Effective Reductions of Mealy Machines

Florian Renkin, Philipp Schlehuber-Caissier, Alexandre Duret-Lutz, and Adrien Pommellet

LRDE, EPITA, Kremlin-Bicêtre, France
`{frenkin,philipp,adl,adrien}@lrde.epita.fr`

Abstract. We revisit the problem of reducing incompletely specified Mealy machines with reactive synthesis in mind. We propose two techniques: the former is inspired by the tool MeMin [1] and solves the minimization problem, the latter is a novel approach derived from simulation-based reductions but may not guarantee a minimized machine. However, we argue that it offers a good enough compromise between the size of the resulting Mealy machine and performance. The proposed methods are benchmarked against MeMin on a large collection of test cases made of well-known instances as well as new ones.

1 Introduction

Program synthesis is a well-established formal method: given a logical specification of a system, it allows one to automatically generate a provably correct implementation. It can be applied to reactive controllers (Fig. 1a): circuits that produce for an input stream of Boolean valuations (here, over Boolean variables a and b) a matching output stream (here, over x and y).

The techniques used to translate a specification (say, a Linear Time Logic formula that relates input and output Boolean variables) into a circuit often rely on automata-theoretic intermediate models such as Mealy machines. These transducers are labeled graphs whose edges associate input valuations to a choice of one or more output valuations, as shown in Fig. 1b.

Since Mealy machines with fewer states result in smaller circuits, reducing and minimizing the size of Mealy machines are well-studied problems [2, 12].

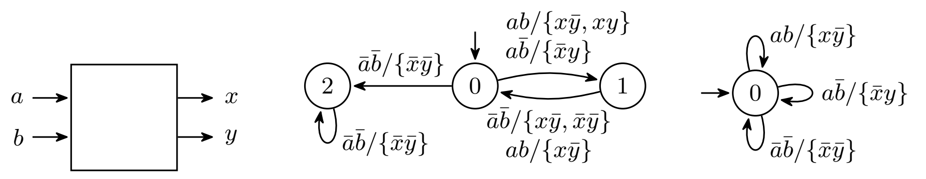

a
b
x
y
2
0
1
$\bar{a}\bar{b}/\{\bar{x}\bar{y}\}$
$ab/\{x\bar{y}, xy\}$
$a\bar{b}/\{\bar{x}y\}$
$\bar{a}\bar{b}/\{\bar{x}\bar{y}\}$
$\bar{a}\bar{b}/\{x\bar{y}, \bar{x}\bar{y}\}$
$ab/\{x\bar{y}\}$
0
$ab/\{x\bar{y}\}$
$a\bar{b}/\{\bar{x}y\}$
$\bar{a}\bar{b}/\{\bar{x}\bar{y}\}$
(a) A reactive controller (b) Original machine (c) Minimal machine

Fig. 1: Minimizing a Mealy machine that models a reactive controller

However, vague specifications may cause incompletely specified machines: for some states (i.e., nodes of the graph) and inputs, there may not exist a unique, explicitly defined output, but a set of valid outputs. Resolving those choices to a single output (among those allowed) will produce a fully specified machine that satisfies the initial specification, however those different choices may have an impact on the minimization of the machine. While minimizing fully specified machines is efficiently solvable [8], the problem is NP-complete for incompletely specified machines [14]. Hence, it may also be worth exploring faster algorithms that seek to reduce the number of states without achieving the optimal result.

Consider Fig. 1b: this machine is incompletely specified, as for instance state 0 allows multiple outputs for input ab (i.e., when both input variables a and b are true) and implicitly allows any output for input $\bar{a}b$ (i.e., only b is true) as it isn't constrained in any way by the specification. We can benefit from this flexibility in unspecified outputs to help reduce the automaton. For instance if we constrain state 2 to behave exactly as state 0 for inputs ab and $a\bar{b}$, then these two states can be merged. Adding further constraints can lead to the single-state machine shown in Fig. 1c. These smaller machines are not *equivalent*, but they are *compatible*: for any input stream, they can only produce output streams that could also have been produced by the original machine.

We properly define *Incompletely specified Generalized Mealy Machines* in Section 2 and provide a SAT-based minimization algorithm in Section 3. Since the minimization of incompletely specified Mealy machines is desirable but not crucial for reactive synthesis, we propose a faster reduction technique yielding "small enough" machines in Section 4. Finally, in Section 5 we benchmark these techniques against the state-of-the-art tool MeMin [1].

2 Definitions

Given a set of propositions (i.e., Boolean variables) X, let $\mathbb{B}^X$ be the set of all possible valuations on X, and let $2^{\mathbb{B}^X}$ be its set of subsets. Any element of $2^{\mathbb{B}^X}$ can be expressed as a Boolean formula over X. The negation of proposition p is denoted $\bar{p}$. We use $\top$ to denote the Boolean formula that is always true, or equivalently the set $\mathbb{B}^X$, and assume that X is clear from the context. A *cube* is a conjunction of propositions or their negations (i.e., literals). As an example, given three propositions a, b and c, the cube $a \wedge \bar{b}$, written $a\bar{b}$, stands for the set of all valuations such that a is true and b is false, i.e. $\{a\bar{b}c, a\bar{b}\bar{c}\}$. Let $\mathbb{K}^X$ stand for the set of all cubes over X. $\mathbb{K}^X$ contains the cube $\top$, that stands for the set of all possible valuations over X. Note that any set of valuations can be represented as a disjunction of disjoint cubes (i.e., not sharing a common valuation).

Definition 1. *An* Incompletely specified Generalized Mealy Machine *(IGMM) is a tuple $M = (I, O, Q, q_{init}, \delta, \lambda)$, where I is a set of* input propositions*, O a set of* output propositions*, Q a finite set of* states*, q_{init} an* initial state*, $\delta\colon \left(Q, \mathbb{B}^I\right) \to Q$ a partial* transition function*, and $\lambda\colon \left(Q, \mathbb{B}^I\right) \to 2^{\mathbb{B}^O} \setminus \{\emptyset\}$ an* output function *such that $\lambda(q, i) = \top$ when $\delta(q, i)$ is undefined. If δ is a total function, we then say that M is* input-complete*.*

It is worth noting that the transition function is input-deterministic but not complete with regards to Q as $\delta(q,i)$ could be undefined. Furthermore, the output function may return many valuations for a given input valuation and state. This is not an unexpected definition from a reactive synthesis point of view, as a given specification may yield multiple compatible output valuations for a given input.

Definition 2 (Semantics of IGMMs). *Let $M = (I, O, Q, q_{init}, \delta, \lambda)$ be an IGMM. For all $u \in \mathbb{B}^I$ and $q \in Q$, if $\delta(q,u)$ is defined, we write that $q \xrightarrow{u/v} \delta(q,u)$ for all $v \in \lambda(q,u)$. Given two infinite sequences of valuations $\iota = i_0 \cdot i_1 \cdot i_2 \cdots \in (\mathbb{B}^I)^\omega$ and $o = o_0 \cdot o_1 \cdot o_2 \cdots \in (\mathbb{B}^O)^\omega$, $(\iota, o) \models M_q$ if and only if:*

- *either there is an infinite sequence of states $(q_j)_{j\geq 0} \in Q^\omega$ such that $q = q_0$ and $q_0 \xrightarrow{i_0/o_0} q_1 \xrightarrow{i_1/o_1} q_2 \xrightarrow{i_2/o_2} \cdots$;*
- *or there is a finite sequence of states $(q_j)_{0\leq j\leq k} \in Q^{k+1}$ such that $q = q_0$, $\delta(q_k, i_k)$ is undefined, and $q_0 \xrightarrow{i_0/o_0} q_1 \xrightarrow{i_1/o_1} \cdots q_k$.*

We then say that starting from state q, M produces output o given the input ι.

Note that if $\delta(q_k, i_k)$ is undefined, the machine is allowed to produce an arbitrary output from then on. Furthermore, given an input word ι, there may be several output words o such that $(\iota, o) \models M_q$ (in accordance with a lax specification).

As an example, consider the input sequence $\iota = ab \cdot \bar{a}\bar{b} \cdot ab \cdot \bar{a}\bar{b} \cdots$ applied to the initial state 0 of the machine shown in Figure 1b. We have $(\iota, o) \models M_0$ if and only if for all $j \in \mathbb{N}$, $o_{2j} \in x$ and $o_{2j+1} \in \bar{y}$, where x and $\bar{y}$ are cubes that respectively represent $\{xy, x\bar{y}\}$ and $\{x\bar{y}, \bar{x}\bar{y}\}$.

Definition 3 (Variation and specialization). *Let $M = (I, O, Q, q_{init}, \delta, \lambda)$ and $M' = (I, O, Q', q'_{init}, \delta', \lambda')$ be two IGMMs. Given two states $q \in Q$, $q' \in Q'$, we say that q' is a:*

- variation *of q if $\forall \iota \in (\mathbb{B}^I)^\omega, \{o \mid (\iota, o) \models M'_{q'}\} \cap \{o \mid (\iota, o) \models M_q\} \neq \emptyset$;*
- specialization *of q if $\forall \iota \in (\mathbb{B}^I)^\omega, \{o \mid (\iota, o) \models M'_{q'}\} \subseteq \{o \mid (\iota, o) \models M_q\}$.*

We say that M' is a variation (resp. specialization) of M if q'_{init} is a variation (resp. specialization) of q_{init}.

Intuitively, all the input-output pairs accepted by a specialization q' in M' are also accepted by q in M. Therefore, if all the outputs produced by state q in M comply with the original specification, then so do the outputs produced by state q' in M'. In order for two states to be a variation of one another, for all possible inputs they must be able to agree on a common output behaviour.

We write $q' \approx q$ (resp. $q' \sqsubseteq q$) if q' is a variation (resp. specialization) of q. Note that $\approx$ is a symmetric but non-transitive relation, while $\sqsubseteq$ is transitive ($\sqsubseteq$ is a preorder).

Our goal in this article is to solve the following problems:

Reducing an IGMM M: finding a specialization of M having at most the same number of states, preferably fewer.

Minimizing an IGMM M: finding a specialization of M having the least number of states.

Consider again the IGMM shown in Figure 1b. The IGMM shown in Figure 1c is a specialization of this machine and has a minimal number of states.

Generalizing inputs and outputs. Note that the output function of an IGMM returns a set of valuations, but it can be rewritten equivalently to output a set of cubes as $\lambda\colon \big(Q, \mathbb{B}^I\big) \to 2^{\mathbb{K}^O}$. As an example, consider $I = \{a\}$ and $O = \{x, y, z\}$; the set of valuations $v = \{\bar{x}yz, \bar{x}y\bar{z}, x\bar{y}z, x\bar{y}\bar{z}\} \in 2^{\mathbb{B}^O}$ is equivalent to the set of cubes $v_c = \{\bar{x}y, x\bar{y}\} \in 2^{\mathbb{K}^O}$.

In the literature, a Mealy machine commonly maps a single input valuation to a single output valuation: its output function is therefore of the form $\lambda\colon \big(Q, \mathbb{B}^I\big) \to \mathbb{B}^O$. The tool MeMin [1] uses a slight generalization by allowing a single output cube, hence $\lambda\colon \big(Q, \mathbb{B}^I\big) \to \mathbb{K}^O$. Thus, unlike our model, neither the common definition nor the tool MeMin can feature an edge outputting the aforementioned set v (or equivalently v_c), as it cannot be represented by a single cube or valuation. Our model is therefore *strictly more expressive*, although it comes at a price for minimization.

Note that, in practice, edges with identical source state, output valuations, and destination state can be merged into a single transition labeled by the set of allowed inputs. Both our tool and MeMin feature this optimization. While it does not change the expressiveness of the underlying model, this more succinct representation of the machines does improve the efficiency of the algorithms detailed in the next section, as they depend on the total number of transitions.

3 SAT-Based Minimization of IGMM

This section builds upon the approach presented by Abel and Reineke [1] for machines with outputs constrained to cubes, and generalizes it to the IGMM model (with more expressive outputs).

3.1 General approach

Definition 4. *Given an IGMM $M = (I, O, Q, q_{init}, \delta, \lambda)$,* a variation class $C \subseteq Q$ *is a set of states such that all elements are pairwise variations, i.e.* $\forall q, q' \in C$, $q' \approx q$. *For any input $i \in \mathbb{B}^I$, we define:*

- *the* successor function $\mathrm{Succ}(C, i) = \bigcup_{q \in C} \{\delta(q, i) \mid \delta(q, i)$ *is defined*$\}$*;*
- *the* output function $\mathrm{Out}(C, i) = \bigcap_{q \in C} \lambda(q, i)$.

Intuitively, the successor function returns the set of all states reachable from a given class under a given input symbol. The output function returns the set of all shared output valuations between the various states in the class.

In the remainder of this section we will call a variation class simply a class, as there is no ambiguity. We consider three important notions concerning classes, or rather sets thereof, of the form $S = \{C_0, \ldots, C_{n-1}\}$.

Definition 5 (Cover condition). *We say that a set of classes S* covers *the machine M if every state of M appears in at least one of the classes.*

Definition 6 (Closure condition). *We say that a set of classes S is* closed *if for all $C_j \in S$ and for all inputs $i \in \mathbb{B}^I$ there exists a $C_k \in S$ such that $\mathrm{Succ}(C_j, i) \subseteq C_k$.*

Definition 7 (Nonemptiness condition). *We say that a class C has a* nonempty output *if $\mathrm{Out}(C, i) \neq \emptyset$ for all inputs $i \in \mathbb{B}^I$.*

The astute reader might have observed that the nonempty output condition is strictly stronger than the condition that all elements in a class have to be pairwise variations of one another. We will see that this distinction is however important, as it gives rise to a different set of clauses in the SAT problem, reducing the total runtime.

Combining these conditions yields the main theorem for this approach. This extends a similar theorem by Abel and Reineke [1, Thm 1] by adding the nonemptiness condition to support the more expressive IGMM model.

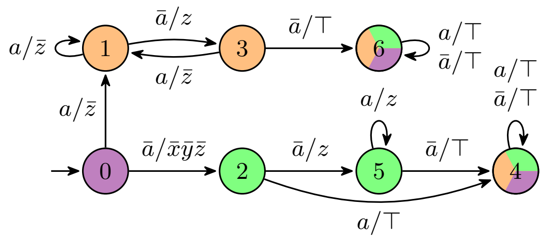
Theorem 1. *Let $M = (I, O, Q, q_{init}, \delta, \lambda)$ be an IGMM and $S = \{C_0, \ldots, C_{n-1}\}$ be a* minimal *(in terms of size) set of classes such that (1) S is* closed, *(2) S* covers *every state of the machine M and (3) each of the classes C_j has a* nonempty output. *Then the IGMM $M' = (I, O, S, q'_{init}, \delta', \lambda')$ where:*

- $q'_{init} = C$ *for some* $C \in S$ *such that* $q_{init} \in C$;
- $\delta'(C_j, i) = \begin{cases} C_k \text{ for some } k \text{ s.t. } \mathrm{Succ}(C_j, i) \subseteq C_k & \text{if } \mathrm{Succ}(C_j, i) \neq \emptyset \\ \text{undefined} & \text{else;} \end{cases}$
- $\lambda'(C_j, i) = \begin{cases} \mathrm{Out}(C_j, i) & \text{if } \mathrm{Succ}(C_j, i) \neq \emptyset \\ \top & \text{else;} \end{cases}$

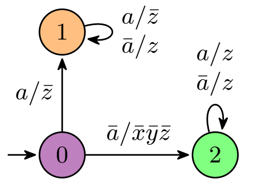
is a specialization *of minimal size (in terms of states) of M.*

Figure 2a illustrates this construction on an example with a single input proposition $I = \{a\}$ (hence two input valuations $\mathbb{B}^I = \{a, \bar{a}\}$), and three output propositions $O = \{x, y, z\}$. To simplify notations, elements of $2^{\mathbb{B}^O}$ are represented as Boolean functions (happening to be cubes in this example) rather than sets.

States have been colored to indicate their possible membership to one of the three variational classes. The SAT solver needs to associate each state to at least one of them in order to satisfy the cover condition (5), while simultaneously respecting conditions (6)–(7). A possible choice would be: $C_0 = \{0\}$, $C_1 = \{1, 3, 6\}$, and $C_2 = \{2, 4, 5\}$. For this choice, the *violet* class C_0 has only a single state, so the closure condition (6) is trivially satisfied. All transitions of the states in the *orange* class C_1 go to states in C_1, also satisfying the condition. The same can be said of the *green* class C_2.



(a) Original IGMM M



(b) Minimal specialization of M

Fig. 2: Minimization example

Finally, we need to check the nonempty output condition (7). Once again, it is trivially satisfied for the *violet* class C_0. For the *orange* and *green* classes, we need to compute their respective output. We get $\mathrm{Out}(C_1, a) = \bar{z}$, $\mathrm{Out}(C_1, \bar{a}) = z$, $\mathrm{Out}(C_2, a) = \bar{z}$ and $\mathrm{Out}(C_2, \bar{a}) = z$. None of the output sets is empty, thus condition (7) is satisfied as well. Note that, since the outgoing transitions of states 4 and 6 are self-loops compatible with all possible output valuations, another valid choice is: $C_0 = \{0, 4, 6\}$, $C_1 = \{1, 3, 4, 6\}$, and $C_2 = \{2, 4, 5, 6\}$.

The corresponding specialization, constructed as described in Theorem 1, is shown in Figure 2b. Note that this machine is input-complete, so the incompleteness of the specification only stems from the possible choices in the outputs.

3.2 Proposed SAT Encoding

We want to design an algorithm that finds a minimal specialization of a given IGMM M. To do so, we will use the following approach, starting from $n = 1$:

- Posit that there are n classes, hence, n states in the minimal machine.
- Design SAT clauses ensuring cover, closure and nonempty outputs.
- Check if the resulting SAT problem is satisfiable.
- If so, construct the minimal machine described in Theorem 1.
- If not, increment n by one and apply the whole process again, unless $n = |Q| - 1$, which serves as a proof that the original machine is already minimal.

Encoding the cover and closure conditions. In order to guarantee that the set of classes $S = \{C_0, \ldots, C_{n-1}\}$ satisfies both the cover and closure conditions and that each class C_j is a variation class, we need two types of literals:

- $s_{q,j}$ should be true if and only if state q belongs to the class C_j;
- $z_{i,k,j}$ should be true if $\mathrm{Succ}(C_k, i) \subseteq C_j$ for $i \in \mathbb{B}^I$.

The cover condition, encoded by Equation (1), guarantees that each state belongs to at least one class.

$$\bigwedge_{q\in Q} \bigvee_{0\le j<n} s_{q,j} \qquad (1) \qquad\qquad \bigwedge_{0\le j<n} \bigwedge_{\substack{q,q'\in Q\\ q\not\approx q'}} \overline{s_{q,j}} \vee \overline{s_{q',j}} \qquad (2)$$

Equation (2) ensures that each class is a variational class: two states q and q' that are not variations of each other cannot belong to the same class.

The closure condition must ensure that for every class C_i and every input symbol $i \in \mathbb{B}^I$, there exists at least one class that contains all the successor states: $\forall k, \forall i, \exists j,\ \mathit{Succ}(C_k, i) \subseteq C_j$. This is expressed by the constraints (3) and (4).

$$\bigwedge_{0\le k<n} \bigwedge_{i\in\mathbb{B}^I} \bigvee_{0\le j<n} z_{i,k,j} \quad (3) \qquad \bigwedge_{0\le j,k<n} \bigwedge_{\substack{q,q'\in Q, i\in\mathbb{B}^I\\ q'=\delta(q,i)}} (z_{i,k,j} \wedge s_{q,k}) \rightarrow s_{q',j} \quad (4)$$

The constraint (3) ensures that at least one C_j contains $\mathrm{Succ}(C_k, i)$, while (4) ensures this mapping of classes matches the transitions of M.

Encoding the nonempty output condition. Each class in S being a variation class is necessary but not sufficient to satisfy the nonempty output condition. We indeed want to guarantee that for any input i, all states in a given class can agree on at least one common output valuation.

However it is possible to have three or more states (like $\textcircled{0}\!\circlearrowleft a/\{xy, x\bar{y}\}$, $\textcircled{1}\!\circlearrowleft a/\{\bar{x}y, x\bar{y}\}$, and $\textcircled{2}\!\circlearrowleft a/\{xy, \bar{x}y\}$) that are all variations of one another, but still cannot agree on a common output.

This situation cannot occur in MeMin since their model uses *cubes* as outputs rather than arbitrary sets of valuations as in our model. A useful property of cubes is that if the pairwise intersections of all cubes in a set are nonempty, then the intersection of all cubes in the set is necessarily nonempty as well.

Since *cubes* are not expressive enough for our model, we will therefore generalize the output as discussed earlier in Section 2: we represent the arbitrary set of valuations produced by the output function λ as a set of cubes whose disjunction yields the original set. For $q \in Q$ and $i \in \mathbb{B}^I$, we partition the set of valuations $\lambda(q, i)$ into cubes, relying on the Minato [11] algorithm, and denote the obtained set of cubes as $\mathrm{CS}(\lambda(q, i))$.

Our approach for ensuring that there exists a common output is to search for disjoint cubes and exclude them from the possible outputs by selectively deactivating them if necessary; an active cube is a set in which we will be looking for an output valuation that the whole class can agree on. To express this, we need two new types of literals:

– $a_{c,q,i}$ should be true iff the particular instance of the cube $c \in \mathrm{CS}(\lambda(q, i))$ used in the output of state q when reading i is *active*;
– $sc_{q,q'}$ should be true iff $\exists C_j \in S$ such that $q \in C_j$ and $q' \in C_j$

The selective deactivation of a cube can then be expressed by the following:

$$\bigwedge_{\substack{q,q'\in Q\\ 0\le j<n}} (s_{q,j} \wedge s_{q',j}) \rightarrow sc_{q,q'} \quad (5) \qquad \bigwedge_{\substack{q\in Q,\, i\in\mathbb{B}^I\\ \delta(q,i) \text{ is defined}}} \bigvee_{c\in\mathrm{CS}(\lambda(q,i))} a_{c,q,i} \quad (6)$$

$$\bigwedge_{\substack{q,q'\in Q,\, i\in\mathbb{B}^I\\ \delta(q,i)\text{ is defined}\\ \delta(q',i)\text{ is defined}}} \bigwedge_{\substack{c\in\mathrm{CS}(\lambda(q,i))\\ c'\in\mathrm{CS}(\lambda(q',i))\\ c\cap c'=\emptyset}} (a_{c,q,i} \wedge a_{c',q',i}) \rightarrow \overline{sc_{q,q'}}. \qquad (7)$$

Constraint (5) ensures that $sc_{q,q'}$ is true if there exists a class containing both q and q', in accordance with the expected definition.

Constraint (6) guarantees that at least one of the cubes in the output $\lambda(q,i)$ is active, causing the restricted output to be nonempty.

Constraint (7) expresses selective deactivation and only needs to be added for a given $q, q' \in Q$ and $i \in \mathbb{B}^I$ if $\delta(q,i)$ and $\delta(q',i)$ are properly defined. This formula guarantees that if there exists a class to which q and q' belong to (i.e., $sc_{q,q'}$ is true) but there also exist disjoint cubes in the partition of their respective outputs, then we deactivate at least one of these: only cubes that intersect can be both activated. Thus, this constraint guarantees the nonempty output condition.

Since encoding an output set requires a number of cubes exponential in $|O|$, the above encoding uses $\mathrm{O}(|Q|(2^{|I|+|O|} + |Q|) + n^2 \cdot 2^{|I|})$ variables as well as $\mathrm{O}(Q^2(n+2^{2|O|})+n^2\cdot 2^{|I|}+|\delta|(2^{|O|}+n^2))$ clauses. We use additional optimizations to limit the number of clauses, and make the algorithm more practical despite its frightening theoretical worst case. In particular the CEGAR approach of Section 3.3 strives to avoid introducing constraints (5)–(7).

3.3 Adjustment of Prior Optimizations

Constructing the SAT problem iteratively starting from $n = 1$ would be grossly inefficient. We can instead notice that two states that are not variations of each other can never be in the same class. Thus, assuming we can find k states that are not pairwise variations of one another, we can infer that we need at least as many classes as there are states in this set, providing a lower bound for n. This idea was first introduced in [1]; however, performing a more careful inspection of the constraints with respect to this "partial solution" allows us to reduce the number of constraints and literals needed.

The nonemptiness condition involves the creation of many literals and clauses and necessitates an expensive preprocessing step to decompose the arbitrary output sets returned by output function ($\lambda\colon \left(Q, \mathbb{B}^I\right) \rightarrow 2^{\mathbb{B}^O} \setminus \{\emptyset\}$) into disjunctions of cubes ($\lambda\colon \left(Q, \mathbb{B}^I\right) \rightarrow 2^{\mathbb{K}^O} \setminus \{\emptyset\}$). We avoid adding unnecessary nonempty output clauses in a counter-example guided fashion. Violation of these conditions can easily be detected before constructing the minimized machine. If detected, a small set of these constraints is added to SAT problem excluding this particular violation. In many cases, this optimization greatly reduces the number of literals and constraints needed, to the extent we can often avoid their use altogether.

From now on, we consider an IGMM with N states $Q = \{q_0, q_1, \ldots, q_{N-1}\}$.

Variation matrix. We first need to determine which states are not pairwise variations of one another in order to extract a partial solution and perform simplifications on the constraints. We will compute a square matrix of size $N \times N$ called mat such that $\mathrm{mat}[k][\ell] = 1$ if and only if $q_k \not\approx q_\ell$ in the following fashion:

1. Initialize all entries of mat to 0.
2. Iterate over all pairs (k, ℓ) with $0 \leq k < \ell < N$. If the entry $\mathrm{mat}[k][\ell]$ is 0, check if $\exists i \in \mathbb{B}^I$ such that $\lambda(q_k, i) \cap \lambda(q_l, i) = \emptyset$. If it exists, $\mathrm{mat}[k][\ell] \leftarrow 1$.
3. For all pairs (k, ℓ) whose associated value $\mathrm{mat}[k][\ell]$ changed from 0 to 1, set all existing predecessor pairs (m, n) with $m < n$ under the same input to 1 as well, that is, $\exists i \in \mathbb{B}^I$ such that $\delta(q_m, i) = q_k$ and $\delta(q_n, n) = q_l$. Note that we may need to propagate these changes to the predecessors of (m, n).

As "being a variation of" is a symmetric, reflexive relation, we only compute the elements above the main diagonal of the matrix. The intuition behind this algorithm is that two states q and q' are not variations of one another if either:

– There exists an input symbol for which the output sets are disjoint.
– There exists a pair of states which are not variations of one another and that can be reached from q and q' under the same input sequence.

The complexity of this algorithm is $\mathrm{O}(|Q|^2 \cdot 2^{|I|})$ if we assume that the disjointness of the output sets can be checked in constant time; see [1]. This assumption is not correct in general: testing disjointness for cubes has a complexity linear in the number of input propositions. On the other hand, testing disjointness for generalized Mealy machines that use arbitrary sets of valuations has a complexity exponential in the number of input propositions. This increased complexity is however counterbalanced by the succinctness the use of arbitrary sets allows.

As an example, given $2m$ output propositions $o_0, \ldots, o_{2m-1}$, consider the set of output valuations expressed as a disjunction of cubes $\bigvee_{0 \leq k < m} o_{2k}\, \overline{o_{2k+1}} \vee \overline{o_{2k}}\, o_{2k+1}$. Exponentially many *disjoint* cubes are needed to represent this set. Thus, a non-deterministic Mealy machine labeled by output cubes will incur an exponential number of computations performed in linear time, whereas a generalized Mealy machine will only perform a single test with exponential runtime.

Computing a partial solution. The partial solution corresponds to a set of states such that none of them is a variation of any other state in the set. Thus, none of these states can belong to the same (variation) class. The size of this set is therefore a lower bound for the number of states in the minimal machine.

Finding the largest partial solution is an NP-hard problem; we therefore use the greedy heuristic described in [1]. For each state q of M, we count the number of states q' such that q is not a variation of q'; call this number nvc_q. We then successively add to the partial solution the states that have the highest nvc_q but are not variations of any state already inserted.

CEGAR approach to ensure the nonempty output condition. Assuming a solution satisfying the cover and closure constraints has already been found, we then need to check if said solution satisfies the nonempty output condition. If this is indeed the case, we can then construct and return a minimal machine.

```
Data: a machine $M = (I, O, Q, q_{init}, \delta, \lambda)$
Result: a minimal specialization $M'$
/* Computing the variation matrix */
bool[][] mat ← isNotVariationOf($M$);
/* Looking for a partial solution P */
set $P$ ← extractPartialSol(mat);
clauses ← empty list;
/* Using the lower bound inferred from P */
for $n \leftarrow |P|$ to $|Q| - 1$ do
    addCoverCondition(clauses, $M$, $P$, mat, $n$);
    addClosureCondition(clauses, $M$, $P$, mat, $n$);
    /* Solving the cover and closure conditions */
    (sat, solution) ← satSolver(clauses);
    while sat do
        if verifyNonEmpty(M, solution) then
            return buildMachine($M$, solution);
        /* Adding the relevant nonemptiness clauses */
        addNonemptinessCondition(clauses, $M$, solution);
        (sat, solution) ← satSolver(clauses);
/* If no solution has been found, return M */
return copyMachine($M$);
```

Algorithm 1: SAT-based minimization

If the condition is not satisfied, we look for one or more combinations of classes and input symbols such that $\mathrm{Succ}(C_k, i) = \emptyset$. We add for the states in C_k and the input symbol i the constraints described in Section 3.2, and for these states and input symbols only. Then we check if the problem is still satisfiable.

If it is not, then we need to increase the number of classes to find a valid solution. If it is, the solution either respects condition (7) and we can return a minimal machine, or it does not and the process of selectively adding constraints is repeated. Either way, this *counter-example guided abstraction refinement* (CEGAR) scheme ensures termination, as the problem is either shown to be unsatisfiable or solved through iterative exclusion of all violations of condition (7).

3.4 Algorithm

The optimizations described previously yield Algorithm 1.

Further optimizations and comparison to MeMin. The proposed algorithm relies on the general approach outline in [1], as well as the SAT encoding for the cover and closure conditions. We find a partial solution by using a similar heuristic and adapt some optimizations found in their source code, which are neither detailed in their paper nor here due to a lack of space.

The main difference lies in the increased expressiveness of the input and output symbols that causes some significant changes. In particular, we added the nonemptiness condition to guarantee correctness, as well as a CEGAR-based implementation to maintain performance. Other improvements mainly stem from a better usage of the partial solution.

For instance, each state q of the partial solution is associated to “its own” class C_j. Since the matching literal $s_{q,j}$ is trivially true, it can be omitted by replacing all its occurrences by true. States belonging to the partial solution have other peculiarities that can be leveraged to reduce the number of possible successor classes, further reducing the amount of literals and clauses needed.

We therefore require fewer literals and clauses, trading a more complex construction of the SAT problem for a reduced memory footprint. The impact of these improvements is detailed in Section 5.

The Mealy machine described by [1] come in two flavors: One with an explicit initial state and a second one where all states are considered to be possible initial states. While our approach does explicit an initial state, it does not further influence the resulting minimal machine when all original states are reachable.

4 Bisimulation with Output Assignment

We introduce in this section another approach tailored to our primary use case, that is, efficient reduction of control strategies in the context of reactive synthesis. This technique, based on the $\sqsubseteq$ specialization relation, yields non-minimal but “relatively small” machines at significantly reduced runtimes.

Given two states q and q' such that $q' \sqsubseteq q$, one idea is to restrict the possible outputs of q to match those of q'. Concretely, for all inputs $i \in \mathbb{B}^I$, we restrict $\lambda(q, i)$ to its subset $\lambda(q', i)$; q and q' thus become bisimilar, allowing us to merge them. In practice, rather than restricting the output first then reducing bisimilar states to their quotient, we instead directly build a machine that is minimal with respect to $\sqsubseteq$ where all transitions going to q are redirected to q'.

Note that if two states q and q' are bisimilar, then necessarily $q' \sqsubseteq q$ and $q \sqsubseteq q'$: therefore, both states will be merged by our approach. As a consequence, the resulting machine is always smaller than the bisimulation quotient of the original machine (as shown in Section 5).

4.1 Reducing Machines with $\sqsubseteq$

Our algorithm builds upon the following theorem:

Theorem 2. *Let $M = (I, O, Q, q_{init}, \delta, \lambda)$ be an IGMM, and $r\colon Q \to Q$ be a mapping satisfying $r(q) \sqsubseteq q$. Define $M' = (I, O, Q', q'_{init}, \delta', \lambda)$ as an IGMM where $Q' = r(Q)$, $q'_{init} = r(q_{init})$ and $\delta'(q, i) = r(\delta(q, i))$ for all states q and input i. Then M' is a specialization of M.*

Intuitively, if a state q is remapped to a state $q' \sqsubseteq q$, then the set of words w that can be output for an input i is simply reduced to a subset of the original output. The smaller the image $r(Q)$, the more significant the reduction performed on the machine. Thus, to find a suitable function r, we map each state q to one of the *minimal elements* of the $\sqsubseteq$ preorder, also called the *representative states.*

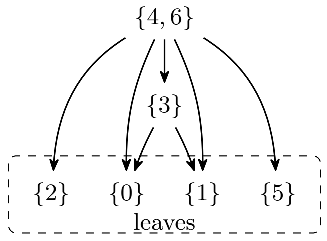


Fig. 3: Specialization graph of the IGMM of Fig. 2a

q	$r(q)$
$0 \rightarrow$	0
$1 \rightarrow$	1
$2 \rightarrow$	2
$3 \rightarrow$	1
$4 \rightarrow$	1
$5 \rightarrow$	5
$6 \rightarrow$	1

Fig. 4: Chosen representative mapping.

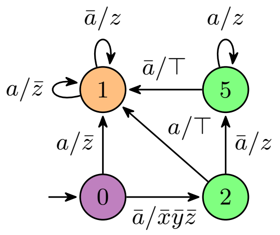


Fig. 5: IGMM obtained by reducing that of Fig. 2a

Definition 8 (Specialization graph). *A* specialization graph *of an IGMM $M = (I, O, Q, q_{init}, \delta, \lambda)$ is the* condensation graph *of the directed graph representing the relation $\sqsubseteq$: the vertices of the specialization graph are sets that form a partition of Q such that two states q and q' belong to the same vertex if $q \sqsubseteq q'$ and $q' \sqsubseteq q$; there is an edge $\{q_1, q_2, ...\} \longrightarrow \{q'_1, q'_2, ...\}$ if and only if $q'_i \sqsubseteq q_j$ for some (or equivalently all) i, j. Note that this graph is necessarily acyclic.*

Fig. 3 shows the specialization graph associated to the machine of Fig. 2a.

Definition 9 (Representative of a state). *Given two states q and q' of an IGMM, q' is a* representative *of q if, in the specialization graph of M, q' belongs to a leaf that can be reached from the vertex containing q. In other words, q' is a representative of q if $q' \sqsubseteq q$ and q' is a minimal element of the $\sqsubseteq$ preorder.*

Note that any state has at least one representative. In Fig. 3 we see that 0 represents 0, 3, 4, and 6. States 3, 4, and 6 can be represented by 0 or 1.

By picking one state in each leaf, we obtain a set of representative states that cover all states of the IGMM. We then apply Theorem 2 to a function r that maps each state to its representative in this cover. In Fig. 3, all leaves are singletons, so the set $\{0, 1, 2, 5\}$ contains representatives for all states. Applying Th. 2 using r from Fig. 4 yields the machine shown in Fig. 5. Note that while this machine is smaller than the original, it is still bigger than the minimal machine of Fig. 2b, as this approach does not appraise the variation relation $\approx$.

4.2 Implementing $\sqsubseteq$

We now introduce an effective decision procedure for $q \sqsubseteq q'$. Note that $\sqsubseteq$ can be defined recursively like a simulation relation. Assuming, without loss of generality, that the IGMM is input-complete, $\sqsubseteq$ is the coarsest relation satisfying:

$$q' \sqsubseteq q \Longrightarrow \forall i \in \mathbb{B}^I, \begin{cases} \lambda(q', i) \subseteq \lambda(q, i) \\ \delta(q', i) \sqsubseteq \delta(q, i) \end{cases}$$

As a consequence, $\sqsubseteq$ can be decided using any technique that is suitable for computing simulation relations [7, 6]. Our implementation relies on a straightforward adaptation of the technique of signatures described by Babiak et al. [4, Sec. 4.2]: for each state q, we compute its *signature* $\mathrm{sig}(q)$, that is, a Boolean formula (represented as a BDD) encoding the outgoing transitions of that state such that $\mathrm{sig}(q) \Rightarrow \mathrm{sig}(q')$ if and only if $q \sqsubseteq q'$. Using these signatures, it becomes easy to build the *specialization graph* and derive a remapping function r.

Note that, even if $\sqsubseteq$ can be computed like a simulation, we do not use it to build a bisimulation quotient. The remapping applied in Th. 2 does not correspond to the quotient of M by the equivalence relation induced by $\sqsubseteq$.

5 Benchmarks

The two approaches described in Sections 3 and 4 have been implemented within Spot 2.10 [5], a toolbox for ω-automata manipulation, and used in our SyntComp'21 submission [15]. The following benchmarks are based on a development version of Spot[1] that features efficient variation checks (verifying whether $q \approx q'$) thanks to an improved representation of cubes.

We benchmark the two proposed approaches against MeMin, against a simple bisimulation-based approach, and against one another. The MeMin tool has already been shown [1] to be superior to existing tools like Bica [13], Stamina [16], and Cosme [3]; we are not aware of more recent contenders. For this reason, we only compare our approaches to MeMin.

In a similar manner to Abel and Reineke [1], we use the ISM benchmarks [10] as well as the MCNC benchmark suite [17]. These benchmarks share a severe drawback: they only feature very small instances. MeMin is able to solve any of these instances in less than a second. We therefore extend the set of benchmarks with our main use-cases: Mealy machines corresponding to control strategies obtained from SYNTCOMP LTL specifications [9].

As mentioned in Section 2, MeMin processes Mealy machines, encoded using the the KISS2 input format [17], whose output can be chosen from a cube. However, the IGMM formalism we promote allows an arbitrary set of output valuations instead. This is particularly relevant for the SYNTCOMP benchmark, as the LTL specifications from which the sample's Mealy machines are derived often fail to fully specify the output. In order to (1) show the benefits of the generalized formalism while (2) still allowing comparisons with MeMin, we prepared two versions of each SYNTCOMP input: the "full" version features arbitrary sets of output valuations that cannot be processed by MeMin, while in the "cube" version said sets have been replaced by the first cube produced by the Minato algorithm [11] on the original output set. The ACM and MCNC benchmarks, on the other hand, already use a single output cube in the first place.

Figure 6 displays a log-log plot comparing our different methods to MeMin, using only the "cube" instances.[2]. The label "*bisim. w/ o.a.*" refers to the approach outlined in Section 4, "*bisim.*", to a simple bisimulation quotient, and "*SAT*", to the approach of Section 3. Points on the black diagonal stand for cases

[1] For instructions to reproduce, see `https://www.lrde.epita.fr/~philipp/forte22/`

[2] A 30 minute timeout was enforced for all instances. The benchmarks were run on an Asus G14 with a Ryzen 4800HS CPU with 16GB of RAM and no swap

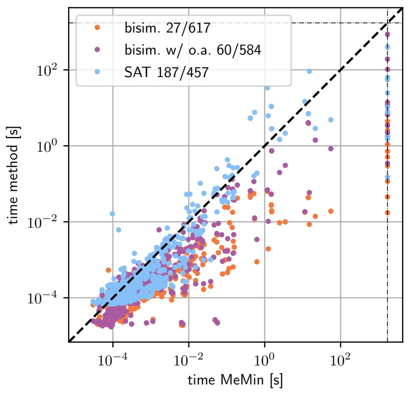


Fig. 6: Log-log plot of runtimes. The legend a/b stands for a cases above diagonal, and b below.

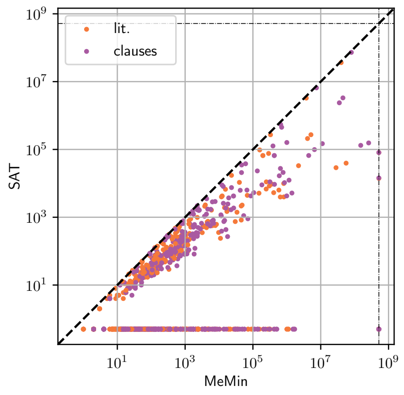


Fig. 7: Comparison of the number of literals and clauses in the encodings.

where MeMin and the method being tested had equal runtime; cases above this line favor MeMin, while cases below favor the aforementioned methods. Points on the dotted line at the edges of the figure represent timeouts. Only MeMin fails this way, on 10 instances. Figure 7 compares the maximal number of literals and clauses used to perform the SAT-based minimization by MeMin or by our implementation. These two figures only describe "cube" instances, as MeMin needs to be able to process the sample machines.

To study the benefits of our IGMM model's generic outputs, Table 1 compares the relative reduction ratios achieved by the various methods w.r.t. other methods as well as the original and minimal size of the sample machines. We use the "full" inputs everywhere with the exception of MeMin.

Interpretation. Reduction via bisimulation solves all instances and has been proven to be by far the fastest method (Fig. 6), but also the coarsest, with a mere 0.94 reduction ratio (Tab.1). Bisimulation with output assignment achieves a better reduction ratio of 0.83, very close to MeMin's 0.81.

In most cases, the proposed SAT-based approaches remain significantly slower than approaches based on bisimulation (Fig. 6). Our SAT-based algorithm is sometimes slower than MeMin's, as the model's increased expressiveness requires a more complex method. However, improving the use of partial solutions and increasing the expressiveness of the input symbols significantly reduce the size of the encoding of the intermediate SAT problems featured in our method (Fig. 7), hence, achieve a lower memory footprint. Points on the horizontal line at the bottom of Figure 7 correspond to instances that have already been proven minimal, since the partial solution is equal to the entire set of states: in these cases, no further reduction is required.

	>(1)	>(2)	>(3)	>(4)	$\frac{size}{orig}$		$\frac{size}{min}$		$\frac{size}{orig}$		$\frac{size}{min}$	
					avg.	md.	avg.	md.	avg.	md.	avg.	md.
original	114	304	271	314	1.00	1.0	6.56	1.0	1.00	1.00	12.23	1.77
(1) bisim (full)		249	214	275	0.94	1.0	1.85	1.0	0.88	1.00	2.72	1.50
(2) bisim w/ o.a. (full)		0	68	84	0.83	1.0	1.55	1.0	0.66	0.67	2.10	1.00
(3) MeMin (minimal cube)		74	0	77	0.81	1.0	1.13	1.0	0.63	0.69	1.27	1.00
(4) SAT (full)		0	0	0	0.77	1.0	1.00	1.0	0.54	0.56	1.00	1.00
	all 634 instances without timeout								314 non-minimal instances without timeout			

Table 1: Statistics about our three reduction algorithms. The leftmost pane counts the number of instances where algorithm (y) yields a smaller result than algorithm (x); as an example, bisimulation with output assignment (2) outperforms standard bisimulation (1) in 249 cases. The middle pane presents mean (avg.) and median (md.) size ratios relative to the original size and the minimal size of the sample machines. The rightmost pane presents similar statistics while ignoring all instances that were already minimal in the first place.

Finally, the increased expressiveness of our model results in significantly smaller minimal machines, as shown by the 1.27 reduction ratio of MeMin's cube-based machines compared to the minimisation of generic IGMMs derived from the same specification. There are also 74 cases where this superior expressiveness allows the bisimulation with output assignment to beat MeMin.

6 Conclusion

We introduced a generalized model for incompletely specified Mealy machines, whose output is an arbitrary choice between multiple possible valuations. We have presented two reduction techniques on this model, and compared them against the state-of-the-art minimization tool MeMin (where the output choices are restricted to a cube).

The first technique is a SAT-based approach inspired by MeMin [1] that yields a minimal machine. Thanks to this generalized model and an improved use of the partial solution, we use substantially fewer clauses and literals.

The second technique yields a reduced yet not necessarily minimal machine by relying on the notion of state specialization. Compared to the SAT-based approach, this technique offers a good compromise between the time spent performing the reduction, and the actual state-space reduction, especially for the cases derived from SYNTCOMP from which our initial motivation originated.

Both techniques are implemented in Spot 2.10. They have been used in our entry to the 2021 Synthesis Competition [15]. Spot comes with Python bindings that make it possible to experiment with these techniques and compare their respective effects[3].

[3] See: `https://spot.lrde.epita.fr/ipynb/synthesis.html`.

References

1. A. Abel and J. Reineke. MeMin: SAT-based exact minimization of incompletely specified Mealy machines. In *ICCAD'15*, pp. 94–101. IEEE Press, 2015.
2. A. Alberto and A. Simao. Minimization of incompletely specified finite state machines based on distinction graphs. In *LATW'09*, pp. 1–6. IEEE, 2009.
3. A. Alberto and A. Simao. Iterative minimization of partial finite state machines. *Open Computer Science*, 3(2):91–103, 2013. URL `https://doi.org/10.2478/s13537-013-0106-0`.
4. T. Babiak, T. Badie, A. Duret-Lutz, M. Křetínský, and J. Strejček. Compositional approach to suspension and other improvements to LTL translation. In *SPIN'13*, *LNCS* 7976, pp. 81–98. Springer, 2013.
5. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA'16*, *LNCS* 9938, pp. 122–129. Springer, 2016.
6. K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *Concur'00*, *LNCS* 1877, pp. 153–167. Springer, 2000.
7. M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of the 36th Symposium on Foundations of Computer Science*, pp. 453–462, 1995.
8. J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Proc. of an International Symposium on the Theory of Machines and Computations*, pp. 189–196. Academic Press, 1971.
9. S. Jacobs and R. Bloem. The 5th reactive synthesis competition-syntcomp 2018. 2020.
10. T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *31st Design Automation Conference*, pp. 684–690. IEEE, 1994.
11. S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *SASIMI'92*, pp. 64–73, 1992.
12. M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, EC-8(3):356–367, 1959.
13. 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*, 18(11):1619–1632, 1999.
14. C. P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Transactions on Computers*, C-22(12):1099–1102, 1973.
15. F. Renkin, P. Schlehuber, A. Duret-Lutz, and A. Pommellet. Improvements to `ltlsynt`. Presented at the SYNT'21 workshop, without proceedings., July 2021. `https://www.lrde.epita.fr/~adl/dl/adl/renkin.21.synt.pdf`.
16. 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*, 13(2):167–177, 1994.
17. S. Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Citeseer, 1991.

This appendix contains supplementary material that we could not fit into the main text. It was part of our submission, but has not been explicitly reviewed.

A Additional Data from our Benchmark

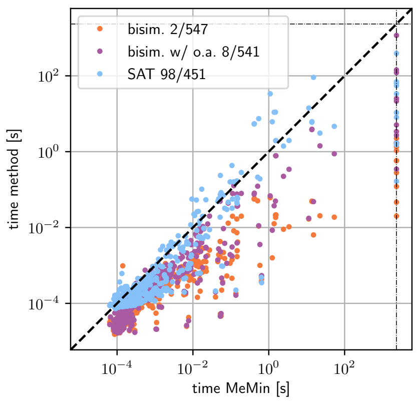
Figures 8a–10a plot the total runtime per instance of the three different benchmark suites (SYNTCOMP, MCNC, ISM) and provide a deeper insight into the results.

Even on "cube" instances, the two bisimulation-based (without and with output assignment) approaches outperform both MeMin and the proposed SAT approach of Section 3 at the cost of sometimes producing larger non-minimal machines. The detailed improvements are shown in Figures 8b–10b. The y axis represents here the ratio $\frac{size}{min}$, where min is the minimal size of Mealy machines computed on the "cube" instances (not the "full" ones).

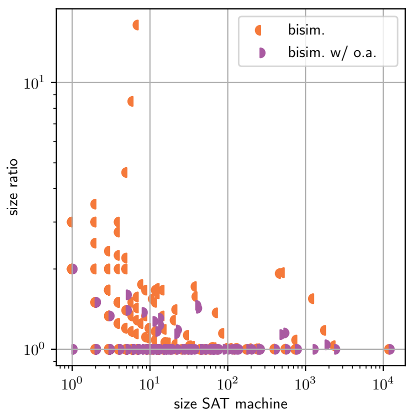
As shown by their size ratio, SYNTCOMP instances (Fig. 8b) significantly benefit from our bisimulation with output assignment approach compared to the other benchmark suites, even when the sample machines are restricted to "cube" instances,

Tables 2–3 complete Table 1 by displaying the corresponding runtime. Since the average runtime is heavily biased towards the largest instances, we also provide a geometric mean and a median runtime.

Finally Figure 12 shows sizes ratio of the reduction with the full instances, for all combined benchmarks. And Figure 11 shows how many instances each technique can solve under a given time.

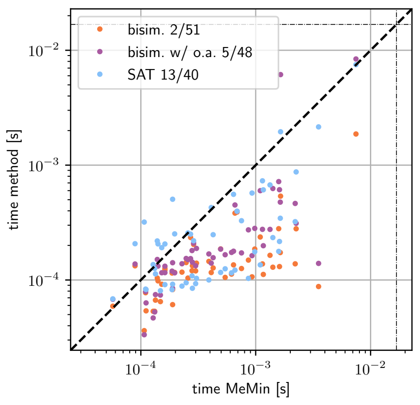


(a) Total runtime for instances derived from SYNTCOMP.

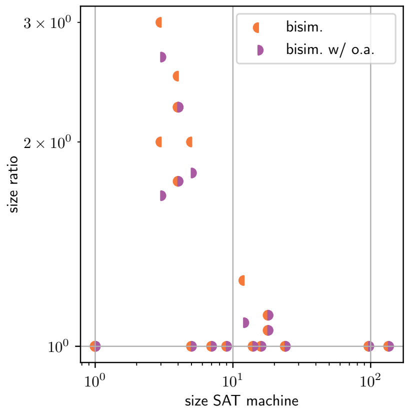


(b) Size ratios for instances derived from SYNTCOMP.

Fig. 8: Details for SYNTCOMP instances

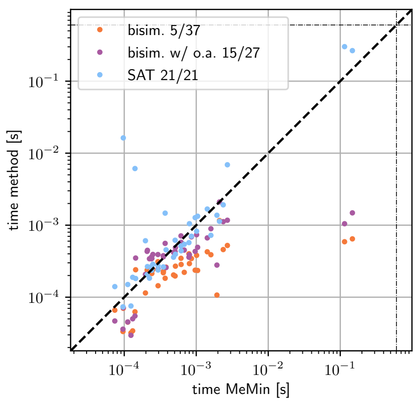


(a) Total runtime for MCNC instances.

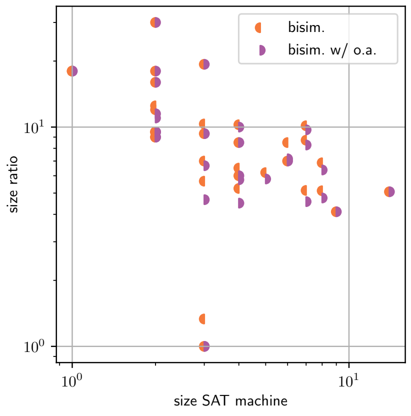


(b) Size ratios for MCNC instances.

Fig. 9: Details for MCNC instances



(a) Total runtime for ISM instances.



(b) Size ratios for ISM instances.

Fig. 10: Details for ISM instances

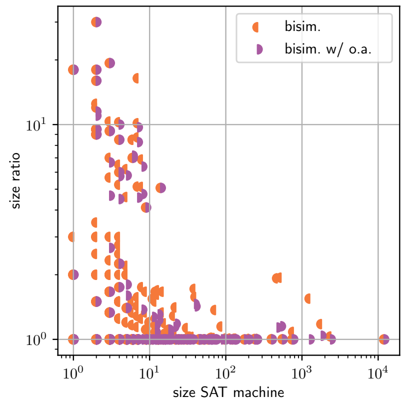


Fig. 11: Size ratios of the resulting machines of the full instances of all combined benchmarks.

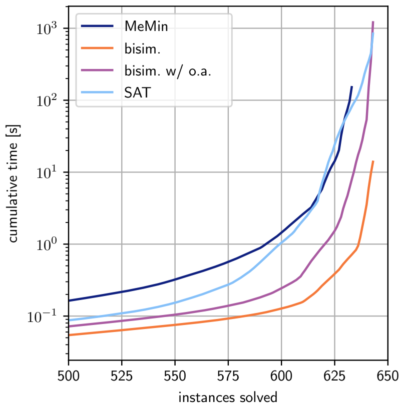


Fig. 12: Cactus plot of the different reduction techniques.

	a.mean	g.mean	median
bisim.	1.3	0.16	0.14
bisim. w/ o.a.	21.9	0.23	0.17
SAT	347.7	0.39	0.18
MeMin	239.6	0.56	0.28

Table 2: Arithmetic means (a.mean), geometric means (g.mean), and medians of the runtimes (in ms) of the four approaches on the set of 634 cases that MeMin was able to minimize.

	a.mean	g.mean	median
bisim.	1.1	0.21	0.18
bisim. w/ o.a.	11.6	0.32	0.21
SAT	602.5	0.73	0.28
MeMin	375.8	0.83	0.37

Table 3: Arithmetic means (a.mean), geometric means (g.mean), and medians of the runtimes (in ms) of the four approaches on the set of 314 *non-minimal* cases that MeMin was able to minimize.