Regular Model Checking Using Solver Technologies and Automata Learning[⋆]

Daniel Neider and Nils Jansen

RWTH Aachen University, Germany

Abstract. *Regular Model Checking* is a popular verification technique where large and even infinite sets of program configurations can be encoded symbolically by finite automata. Thereby, the handling of regular sets of *initial* and *bad configurations* often imposes a serious restriction in practical applications. We present two new algorithms both utilizing modern *solver technologies* and *automata learning.* The first one works in a CEGAR-like fashion by iteratively refining an abstraction of the reachable state space using counterexamples, while the second one is based on Angluin's prominent learning algorithm. We show the feasibility and competitiveness of our approaches on different benchmarks and compare them to other established tools.

1 Introduction

Model Checking is a prominent technique designed for the verification of safety-critical systems [1,2]. Combined with the feature of *counterexample* generation, this may not only help to show the defectiveness of a system but also to identify and correct its errors. Classic model checking is based on a rigorous exploration of state spaces, which leads to serious problems considering the large size of models for real world scenarios. Hence, for large or even *infinite* systems, feasible abstraction techniques or finite representations are needed.

One natural approach to overcome this problem is to encode states of a system, e. g., configurations of a program, as finite words and symbolically describe such systems by regular languages. *Regular Model Checking* [3] refers to a technique where the set of the program's *initial configurations* is given as a regular set while the program's transitions are defined in terms of a *finite state transducer*. Additionally, a regular set of *bad configurations* is considered that describes configurations of the program that must not occur during the program's execution. Although the regular sets and the transducer need to be devised manually from the system in question, Regular Model Checking has been applied to many practical examples with infinite state-spaces [4,5,6]. In [4], the authors also describe for some examples how a concrete system can be transformed into a Regular Model Checking instance.

[⋆] This work was partly supported by the German Research Council (DFG)as part of the research project CEBug (AB 461/1-1).

Tools for Regular Model Checking such as T(O)RMC [5], FASTER [6], and LEVER [7] compute regular sets that either encode the exact set of reachable configurations or overapproximate them. If such a set having an empty intersection with the bad configurations is identified, it is called a *proof* and serves as a witness that the program is correct. Note, that the problem of Regular Model Checking is undecidable in general. Thus, corresponding tools are necessarily based on *semi-algorithms*, i.e., algorithms that are not guaranteed to terminate on every input, but find a solution if one exists. Nonetheless, there is a large number of practical applications where good results are achieved (see, e.g., [5] and [6]).

A major drawback of all of these tools is that the computations are very expensive if the automata defining the sets of initial and bad configurations become large. Take, for instance, tools such as T(O)RMC and FASTER. They start with a DFA for the initial configurations, successively iterate the transducer, and then apply *widening* or *acceleration* to extrapolate infinite behavior. However, if the initial DFA is large, the performance of these approaches is often poor.

In this paper, we overcome this problem by combining advantages of state-of-the-art SAT and SMT solvers with automata learning techniques. Intuitively, our approach is a combination of two existing methods. The first is a SAT and SMT-based method for Regular Model Checking, which has recently been introduced in [8]. The second is an automata learning technique as described, e.g., in [9,10].

The fundamental idea of our approach is to abstract from the exact sets of initial and bad configurations by sampling them. More precisely, our approach works by generating a proof from a sample $\mathcal{S} = (S_+, S_-)$ containing finite (and small) approximations of the sets of initial and bad configurations. Using the sample sets, we compute a DFA that is consistent with these sets and *inductive*, i.e., closed, with respect to the transducer by means of SAT or SMT solvers (cf. Section 3). The resulting DFA contains at least the configurations reachable from S_+ via the transitions defined by the transducer and does not contain any configurations in S_-. If all original initial configurations and no bad configurations are contained, the DFA is a proof. If this is not the case, the respective approximation has to be refined and the process is iterated.

We propose two algorithms here that differ in the strategy to sample and refine sets of program configurations. Both are based on the popular learning framework introduced by Angluin [11], in which a regular language is learned in interaction with a so-called teacher that possesses knowledge about the language in question. The first algorithm (cf. Section 4.2) straightforwardly follows the idea of the CEGAR framework [12]: if the abstraction of either the initial or the bad configurations is too coarse to compute a satisfactory proof, a counterexample is given by the teacher and the abstraction is refined accordingly. The second one (cf. Section 4.3) follows a more elaborated procedure based on Angluin's learning algorithm [11], where additional queries ask whether individual configurations belong to a proof. These queries refine the abstraction further and remove the need of generating a new automaton at every step. Before we present our learning algorithms, Section 4.1 describes how an appropriate teacher can be built.

Our approach has several advantages. First, the canonical usage of established learning algorithms offers an effective way to sample and refine the abstraction of the program. Second, our technique is applicable even if the sets of initial and bad configurations are no longer regular—as long as an appropriate teacher can be constructed (e.g., for visibly or deterministic context free languages). Finally, learning algorithms typically produce small results, which highly increases the practical applicability of our approach. We demonstrate the latter claim in Section 5 by comparing a prototype of our approach to established tools.

2 Preliminaries

Finite Automata and Transducers. An *alphabet* Σ is a finite, non-empty set. A *word* $w = a_0 \dots a_n$ is a finite sequence of symbols $a_i \in \Sigma$ for $i = 0, \dots, n$; in particular, the *empty word* ε is the empty sequence. The *concatenation* of two words $u = a_0 \dots a_n$ and $v = b_0 \dots b_m$ is the word $u \cdot v = uv = a_0 \dots a_n b_0 \dots b_m$. If $u = vw$ for $u, v, w \in \Sigma^*$, we call v a *prefix* and w a *suffix* of u.

The set Σ^* is the set of all (finite) words over the alphabet Σ. A subset $L \subseteq \Sigma^*$ is called a *language*. For a language $L \subseteq \Sigma^*$, let the set of all prefixes of words in L be $\mathit{Pref}(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^* \colon uv \in L\}$.

A *(nondeterministic) finite automaton (NFA)* is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ consisting of a finite, non-empty set Q of states, an input alphabet Σ, an initial state $q_0 \in Q$, a transition relation $\Delta \subseteq Q \times \Sigma \times Q$, and a set $F \subseteq Q$ of final states. A *run* of an NFA $\mathcal{A}$ on a word $u = a_0 \dots a_n$ from a state $q \in Q$ is a sequence $\rho = q_0 \dots q_{n+1}$ such that $q_0 = q$ and $(q_i, a_i, q_{i+1}) \in \Delta$ for $i = 0, \dots, n$; as abbreviation we write $\mathcal{A} \colon q_0 \xrightarrow{u} q_{n+1}$. A word u is *accepted* by $\mathcal{A}$ if $\mathcal{A} \colon q_0 \xrightarrow{u} q$ with $q \in F$. The language $L(\mathcal{A}) = \{u \in \Sigma^* \mid \mathcal{A} \colon q_0 \xrightarrow{u} q, q \in F\}$ is the language of all words accepted by $\mathcal{A}$. A language L is called *regular* if there exists an NFA $\mathcal{A}$ such that $L = L(\mathcal{A})$. To measure the "complexity" of NFAs, we define the *size* of an NFA as $|Q|$, i.e., the number of its states.

A *deterministic finite automaton (DFA)* is an NFA where for all $p \in Q$ and $a \in \Sigma$ there exists a unique $q \in Q$ with $(p, a, q) \in \Delta$. In the case of DFAs, we substitute the transition relation Δ with a transition function $\delta \colon Q \times \Sigma \to Q$.

A *(finite-state) transducer* $\mathcal{T}$ is a special NFA working over the alphabet $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})$ with transitions of the form $(p, (a, b), q)$, $(p, (a, \varepsilon), q)$, and $(p, (\varepsilon, b), q)$. A transducer reads pairs of words and moves from state p to state q on reading $(u, v) \in \Sigma^* \times \Sigma^*$, denoted by $\mathcal{T} \colon p \xrightarrow{(u,q)} q$, if a sequence of transitions exists whose labels yield the pair (u, v) when concatenated componentwise. Rather than a regular language, a transducer accepts (or defines) a relation $R(\mathcal{T}) \subseteq \Sigma^* \times \Sigma^*$ where $R(\mathcal{T}) = \{(u, v) \mid \mathcal{T} \colon q_0 \xrightarrow{(u,v)} q, q \in F\}$. A relation $R \subseteq \Sigma^* \times \Sigma^*$ is called *rational* if there exists a transducer $\mathcal{T}$ such that $R = R(\mathcal{T})$.

For a relation $R \subseteq \Sigma^* \times \Sigma^*$ let R^* denote the reflexive and transitive closure of R. Moreover, for a language $L \subseteq \Sigma^*$ let $R(L)$ be the image of L under R defined by $R(L) = \{v \in \Sigma^* \mid \exists u \in L \colon (u, v) \in R\}$. Finally, if $R(L) \subseteq L$ holds, we call L a *regular invariant* or *inductive (with respect to R)*. Analogously, if $R(L(\mathcal{A})) \subseteq L(\mathcal{A})$ for some NFA (or DFA) $\mathcal{A}$, we call $\mathcal{A}$ inductive.

Regular Model Checking. In Regular Model Checking, a program $\mathcal{P} = (I, T)$ consists of a regular set $I \subseteq \Sigma^*$ of *initial configurations* over an a priori fixed alphabet Σ and a rational relation $T \subseteq \Sigma^* \times \Sigma^*$ defining the *transitions.* Regular Model Checking now asks whether there exists a path along the transitions from some initial configuration into a given regular set $B \subseteq \Sigma^*$ of *bad configurations*, which must never be reached. In other words, we are interested in answering the decision problem "Given a program $\mathcal{P} = (I, T)$ and a regular set $B \subseteq \Sigma^*$. Does $T^*(I) \cap B = \emptyset$ hold?". If the intersection is non-empty, we know that the program is erroneous. Note that Regular Model Checking is undecidable in general as rational relations are powerful enough to encode computations of Turing machines. Thus, the algorithms presented here are necessarily semi-algorithms.

A well-established approach to solve the Regular Model Checking problem used, e.g., by T(O)RMC [5] or LEVER [7] is to compute so-called *proofs.* Formally, a proof is a regular set $P \subseteq \Sigma^*$ such that $I \subseteq P$, $B \cap P = \emptyset$, and $T(P) \subseteq P$; for convenience, we also call an NFA (or DFA) $\mathcal{A}$ a proof if $L(\mathcal{A})$ is a proof. Note that any proof contains at least the set of reachable configurations and is, therefore, sufficient to prove a program correct. The advantage of computing a proof rather than the set of reachable configurations is that a proof might exists even if the set of reachable states itself is not regular.

Logics. In the remainder of this paper, we use both the propositional logic over Boolean variables and the quantifier-free logic over the integers with uninterpreted functions. *Formulas*, denoted by φ or—if the free variables are of interest—$\varphi(x_1, \ldots, x_n)$, are defined in the usual way. Moreover, a *model* of a formula $\varphi(x_1, \ldots, x_n)$ is a mapping $\mathfrak{M}\colon \{x_1, \ldots, x_n\} \to D_1 \times \ldots \times D_n$ that assigns to each variable x_i a value from the domain D_i of x_i such that φ evaluates to `true`. Moreover, if φ contains uninterpreted functions, then the model has to provide an interpretation of these functions. If $\mathfrak{M}$ is a model of φ, we write $\mathfrak{M} \models \varphi$.

Formulas defined in propositional logic can be solved by SAT solvers, and formulas defined in quantifier-free logic over the integers with uninterpreted functions can be solved by SAT-modulo-theories solvers (SMT), which is a generalization of the classical propositional satisfiability problem (SAT). Compared to SAT problems, in an SMT formula atomic propositions may be replaced by atoms of a given theory, in our case *uninterpreted functions.* Several tools for solving SAT and SMT formulae are available, e.g., GLUCOSER and Z3, respectively.

3 Inferring Inductive DFAs from Finite Samples

In this section, we present a solver-based approach to compute inductive DFAs from a finite sample of initial and bad configurations of a program. Remember that the procedure of our Regular Model Checking approach works roughly as follows: we provide a procedure to compute a DFA that is consistent with a finite sample and inductive with respect to a given transducer. Starting with an empty sample, we use automata learning techniques to extend the sample, and compute a consistent and inductive DFA after each extension. This process continues until enough information has been learned and the computed DFA is a proof.

Our solver-based approach is a novel combination of techniques presented in [13] and [8], which both work in a slightly different setting. For the reader's convenience, however, we recap some principles of these techniques here.

Let us first fix the definitions of samples and consistency, which we already used informally above. A *sample* is a pair $\mathcal{S} = (S_+, S_-)$ consisting of two disjoint and finite sets $S_+, S_- \subseteq \Sigma^*$ over the same alphabet Σ. Intuitively, the set S_+ contains words that have to be accepted by an automaton whereas the set S_- contains such words that have to be rejected. A NFA (or DFA) $\mathcal{A}$ is said to be *consistent* with a sample $\mathcal{S}$ if it accepts all words in S_+ and rejects all words in S_-, i.e., if $S_+ \subseteq L(\mathcal{A})$ and $S_- \cap L(\mathcal{A}) = \emptyset$.

In the following, let a sample $\mathcal{S}$ and a transducer $\mathcal{T}$ over a common alphabet Σ be given. We compute a consistent and inductive DFA by constructing (and solving) logical formulas $\varphi_n^{\mathcal{S},\mathcal{T}}$ that depend on the sample $\mathcal{S}$, the transducer $\mathcal{T}$, and a natural number $n > 0$. A formula $\varphi_n^{\mathcal{S},\mathcal{T}}$ will have the following properties:

- $\varphi_n^{\mathcal{S},\mathcal{T}}$ is satisfiable if and only if there exists a DFA $\mathcal{A}$ with n states such that $\mathcal{A}$ is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$.
- If $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{T}}$, then we can use $\mathfrak{M}$ to derive a DFA $\mathcal{A}_{\mathfrak{M}}$ that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$.

Using these properties, a straightforward algorithm to find a DFA consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ is depicted in Algorithm 1. The idea is to increase the value of n until $\varphi_n^{\mathcal{S},\mathcal{T}}$ becomes satisfiable. If a consistent and inductive DFA exists, the process terminates eventually, and $\mathcal{A}_{\mathfrak{M}}$ is such a DFA. However, such a DFA does not always exist, e.g., in the simple case that a configuration in S_- is reachable via the transitions from configurations in S_+.

Algorithm 1: Computing minimal consistent and inductive DFAs.

```
Input: a sample $\mathcal{S}$ and a transducer $\mathcal{T}$ over a common alphabet $\Sigma$.
$n \coloneqq 0$;
repeat
    $n \coloneqq n + 1$;
    Construct and solve $\varphi_n^{\mathcal{S},\mathcal{T}}$;
until $\varphi_n^{\mathcal{S},\mathcal{T}}$ is satisfiable (with model $\mathfrak{M}$);
Construct and return $\mathcal{A}_{\mathfrak{M}}$;
```

Note that Algorithm 1 does not only compute a DFA that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ but a smallest such DFA in terms of the number of states. Although this fact is not important here, it will be crucial for proving the termination of the algorithms we will present in Section 4. Also note that a binary search is a more efficient way to find the minimal value for n such that $\varphi_n^{\mathcal{S},\mathcal{T}}$ is satisfiable. Let us sum up by stating the main result of this section.

Theorem 1. *Let a sample S and a transducer $\mathcal{T}$ over a common alphabet Σ be given. If a DFA consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ exists, say with k states, then Algorithm 1 terminates after at most k steps. Moreover, $\mathcal{A}_{\mathfrak{M}}$ is a smallest DFA that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$.*

Proof (of Theorem 1). The proof is straightforward and relies on the fact that $\varphi_n^{\mathcal{S},\mathcal{T}}$ has indeed the desired properties (cf. Lemma 3 on page 8). Let a sample $\mathcal{S}$ and a transducer $\mathcal{T}$ be given. Suppose that a DFA consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ exists, say with k states. Then, the formula $\varphi_n^{\mathcal{S},\mathcal{T}}$ is satisfiable for all $n \geq k$. Moreover, if $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{T}}$, then $\mathcal{A}_{\mathfrak{M}}$ is a DFA with n states that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$. Since we increase n by one in every iteration, we eventually find the smallest value for which $\varphi_n^{\mathcal{S},\mathcal{T}}$ is satisfiable (after at most k steps) and, hence, a smallest DFA. □

In the remainder of this section, we will implement the formula $\varphi_n^{\mathcal{S},\mathcal{T}}$ in two different logics: propositional Boolean logic, and the quantifier free fragment of Presburger arithmetic with uninterpreted functions. We will present the implementation in propositional Boolean logic in detail, but only sketch the implementation in Presburger arithmetic as the general idea is similar.

Finally, note that the application of SAT and SMT solvers in this setting is justified as already the special case of finding a minimal DFA that is consistent with a sample is computationally hard—in this case, the transducer defines the identity relation. To be more precise, Gold [14] showed that the corresponding decision problem "Given a sample $\mathcal{S}$ and a natural number k. Does a DFA with k states consistent with $\mathcal{S}$ exist?" is NP-complete. Moreover, there exist highly-optimized logic solvers that can solve even large problems efficiently.

SAT-based Approach. Next, we present a formula in propositional Boolean logic that encodes a DFA with a fixed number $n > 1$ of states that is consistent with a given sample $\mathcal{S} = (S_+, S_-)$ and inductive with respect to a transducer $\mathcal{T} = (Q^{\mathcal{T}}, (\Sigma \cup \{\varepsilon\})^2, q_0^{\mathcal{T}}, \Delta^{\mathcal{T}}, F^{\mathcal{T}})$. The state set of the resulting DFA will be $Q = \{q_0, \ldots, q_{n-1}\}$ with initial state q_0. To encode a DFA, we make a simple observation: if we fix the set of states, the initial state (e.g., as above), and the input alphabet Σ, then every DFA is uniquely determined by its transition function δ and final states F. Our encoding exploits this fact and uses Boolean variables $d_{p,a,q}$ and f_q with $p, q \in Q$ and $a \in \Sigma$. Their meaning is that if $d_{p,a,q}$ is `true`, then $\delta(p,a) = q$. Analogously, if f_q is `true`, then it means that $q \in F$. Note that this idea is used in [13], although not stated in this explicit form.

To make sure that the variables $d_{p,a,q}$ in fact encode a transition function of a DFA, we impose the following constraints.

$$\neg d_{p,a,q} \vee \neg d_{p,a,q'} \qquad p, q, q' \in Q,\ q \neq q',\ a \in \Sigma \tag{1}$$

$$\bigvee_{q \in Q} d_{p,a,q} \qquad p \in Q,\ a \in \Sigma \tag{2}$$

Constraints of type (1) make sure that the variables $d_{p,a,q}$ encode a deterministic function whereas constraints of type (2) enforce the function to be complete.

Now, let $\varphi_n^{\mathrm{DFA}}(\overline{d}, \overline{f})$ be the conjunction of these constraints where $\overline{d}$ is the vector of all variables $d_{p,a,q}$ and $\overline{f}$ is the vector of all variables f_q. From a model $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}}(\overline{d}, \overline{f})$, we can derive a DFA $\mathcal{A}_{\mathfrak{M}} = (\{q_0, \ldots, q_{n-1}\}, \Sigma, q_0, \delta, F)$ in a straightforward manner: we set $\delta(p,a) = q$ for the unique q such that $\mathfrak{M}(d_{p,a,q}) = \texttt{true}$ and $q \in F$ if and only if $\mathfrak{M}(f_q) = \texttt{true}$.

To guarantee that $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$, we impose further constraints on the formula φ_n^{DFA}. We do so by introducing two auxiliary formulas $\varphi_n^{\mathcal{S}}$ as well as $\varphi_n^{\mathcal{T}}$, whose meaning is the following:

– If $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \varphi_n^{\mathcal{S}}$, then $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$ and $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$.
– If $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \varphi_n^{\mathcal{T}}$, then $R(\mathcal{T})(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.

It is not hard to see that if $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \varphi_n^{\mathcal{S}} \wedge \varphi_n^{\mathcal{T}}$, then $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$. When presenting both formulas in the following, we will describe their influence on the resulting DFA $\mathcal{A}_{\mathfrak{M}}$ rather than on the variables $d_{p,a,q}$ and f_q. We thereby implicitly assume that the formulas are satisfiable.

Let us begin by describing the formula $\varphi_n^{\mathcal{S}}$, which originally was proposed by Heule and Verwer [13]. The general idea is to consider runs of the DFA $\mathcal{A}_{\mathfrak{M}}$ on words from $\mathcal{S}$ and their prefixes. To this end, we introduce auxiliary variables $x_{u,q}$ for $u \in \mathit{Pref}(S_+ \cup S_-)$ and $q \in Q$. The meaning of these variables is that if $\mathcal{A}_{\mathfrak{M}}$ reaches state q after reading a word $u \in \mathit{Pref}(S_+ \cup S_-)$, then $x_{u,q}$ is set to `true`. To establish this, we use the following constraints.

$$x_{\varepsilon,q_0} \tag{3}$$

$$(x_{u,p} \wedge d_{p,a,q}) \rightarrow x_{ua,q} \qquad ua \in \mathit{Pref}(S_+ \cup S_-),\ a \in \Sigma,\ p,q \in Q \tag{4}$$

$$x_{u,q} \rightarrow f_q \qquad u \in S_+,\ q \in Q \tag{5}$$

$$x_{u,q} \rightarrow \neg f_q \qquad u \in S_-,\ q \in Q \tag{6}$$

Constraint (3) ensures that the variable x_{ε,q_0} is set to `true` since $\mathcal{A}\colon q_0 \xrightarrow{\varepsilon} q_0$ holds by definition for every DFA $\mathcal{A}$. Constraints of type (4) describe how the run of $\mathcal{A}_{\mathfrak{M}}$ on some input develops: if $\mathcal{A}_{\mathfrak{M}}$ reaches state p after reading u and $\delta(p,a) = q$, then $\mathcal{A}_{\mathfrak{M}}$ will reach state q after reading ua. Constraints of type (5) and (6) assure that words from S_+ and S_- are accepted and rejected, respectively.

Let $\varphi_n^{\mathcal{S}}(\overline{d}, \overline{f}, \overline{x})$ be the conjunction of constraints(3) to (6) where $\overline{d}$ and $\overline{f}$ are as above and $\overline{x}$ is the vector of all variables $x_{u,q}$. Then, we obtain the following.

Lemma 1 (Consistency with $\mathcal{S}$, [13]). *Let $\mathcal{S} = (S_+, S_-)$ be a sample and $\mathfrak{M} \models \varphi_n^{DFA}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{S}}(\overline{d}, \overline{f}, \overline{x})$ for some $n \in \mathbb{N}$. Then, $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$, i.e., $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$ and $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$.*

Lemma 1 can be proved by an induction over the length of the words from the sample. For further details we refer to [13].

The formula $\varphi_n^{\mathcal{T}}$ has recently been introduced also in the context of Regular Model Checking [8]. The basic idea is to keep track of the parallel behavior of the transducer $\mathcal{T}$ and the DFA $\mathcal{A}_{\mathfrak{M}}$. More precisely, we need to establish that if a pair (u,v) of words is accepted by $\mathcal{T}$ and $u \in L(\mathcal{A}_{\mathfrak{M}})$, then $v \in L(\mathcal{A}_{\mathfrak{M}})$ holds, too. To this end, we introduce new auxiliary variables $y_{q,q',q''}$ with $q, q'' \in Q$ and $q' \in Q^{\mathcal{T}}$. Their meaning is that $\mathcal{T}\colon q_0^{\mathcal{T}} \xrightarrow{(u,v)} q'$, $\mathcal{A}_{\mathfrak{M}}\colon q_0 \xrightarrow{u} q$, and $\mathcal{A}_{\mathfrak{M}}\colon q_0 \xrightarrow{v} q''$, then $y_{q,q',q''}$ is set to `true`. The condition stated intuitively above can then be expressed using the following constraints.

$$y_{q_0,q_0^{\mathcal{T}},q_0} \tag{7}$$

$$(y_{p,p',p''} \land d_{p,a,q} \land d_{p'',b,q''}) \rightarrow y_{q,q',q''} \qquad (p',(a,b),q') \in \Delta^{\mathcal{T}},\ a,b \in \Sigma,\ p,p'',q,q'' \in Q,\ p',q' \in Q^{\mathcal{T}} \tag{8}$$

$$(y_{p,p',p''} \land d_{p,a,q}) \rightarrow y_{q,q',p''} \qquad (p',(a,\varepsilon),q') \in \Delta^{\mathcal{T}},\ a \in \Sigma,\ p,p'',q \in Q,\ p',q' \in Q^{\mathcal{T}} \tag{9}$$

$$(y_{p,p',p''} \land d_{p'',b,q''}) \rightarrow y_{p,q',q''} \qquad (p',(\varepsilon,b),q') \in \Delta^{\mathcal{T}},\ b \in \Sigma,\ p,q,q'' \in Q,\ p',q' \in Q^{\mathcal{T}} \tag{10}$$

$$(y_{q,q',q''} \land f_q) \rightarrow f_{q''} \qquad q,q'' \in Q,\ q' \in F^{\mathcal{T}} \tag{11}$$

Constraint (7) makes sure that $y_{q_0,q_0^{\mathcal{T}},q_0}$ is set to `true` since $\mathcal{T}\colon q_0^{\mathcal{T}} \xrightarrow{\varepsilon} q_0^{\mathcal{T}}$ and $\mathcal{A}_{\mathfrak{M}}\colon q_0 \xrightarrow{\varepsilon} q_0$ holds by definition of runs. Moreover, constraints of the types (8) to (10) describe how the parallel behavior of $\mathcal{T}$ and $\mathcal{A}_{\mathfrak{M}}$ develops depending on the type of $\mathcal{T}$'s transitions. This is done in a similar manner as the constraints (4) of the formula $\varphi_n^{\mathcal{S}}$. Finally, constraints of type (11) state that if (u,v) is accepted by $\mathcal{T}$ and u is accepted by $\mathcal{A}_{\mathfrak{M}}$, then v has to be accepted by $\mathcal{A}_{\mathfrak{M}}$, too.

Let $\varphi_n^{\mathcal{T}}(\overline{d},\overline{f},\overline{y})$ be the conjunction of constraints (7) to (11) where $\overline{d}$ and $\overline{f}$ are as above and $\overline{y}$ is the vector of all variables $y_{q,q'q,''}$. Then, we obtain the following lemma.

Lemma 2 (Inductivity with respect to $\mathcal{T}$, [8]). *Let $\mathcal{T}$ be a finite state transducer and $\mathfrak{M} \models \varphi_n^{DFA}(\overline{d},\overline{f}) \land \varphi_n^{\mathcal{T}}(\overline{d},\overline{f},\overline{y})$ for some $n \in \mathbb{N}$. Then, $\mathcal{A}_{\mathfrak{M}}$ is inductive with respect to $\mathcal{T}$, i.e., $R(\mathcal{T})(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.*

The proof of Lemma 2 uses an induction similar to the proof of Lemma 1. This time, however, the induction is over the number of $\mathcal{T}$'s transition used on a run. We refer to [8] further details. Let us now sum up.

Lemma 3. *Let S be a sample and $\mathcal{T}$ a transducer over a common alphabet Σ, $n \in \mathbb{N}$, and*

$$\varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d},\overline{f},\overline{x},\overline{y}) = \varphi_n^{DFA}(\overline{d},\overline{f}) \land \varphi_n^{\mathcal{S}}(\overline{d},\overline{f},\overline{x}) \land \varphi_n^{\mathcal{T}}(\overline{d},\overline{f},\overline{y}).$$

Then, $\varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d},\overline{f},\overline{x},\overline{y})$ is satisfiable if and only if there exists a DFA with n states that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$.

Proof (of Lemma 3). The direction from left to right is a straightforward application of Lemma 1 and Lemma 2. Let $\varphi_n^{\mathcal{S},\mathcal{T}}$ be satisfiable and $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{T}}$. Then, $\mathcal{A}_{\mathfrak{M}}$ is a DFA with n states, consistent with $\mathcal{S}$, and inductive with respect to $\mathcal{T}$.

For the reverse direction, suppose that there exists a DFA $\mathcal{A} = (Q,\Sigma,q_0,\delta,F)$ with n states that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$. Based on $\mathcal{A}$, we can find a model $\mathfrak{M}$ for the formula $\varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d},\overline{f},\overline{x},\overline{y})$: let $\mathfrak{M}(d_{p,a,q}) =$ `true` if and only if $\delta(p,a) = q$ and $\mathfrak{M}(f_q) =$ `true` if and only if $q \in F$. The values of $\mathfrak{M}(x_{u,q})$ and $\mathfrak{M}(y_{q,q',q''})$ can then be derived accordingly. □

Finally, note that $\varphi_n^{\mathcal{S},\mathcal{T}}$ can easily be turned into conjunctive normal form for a SAT solver. In total, the formula comprises $\mathcal{O}(n^2|\Sigma|+n|\mathit{Pref}(S_+ \cup S_-)|+n^2|Q^{\mathcal{T}}|)$ variables and $\mathcal{O}(n^3|\Sigma| + n^2|\mathit{Pref}(S_+ \cup S_-)| + n^4|\Delta^{\mathcal{T}}| + n^2|F^{\mathcal{T}}|)$ clauses.

SMT-based Approach. We now sketch the the implementation of $\varphi_n^{\mathcal{S},\mathcal{T}}$ in SMT logic. To this end, we assume without loss of generality that all automata have a special format: the set of states is $Q = \{0, \dots, n-1\}$, $q_0 = 0$, and the input alphabet is $\Sigma = \{0, \dots, m-1\}$.

Our approach is to encode the automaton directly into the formula utilizing two uninterpreted functions $d \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $f \colon \mathbb{N} \to \{0,1\}$ where d represents the transitions and f the final states. Moreover, we use two additional uninterpreted functions $x \colon \mathbb{N} \to \mathbb{N}$ and $y \colon \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \{0,1\}$, which have the same meaning as the variables $x_{u,q}$ and $y_{q,q'q,''}$ in the SAT-based approach.

Uninterpreted functions allow us to formulate constraints 3 to 11 of the previous section in a convenient manner. Constraints 8, for instance, can be expressed as $y(i,i',i'') \to y(d(i,a),j',d(i'',b))$ where $i,i'' \in Q$, $i',j' \in Q^{\mathcal{T}}$, and $(i',(a,b),j') \in \Delta^{\mathcal{T}}$. As the SMT implementation is analogous to the implementation in Boolean propositional logic, we skip the details here and refer to [8] for a more detailed description. However, let us mention that $\varphi_n^{\mathcal{S},\mathcal{T}}$ comprises $\mathcal{O}(n|\Sigma| + |\mathit{Pref}(S_+ \cup S_-)| + n^2(|\Delta^{\mathcal{T}}| + |F^{\mathcal{T}}|))$ constraints.

4 Learning-based Regular Model Checking

This section presents two algorithms based on algorithmic learning and solver technologies (introduced in the previous section) to compute proofs in Regular Model Checking. In contrast to most existing approaches, our idea is to learn a proof rather than to compute one in a constructive manner.

The learning framework we use was originally introduced by Angluin [11]. In this setting, a *learner* learns a regular *target language* $L \subseteq \Sigma^*$ over an a priori fixed alphabet Σ in interaction with a *teacher*. To do so, the learner can pose two different types of queries: *membership* and *equivalence queries*. On a membership query, the learner queries whether a word $w \in \Sigma^*$ belongs to the target language. The teacher answer either $w \in L$ or $w \notin L$. On an equivalence query, the learner conjectures a regular language, typically given as a DFA $\mathcal{A}$, and the teacher checks if $L(\mathcal{A}) = L$. If this is the case, he returns "yes". Otherwise, he returns a counterexample $w \in L(\mathcal{A}) \Leftrightarrow w \notin L$ as a witness that $L(\mathcal{A})$ and L are different.

Clearly, in our setting we cannot build a teacher that can answer arbitrary membership queries as this would mean to already solve the Regular Model Checking problem. Moreover, answering equivalence queries is possible, but there seems to be no way of finding counterexamples. Thus, we move to a slightly different learning scenario in which answering queries is possible: we allow the teacher to answer "don't know", denoted by ?, to membership queries, and we will only conjecture DFAs on equivalence queries that are inductive with respect to the transducer. This way, the teacher only needs to check whether the proposed conjecture classifies the initial and bad configurations correctly.

Employing learning techniques in an Angluin-like learning scenario is in general a two-step process. First, we need to construct a teacher that is able to answer membership and equivalence queries (cf. Section 4.1). Second, we have to develop a learning algorithm that learns from this teacher. For the latter task, we will present two algorithms. The first (cf. Section 4.2) follows the principles of the successful CEGAR approach [12]. The second (cf. Section 4.3) is based on Angluin's prominent algorithm for learning regular languages [11].

Both algorithms share the same fundamental idea. The learner supposes that the teacher knows a proof and asks the teacher if configurations belong to the proof or not. The problem is, that the teacher does not know a proof. However, he can clearly answer queries if the configuration in question belongs to the set of initial or bad configurations. If this is not the case, he simply returns "don't know". Once a learner has gathered enough information, he conjectures an inductive DFA consistent with the information obtained so far. To answer this equivalence query, the teacher only needs to check whether the conjecture classifies the initial and bad configurations correctly since we assume any conjecture to be inductive. If the check fails, the teacher can easily find a counterexample.

A similar scenario called *learning from inexperienced teachers* was investigated in [9] and subsequently in [10]. In [10], also the general idea of a CEGAR-style and Angluin-style learner have been discussed. Note, however, that the inexperienced teacher setting is simpler and does not involve computing inductive automata.

4.1 An Inexperienced Teacher for Regular Model Checking

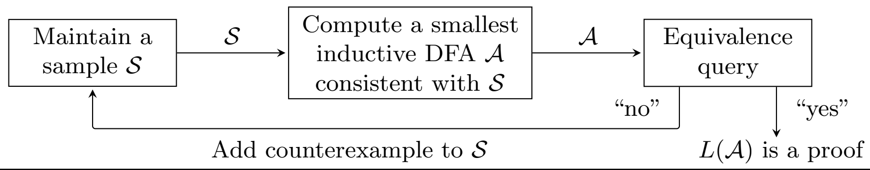
Implementing a teacher for our setting is simple. On a membership query $w \in \Sigma^*$, the teacher returns "yes" if $w \in I$, "no" if $w \in B$, and "?" in any other case.

On an equivalence query with an DFA $\mathcal{A}$, the teacher checks whether $I \subseteq L(\mathcal{A})$ and $B \cap L(\mathcal{A}) = \emptyset$ holds and returns "yes" if so. If this is not the case, he returns a counterexample $w \in I$ and $w \notin L(\mathcal{A})$, or $w \in B \cap L(\mathcal{A})$. Note that this check is in fact enough to ensure that $\mathcal{A}$ is a proof since we assume that every conjecture provided on an equivalence query is inductive with respect to the transducer $\mathcal{T}$. Furthermore, note that all these checks are decidable for regular languages.

4.2 The CEGAR-style Learner

The CEGAR-style learner (sketched as Algorithm 2) maintains a sample $\mathcal{S} = (S_+, S_-)$ as finite abstraction of the (potentially infinite) sets of initial and bad configurations. In every iteration, the learner computes a minimal DFA $\mathcal{A}$ consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ using one of the techniques introduced in Section 3 in a black-box fashion. The DFA $\mathcal{A}$ is then conjectured on an equivalence query. If the teacher replies "yes", the process terminates. If the teacher returns a counterexample w, we refine our abstraction. As a counterexample either satisfies $w \in I$ and $w \notin L(\mathcal{A})$, or $w \in B \cap L(\mathcal{A})$, we add w to S_+ in the first case and w to S_- in the latter case. This excludes a spurious behavior of further conjectures on w. Then, we continue with the next iteration.

Algorithm 2: The CEGAR-style learner


Maintain a sample $\mathcal{S}$
$\mathcal{S}$
Compute a smallest inductive DFA $\mathcal{A}$ consistent with $\mathcal{S}$
$\mathcal{A}$
Equivalence query
"no"
"yes"
Add counterexample to $\mathcal{S}$
$L(\mathcal{A})$ is a proof

Algorithm 2 follows the CEGAR approach in the following sense. The DFA $\mathcal{A}$ produced from the sample in every iteration is an abstraction of the reachable part of the program. In the beginning, the sample contains only a few words and our algorithm will produce very coarse abstractions. An equivalence check with the abstraction reveals if a proof has been found. If this is not the case, counterexamples are used to refine the abstraction until a proof can be identified.

We can now state the main result of this section.

Theorem 2. *Let $P = (I, T)$ be a program and B a regular set of bad configurations with $B \cap I = \emptyset$. If a proof that $\mathcal{P}$ is correct with respect to B exists, Algorithm 2 terminates and returns a (smallest) proof.*

Proof (of Theorem 2). Due to the nature of equivalence queries, we know that the result of Algorithm 2 is in fact a proof once the algorithm terminates. Thus, it is enough to prove the termination of Algorithm 2 if a proof exists.

To this end, suppose that a proof exists, say with k states. We observe that Algorithm 2 never conjectures the same DFA twice and that the size of the conjectures increases monotonically. This can be seen as follows. Assume that the conjecture $\mathcal{A}_i$ of iteration i has n_i states and the conjecture $\mathcal{A}_{i+1}$ of iteration $i+1$ has $n_{i+1} < n_i$ states. Since the sample of iteration $i+1$ results from the one of iteration i by adding one word to S_+ or S_-, $\mathcal{A}_{i+1}$ is necessarily consistent with the sample of iteration i, but has fewer states than $\mathcal{A}_i$. This is a contradiction to the fact that we only produce minimal DFAs.

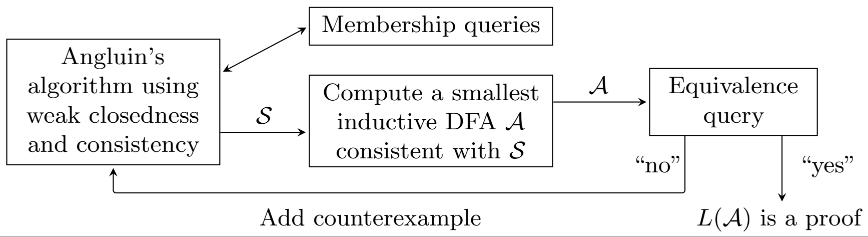
A second observation is that every proof is consistent with the samples produced by Algorithm 2 as the sample contains only counterexamples for which the teacher told us their classification; in particular, this holds for any smallest proof. Moreover, since Algorithm 1 always returns a smallest consistent and inductive DFA, it will eventually find a smallest proof as a solution once S_+ and S_- are large enough to rule out any smaller consistent and inductive DFA—regardless of the concrete choice of elements in S_+ and S_-. $\square$

4.3 The Angluin-style Learner

Our Angluin-style learning algorithm is an extension of the CEGAR-style learner. Like Angluin's algorithm, it accelerates the learning process by additionally posing membership queries to gather further information before constructing a conjecture. Our general idea is to lift well-established querying techniques provided by Angluin's algorithm to our setting. Hence, when presenting the Angluin-style learner below, we assume a basic understanding of Angluin's algorithm [11].

Our Angluin-style learner, sketched as Algorithm 3, is an adaptation of a learning algorithm proposed by Grinchtein, Leucker, and Piterman [9]. The learner maintains a prefix-closed set $R \subseteq \Sigma^*$ and a set $S \subseteq \Sigma^*$ of words. Moreover, the learner organizes the learned data in a so-called *table* $T\colon (R \cup R \cdot \Sigma) \cdot S \to \{0, 1, ?\}$, which it fills by posing membership queries; the value of $T(u)$ is the answer to a membership query on u. The words from R are candidates for identifying states of a conjecture, and the words from S are used to distinguish such states. Using this intuition, we define two words $r, r' \in R \cup R \cdot \Sigma$ to be *equivalent* (i.e., to potentially represent the same state), denoted by $r \approx r'$, if they cannot be distinguished by words from S, i.e., $T(rs) \neq ?$ and $T(r's) \neq ?$ implies $T(rs) = T(r's)$ for all $s \in S$. In other words, r and r' are equivalent, if the ?-entries in the table can be resolved in such a way that $T(rs) = T(r's)$ holds for all $s \in S$.

Algorithm 3: The Angluin-style learner


Membership queries
Angluin's algorithm using weak closedness and consistency
$\mathcal{S}$
Compute a smallest inductive DFA $\mathcal{A}$ consistent with $\mathcal{S}$
$\mathcal{A}$
Equivalence query
"no"
"yes"
Add counterexample
$L(\mathcal{A})$ is a proof

Our Angluin-style learner works like Angluin's algorithm, which makes the table closed and consistent in every iteration. Since we need to handle ?-entries, we switch to a weak notion of closedness and consistency as introduced in [9]:

- A table T is *weakly closed* if for $r \in R$ and $a \in \Sigma$ there exists an $r' \in R$ such that $ra \approx r'$. If this is not satisfied, the algorithm adds ra to R.
- A table T is *weakly consistent* if $r \approx r'$ implies $ra \approx r'a$ for $r, r' \in R$, $a \in \Sigma$. If T is not weakly consistent, then there exists an $s \in S$ such that $T(ras) \neq ?$, $T(r'as) \neq ?$, and $T(ras) \neq T(r'as)$, and the algorithm adds as to S.

Once T is weakly closed and weakly consistent, the Angluin-style learner turns the table into a sample $\mathcal{S} = (S_+, S_-)$ where $S_+ = \{rs \mid r \in R, s \in S, T(rs) = 1\}$ and $S_- = \{rs \mid r \in R, s \in S, T(rs) = 0\}$. Then, it applies one of the approaches of Section 3 as a black-box to derive a smallest DFA $\mathcal{A}$ consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ and submits $\mathcal{A}$ to an equivalence query. If the teacher replies "yes", the learning terminates. Otherwise, the algorithm adds the returned counterexample and all of its prefixes to R and continues with the next iteration.

The following theorem states the correctness of Algorithm 3. The proof is similar to the proof of Theorem 2 and, hence, skipped here.

Theorem 3. *Let $P = (I, T)$ be a program and B a regular set of bad configurations with $B \cap I = \emptyset$. If a proof that $\mathcal{P}$ is correct with respect to B exists, Algorithm 3 terminates and returns a (smallest) proof.*

5 Related Work and Experiments

Related Work. We are aware of three established tools for Regular Model Checking: T(O)RMC, FASTER, and LEVER. T(O)RMC iterates the given transducer and tries to identify differences between the iterations. These differences are extrapolated using *widening*, which approximates the limit of an infinite iteration of the transducer. The drawback of this method is that the bad configurations are not taken into account during the computation. If the the result is not disjoint to the bad configurations, the process has to be restarted with additional user input, which requires expert knowledge about the problem at hand. Moreover, T(O)RMC requires DFAs as input whereas our approach also works with NFAs, which can be exponentially smaller than equivalent DFAs.

FASTER computes the *exact* set of reachable configurations using *acceleration*, i.e., by computating in-the-limit effects of iterating cycles. This might lead to non-termination if the set of reachable configurations is not regular. In contrast, our learners always find a proof if one exists. Moreover, FASTER is originally designed for *integer linear systems* over Presburger formulas, which are internally translated into a Regular Model Checking problem. Thus, we can compare our approaches only on examples that are expressible as integer linear systems.

The LEVER tool uses Angluin's learning algorithm to learn proofs. The main difference from our approach lies in the fact that LEVER does not learn a proof directly, but a set of configurations that is augmented with distance information. These distance information encode how often the transducer has to be applied to reach a given configuration. The problem here is that these augmented sets are often not regular although a proof exists. Then, in contrast to our approaches, LEVER is not guaranteed to terminate. Unfortunately, LEVER is no longer publicly available, and, hence, were not able to compare it to our approaches.

Experiments. To assess the performance of our learning algorithms, we implemented a C++ prototype of both the CEGAR-style and the Angluin-style learner. The implementation uses the `libalf` automaton learning library as well as the GLUCOSER SAT solver and the Z3 SMT solver. We used two different benchmarks suits: integer linear systems and examples in which the size of the DFAs specifying initial and bad configurations were successively enlarged. For all experiments, we used a PC with an Intel Q9550 CPU and 4GB of RAM (at most 500MB were ever used) running Linux.

The first benchmark suit contains integer linear systems available at the FASTER and T(O)RMC websites. Table 1 shows results for FASTER, T(O)RMC[1], our CEGAR-style approach (CEGAR), and our Angluin-style approach (Angluin) on a simple petri net, the *Berkeley cache coherence protocol*, the *Synapse cache coherence protocol*, a lift protocol, the M.E.S.I. Cache Coherence Protocol

[1] Please note that the 64-bit version of T(O)RMC did not work properly. The 32-bit version partly worked but suffered from severe memory access violation and leaks. For some benchmarks, T(O)RMC crashed and we were not able to obtain a result. Furthermore, it was not possible to generate new benchmarks for T(O)RMC.

and several more. Due to space constraints, we can provide only a selection of our results. However, all experiments showed the same qualitative results. Table 1 shows the running times in seconds. FASTER performs best on these examples, with the exception of petri net, where we had to pick FASTER-specific parameters by hand. On most examples, T(O)RMC is outperformed by the other tools.

Experiment	Angluin		CEGAR		T(O)RMC	FASTER
	Glucoser	Z3	Glucoser	Z3		
petri net	0.12	0.15	0.11	0.11	0.02	1.13
berkeley	0.62	0.92	1.29	1.45	4.23	0.03
synapse	0.04	0.07	0.06	0.16	0.19	0.03
lift	0.01	0.01	0.01	0.02	5.54	0.15
mesi	0.58	2.64	1.55	6.24	5.52	0.04

Table 1. Results for integer linear systems

The second benchmark suite demonstrates the advantages of our tool when confronted with large automata for initial and bad configurations. The suite contains examples of a modulo-counter and the well-known token ring protocol where we successively increased the size of the input automata. Table 2 shows the results together with the sizes of both initial and bad automata. In these experiments, we also compared ourselves to the mere SAT-based approach of [8]. "TO" indicates a timeout after 300 seconds and "–" indicates that the experiment could not be performed as we were not able to generate new benchmarks. We observe that T(O)RMC is clearly outperfomed by the CEGAR-style learner on the token-ring benchmark, but the solver-only method is the fastest. The best algorithm for the modulo-counter is the Angluin-style learner using GLUCOSER.

Table 2. Results for Modulo counter and Token-ring

Experiment	Size	Angluin		CEGAR		Solver only	T(O)RMC
	init / bad	Glucoser	Z3	Glucoser	Z3	Glucoser	
Token-ring	50 / 3	1.23	1.52	0.07	0.14	0.02	0.31
	150 / 3	132.70	137.74	0.95	1.11	0.04	6.78
	250 / 3	TO	TO	4.13	3.43	0.04	31.57
	350 / 3	TO	TO	11.02	13.53	0.04	89.66
	450 / 3	TO	TO	24.65	24.90	0.04	203.36
Modulo counter	14 / 125	0.29	0.41	0.75	1.03	0.24	–
	14 / 156	0.58	0.99	1.75	2.09	0.29	–
	34 / 187	1.13	3.52	4.04	6.48	1.29	–
	34 / 218	2.49	20.42	6.45	47.84	27.49	–
	82 / 249	21.27	100.48	45.23	178.59	TO	–

In total, we observe two things. First, our experiments show that we can handle problem instances specified for tools such as T(O)RMC and FASTER with competitive running times. Second, we observe that there is no superior algorithm. In particular, there are examples where the CEGAR-style outperforms the Angluin-style learner and vice versa, but both clearly beat T(O)RMC and the solver-only approach on the second benchmark suite. For the benchmarks at hand the implementations using the GLUCOSER SAT solver were always slightly faster than the one using the Z3 SMT solver. Note, however, that this might be different for larger instances as the size of the generated formulas grows faster for SAT than for SMT formulas. Moreover, note that our implementation is only an early prototype whereas T(O)RMC and FASTER are highly optimized tools.

6 Conclusion and Future Work

We presented two new algorithms for Regular Model Checking that combine off-the-shelf SAT and SMT solver technologies with automata learning. Our prototype implementation turned out to be competitive to FASTER and T(O)RMC, especially for large input automata. Moreover, our approaches work out-of-the-box, do not require expert knowledge, and always find a proof if one exist.

As future work we would like to investigate the applicability of nonregular sets of initial and bad configurations that still allow answering membership and equivalence queries, e.g., visibly or deterministic context-free languages. Another interesting field of further research is to use nondeterministic rather than deterministic automata as representation of proofs. This will increase the size of the formula $\varphi_n^{\mathcal{S},\mathcal{T}}$ on the one hand, but might yield an exponentially smaller result on the other hand. Furthermore, we will consider an incremental SAT approach, where clauses learnt during the solving process are reused in order to avoid complete restarts of the solver.

References

1. Jr., E.M.C., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
3. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Proc. of CAV. Volume 1855 of LNCS, Springer (2000) 403–418
4. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Proc. of CAV. Volume 1427 of LNCS, Springer (1998) 88–97
5. Legay, A.: T(o)rmc: A tool for (omega)-regular model checking. In: Proc. of CAV. Volume 5123 of LNCS, Springer (2008) 548–551
6. Bardin, S., Finkel, A., Leroux, J.: Faster acceleration of counter automata in practice. In: Proc. of TACAS. Volume 2988 of LNCS (2004) 576–590
7. Vardhan, A., Viswanathan, M.: Lever: A tool for learning based verification. In: Proc. of CAV. Volume 4144 of LNCS, Springer (2006) 471–474
8. Neider, D.: Computing minimal separating dfas and regular invariants using sat and smt solvers. In: Proc. of ATVA. Volume 7561 of LNCS, Springer (2012) 354–369
9. Grinchtein, O., Leucker, M., Piterman, N.: Inferring network invariants automatically. In: Proc. of IJCAR. Volume 4130 of LNCS, Springer (2006) 483–497
10. Leucker, M., Neider, D.: Learning minimal deterministic automata from inexperienced teachers. In: Proc. of Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). Volume 7609 of LNCS, Springer (2012) 524–538
11. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2) (1987) 87–106
12. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. of CAV. Volume 1855 of LNCS, Springer (2000) 154–169
13. Heule, M., Verwer, S.: Exact dfa identification using sat solvers. In: Proc. of Grammatical Inference (ICGI). Volume 6339 of LNCS, Springer (2010) 66–79
14. Gold, E.M.: Complexity of automaton identification from given data. Information and Control **37**(3) (1978) 302–320