Computing Minimal Separating DFAs and Regular Invariants Using SAT and SMT Solvers

Daniel Neider

Lehrstuhl für Informatik 7, RWTH Aachen University, Germany

Abstract. We develop a generic technique to compute minimal separating DFAs (deterministic finite automata) and regular invariants. Our technique works by expressing the desired properties of a solution in terms of logical formulae and using SAT or SMT solvers to find solutions. We apply our technique to three concrete problems: computing minimal separating DFAs (e.g., used in compositional verification), regular model checking, and synthesizing loop invariants of integer programs that are expressible in Presburger arithmetic.

1 Introduction

In this paper, we present a generic technique based on SAT or SMT solvers to compute minimal separating DFAs (deterministic finite automata) and regular invariants.

A separating DFA is a DFA that separates two disjoint regular languages, i.e., whose accepted language contains the first language and is disjoint to the second. A minimal separating DFA is a separating DFA that has the least number of states among all separating DFAs. Many well known problems can be reduced to the problem of finding a minimal separating DFA. For instance, in [5] minimal separating DFAs are used in the context of compositional verification. Two other examples are the minimization of incomplete specified DFAs [13] and the computation of minimal DFAs that are consistent with a set of positively and negatively classified samples [3]. Note that finding a minimal separating DFA is in fact a non-trivial task since the corresponding decision problem "Given two disjoint regular languages. Does a separating DFA with $k \in \mathbb{N}$ states exist?" is NP-hard (cf. [14] where this is shown for two disjoint finite languages).

Regular invariants are defined in terms of binary relations T specified by finite-state transducers. Intuitively, a regular invariant is a regular language L (alternatively a DFA) that is invariant (or closed) under T, i.e., for all $(u, v) \in T$ with $u \in L$ also $v \in L$ is satisfied. Regular invariants occur, e.g., in regular model checking [4]. There, one way to prove a program correct is to find a regular invariant with respect to the transitions of the program that overapproximates the set of initial configurations and has an empty intersection with the set of bad configurations. In this sense, regular invariants extend the concept of separating DFAs.

The main contribution of this work is a technique to compute minimal separating DFAs and regular invariants. Our technique works as follows. It takes a problem instance, translates it into a logical formula of size polynomial in the input, and uses a SAT or SMT solver to find a solution. More precisely, our technique creates a logical formula φ_n that depends on the regular languages given as input as well as a natural number $n \geq 1$. Moreover, φ_n has the following two properties. First, φ_n is satisfiable if and only if there exists a DFA with n states that possesses the desired properties. Second, a model of φ_n allows to derive a DFA with these properties. Starting with $n = 1$ we increase n until φ_n becomes satisfiable. This guarantees to find a suitable DFA (if one exists).

We implement the formula φ_n in two logics: propositional logic over Boolean variables and the quantifier-free logic over the structure $(\mathbb{N}, <, =)$ where we allow constants $c \in \mathbb{N}$ and uninterpreted functions. We choose these logics because highly-optimized off-the-shelf solvers are available in both cases, e.g., the MiniSat SAT solver in the first case and Microsoft's Z3 SMT solver in the latter. Since the decision problem variant of computing minimal separating DFAs and regular invariants is NP-hard, using SAT and SMT solvers is a natural approach.

We apply our technique to three concrete problems: computing minimal separating DFAs (Section 3), regular model checking (Section 4.1), and synthesizing loop invariants of integer programs that are expressible in Presburger arithmetic (Section 4.2). The latter two can be subsumed under the topic regular invariants. Each mentioned section begins with an introduction to the setting, then describes the application of our technique, and finally concludes with related work and experiments.

2 Preliminaries

Words, Languages and Finite automata. An *alphabet* Σ is a finite set of *symbols*. A *word* $w = a_1 \dots a_n$ is a finite, possibly empty, sequence of symbols $a_i \in \Sigma$. If a sequence is empty, we call it the *empty word*, denoted by ε. The *concatenation* of two words $u = a_1 \dots a_n$ and $v = b_1 \dots b_m$ is the word $uv = a_1 \dots a_n b_1 \dots b_m$. Σ^* is the set of all finite words over Σ. A subset $L \subseteq \Sigma^*$ is called a *language*.

A *(nondeterministic) finite automaton (NFA)* is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ where Q is a finite, non-empty set of states, Σ is the input alphabet, $q_0 \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states. The size of an NFA is the number $|Q|$ of its states. For $w = a_1 \dots a_n \in \Sigma^*$, a *run* of $\mathcal{A}$ on w from a state q is a sequence $q_1, \dots, q_{n+1}$ such that $q_1 = q$ and $(q_i, a_i, q_{i+1}) \in \Delta$ for $1 \leq i \leq n$; we also write $\mathcal{A}\colon q \xrightarrow{w} q_{n+1}$ for short. A word w is *accepted* by $\mathcal{A}$ if $\mathcal{A}\colon q_0 \xrightarrow{w} q$ with $q \in F$. The *language accepted by* $\mathcal{A}$ is the set $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A}\colon q_0 \xrightarrow{w} q, q \in F\}$. A language $L \subseteq \Sigma^*$ is called *regular* if there exists an NFA $\mathcal{A}$ such that $L = L(\mathcal{A})$.

A *deterministic finite automaton (DFA)* is an NFA where for every $p \in Q$ and $a \in \Sigma$ a unique $q \in Q$ exists such that $(p, a, q) \in \Delta$. In this case, Δ defines a function $\delta\colon Q \times \Sigma \to Q$, and we replace Δ by δ in the definition of a DFA.

Logics, Formulae, and Satisfiability. In this paper, we consider three logics: propositional logic over Boolean variables, quantifier-free logic over the structure $(\mathbb{N}, <, =)$ with uninterpreted functions, and Presburger arithmetic. We use the first two logics to implement our technique in Section 3.1 and Section 3.2, respectively. The third logic is considered in Section 4.2 in the context of synthesizing loop invariants of integer programs.

In propositional logic, a *formula* φ—often called SAT formula—is built from Boolean variables $x_1, \ldots, x_n$ and the logical connectives $\wedge$, $\vee$, $\rightarrow$ and $\neg$ with their usual meaning. The set $\mathrm{Var}(\varphi)$ is the set of variables occurring in φ; if the variables occurring in φ are of special interest, we also write $\varphi(x_1, \ldots, x_n)$. A *model of* φ is a mapping $\mathfrak{M}\colon \mathrm{Var}(\varphi) \rightarrow \{0,1\}$ (0 representing `false`, 1 representing `true`) such that φ evaluates to `true` if all variables x_i in φ are substituted by $\mathfrak{M}(x_i)$; we then write $\mathfrak{M} \models \varphi$. A formula φ is said to be in *conjunctive normal form* if $\varphi = \bigwedge_{i=1}^{r} c_i$ where $c_i = \bigvee_{j=1}^{s_i} l_{ij}$ is a *clause* and l_{ij} is either a Boolean variable or its negation. Satisfiability of a formula can be decided by a SAT solver, and if the formula is satisfiable, then the solver can provide a model. Note that most SAT solver require the input to be in conjunctive normal form.

The second logic we consider is the usual quantifier-free logic over the structure $(\mathbb{N}, <, =)$. We additionally allow constants $c \in \mathbb{N}$ and enrich this logic with uninterpreted functions. Each such function is of the form $f\colon \mathbb{N} \times \ldots \times \mathbb{N} \rightarrow \mathbb{N}$ and represents an unknown function of which only the name f and its signature is known. In our case, it turns out that functions of the form $\mathbb{N} \times \ldots \times \mathbb{N} \rightarrow \{\mathtt{true}, \mathtt{false}\}$, which can easily be simulated, are also helpful and used later on. The concept of a model $\mathfrak{M}$ is lifted to this kind of formulae in the natural way. If f is an uninterpreted function occurring in a formula φ and $\mathfrak{M} \models \varphi$, then we write $f_{\mathfrak{M}}$ to denote the function defined by the model $\mathfrak{M}$. Satisfiability of this kind of formulae can be decided with today's *satisfiability modulo theory (SMT)* solvers, and, hence, we bravely refer to them as SMT formulae.

Presburger arithmetic is the first order logic over the structure $(\mathbb{Z}, +, \leq, 0)$. For convenience, we add syntactic sugar to this logic by also allowing to use the relations $<$, $>$ and arbitrary constants $c \in \mathbb{Z}$. Presburger formulae have the nice property that they define regular languages. More formally, each Presburger formula $\varphi(x_1, \ldots, x_n)$ can be translated into a DFA $\mathcal{A}^{\varphi}$ working over the alphabet $\{0,1\}^n$ (see, e.g., [11]). Intuitively, the automaton $\mathcal{A}^{\varphi}$ accepts exactly the set of binary strings encoding integer values that satisfy φ. Note, however, that not every regular language represents a Presburger formula. Nonetheless, in [11] the problem to decide whether a regular language represents a Presburger formula is shown to be decidable in polynomial time. Moreover, in this case, a formula defining the language can be computed in polynomial time.

3 Minimal Separating DFAs

The first application of our technique is to compute minimal separating DFAs. To this end, let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages over a fixed alphabet Σ. A DFA $\mathcal{A}$ with $L_1 \subseteq L(\mathcal{A})$ and $L_2 \cap L(\mathcal{A}) = \emptyset$ is said to be a *separating DFA* since it separates both languages. A minimal separating DFA is a separating DFA of minimal size. Note that minimal separating DFAs are not unique for fixed L_1, L_2 since their behavior on words not belonging to L_1 or

L_2 is unspecified. In fact, computing a minimal separating DFA for two disjoint regular languages is computationally hard (cf. [14]).

Minimal separating DFAs are helpful in various contexts. For instance, a direct application to compositional verification is described in [5]. Moreover, the well-known task of minimizing incompletely specified DFAs [13] can also be phrased in terms of minimal separating DFAs. To this end, an incompletely specified DFA is defined as a DFA $\mathcal{B} = (Q, \Sigma, q_0, \delta, \mathrm{Acc}, \mathrm{Rej}, \mathrm{Unspec})$ whose states are partitioned into accepting, rejecting, and unspecified states. The task of minimizing an incompletely specified DFA now is to compute a DFA of minimal size that accepts all words that lead to an accepting state in $\mathcal{B}$ and rejects all words that lead to a rejecting state in $\mathcal{B}$. This, however, is the same as computing a minimal separating DFA for the languages $L_1 = \{u \in \Sigma^* \mid \mathcal{B}\colon q_0 \xrightarrow{u} q, q \in \mathrm{Acc}\}$ and $L_2 = \{u \in \Sigma^* \mid \mathcal{B}\colon q_0 \xrightarrow{u} q, q \in \mathrm{Rej}\}$. Finally, let us mention Biermann's task of computing a minimal DFA that agrees with a finite set of positively and negative classified samples [3]. Here, L_1 and L_2 are both finite languages.

Let us now describe our technique to compute minimal separating DFAs. For the rest of this section, fix an alphabet Σ, and let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages. Let us assume that L_1 and L_2 are given by two NFAs $\mathcal{A}_1 = (Q_1, \Sigma, q_0^1, \Delta_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, q_0^2, \Delta_2, F_2)$, respectively.

We search for a separating DFA by creating a formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ that depends on the NFAs $\mathcal{A}_1$, $\mathcal{A}_2$, a natural number $n \geq 1$, and has the following properties:

- $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ is satisfiable if and only if there exists a separating DFA $\mathcal{A}$ with n states, i.e., $L_1 \subseteq L(\mathcal{A})$ and $L_2 \cap L(\mathcal{A}) = \emptyset$.
- If $\mathfrak{M} \models \varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$, then $\mathfrak{M}$ can be used to derive a DFA separating L_1 and L_2.

Clearly, if $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ is satisfiable for a given value of n, then we have found a separating DFA. However, if the formula is unsatisfiable, then the parameter n has been chosen too small and we need to increment it. Since a separating DFA always exist, e.g., the DFA accepting exactly the language L_1, this process terminates eventually. The algorithm in pseudo code is shown in Algorithm 1. Note that an even faster approach is to use a binary search to find the minimal value of n rather than incrementing n by one in each iteration. Providing an intelligent starting value of n is difficult, but the size of $\mathcal{A}_1$ is a natural choice.

We can now state the main result of this section.

Theorem 1. *Let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages. If a minimal separating DFA has k states, then Algorithm 1 terminates after k iterations and returns a minimal separating DFA.*

Proof. Let us first state that a DFA separating L_1 and L_2 always exist, e.g., the DFA accepting exactly the language L_1. Then, the proof of Theorem 1 is a straight-forward application of the fact that $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ has indeed the desired properties (cf. Lemma 2 on page 360): if a minimal separating DFA has k states, then $\varphi_l^{\mathcal{A}_1,\mathcal{A}_2}$ is satisfiable for all $l \geq k$, and we find the minimal value k since we increase n by one in every iteration. $\square$

Algorithm 1: Computing minimal separating DFAs.

```
Input: two disjoint regular languages $L_1, L_2 \subseteq \Sigma^*$ given as NFAs $\mathcal{A}_1, \mathcal{A}_2$.
$n := 0$;
repeat
    $n := n + 1$;
    Construct and solve $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$;
until $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ is satisfiable;
Construct and return $\mathcal{A}_{\mathfrak{M}}$;
```

In the following two subsections, we utilize two different logics to implement the formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$. The first is the propositional logic over Boolean variables, which we consider in Section 3.1. In Section 3.2, we consider the quantifier-free logic over the structure $(\mathbb{N}, <, =)$ with uninterpreted functions. However, before we continue to implement $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$, let us briefly discuss related work.

Related work. Chen et al. [5] propose an algorithm to compute minimal separating DFAs, which is based on algorithmic learning. Internally, they use an algorithm to minimize incompletely specified DFAs as a subroutine. In fact, every algorithm that minimizes incompletely specified DFAs (e.g., [13]) can be used to compute minimal separating DFAs. Note, however, that these algorithms typically require deterministic automata as input whereas we allow NFAs.

With the emerge of fast and efficient SAT and SMT solvers and since the problem itself is computationally hard (cf. [14]), it seems justified to promote a SAT and SMT based approach in this context. Due to the lack of publicly available tools we did not yet compare our technique to other approaches.

3.1 Finding Separating DFAs Using SAT Formulae

In the following, we describe a formula in the propositional logic over Boolean variables that, if satisfiable, encodes a DFA with a fixed number $n \geq 1$ of states that separates L_1 and L_2. The automaton will have the state set $Q = \{q_0, \ldots, q_{n-1}\}$, and the initial state $q_0 \in Q$. To encode a DFA, we make the following simple observation: if the set of states and the initial state are fixed (e.g., as above), then each DFA is completely defined by its transition function and final states. Our encoding exploits this fact and uses Boolean variables $d_{p,a,q}$ and f_q with $p, q \in Q$ and $a \in \Sigma$. If $d_{p,a,q}$ is assigned to `true`, it means that $\delta(p, a) = q$. Similarly, if f_q is assigned to `true`, then state q is a final state.

To make sure that the variables $d_{p,a,q}$ indeed encode a deterministic transition function, 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 $d_{p,a,q}$ in fact encode a well-defined function, i.e., that for every state $p \in Q$ and input $a \in \Sigma$ there is at most one $q \in Q$ such that $d_{p,a,q}$ is set to `true`. Constraints of type (2), on the other hand, ascertain that the transition function is total. The latter constraints are not needed in general and might be skipped.

Let $\varphi_n^{\text{DEA}}(\overline{d}, \overline{f})$ be the conjunction of the constraints of type (1) and (2) where $\overline{d}$ is the vector of all variables $d_{p,a,q}$ and $\overline{f}$ the vector of all variables f_q. From a model $\mathfrak{M}$ of the formula $\varphi_n^{\text{DEA}}(\overline{d}, \overline{f})$ it is straight-forward to derive a DFA $\mathcal{A}_{\mathfrak{M}} = (\{q_0, \ldots, q_{n-1}\}, \Sigma, q_0, \delta, F)$: $\delta(p,a) = q$ for the unique q such that $\mathfrak{M}(d_{p,a,q}) = 1$, and $F = \{q \mid \mathfrak{M}(f_q) = 1\}$.

Until now, $L(\mathcal{A}_{\mathfrak{M}})$ is unspecified. Thus, to guarantee that $\mathcal{A}_{\mathfrak{M}}$ is a separating DFA, we need to impose further constraints on the formula φ_n^{DEA}. We do so by introducing two auxiliary formulae $\varphi_n^{\mathcal{A}_1 \subseteq}$ and $\varphi_n^{\neg \mathcal{A}_2}$ that express the following:

- If $\mathfrak{M} \models \varphi_n^{\text{DEA}} \wedge \varphi_n^{\mathcal{A}_1 \subseteq}$, then $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.
- If $\mathfrak{M} \models \varphi_n^{\text{DEA}} \wedge \varphi_n^{\neg \mathcal{A}_2}$, then $L(\mathcal{A}_2) \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$.

Clearly, if $\mathfrak{M} \models \varphi_n^{\text{DEA}} \wedge \varphi_n^{\mathcal{A}_1 \subseteq} \wedge \varphi_n^{\neg \mathcal{A}_2}$, then $\mathcal{A}_{\mathfrak{M}}$ is DFA separating L_1 and L_2.

The idea behind the formulae $\varphi_n^{\mathcal{A}_1 \subseteq}$ and $\varphi_n^{\neg \mathcal{A}_2}$ is to impose restrictions on the variables $d_{p,a,q}$ and f_q, which determine the automaton $\mathcal{A}_{\mathfrak{M}}$. Keeping this in mind, it is easier to explain the formulae by directly referring to the automaton $\mathcal{A}_{\mathfrak{M}}$ rather than to describe their influence on the variables $d_{p,a,q}$ and f_q. Note, however, that we thereby implicitly assume that the corresponding formula is satisfiable and that $\mathfrak{M}$ is a model.

The idea of the formula $\varphi_n^{\mathcal{A}_1 \subseteq}$ is to keep track of the parallel behavior of the automaton $\mathcal{A}_1$ and $\mathcal{A}_{\mathfrak{M}}$. For that, we use new auxiliary variables $x_{q,q'}$ where $q \in Q$ and $q' \in Q_1$. Intuitively, we want to establish for all models $\mathfrak{M} \models \varphi_n^{\text{DEA}} \wedge \varphi_n^{\mathcal{A}_1 \subseteq}$ that if some input $w \in \Sigma^*$ induces the runs $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{w} q$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{w} q'$, then $x_{q,q'}$ is assigned to `true`. This is done by the following constraints.

$$x_{q_0,q_0^1} \tag{3}$$

$$(x_{p,p'} \wedge d_{p,a,q}) \rightarrow x_{q,q'} \qquad p,q \in Q,\ p',q' \in Q_1,\ a \in \Sigma,\ (p',a,q') \in \Delta_1 \tag{4}$$

The constraint (3) requires the variable x_{q_0,q_0^1} to be assigned to `true` because ε induces the runs $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{\varepsilon} q_0$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{\varepsilon} q_0^1$. The constraints of type (4) describe how to propagate the values of the variables $x_{q,q'}$ step-by-step: if there exists a word u such that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{u} p$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{u} p'$, i.e., $x_{p,p'}$ is assigned to `true`, and there are transitions $\delta(p,a) = q$ in $\mathcal{A}_{\mathfrak{M}}$ and $(p',a,q') \in \Delta_1$, then $x_{q,q'}$ has to be assigned to `true`, too.

Using the variables $x_{q,q'}$ we can now express that $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_{\mathfrak{M}})$ as done below by the constraints of type (5). These constraints state that if a word leads to an accepting state in $\mathcal{A}_1$, then it also leads to an accepting state in $\mathcal{A}_{\mathfrak{M}}$.

$$x_{q,q'} \rightarrow f_q \qquad q \in Q,\ q' \in F_1 \tag{5}$$

Let $\varphi_n^{\mathcal{A}_1 \subseteq}(\overline{d}, \overline{f}, \overline{x})$ be the conjunction of the constraints of type (3) to (5) where $\overline{d}$, $\overline{f}$ are as described above, and $\overline{x}$ is the vector of new variables $x_{q,q'}$ with $q \in Q$ and $q' \in Q_1$. Then, we obtain the following lemma.

Lemma 1. *If* $\mathfrak{M} \models \varphi_n^{DEA}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{A}_1 \subseteq}(\overline{d}, \overline{f}, \overline{x})$*, then* $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_{\mathfrak{M}})$*.*

Proof. The proof follows the same line of arguments that we used in our intuitive explanation above. Let $\mathfrak{M} \models \varphi_n^{\mathrm{DEA}}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{A}_1 \subseteq}(\overline{d}, \overline{f}, \overline{x})$.

First, let us show by induction that if there exists a word w such that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{w} q$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{w} q'$, then $x_{q,q'}$ is assigned to `true`. For $w = \varepsilon$ the claim holds since $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{\varepsilon} q_0$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{\varepsilon} q_0^1$, and constraint (3) forces x_{q_0,q_0^1} to be set to `true`. For $w = ua$ assume that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{u} p \xrightarrow{a} q$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{u} p' \xrightarrow{a} q'$. In particular, this means that there are transitions $\delta(p, a) = q$ in $\mathcal{A}_{\mathfrak{M}}$, i.e. $\mathfrak{M}(d_{p,a,q}) = 1$, and $(p', a, q') \in \Delta_1$. Moreover, the induction hypothesis yields that $x_{p,p'}$ is assigned to `true`. Then, however, the constraints of type (4) force $x_{q,q'}$ to be set to `true`, too.

Now, let $w \in L(\mathcal{A}_1)$. Then, there exists $q \in Q$ and $q' \in F_1$ such that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{w} q$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{w} q'$. Hence, $\mathfrak{M}(x_{q,q'}) = 1$. In this case, the constraints of type (5) assure that $\mathfrak{M}(f_q) = 1$, and, hence, $w \in L(\mathcal{A}_{\mathfrak{M}})$. □

The formula $\varphi_n^{\neg \mathcal{A}_2}$ works analogous to $\varphi_n^{\mathcal{A}_1 \subseteq}$. We introduce new auxiliary variables $y_{q,q'}$ with $q \in Q, q' \in Q_2$ and modify the constraints of type (3) and (4) accordingly. Note that we need to change the constraints of type (5) slightly such that they now express that whenever a word is accepted by $\mathcal{A}_2$, then it is rejected by $\mathcal{A}_{\mathfrak{M}}$. The constraints on the variables $y_{q,q'}$ are listed below.

$$y_{q_0,q_0^2} \tag{6}$$

$$(y_{p,p'} \wedge d_{p,a,q}) \rightarrow y_{q,q'} \qquad p, q \in Q,\ p', q' \in Q_2,\ a \in \Sigma,\ (p', a, q') \in \Delta_2 \tag{7}$$

$$y_{q,q'} \rightarrow \neg f_q \qquad q \in Q,\ q' \in F_2 \tag{8}$$

Let $\varphi_n^{\neg \mathcal{A}_2}(\overline{d}, \overline{f}, \overline{y})$ be the conjunction of the constraints of type (6) to (8) where $\overline{d}$, $\overline{f}$ are as described above, and $\overline{y}$ is the vector of new variables $y_{q,q'}$ with $q \in Q$ and $q' \in Q_2$. Analogous to Lemma 1 we obtain $L(\mathcal{A}_2) \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$ if $\mathfrak{M} \models \varphi_n^{\mathrm{DEA}}(\overline{d}, \overline{f}) \wedge \varphi_n^{\neg \mathcal{A}_2}(\overline{d}, \overline{f}, \overline{y})$.

We can now combine all subformulae and obtain the following result.

Lemma 2. *Let* $L_1, L_2 \subseteq \Sigma^*$ *be two disjoint regular languages,* $n \in \mathbb{N}$ *and*

$$\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}(\overline{d}, \overline{f}, \overline{x}, \overline{y}) = \varphi_n^{DFA}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{A}_1 \subseteq}(\overline{d}, \overline{f}, \overline{x}) \wedge \varphi_n^{\neg \mathcal{A}_2}(\overline{d}, \overline{f}, \overline{y}).$$

Then, $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}(\overline{d}, \overline{f}, \overline{x}, \overline{y})$ *is satisfiable if and only if there exists a DFA with* n *states that separates* L_1 *and* L_2*.*

Proof. The direction from left to right is a straight-forward application of the properties of the formulae $\varphi_n^{\mathcal{A}_1 \subseteq}$ and $\varphi_n^{\neg \mathcal{A}_2}$ (cf. Lemma 1). Let $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ be satisfiable and $\mathfrak{M} \models \varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$. Then, $\mathcal{A}_{\mathfrak{M}}$ is an automaton with n states that separates L_1 and L_2.

For the reverse direction, let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DFA with n states that separates L_1 and L_2. From $\mathcal{A}$ we can derive a model $\mathfrak{M}$ as follows: we set $\mathfrak{M}(d_{p,a,q}) = 1$ if and only if $\delta(p, a) = q$, and $\mathfrak{M}(f_q) = 1$ if and only if $q \in F$. Values for the variables $x_{q,q'}$ and $y_{q,q'}$ can be derived by looking at the states reachable in the products of $\mathcal{A}$ and $\mathcal{A}_1$ as well as $\mathcal{A}$ and $\mathcal{A}_2$, respectively. □

Finally, let us note that $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ can easily be turned into conjunctive normal form by applying that $A \rightarrow B$ is logically equivalent to $\neg A \vee B$ and De Morgan's law. In conjunctive normal form, $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ ranges over $\mathcal{O}(n^2|\Sigma| + n(|Q_1| + |Q_2|))$ variables and comprises $\mathcal{O}(n^2(|\Delta_1| + |\Delta_2|) + n(|F_1| + |F_2|))$ clauses.

3.2 Finding Separating DFAs Using SMT Formulae

In this section, we implement the formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ in the quantifier-free logic over the structure $(\mathbb{N}, <, =)$ with constants $c \in \mathbb{N}$ and uninterpreted functions. To this end, let us assume that all automata have a special form: the set of states is $Q = \{0, 1, \dots, n-1\}$, the initial state is 0, and the alphabet is $\Sigma = \{0, 1, \dots, m-1\}$. We can easily achieve this form by renaming the states of the automaton and symbols of the alphabet.

The formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ is based on the very same idea as in Section 3.1, but uses two functions $d \colon \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ and $f \colon \mathbb{N} \rightarrow \{0, 1\}$ to encode the automaton we are searching for; the function d encodes the transitions whereas f encodes the set of final states. By means of these functions, it is easy to derive an automaton $\mathcal{A}_{\mathfrak{M}} = (\{0, \dots, n-1\}, \Sigma, 0, \delta, F)$ from a model $\mathfrak{M} \models \varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$: we define $\delta(i, a) = d_{\mathfrak{M}}(i, a)$ for $i \in Q$, $a \in \Sigma$ and $i \in F \Leftrightarrow f_{\mathfrak{M}}(i) = 1$. To ensure that δ is well-defined, we will make sure that $d(i, a) < n$ is satisfied for $0 \leq i < n$ and $0 \leq a < m$.

To express that $\mathcal{A}_{\mathfrak{M}}$ separates L_1 and L_2, we additionally utilize two auxiliary functions $x \colon \mathbb{N} \times \mathbb{N} \rightarrow \{0, 1\}$ and $y \colon \mathbb{N} \times \mathbb{N} \rightarrow \{0, 1\}$, which have the same meaning as the equally named variables in Section 3.1. By means of the functions d, f, x, y, we can now rephrase the constraints of Section 3.1 as follows.

$$\begin{array}{rl}
d(i,a) < n & i \in Q,\ a \in \Sigma \\
x(0,0) \wedge y(0,0) & \\
x(i,i') \rightarrow x(d(i,a),j') & i \in Q,\ i',j' \in Q_1,\ a \in \Sigma,\ (i',a,j') \in \Delta_1 \\
x(i,i') \rightarrow f(i) & i \in Q,\ i' \in F_1 \\
y(i,i') \rightarrow y(d(i,a),j') & i \in Q,\ i',j' \in Q_2,\ a \in \Sigma,\ (i',a,j') \in \Delta_2 \\
y(i,i') \rightarrow \neg f(i) & i \in Q,\ i' \in F_2
\end{array}$$

Let $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}(d, f, x, y)$ be the conjunction of these constraints. Then, we obtain the following lemma. The proof is analogous to the proof of Lemma 2 and, therefore, skipped.

Lemma 3. *Let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages and $n \in \mathbb{N}$. Then, $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}(d, f, x, y)$ is satisfiable if and only if there exists a DFA $\mathcal{A}$ with n that separates L_1 and L_2.*

Finally, let us briefly remark that the formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}(d, f, x, y)$ comprises $\mathcal{O}(n|\Sigma| + n(|\Delta_1| + |\Delta_2|) + n(|F_1| + |F_2|))$ constraints.

4 Regular Invariants

Let us now extend the technique of the previous section to the task of synthesizing regular invariants. Regular invariants are defined in terms of finite-state transducers. A *finite-state transducer* is basically an NFA (or DFA) $\mathcal{A}$ that works over the alphabet $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})$ and reads pairs (u, v) of words. We write $\mathcal{A} \colon p \xrightarrow{(u,v)} q$ if there exists a sequence of transitions in $\mathcal{A}$ that leads from state p to q and where the labels along the transitions yield the pair (u, v). Rather than a language, a finite-state transducer accepts (or defines) a relation $R(\mathcal{A}) \subseteq \Sigma^* \times \Sigma^*$ where $R(\mathcal{A}) = \{(u, v) \mid \mathcal{A} \colon q_0 \xrightarrow{(u,v)} q, q \in F\}$. A relation $R \subseteq \Sigma^* \times \Sigma^*$ is called *rational* if there exists a finite-state transducer $\mathcal{A}$ such that $R = R(\mathcal{A})$.

For a relation $R \subseteq \Sigma^* \times \Sigma^*$ we denote the reflexive and transitive closure by R^*. For a language $L \subseteq \Sigma^*$ and a relation $R \subseteq \Sigma^* \times \Sigma^*$ let $R(L) = \{v \in \Sigma^* \mid \exists u \in A \colon (u, v) \in R\}$ be the image of L under R. Finally, we call a (regular) language $L \subseteq \Sigma^*$ a *(regular) invariant* if $R(L) \subseteq L$.

Synthesizing regular invariants (that of course possess additional properties) is an important task in various applications. In the next two subsections we show how our technique can be applied to two prominent settings: regular model checking (in Section 4.1) and synthesis of loop invariants (in Section 4.2).

4.1 Regular Model Checking

In the regular model checking framework [4], configurations of a program (or system) are modeled as finite words over a fixed alphabet Σ. The program itself is a tuple $\mathcal{P} = (I, T)$ consisting of a regular set $I \subseteq \Sigma^*$ of *initial configurations* and a rational relation $T \subseteq \Sigma^* \times \Sigma^*$ defining the *transitions*, i.e., the successor relation on the configurations. Regular model checking—more precise, the regular model checking problem—is the decision problem

“Given a program $\mathcal{P} = (I, T)$ and a regular set $B \subseteq \Sigma^*$ of *bad configurations.* Does $T^*(I) \cap B = \emptyset$ hold?”.

In other words, the model checking problem asks whether there is a path in $\mathcal{P}$ that leads from an initial configuration into the set of bad configurations. In this case, the program is erroneous. Note that the model checking problem is in general undecidable (cf. [4]).

Many tools for regular model checking such as T(O)RMC [10], which is based on LASH, or LEVER [16] try to compute a regular set that overapproximates the reachable configurations, is an invariant, and has an empty intersection with the set of bad configurations. More formally, these tools search for a regular set $P \subseteq \Sigma^*$ with $I \subseteq P$, $B \cap P = \emptyset$, and $T(P) \subseteq P$. We call such a set a *proof* that the program $\mathcal{P}$ is correct (with respect to B) since it proves that there does not exist a path from any initial configuration to a bad one. In other words, a proof is a regular invariant with the additional properties $I \subseteq P$ and $B \cap P = \emptyset$.

We can extend our technique from the previous section to compute proofs. To this end, let us assume that I and B are given as two NFAs $\mathcal{A}^I$ and $\mathcal{A}^B$ and that T is given as a finite-state transducer $\mathcal{A}^T = (Q^T, (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}), q_0^T, \Delta^T, F^T)$.

Analogous to Section 3, we create a formula $\psi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$ that is satisfiable if and only if there exists a DFA $\mathcal{A}$ with n states such that $L(\mathcal{A})$ is a proof. In the following, we show how this is done for SAT formulae. The adaptation for SMT formulae can be done in a straight-forward manner.

Following the definition of a proof from above, we define $\psi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$ as

$$\psi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T} := \varphi_n^{\text{DEA}} \wedge \varphi_n^{\mathcal{A}^I \subseteq} \wedge \varphi_n^{\neg \mathcal{A}^B} \wedge \varphi_n^{\text{inv } \mathcal{A}^T}$$

where the subformulae $\varphi_n^{\mathcal{A}^I \subseteq}$ and $\varphi_n^{\neg \mathcal{A}^B}$ are as in Section 3.1, and $\varphi_n^{\text{inv } \mathcal{A}^T}$ ensures that if $\mathfrak{M} \models \varphi_n^{\text{DEA}} \wedge \varphi_n^{\text{inv } \mathcal{A}^T}$, then $T(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.

In other words, the formula $\varphi_n^{\text{inv } \mathcal{A}^T}$ needs to make sure that $L(\mathcal{A}_{\mathfrak{M}})$ is an invariant. To this end, we consider the parallel behavior of $\mathcal{A}_{\mathfrak{M}}$ and $\mathcal{A}^T$ analogous to Section 3.1. This time, however, the situation is more involved since $\mathcal{A}^T$ works on pairs (u, v) of words.

We need to establish that if (u, v) is accepted by $\mathcal{A}^T$ and $u \in L(\mathcal{A}_{\mathfrak{M}})$, then $v \in L(\mathcal{A}_{\mathfrak{M}})$ holds, too. In other words, this means that if $\mathcal{A}^T$ reaches a final state after reading (u, v) and $\mathcal{A}_{\mathfrak{M}}$ reaches a final state after reading u, then $\mathcal{A}_{\mathfrak{M}}$ must also reach a final state after reading v. This condition can be expressed using auxiliary variables $z_{q,q',q''}$ with $q, q'' \in Q$ and $q' \in Q^T$. Their meaning is that if $\mathcal{A}^T \colon q_0^T \xrightarrow{(u,v)} q'$, $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{u} q$, and $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{v} q''$, then $z_{q,q',q''}$ is set to `true`. The following constraints define the formula $\varphi_n^{\text{inv } \mathcal{A}^T}$.

$$z_{q_0, q_0^B, q_0} \tag{9}$$

$$(z_{p,p',p''} \wedge d_{p,a,q} \wedge d_{p'',b,q''}) \rightarrow z_{q,q',q''} \qquad p, p'', q, q'' \in Q,\ a, b \in \Sigma,\ p', q' \in Q^T,\ (p', (a, b), q') \in \Delta^T \tag{10}$$

$$(z_{p,p',p''} \wedge d_{p,a,q}) \rightarrow z_{q,q',p''} \qquad p, p'', q \in Q,\ a \in \Sigma,\ p', q' \in Q^T,\ (p', (a, \varepsilon), q') \in \Delta^T \tag{11}$$

$$(z_{p,p',p''} \wedge d_{p'',b,q''}) \rightarrow z_{p,q',q''} \qquad p, p'', q'' \in Q,\ b \in \Sigma,\ p', q' \in Q^T,\ (p', (\varepsilon, b), q') \in \Delta^T \tag{12}$$

$$(z_{q,q',q''} \wedge f_q) \rightarrow f_{q''} \qquad q, q'' \in Q,\ q' \in F^T \tag{13}$$

Let $\varphi_n^{\text{inv } \mathcal{A}^T}(\overline{d}, \overline{f}, \overline{z})$ be the conjunction of the constraints of type (9) to (13) where $\overline{d}$, $\overline{f}$ are as described in Section 3.1, and $\overline{z}$ is the vector of new variables $z_{q,q',q''}$. Then, we obtain $T(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$ in analogy to Lemma 1.

We can now combine all subformulae and obtain an algorithm to compute proofs in regular model checking. The algorithm is shown in Algorithm 2. Let us remark that $\psi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$ ranges over $\mathcal{O}(n^2|\Sigma| + n(|Q_I| + |Q_B| + n|Q_T|))$ variables and comprises $\mathcal{O}(n^2(|\Delta^I| + |\Delta^B| + n^2|\Delta^T|) + n(|F^I| + |F^B| + n|F^T|))$ clauses.

Using Algorithm 2 we can establish Theorem 2. The proof of Theorem 2 is done analogously to Theorem 1 and, hence, skipped.

Algorithm 2: Computing proofs using SAT or SMT solvers.

```
Input: a program $\mathcal{P} = (I, T)$ and a regular set $B$ of bad configurations.
$n := 0$;
repeat
    $n := n + 1$;
    Construct and solve $\psi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$;
until $\psi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$ is satisfiable;
Construct and return $\mathcal{A}_{\mathfrak{M}}$;
```

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

Note that we cannot guarantee that Algorithm 2 terminates since there might not exist a proof at all. Hence, Algorithm 2 is necessarily a semi algorithm (the regular model checking problem is undecidable). This, however, is of course also true for all other algorithms.

Related Work and Experiments. There are various tools for regular model checking, which have been successfully applied in practice. Prominent examples are T(O)RMC [10] (based on LASH), LEVER [16], and FAST [1].

T(O)RMC computes overapproximations of the set of reachable configurations using various extrapolation techniques. However, T(O)RMC does not consider the set of bad configurations directly and, hence, cannot guarantee to find an overapproximation having an empty intersection with the set of bad configurations. In such a situation, T(O)RMC has to be restarted with different settings until a proof has been found. This is a resource consuming process, and choosing "good" settings requires in-depth knowledge about both T(O)RMC and the particular application domain. By contrast, our technique is generic and does not need any particular knowledge about the domain. Moreover, T(O)RMC requires DFAs as input whereas our technique can also handle NFAs. Thus, the input for our technique can be exponentially smaller than an equivalent input for T(O)RMC.

LEVER computes proofs using automata learning techniques. Thereby, it does not learn a proof directly, but a set of configurations that are enriched with additional information. This is necessary to be able to answer queries posed by the learning algorithm. However, the problem with this approach is that even if a proof exists it can no longer be guaranteed that these augmented sets are regular. In this case, LEVER cannot find a proof and might not terminate. This is a drawback compared to our technique, which always finds a proof if one exists. Note, however, that learning based techniques treat the input automata as black boxes whereas we assume them to be known as white boxes.

FAST computes the exact set of reachable configurations by means of acceleration techniques. Also here, this set is not necessarily regular (in general not even computable) although a proof might exist. In such situations, FAST fails and does not terminate.

At this point let us note that our approach is different from bounded model checking [2] although it seems quite similar. The idea of bounded model checking is to search for a bad execution whose length is bounded by some integer k. The value of k is increased until a bug is found or some a priori defined bound is reached. We, on the other hand, always consider the whole program, but parametrize the size of a proof we are looking for.

We implemented a proof-of-concept based on the MiniSat SAT solver and Microsoft's Z3 SMT solver. To evaluate our technique, we ran experiments suggested in [9]. These experiments comprise a token-ring protocol and programs relying on so-called "arithmetic relations". The results are so far promising and show that our technique performs well not only on toy examples but also on non-trivial medium size examples. This holds especially in situation where it is a priori known that proofs are small. In our experiments, we could not find any significant difference between the SAT and SMT based approach.

Finally, let us mention that our proof-of-concept is not meant to compete with the above-named tools, mainly for two reasons. First, it is by far not as optimized as the mature tools, and further optimizations remain to be investigated. Second, it is a generic technique not solely dedicated to regular model checking.

4.2 Synthesizing Loop Invariants in Presburger Arithmetic

In this section, we consider imperative programs that work over integer variables. An example of such a program is depicted in Figure 1.

Our goal is to compute (or synthesize) loop invariants expressible in Presburger arithmetic for annotated loops. Thereby, we assume the following setting. Program states are described by Presburger formulae $\varphi(\overline{x})$ that range over all program variables $\overline{x} = (x_1, \ldots, x_n) \in \mathbb{Z}^n$. The annotated loop is given as a precondition $\varphi_{\text{pre}}(\overline{x})$ of the loop, a postcondition $\varphi_{\text{post}}(\overline{x})$, and a formula $\varphi_{\text{loop}}(\overline{x}, \overline{x}')$ that describes the effect of the loop on the program variables. Thereby, $\overline{x}$ corresponds to the variables before the loop body is executed whereas $\overline{x}'$ corresponds to the (altered) program variables after the loop body has been executed.

```
Input: x

r = 0;
y = x;
while(y > 0) {
    r = r + 3;
    y = y - 1;
}
```

Fig. 1. An example program over integer variables

$$\begin{aligned}
G(x,y,r) &\coloneqq y > 0\\
\varphi_{\text{pre}}(x,y,r) &\coloneqq r = 0 \land y = x\\
\varphi_{\text{post}}(x,y,r) &\coloneqq y = 0 \land\\
&\qquad\qquad \exists t\colon x + x = t \land x + t = r\\
\varphi_{\text{loop}}(x,y,r,x',y',r') &\coloneqq x' = x \land y' = y - 1 \land\\
&\qquad r' = r + 3
\end{aligned}$$

Fig. 2. Presburger formulae describing the loop of the program in Figure 1

Finally, let $G(\overline{x})$ be the loop guard. Figure 2 shows an example of formulae that describe the loop of the program in Figure 1.

Intuitively, a loop invariant is a statement about the states of a program that is true before and after every iteration of the loop. Formally, we define a *loop invariant* as a set *Inv* of program states that satisfies the following properties:

- $\varphi_{\text{pre}}(\overline{x}) \rightarrow \overline{x} \in \mathit{Inv}$,
- $(\overline{x} \in \mathit{Inv} \wedge \neg G(\overline{x})) \rightarrow \varphi_{\text{post}}(\overline{x})$, and
- $(\overline{x} \in \mathit{Inv} \wedge G(\overline{x}) \wedge \varphi_{\text{loop}}(\overline{x}, \overline{x}')) \rightarrow \overline{x}' \in \mathit{Inv}$.

In the example of Figure 1 and 2, the set of program states satisfying the condition $3(x - y) = r$ is a loop invariant. In fact, this loop invariant is exactly what our technique (described below) computes in this example.

Since we are interested in loop invariants that can be expressed in Presburger arithmetic, we want to compute *regular loop invariants*, i.e., loop invariants that are regular languages, and translate them into Presburger formulae. To this end, we turn the formulae φ_{pre}, φ_{post}, and G into DFAs $\mathcal{A}^{\varphi_{\text{pre}}}$, $\mathcal{A}^{\varphi_{\text{post}}}$, and $\mathcal{A}^{G}$ working over the alphabet $\Sigma = \{0, 1\}^n$ and φ_{loop} into a finite-state transducer $\mathcal{A}^{\varphi_{\text{loop}}}$ working over the alphabet $\Sigma \times \Sigma$. Then, we can reformulate the definition of loop invariants from above as follows, where now a loop invariant is a set $\mathit{Inv} \subseteq \Sigma^*$ that matches the encoding of program states used by $\mathcal{A}^{\varphi_{\text{pre}}}$, $\mathcal{A}^{G}$, etc.

- $L(\mathcal{A}^{\varphi_{\text{pre}}}) \subseteq \mathit{Inv}$,
- $(\Sigma^* \setminus (L(\mathcal{A}^{G}) \cup L(\mathcal{A}^{\varphi_{\text{post}}}))) \cap \mathit{Inv} = \emptyset$
  (which is true if and only if $(\mathit{Inv} \cap (\Sigma^* \setminus L(\mathcal{A}^{G}))) \subseteq L(\mathcal{A}^{\varphi_{\text{post}}})$), and
- $(R(\mathcal{A}^{\varphi_{\text{loop}}}) \cap (L(\mathcal{A}^{G}) \times \Sigma^*))(\mathit{Inv}) \subseteq \mathit{Inv}$.

If phrased this way, and if we set $I = L(\mathcal{A}^{\varphi_{\text{pre}}})$, $B = \Sigma^* \setminus (L(\mathcal{A}^{G}) \cup L(\mathcal{A}^{\varphi_{\text{post}}}))$, and $T = (R(\mathcal{A}^{\varphi_{\text{loop}}}) \cap (L(\mathcal{A}^{G}) \times \Sigma^*))$, then computing (or synthesizing) regular loop invariants boils down to computing proofs in the sense of Section 4.1. This leads to the main result of this section.

Theorem 3. *Let Presburger formulae φ_{pre}, φ_{post}, φ_{loop}, and G for a loop of an integer program be given, and let $I = L(\mathcal{A}^{\varphi_{pre}})$, $B = \Sigma^* \setminus (L(\mathcal{A}^{G}) \cup L(\mathcal{A}^{\varphi_{post}}))$, and $T = (R(\mathcal{A}^{\varphi_{loop}}) \cap (L(\mathcal{A}^{G}) \times \Sigma^*))$. Then, Algorithm 2 with input $\mathcal{P} = (I, T)$ and B terminates and returns a regular loop invariant if one exists.*

Once Algorithm 2 returns a regular loop invariant, we can try to translate it into a Presburger formula according to [11]. However, even if this is not possible, a (regular) loop invariant is enough for the verification of programs as it proves that the postcondition holds once the loop terminates. Nonetheless, it would be interesting to investigate whether we can impose further constraints on the formula $\varphi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$ (corresponding to the characterization in [11]) that guarantee that a computed loop invariant can be translated into a Presburger formula.

Related Work and Experiments. Many techniques to compute loop invariants (for integer programs) have been proposed, e.g., abstract interpretation (used in [8]) or template-based approaches (used in [15]) to name just two examples. The idea of the latter is to search for loop invariants only in a restricted (often "simple") domain such as linear inequalities or polyhedra. In this sense, we use Presburger formulae, or DFAs, as templates. As to our knowledge not much research has been devoted to this type of templates. Moreover, the idea of using tools for regular model checking to synthesize loop invariants is novel.

To try our approach, we extended the SAT-based proof-of-concept of Section 4.1 using Mona [7] to translate Presburger formulae into DFAs. We considered example programs (mostly fragments taken from real world software) that are delivered with the InvGen toolkit [6]. About 10% of these examples (nine in total) were suitable for our setting, i.e., they contained a single loop, the effect of the loop could be expressed in Presburger arithmetic, etc.

Table 1. Experimental results of InvGen's "C test suite" examples

Experiment	Size	Presburger
down	3	yes
gulwani_cegar2	3	yes
half	7	
ken-imp	3	
NetBSD_g_Ctoc	2	yes
simple	2	yes
simple_if	2	yes
split	6	
up-nd	2	yes

Table 1 shows the results of our experiments. The column "Size" shows how many states a resulting DFA comprises. The column "Presburger" indicates whether a loop invariant could be translated into a Presburger formula. Since we did not use software for this task, a blank entry indicates that a DFA did not obviously encode a formula. All experiments were done on an Intel Q9550 CPU running Linux, each in less that 30 seconds with at most 300 MB of RAM used. As Table 1 depicts, our implementation found loop invariants for all examples. In comparison, InvGen also found loop invariants for all examples and used roughly the same amount of time and memory. This shows that our technique can match InvGen (where it is applicable).

However, we did not benchmark our technique with template-based tools other than InvGen. Such a comparison seems to be unsatisfactory for two reasons. First, there are very few (and perhaps uninteresting) examples that have two equally complex solutions for either template and would allow a fair comparison. Second, our implementation is a proof-of-concept, and it is doubtful whether it can compete with optimized tools on their type of templates.

5 Conclusion

We presented a generic technique to compute minimal separating DFAs and, based on that, regular invariants. The main idea is to express the desired properties of a DFA in terms of a logical formula and to use a SAT or SMT solver to find a solution. We applied this technique to the task of computing minimal separating DFAs directly as well as to regular model checking and to the synthesis of loop invariants of integer programs. Our experiments showed that SAT and SMT solvers are useful tools for these tasks.

The way we compute automata allows us to compute minimal DFAs (although that might not be needed in every situation). Moreover, we can easily adapt our technique to also compute minimal NFAs. On the one hand, this has the advantage that NFAs as solution can be exponentially smaller than DFAs, and, thus, φ_n might be satisfiable for much smaller n. The disadvantage, on the other hand, is that encoding NFAs enlarges φ_n significantly.

Finally, let us note that our work is meant as a showcase how to use SAT and SMT solvers to compute (minimal) DFAs that possess certain properties with respect to other given regular languages. In this sense, one can think of our technique as a generic toolkit from which an algorithm for a concrete problem can be instantiated. We hope that such a toolkit comes in handy for other researches and may be applied in different fields, too. For instance, our technique can also be used to compute winning strategies for games on automatic graphs [12].

Acknowledgment. I thank Madhusudan Parthasarathy and Christof Löding very much for drawing my attention to this topic and for their support.

References

1. Bardin, S., Leroux, J., Point, G.: FAST Extended Release. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 63–66. Springer, Heidelberg (2006)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)
3. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Transactions on Computers C-21(6), 592–597 (1972)
4. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular Model Checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
5. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning Minimal Separating DFA's for Compositional Verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
6. Gupta, A., Rybalchenko, A.: InvGen: An Efficient Invariant Generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
7. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic Second-order Logic in Practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
8. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
9. Legay, A.: Generic Techniques for the Verification of Infinite-State Systems. Ph.D. thesis, Universite de Liege (2007)

10. Legay, A.: T(O)RMC: A Tool for (ω)-Regular Model Checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 548–551. Springer, Heidelberg (2008)
11. Leroux, J.: A polynomial time presburger criterion and synthesis for number decision diagrams. In: LICS, pp. 147–156. IEEE Computer Society (2005)
12. Neider, D.: Reachability Games on Automatic Graphs. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 222–230. Springer, Heidelberg (2011)
13. Pena, J.M., Oliveira, A.L.: A new algorithm for the reduction of incompletely specified finite state machines. In: ICCAD, pp. 482–489 (1998)
14. Pfleeger, C.: State reduction in incompletely specified finite-state machines. IEEE Transactions on Computers C-22(12), 1099–1102 (1973)
15. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable Analysis of Linear Systems Using Mathematical Programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
16. Vardhan, A., Viswanathan, M.: LEVER: A Tool for Learning Based Verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 471–474. Springer, Heidelberg (2006)