We then iterate over the sorted list of states. Whenever we encounter a state that is incompatible to all states in S, we add this state to S.

V. Implementation

We have implemented our approach in a tool called MeMin. MeMin is available for download from[1]. The tool is written in C++, and it uses MiniSat [35] as SAT solver. MiniSat is an open-source SAT solver that can be used as a library and that provides a simple API.

Like many previous tools, MeMin accepts inputs in the widely used KISS2 input format [36]. In this format, Mealy machines are essentially described by a set of 4-tuples of the form $(input, currentState, nextState, output)$. $input$ and $output$ are sequences of $\{0, 1, -\}$, where $-$ means that the corresponding bit is not specified. In the following paragraphs, we describe how we deal with some of the implications of using this specification format.

A. Dealing with partially specified outputs

The KISS2 format allows for partially specified outputs, i.e., outputs in which a subset of the bits are undefined. A machine M covers such a partially specified machine M' iff the set of possible outputs of M is a (non-empty) subset of the set of possible outputs of M'.

[1] http://embedded.cs.uni-saarland.de/MeMin.php

We incorporate this generalization into our framework in a similar way as the authors of [10]. We define two outputs to be compatible if all of their bits at the same position are either the same, or at least one of the bits is undefined. We then use this definition of the compatibility of outputs in the definition of the compatibility of states.

B. Dealing with partially specified inputs

Partially specified inputs are used as a shorthand for sets of inputs. One straight-forward way to deal with this would be to just add transitions for all concrete inputs that are described by a partially specified input. However, this approach is only viable when the number of unspecified bits is small.

Instead, we use the following approach. We first partition the set of states into equivalence classes such that two states are in the same class if they are transitively compatible. For each equivalence classes C we then compute a set of disjoint partially specified inputs D such that all intersections of inputs from C can be expressed by a combination of inputs from D. Finally, we replace all transitions of states in C with transitions that have the corresponding inputs from D.

Furthermore, for the closure constraints in Section IV-B, if there are multiple inputs that have exactly the same output/next state behavior, we only need to consider one of these inputs.

C. Undefined reset states

The KISS2 format allows for the optional definition of a reset state. According to the specification [36], if no reset state is specified, the first state encountered in the transition list is implicitly assumed to be the reset state. However, several of the benchmarks we use in the evaluation would not make much sense with this specification, as for example in *rubin2250*, only three (out of 2250) states are reachable from the first state. We therefore added a command line parameter to our tool that controls whether the first state should be the reset state, or any state might be a reset state if no explicit reset state is specified.

VI. EVALUATION

In this section, we first evaluate our approach on two sets of standard benchmarks against two other exact and two heuristic techniques. We then investigate how the precomputation of a partial solution, as described in Section IV-C, influences the execution time.

A. Benchmarks

We compare the performance of our implementation with BICA [10], which is based on Angluin's learning algorithm, and STAMINA (exact mode) [14], which is a popular implementation of the explicit version of the two-step standard approach. Furthermore, we also compare our tool with STAMINA (heuristic mode), and COSME [9], which is another, recently proposed, heuristic technique.

There are two standard sets of benchmarks that were used in the evaluations of previous techniques: The ISM benchmarks, used by, e.g., [8], [9], [10], [11], [12], and the MCNC benchmarks, used by, e.g., [13], [14], [15], [16], [12], [17], [18], [19].

The ISM benchmarks contain several examples that exhibit a very large number of prime compatibles, and are therefore not solvable by techniques that are based on the explicit enumeration of prime compatibles, such as STAMINA. The machines after minimization are, however, rather small (at most 14 states), which makes them amenable to the learning-based approach.

Most benchmarks from the MCNC suite, on the other hand, can be easily solved by the standard approach. However, some of the minimized machines are significantly larger (up to 135 states), which makes these benchmarks harder for the learning-based technique, as well as for the technique based on implicit enumeration, where "the representation becomes inefficient", "when there are many states and few compatibles" [8].

The scatter plots in Figure 1 and 2 show the results of the different tools on both sets of benchmarks. MEMIN reported a correct result on all benchmarks. Cases, where the other tool did not return a correct result are indicated with orange and red (for the exact approaches, a non-minimal result is considered to be incorrect). The reported runtimes are the averages over 5 runs; the timeout was set to 5 minutes. The standard deviations of the runtimes were in all cases smaller than 4.5%. A table with all results is available online[2].

1) ISM benchmarks: This set of benchmarks was compiled by the authors of the ISM tool [8]. The benchmarks come from a variety of sources, including asynchronous synthesis procedures, FSMs constructed to be compatible with a given collection of examples of input/output behavior, FSMs that are part of a surrounding network of FSMs, FSMs contructed to have an exponential number (up to 2^{1500}) of prime compatibles, and randomly generated machines.

MEMIN was able to solve all 34 benchmarks in under 0.2s, and 30 of them even in less than 10ms. BICA was on all benchmarks at least 16 times slower; it could solve only one benchmark in less than 10ms. 16 benchmarks took more than 100ms, 7 more than 1s, and one could not finish within a timeout of 5 minutes.

STAMINA (exact mode) was not able to solve 13 benchmarks on our machine. This mostly corresponds to what was also reported in previous publications, except for ifsm1, for which [10] reported a time about twice as high as for *ifsm2*, and *fo.20* and *th.30*, for which [9] reported around 36s and 84s, respectively, though it is not clear whether they used the exact or the heuristic mode of STAMINA. Furthermore, for 5 benchmarks, the results of STAMINA (exact mode) were not minimal.

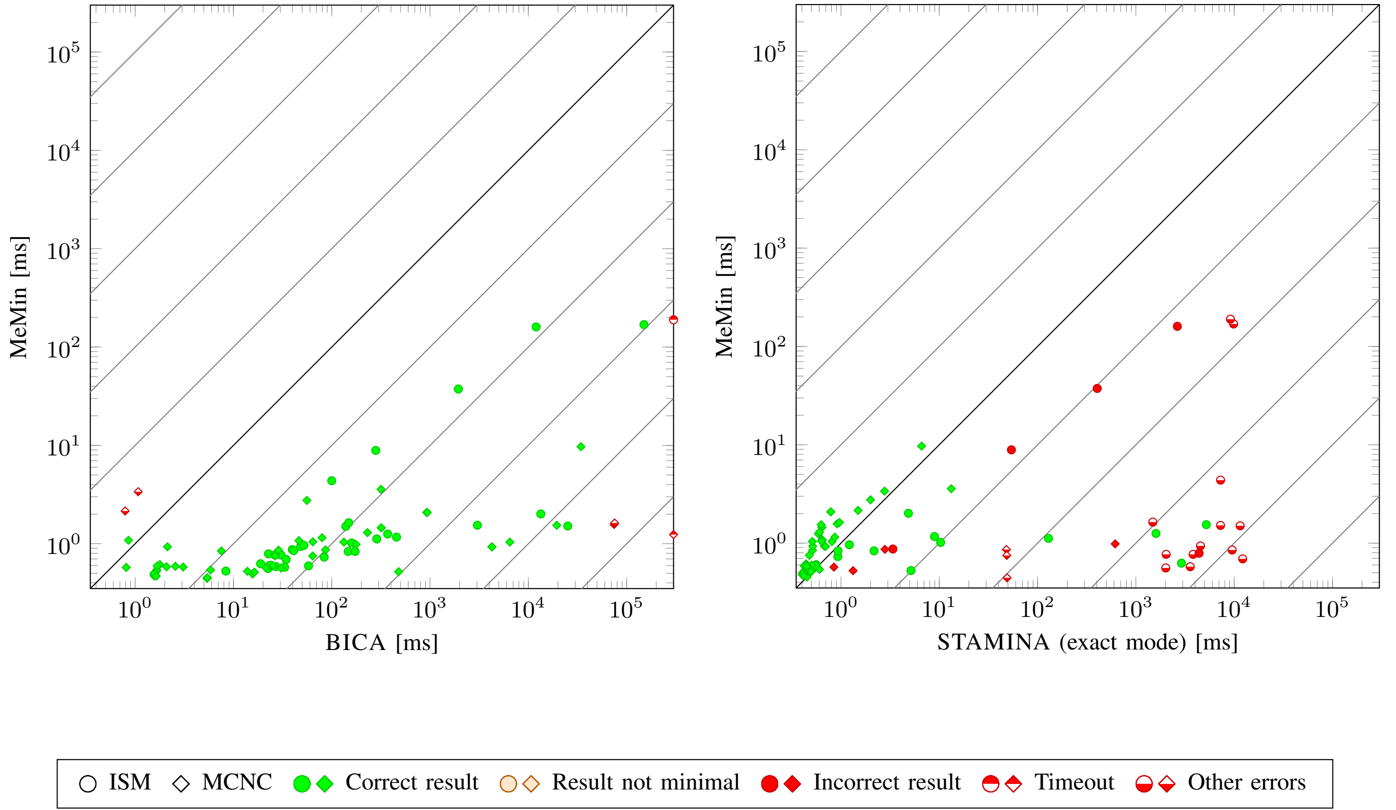
The execution time of the heuristic mode of STAMINA was in only one case significantly different from the exact mode. 15 benchmarks lead to error messages on our machine. [12] reported results for the heuristic mode of STAMINA for these cases. For the *fo.** and *th.** benchmarks, the resulting machines were in many cases significantly larger than the minimal machines (e.g., 24 states instead of 8 for *th.55*, or 14 instead of 7 for *fo.70*).

COSME was in two cases (*fo.60* and *fo.70*) significantly faster than MEMIN, however, the minimized machines had 3 and 2 additional states, respectively. In 7 cases, COSME was more than 10 times slower than MEMIN, and in 12 cases the resulting machines did not have the minimal number of states.

2) MCNC benchmarks: Benchmarks from the MCNC suite [37] are widely used in logic synthesis. Both MEMIN and STAMINA were able to solve almost all benchmarks in less than 10ms. However, STAMINA was unable to solve three cases, and reported non-minimal results in four cases. BICA, on the other hand, needed more than 100ms in 17 cases, and more than 1s in 8 cases. On one benchmark,

[2]http://embedded.cs.uni-saarland.de/tools/MeMin/results.pdf

Fig. 1: Benchmark results - Exact approaches



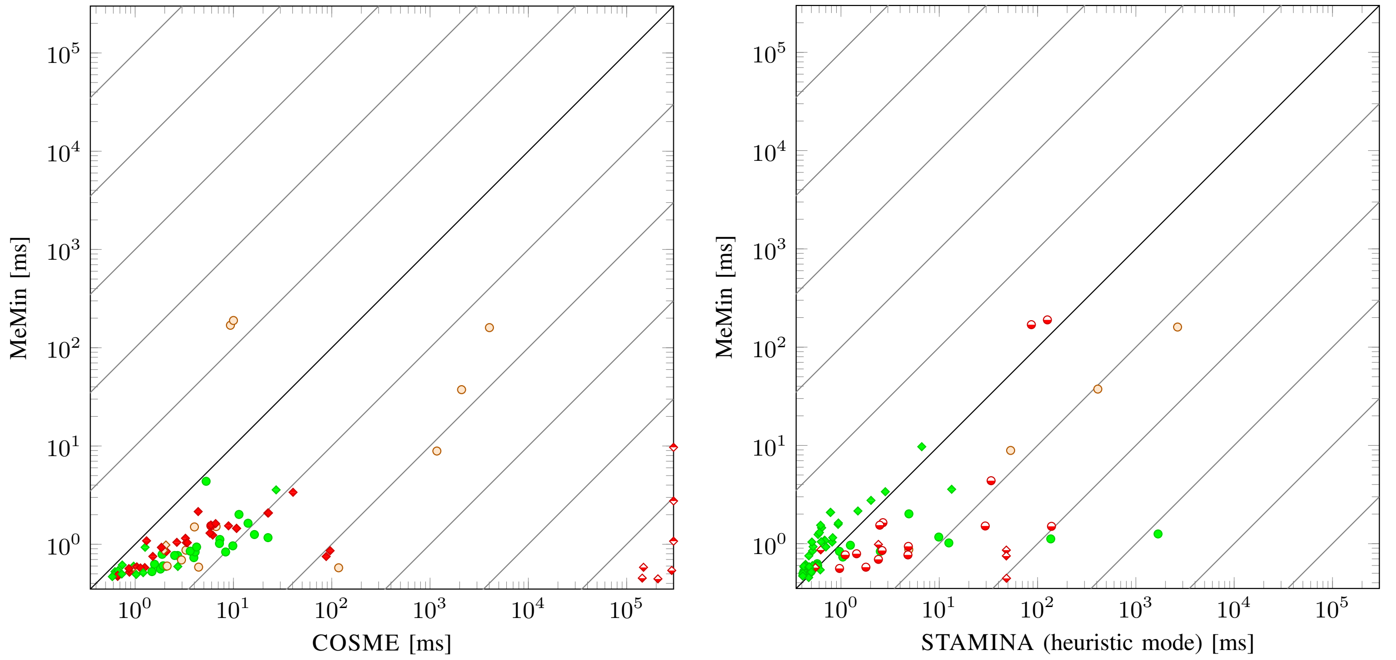


Fig. 2: Benchmark results - Heuristic approaches

BICA did not terminate within a timeout of 5 minutes. On two benchmarks, it ran out of memory. COSME reported invalid results on a large number of these benchmarks. We assume that this might be due to a bug in handling partially specified inputs, which are used by many MCNC benchmarks. We noticed that the resulting machines sometimes had multiple transitions for the same inputs.

B. Evaluation of MeMin

In this section, we analyze our approach in detail. Table I shows the results for several experiments. The column "SAT only" shows the execution time for a naive implementation that does not precompute a partial solution, and uses 1 as the lower bound. The column "SAT+LB" gives the time for an implementation that uses the size of the partial solution as the lower bound, but does not add the constraints from Section IV-C to the SAT problem. The column "SAT+LB+PS" shows the time for the implementation that also adds these constraints (i.e., the same implementation that was also used for the scatter plots). For this implementation, the column "SAT share" displays the time the SAT solver needed in relation to the total execution time. The column $|Q|$ shows the number of states before minimization, $|Q_m|$ the number of states after minimization, and $|P|$ the size of the partial solution. For space reasons, we omitted the benchmarks, for which all of these implementations needed less than 2ms and for which the size of the partial solution was equal to the size of the minimized machine.

All benchmarks, for which the minimized machines have more than 9 states could not be solved by the naive implementation within a timeout of 5 minutes. On the other hand, the solution that uses the lower bound was able to solve all but one benchmark in under 5 minutes (and all but 5 in under 1s). We observed that the SAT solver typically needs much more time to determine that there is no solution for $|Q_m| - 1$ than to find a solution of size $|Q_m|$. For many benchmarks, the size of the partial solution already corresponds to the size of the minimal solution; thus the second implementation often does not need to consider the $|Q_m| - 1$ case.

The additional constraints used in the third implementation were in particular helpful for larger machines (e.g., *s298* and *scf*), as well as those machines, where $|P| < |Q_m|$. The most noticeable speed-up was on *fo.60* and *fo.70* (more than 600 times faster), and on *ifsm1*, for which the first and second implementation did not terminate within 5 minutes, while the final implementation was able to solve the benchmark in 1.6ms.

C. Other tools

A number of other state minimization tools have been described in the literature. Unfortunately, for the approaches described in [8], [12], [16], [15], [13], we were unable to obtain working implementations.

For the ISM tool [8], which implements the implicit enumeration approach based on BDDs, there is no public release available [38]. However, the authors of BICA compared their tool with ISM on the same set of benchmarks that we used in Section VI-A1. ISM was slightly faster than BICA on only one benchmark (ifsm1). It was more than 100 times slower than BICA on 5 benchmarks, and was unable to solve 7 benchmarks that BICA was able to solve.

Slim [12] is not available because it is Fujitsu's proprietary [39]. Also, the source code of VOID [16] is not available any more [40]. While we were able to obtain a copy of CHESMIN [11] from the authors, unfortunately, on our machine, the version we obtained yields segmentation faults on most benchmarks.

TABLE I: Evaluation of MEMIN

					Solution times [ms]			
	Benchmark	$\lvert Q\rvert$	$\lvert Q_m\rvert$	$\lvert P\rvert$	SAT only	SAT+LB	SAT+LB+PS	SAT share
ISM benchmarks	alex1	42	6	6	5.9	2.9	1.1	35%
	intel_edge.dummy	28	4	3	1.7	1.6	.8	36%
	isend	40	4	4	2.1	1.6	1.0	35%
	vbe4a	58	3	3	2.2	1.8	1.2	32%
	vmebus.master.m	32	2	2	2.7	2.5	2.0	14%
	fo.16	17	3	2	1.1	1.1	.6	30%
	fo.30	31	3	2	1.2	1.2	.7	38%
	fo.40	41	4	4	3.2	2.9	1.5	66%
	fo.50	51	6	5	16.0	15.1	4.3	87%
	fo.60	61	7	6	116423.5	117156.3	169.2	99%
	fo.70	71	7	4	116371.3	114812.6	189.4	99%
	th.20	21	4	3	1.4	1.3	.7	42%
	th.25	26	4	3	1.3	1.2	.7	40%
	th.30	31	5	5	2.2	1.7	.6	31%
	th.35	36	7	7	8.1	2.0	.8	42%
	th.40	41	8	8	40.1	2.2	.9	45%
	th.55	55	8	8	94.1	8.5	1.5	63%
	ifsm0	38	3	3	2.2	1.9	.9	9%
	ifsm1	74	14	13	timeout	timeout	1.6	54%
	ifsm2	48	9	9	1616.7	7.2	1.5	35%
	rubin1200	1200	3	3	39.6	37.8	37.4	13%
	rubin2250	2250	3	3	163.9	159.3	160.1	8%
	rubin600	600	3	3	10.1	9.3	8.8	20%
MCNC benchmarks	bbara	10	7	7	7.8	1.4	.5	9%
	bbsse	16	13	13	timeout	8.1	.9	14%
	bbtas	6	6	6	2.2	1.2	.5	8%
	cse	16	16	16	timeout	4.2	.8	13%
	dk14	7	7	7	10.1	1.6	.6	9%
	dk16	27	27	27	timeout	32.7	.9	16%
	dk17	8	8	8	30.9	1.6	.5	9%
	dk27	7	7	7	5.8	1.3	.4	9%
	dk512	15	15	15	timeout	3.4	.5	12%
	ex1	20	18	18	timeout	9.3	1.0	12%
	ex2	19	5	4	3.8	3.5	.9	48%
	ex3	10	4	2	1.6	1.6	.8	47%
	ex4	14	14	14	timeout	3.0	.5	13%
	ex5	9	3	2	1.1	1.0	.5	22%
	ex6	8	8	8	39.3	1.6	.5	11%
	keyb	19	19	19	timeout	6.9	1.1	13%
	kirkman	16	16	16	timeout	5.9	2.1	9%
	mark1	15	12	12	timeout	9.5	1.0	35%
	opus	10	9	9	265.3	1.7	.5	10%
	planet1	48	48	48	timeout	78.7	1.4	26%
	planet	48	48	48	timeout	78.3	1.4	26%
	pma	24	24	24	timeout	8.1	.8	17%
	s1488	48	48	48	timeout	137.7	2.0	22%
	s1494	48	48	48	timeout	134.0	2.0	23%
	s1a	20	1	1	3.3	3.2	2.7	8%
	s1	20	20	20	timeout	8.3	1.0	15%
	s208	18	18	18	timeout	7.6	1.0	13%
	s298	218	135	135	timeout	5273.0	9.7	36%
	s386	13	13	13	timeout	3.0	.7	13%
	s420	18	18	18	timeout	8.7	1.0	13%
	s510	47	47	47	timeout	89.6	1.2	30%
	s820	25	24	24	timeout	16.6	1.5	15%
	s832	25	24	24	timeout	18.9	1.6	15%
	sand	32	32	32	timeout	32.9	1.5	18%
	scf	121	97	97	timeout	1492.3	3.3	45%
	shiftreg	8	8	8	37.6	1.7	.4	9%
	sse	16	13	13	timeout	8.2	.9	14%
	styr	30	30	30	timeout	29.4	1.2	17%
	tbk	32	16	16	timeout	35.5	3.5	6%
	tma	20	18	18	timeout	4.7	.7	20%

D. Experimental setup

All experiments were run on an Intel Core i5-4590 (3.3GHz) with 4GB of RAM. We disabled dynamic frequency scaling, and copied all executables and benchmark files to a RAM disk, to minimize timing variations due to hard drive accesses. The execution times were measured using the perf[3] tool.

We used BICA in version 5.0.3. We specified a hash table of size 100,000 (parameter "-h 100,000"). With the default size of 10,000 some benchmarks failed with the error message "Too many collisions. Specify a large hash table." We used the version of STAMINA that is included in SIS 1.3.6[4]. For the exact mode, we specified the parameter "-s 0", and for the heuristic mode "-s 3" (which combines the "tight upper bound" and "isomorphic" heuristics). We used a version of COSME (as of Aug. 2010) that was provided to us by one of the authors. We specified the same parameters as in the evaluation in [9] ("--comparemode --shownum --xincha").

[3]https://perf.wiki.kernel.org/

[4]http://embedded.eecs.berkeley.edu/Alumni/pchong/sis.html

VII. Conclusions

With respect to the set of benchmarks used to evaluate previous approaches, one can consider the problem of minimizing incompletely specified Mealy machines to be solved: Our method can solve all of these benchmarks in less than 0.2 seconds, and all but four in less than 10ms.

To evaluate the limits of our approach, one problem that future work will need to address is to identify a new set of challenging, realistic benchmarks.

Acknowledgments

We would like to thank Alex D. B. Alberto for providing us with the ISM-benchmarks and the source code of COSME. This work was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See http://www.avacs.org/ for more information.

References

[1] I. Ahmad, "A distributed algorithm for finding prime compatibles on network of workstations," *Microprocessors and Microsystems*, vol. 25, no. 4, pp. 195 – 202, 2001.

[2] A. Alberto and A. Simao, "Minimization of incompletely specified finite state machines based on distinction graphs," in *10th Latin American Test Workshop (LATW '09)*, March 2009, pp. 1–6.

[3] J. Echavarria, A. Morales-Reyes, R. Cumplido, and M. Salido, "FSM merging and reduction for IP cores watermarking using genetic algorithms," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2014, pp. 1–7.

[4] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. Academic Press, New York, 1971, pp. 189–196.

[5] C. Pfleeger, "State reduction in incompletely specified finite-state machines," *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1099–1102, Dec 1973.

[6] M. Paull and S. Unger, "Minimizing the number of states in incompletely specified sequential switching functions," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 356–367, Sept 1959.

[7] A. Grasselli and F. Luccio, "A method for minimizing the number of internal states in incompletely specified sequential networks," *IEEE Transactions on Electronic Computers*, vol. EC-14, no. 3, June 1965.

[8] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "A fully implicit algorithm for exact state minimization," in *31st Conference on Design Automation*, June 1994, pp. 684–690.

[9] A. D. B. Alberto and A. Simao, "Iterative minimization of partial finite state machines," *Central European Journal of Computer Science*, vol. 3, no. 2, pp. 91–103, 2013. [Online]. Available: http://dx.doi.org/10.2478/s13537-013-0106-0

[10] J. Pena and A. Oliveira, "A new algorithm for exact reduction of incompletely specified finite state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 11, pp. 1619–1632, Nov 1999.

[11] S. Gören and F. J. Ferguson, "On state reduction of incompletely specified finite state machines," *Comput. Electr. Eng.*, vol. 33, no. 1, pp. 58–69, Jan. 2007.

[12] H. Higuchi and Y. Matsunaga, "A fast state reduction algorithm for incompletely specified finite state machine," in *Proceedings of the 33rd Annual Design Automation Conference*. ACM, 1996, pp. 463–466.

[13] A. S. Klimowicz and V. V. Solov'ev, "Minimization of incompletely specified mealy finite-state machines by merging two internal states," *J. Comput. Syst. Sci. Int.*, vol. 52, no. 3, pp. 400–409, May 2013.

[14] J.-K. Rho, G. Hachtel, F. Somenzi, and R. Jacoby, "Exact and heuristic algorithms for the minimization of incompletely specified state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 2, pp. 167–177, Feb 1994.

[15] H. Hu, H.-X. Xue, and J.-N. Bian, "HSM2: A new heuristic state minimization algorithm for finite state machine," *Journal of Computer Science and Technology*, vol. 19, no. 5, pp. 729–733, 2004.

[16] I. Ahmad and A. S. Das, "A heuristic algorithm for the minimization of incompletely specified finite state machines," *Computers & Electrical Engineering*, vol. 27, no. 2, pp. 159–172, 2001.

[17] J. Sanchez, A. Garnica, and J. Lanchares, "A genetic algorithm for reducing the number of states in incompletely specified finite state machines," *Microelectronics journal*, vol. 26, no. 5, pp. 463–470, 1995.

[18] R. Puri and J. Gu, "An efficient algorithm to search for minimal closed covers in sequential machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 6, Jun. 1993.

[19] L. N. Kannan and D. Sarma, "Fast heuristic algorithms for finite state machine minimization," in *Proceedings of the Conference on European Design Automation*, 1991, pp. 192–196.

[20] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, Sep. 1955.

[21] D. Aufenkamp, "Analysis of sequential machines II," *IRE Transactions on Electronic Computers*, vol. EC-7, no. 4, pp. 299–306, Dec 1958.

[22] S. Ginsburg, "On the reduction of superfluous states in a sequential machine," *J. ACM*, vol. 6, no. 2, pp. 259–282, Apr. 1959.

[23] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Functional Optimization*, 1st ed. Springer Publishing Company, 2010.

[24] S. De Sarkar, A. Basu, and A. Choudhury, "Simplification of incompletely specified flow tables with the help of prime closed sets," *IEEE Transactions on Computers*, vol. C-18, no. 10, pp. 953–956, Oct 1969.

[25] R. Bennetts, "An improved method of prime c-class derivation in the state reduction of sequential networks," *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 229–231, 1971.

[26] H. Higuchi and Y. Matsunaga, "Implicit prime compatible generation for minimizing incompletely specified finite state machines," in *Proceedings of the Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '95. ACM, 1995.

[27] S. Liao and S. Devadas, "Solving covering problems using LPR-based lower bounds," in *Proceedings of the 34th Annual Design Automation Conference*, ser. DAC '97. ACM, 1997, pp. 117–120.

[28] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli, "Explicit and implicit algorithms for binate covering problems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 7, pp. 677–691, Jul 1997.

[29] F. Rubin, "Worst case bounds for maximal compatible subsets," *IEEE Transactions on Computers*, vol. C-24, no. 8, pp. 830–831, Aug. 1975.

[30] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[31] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 592–597, 1972.

[32] O. Grinchtein and M. Leucker, "Learning finite-state machines from inexperienced teachers," in *Grammatical Inference: Algorithms and Applications*. Springer, 2006, pp. 344–345.

[33] R. Bennetts, J. Washington, and D. Lewin, "A computer algorithm for state table reduction," *Radio and Electronic Engineer*, vol. 42, no. 11, pp. 513–520, November 1972.

[34] M. Avedillo, J. Quintana, and J. Huertas, "New approach to the state reduction in incompletely specified sequential machines," in *IEEE International Symposium on Circuits and Systems*, 1990, pp. 440–443.

[35] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 2919, pp. 502–518.

[36] E. M. Sentovich *et al.*, "SIS: A system for sequential circuit synthesis," Tech. Rep., 1992.

[37] S. Yang, *Logic synthesis and optimization benchmarks user guide Version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.

[38] T. Villa, Personal communication, March 2015.

[39] H. Higuchi, Personal communication, Fujitsu Laboratories, March 2015.

[40] I. Ahmad, Personal communication, March 2015.