T U M

I N S T I T U T F Ü R I N F O R M A T I K

Inferring Network Invariants Automatically

Olga Grinchtein, Martin Leucker, Nir Piterman

TUM–I0603
März 06

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-03-I0603-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Inferring Network Invariants Automatically

Olga Grinchtein[1], Martin Leucker[2], and Nir Piterman[3]

[1] Department of Computer Systems, Uppsala University, Sweden
[2] IT Department, TU Munich, Germany
[3] EPFL, Lausanne, Switzerland

Abstract. Verification by network invariants is a heuristic to solve uniform verification of parameterized systems. Given a system P, a network invariant for P is a system that abstracts the composition of every number of copies of P running in parallel. If there is such a network invariant, by reasoning about it, uniform verification with respect to the family $P[1] \parallel \cdots \parallel P[n]$ can be carried out. In this paper, we propose a procedure that searches systematically for a network invariant satisfying a given safety property. The search is based on algorithms for learning finite automata due to Angluin and Biermann. We optimize the search by combining both algorithms for improving successive possible invariants. We also show how to reduce the learning problem to SAT, allowing efficient SAT solvers to be used, which turns out to yield a very competitive learning algorithm. The overall search procedure finds a minimal such invariant, if it exists.

1 Introduction

One of the most challenging problems in verification is the *uniform verification of parameterized systems*. Given a parameterized system $S(n) = P[1] \parallel \cdots \parallel P[n]$ and a property φ, uniform verification attempts to verify that $S(n)$ satisfies φ for every $n > 1$. The problem is in general undecidable [AK86]. One possible approach is to look for restricted families of systems for which the problem is decidable (cf. [EK00,CTTV04]). Another approach is to look for sound but incomplete methods (e.g., explicit induction [EN95], regular model checking [JN00,PS00], or environment abstraction [CTV06]).

Here, we attack uniform verification of parameterized systems using the heuristic of network invariants [WL89,KM95]. In simple words[1], a *network invariant* for a given finite system P is a finite system I that abstracts the composition of every number of copies of P running in parallel. Thus, the network invariant contains all possible computations of every number of copies of P. If we find such a network invariant I, we can solve uniform verification with respect to the family $S(n) = P[1] \parallel \cdots \parallel P[n]$ by reasoning about I.

The general idea proposed in [WL89] and turned into a working method in [KM95], is to show by induction that I is a network invariant for P. The induction base is to prove that $P \sqsubseteq I$, for a suitable abstraction relation $\sqsubseteq$. The induction step is to show that $P \parallel I \sqsubseteq I$. After establishing that I is a network invariant we can prove $I \models \varphi$, turning I to a *proper* network invariant with respect to φ. Then we conclude that $S(n) \models \varphi$ for every value of n.

[1] We give a precise definition of the network invariants approach in Section 2.

Coming up with a proper network invariant is usually an iterative process. We start with divining a candidate for a network invariant. Then, we try to prove by induction that it is a network invariant. When the candidate system is nondeterministic this usually involves deductive proofs [KPSZ02][2]. During this stage we usually need to refine the candidate until getting a network invariant. The final step is checking that this invariant is proper (by automatically model checking the system versus φ). If it is not, we have to continue refining our candidate until a proper network invariant is found. Coming up with the candidate network invariant requires great knowledge of the parameterized system in question and proving abstraction using deductive methods requires great expertise in deductive proofs and tools. Whether a network invariant exists is undecidable [WL89], hence all this effort can be done in vain.

In this paper, we propose a procedure searching systematically for a network invariant satisfying a given safety property. If one exists, the procedure finds a proper invariant with a minimal number of states. If no proper invariant exists, our procedure in general diverges (though in some cases it may terminate and report that no proper invariant exists). In the light of the undecidability result for the problem, this seems reasonable.

Network invariants are usually explained in the setting of *transition structures*[KP00]. Here, we use learning algorithms that are best explained in terms of deterministic finite state machines (DFAs). Operations like parallel composition are not very natural in the context of DFAs (while standard in the context of transition structures). Porting the learning algorithms to the context of transition structures is not complicated, however, explaining the learning algorithms in the context of transition structures is unnatural and renders the exposition hard to follow. Thus, we explain our work in the setting of checking safety properties of networks that are described in terms of (the parallel product of) DFAs.

As mentioned, this paper is about searching for network invariants. As the class of DFAs is enumerable, a naïve algorithm would be to enumerate all possible DFAs and check one after the other whether it is a proper invariant. Clearly, this algorithm is not feasible in practice. We improve the naïve search for a minimal proper invariant by employing *learning algorithms*. The learning algorithm queries for additional information like which strings should be accepted by the network invariant and which rejected. This information is gathered by checking the system P and the property φ. When the learning algorithm proposes an automaton that is not an invariant, we identify some string that should be accepted or rejected by a real invariant. This information can be fed back to the learning algorithm to improve the candidate for invariant.

Two types of inference (or learning) algorithms for DFAs can be distinguished, so-called *online* and *offline* algorithms. Online algorithms, such as Angluin's L^* algorithm [Ang87], query whether strings are in the language, before coming up with an automaton. Offline algorithms get a fixed set of examples and no further queries are allowed before computing a minimal DFA conforming to the examples. Typical algorithms of this type are based on a characterization in terms of a constraint satisfaction problem (CSP) over the natural numbers due to Biermann [BF72].

[2] For a recent attempt at mechanizing this step see [KPP05].

Clearly, an online algorithm like Angluin's should perform better than offline algorithms like Biermann's. Indeed, Angluin's algorithm is polynomial while without the ability to ask further queries the problem is known to be NP-complete [Gol78]. In our setting, however, we cannot rely completely on Angluin's algorithm. The definition of a network invariant does not identify an automaton completely. In consequence, in some cases we identify behaviors that can either be added to or equally well be removed from the candidate invariant. Thus, queries may be answered by *maybe*.

We therefore define an algorithm that is a combination of an online algorithm and an offline algorithm and is inspired by [PO98]. Similar to Angluin's algorithm, we round off the information on the automaton in question by asking queries. As queries can be answered by *maybe*, we may not be able to complete the information as in Angluin's setting to compute a DFA directly. For this, we use Biermann's approach for obtaining a DFA based on the enriched information. Our combination is conservative in the sense that in case all queries are answered by either *yes* or *no*, we obtain the same efficiency as for Angluin's algorithm. Furthermore, the encoding in terms of CSP is optimized based on the information collected in Angluin's algorithm. Both advantages are in contrast to the combination proposed in [PO98].

While in [OS01] an efficient implementation for solving the resulting CSP problem is explained, we give an encoding as a SAT problem featuring a simple yet—as the examples show—very efficient inference algorithm by employing powerful SAT solvers.

We note that there are efficient algorithms inferring a DFA that is not necessarily of minimal size, like [Lan92], [OG92] (known as RPNI). However, getting a minimum size automaton is essential to obtain our semi-computability result, which is why we cannot use these algorithms.

To validate our approach in practice, we have implemented it and tailored it to the intensively studied setting of *transition structures* [KP00]. Our implementation is based on the verification tool TLV [PS96] and on the SAT solver ZCHAFF [MMZ$^+$01]. We have tested our implementation on a well-studied example of mutual-exclusion protocol. We establish that the protocol is safe (i.e., no two processes are in the critical section simultaneously). The proper network invariant for this example is obtained in about 2 seconds.

Automatic inference of network invariants has been studied in [LHR97]. Their solution is based on *heuristically* solving a recursive equation for I. For some examples, a proper invariant has been found within seconds, while for others, no proper invariant was obtained automatically, though one exists. In contrast, in the case that a proper invariant exists, our algorithm would find one. Furthermore, the optimized yet systematic search for a proper invariant allows to inform the user of our tool about the current progress by saying up-to which size all possible invariants have been rejected. Such a requirement is extremely important especially for semi-terminating algorithms. In other words, while the general problem studied in this paper is undecidable, our algorithm decides the restricted problem of whether, for a given natural number n, a proper network invariant with at most n states exists.

Recently, several applications of learning techniques for verification problems have been proposed. In [HV05,VSVA04b], learning was used in the setting of regular model checking to verify safety properties. The approach was extended in [VSVA05] to check-

ing ω-regular properties. An application to verify FIFO automata is given in [VSVA04a]. None of the approaches deals with queries that are possibly answered by *maybe*. Therefore, these papers do not address the combination of learning algorithms. Less related combinations of verification and learning are reported for example in [AMN05,CCST05].

Contribution To the best of our knowledge, this is the first time learning techniques are applied to the problem of finding network invariants and the first efficient realization of a learning problem in terms of SAT solving. Furthermore, our combination of Angluin's L^* and Biermann's approach is more efficient than that of [PO98] due to additional optimizations.

Outline We recall the framework of verification by network invariants tailored to the setting of DFAs in the next section. Section 3 recalls Angluin's and Biermann's inference algorithms, presents a simple combination of both of them, reduces it to SAT, and discusses some optimizations. The search procedure finding proper network invariants is described in Section 4. We examine the case study in Section 5, before we draw final conclusions.

Acknowledgment We thank Bengt Jonsson and Christian Schallhart for valuable comments and discussions.

2 Verification by Network Invariants

We recall the notion and notation of verification by network invariants, tailored to the setting of checking safety properties of system families built-up by DFAs.

For $n \in \mathbb{N}$, let $[n] := \{1, \ldots, n\}$. For the rest of this section, we fix an alphabet Σ. A *deterministic finite automaton* (DFA) $\mathcal{A} = (Q, q_0, \delta, Q^+)$ over Σ consists of a finite set of *states* Q, an *initial state* $q_0 \in Q$, a *transition function* $\delta : Q \times \Sigma \to Q$, and a set $Q^+ \subseteq Q$ of *accepting states*. A *run* of $\mathcal{A}$ is a sequence $r{=}q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ such that $a_i \in \Sigma$, $q_i \in Q$ and $\delta(q_{i-1}, a_i) = q_i$ for all $i \in [n]$. It is called *accepting* iff $q_n \in Q^+$. We say that r is a run *over* $w = a_1 a_2 \ldots a_n$ and say that w is *accepted* if r is accepting. The *language* accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of accepted strings. We extend δ to strings as usual by $\delta(q, \lambda) = q$ and $\delta(q, ua) = \delta(\delta(q, u), a)$, where λ denotes the empty string. Let $\mathfrak{D}$ denote the class of DFAs (over Σ).

In order to reason about network invariants we have to consider *abstraction relations*, *safety properties*, and *parallel composition*. These notions are well established in the context of transition structures [KP00], however, in the context of DFAs may seem out of place. In what follows, we should have in mind the properties of these notions. These are the properties that are required to make the algorithm work. In the context of transition structures, these notions are well known, can be checked (where appropriate), and have the required properties.

An *abstraction relation* on $\mathfrak{D}$ is a reflexive and transitive relation $\sqsubseteq \subseteq \mathfrak{D} \times \mathfrak{D}$. Here, let $\sqsubseteq \subseteq \mathfrak{D} \times \mathfrak{D}$ be defined by $\mathcal{A} \sqsubseteq \mathcal{B}$ iff $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. A *safety property* is a DFA φ that has a *prefix-closed* language, i.e., $ua \in \mathcal{L}(\varphi)$ implies $u \in \mathcal{L}(\varphi)$, defining the intended correct behavior. Thus, a system $\mathcal{A} \in \mathfrak{D}$ satisfies φ, denoted by $\mathcal{A} \models \varphi$, iff $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\varphi)$. Clearly, in our setting, the abstraction relation is *sound* with respect to safety properties: For $\mathcal{A}, \mathcal{B} \in \mathfrak{D}$, $\varphi \in \mathfrak{D}$, $\mathcal{A} \sqsubseteq \mathcal{B}$ and $\mathcal{B} \models \varphi$ implies $\mathcal{A} \models \varphi$, as $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\varphi)$ implies $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\varphi)$.

A *parallel operator* on $\mathfrak{D}$ is a mapping $\|\colon \mathfrak{D} \times \mathfrak{D} \to \mathfrak{D}$. For notational simplicity, we assume $\|$ to be *associative* and *commutative*, although this is not essential. We call $\|$ *compatible* with respect to $\sqsubseteq$ if, for all $\mathcal{C} \in \mathfrak{D}$, $\mathcal{A} \sqsubseteq \mathcal{B} \implies \mathcal{A} \parallel \mathcal{C} \sqsubseteq \mathcal{B} \parallel \mathcal{C}$. Let us fix a parallel operator that is compatible with $\sqsubseteq$ for the rest of this paper.

A *projection operator* for $\mathcal{A} \parallel \mathcal{B}$ onto $\mathcal{B}$ is a mapping $pr^{\mathcal{B}}_{\mathcal{A}\parallel\mathcal{B}} : \mathcal{L}(\mathcal{A} \parallel \mathcal{B}) \to \mathcal{L}(\mathcal{B})$ such that whenever $w \in \mathcal{L}(\mathcal{A} \parallel \mathcal{B})$ then for all $\mathcal{B}'$ with $pr^{\mathcal{B}}_{\mathcal{A}\parallel\mathcal{B}}(w) \in \mathcal{L}(\mathcal{B}')$ also $w \in \mathcal{L}(\mathcal{A} \parallel \mathcal{B}')$. In other words, (at least) the projection of w has to be removed from $\mathcal{B}$ to (eventually) remove w from the parallel product.

Definition 1. *For $P \in \mathfrak{D}$, we call $I \in \mathfrak{D}$ a* network invariant*, iff* **(I1)** $P \sqsubseteq I$ *and* **(I2)** $P \parallel I \sqsubseteq I$*. If furthermore for $S \in \mathfrak{D}$ and a safety property $\varphi \in \mathfrak{D}$ we have* **(P)** $S \parallel I \models \varphi$ *we call I a* proper network invariant *for (S, P, φ).*

Often, we just say (proper) invariant instead of (proper) network invariant.

Theorem 1. [WL89] *Let S, P, I and φ in $\mathfrak{D}$ such that I is a proper network invariant for (S, P, φ). Then $S \parallel P[1] \parallel \cdots \parallel P[n] \models \varphi$, for all $n \in \mathbb{N}$ where, for every i, $P[i]$ is a copy of P.*

The problem studied in this paper can be phrased as follows:

Definition 2 (Proper Network Invariant Problem). *For systems S, P and a safety property φ in $\mathfrak{D}$, the* proper network invariant problem *is to compute a proper network invariant for (S, P, φ) (if it exists).*

Proposition 1. *The proper network invariant problem is semi-computable.*

Let us give a simple (but non-satisfactory) solution to the problem, based on the observation that the class of DFAs over a fixed alphabet is enumerable. Thus, let $I_1, I_2, \ldots$ be an enumeration of DFAs.

- Check whether (I1) and (I2) hold for P and I_i. If yes, I_i is a network invariant.
- If so, check whether $S \parallel I_i \models \varphi$. If yes, a proper invariant has been found and the procedure stops. If not, continue with $i + 1$.

It other words, the procedure finds a proper invariant, if one exists. Additionally, in case that DFAs are enumerated according to number of states, the resulting proper invariant is minimal with respect to its number of states.

Of course, the algorithm outlined above is, in a way, naïve, and clearly inefficient in practice. We use learning techniques to accelerate the search for the proper invariants in question. Our procedure still produces a minimal proper invariant, if one exists. Conditions (I1), (I2), and (P) are used for inferring properties of the proper invariant we are looking for.

3 Inference of Deterministic Finite Automata

3.1 Angluin's algorithm

Angluin's learning algorithm [Ang87] is designed for learning a regular language, $\mathcal{L} \subseteq \Sigma^*$, by constructing a minimal DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$. In this algorithm a $Learner$, who initially knows nothing about $\mathcal{L}$, is trying to learn $\mathcal{L}$ by asking a $Teacher$, who knows $\mathcal{L}$, two kinds of queries:

– A *membership query* consists of asking whether a string $w \in \Sigma^*$ is in $\mathcal{L}$.
– An *equivalence query* consists of asking whether a hypothesized DFA $\mathcal{H}$ is correct, i.e., whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. The $Teacher$ answers *yes* if $\mathcal{H}$ is correct, or else supplies a counterexample w, either in $\mathcal{L} \setminus \mathcal{L}(\mathcal{H})$ or in $\mathcal{L}(\mathcal{H}) \setminus \mathcal{L}$.

The $Learner$ maintains a prefix-closed set $U \subseteq \Sigma^*$ of prefixes, which are candidates for identifying states, and a suffix-closed set $V \subseteq \Sigma^*$ of suffixes, which are used to distinguish such states. The sets U and V are increased when needed during the algorithm. The $Learner$ makes membership queries for all words in $(U \cup U\Sigma)V$, and organizes the results into a *table* T that maps each $u \in (U \cup U\Sigma)$ to a mapping $T(u) : V \mapsto \{+, -\}$ where $+$ represents accepted and $-$ not accepted. In [Ang87], each function $T(u)$ is called a *row*. When T is

– *closed*, meaning that for each $u \in U$, $a \in \Sigma$ there is a $u' \in U$ such that $T(ua) = T(u')$, and
– *consistent*, meaning that $T(u) = T(u')$ implies $T(ua) = T(u'a)$,

the $Learner$ constructs a hypothesized DFA $\mathcal{H} = (Q, q_0, \delta, Q^+)$, where (a) $Q = \{T(u) \mid u \in U\}$ is the set of distinct rows, (b) q_0 is the row $T(\lambda)$, (c) δ is defined by $\delta(T(u), a) = T(ua)$, and (d) $Q^+ = \{T(u) \mid u \in U, T(u)(\lambda) = +\}$ and submits $\mathcal{H}$ as an equivalence query. If the answer is *yes*, the learning procedure is completed, otherwise the returned counterexample is used to extend U and V, and subsequent membership queries are performed in order to make the new table closed and consistent producing a new hypothesized DFA, etc.

In our setting, queries are no longer answered by either *yes* or *no*, but also by *maybe*, denoted by ?. We therefore list the necessary changes to Angluin's algorithm. We keep the idea of a table but now, for every $u \in (U \cup U\Sigma)$, we get a mapping $T(u) : V \to \{+, -, ?\}$. For $u, u' \in (U \cup U\Sigma)$, we say that rows $T(u)$ and $T(v)$ *look similar*, denoted by $T(u) \equiv T(u')$, iff, for all $v \in V$, $T(u)(v) \neq ?$ and $T(u')(v) \neq ?$ implies $T(u)(v) = T(u')(v)$. Otherwise, we say that $T(u)$ and $T(v)$ are *obviously different*. We call T

– *weakly closed* if for each $u \in U$, $a \in \Sigma$ there is a $u' \in U$ such that $T(ua) \equiv T(u')$, and
– *weakly consistent* if $T(u) \equiv T(u')$ implies $T(ua) \equiv T(u'a)$.

Angluin's algorithm works as before, but using the weak versions of closed and consistent. However, extracting a DFA from a weakly closed and weakly consistent table is no longer straightforward. For this, we rely on Biermann's approach, described next.

3.2 Biermann's algorithm

Biermann's learning algorithm [BF72] is also designed for learning a DFA $\mathcal{A}$. This time we are given a set of strings that are to be accepted by $\mathcal{A}$ and a set of strings that are to be rejected by $\mathcal{A}$. There is no possibility of asking queries and we have to supply a minimal possible DFA that accepts / rejects these strings. The set of positive and negative strings are called *sample*. We now formally describe samples and Biermann's algorithm.

A *sample* is a set of strings that, by the language in question, should either be accepted, denoted by $+$, or rejected, denoted by $-$. For technical reasons, it is convenient to work with prefix-closed samples. As the samples given to us are not necessarily prefix closed we introduce the value *maybe*, denoted by ?. Formally, a *sample* is a partial function $O : \Sigma^* \to \{+, -, ?\}$ that is defined for u whenever it is defined for some

ua. For a string u the sample O yields whether u should be *accepted*, *rejected*, or we do not know. For strings u and u', we say that O *disagrees* on u and u' if $O(u) \neq ?$, $O(u') \neq ?$, and $O(u) \neq O(u')$. Clearly, Angluin's table (including entries with ?) can easily be translated to a sample, possibly by adding prefixes to $(U \cup U\Sigma)V$ with value ? to obtain a prefix-closed domain. An automaton $\mathcal{A}$ is said to *conform* with a sample O, if whenever O is defined for u we have $O(u) = +$ implies $u \in \mathcal{L}(\mathcal{A})$ and $O(u) = -$ implies $u \notin \mathcal{L}(\mathcal{A})$.

Given a sample O and a DFA $\mathcal{A}$ that is conform to O, let S_u denote the state reached in $\mathcal{A}$ when reading u. As long as we do not have $\mathcal{A}$, we can treat S_u as a variable ranging over states and derive constraints for the assignments of such a variable. More precisely, let $\mathrm{CSP}(O)$ denote the set of equations

$$\begin{aligned} &\{S_u \neq S_{u'} && \mid O \text{ disagrees on } u \text{ and } u'\} && \text{(C1)} \\ \cup\, &\{S_u = S_{u'} \Rightarrow S_{ua} = S_{u'a} \mid a \in \Sigma, ua, u'a \in \mathcal{D}(O)\} && && \text{(C2)} \end{aligned}$$

Let the domain of $\mathcal{D}(\mathrm{CSP}(O))$ comprise the set of variables S_u used in the constraints. A *solution* of $\mathrm{CSP}(O)$ is mapping $\Gamma : \mathcal{D}(\mathrm{CSP}(O)) \to \mathbb{N}$ fulfilling the equations over the naturals, defined in the usual manner. The set $\mathrm{CSP}(O)$ is *solvable* over $[N]$ iff there is a solution with range $[N]$. It is easy to see that every solution of the CSP problem over the natural numbers can be turned into an automaton conforming with O.

Lemma 1 (Learning as CSP, [BF72]). *For a sample O, a DFA with N states conforming to O exists iff $\mathrm{CSP}(O)$ is solvable over $[N]$.*

We note that taking a different value for every S_u, trivially solves the CSP problem. Thus, a solution with minimum range exists and yields a DFA with a minimal number of states.

3.3 Pruning the search space of the CSP problem

In general, one finds a minimum DFA by trying to solve the corresponding CSP problem with subsequently larger integer ranges. However, before doing so, let us make a simple yet important observation to simplify the CSP problem. We call a bijection $\iota : [N] \to [N]$ a *renaming* and say that Γ and Γ' are *equivalent modulo* renaming iff there is a renaming ι such that $\Gamma = \iota \circ \Gamma'$.

Since names or numbers of states have no influence on the accepted language of an automaton, we get

Lemma 2 (Name irrelevance). *For a sample O, $\Gamma : \mathcal{D}(\mathrm{CSP}(O)) \to [N]$ is a solution for $\mathrm{CSP}(O)$ iff for every renaming $\iota : [N] \to [N]$, $\iota \circ \Gamma$ is a solution of $\mathrm{CSP}(O)$.*

The previous lemma can be used to prune the search space for a solution: We can assign numbers to state variables, provided different numbers are used for different states.

Definition 3 (Obviously different). *S_u and $S_{u'}$ are said to be* obviously different *iff there is some $v \in \Sigma^*$ such that O disagrees on uv and $u'v$. Otherwise, we say that S_u and $S_{u'}$* look similar.

A CSP problem with M obviously different variables needs at least M different states, which gives us together with Lemma 1:

Lemma 3 (Lower bound). *Let M be the number of obviously different variables. Then $\mathrm{CSP}(O)$ is not solvable over all $[N]$ with $N < M$.*

Note that solvability over $[M]$ is not guaranteed, as can easily be seen.

As a solution to the constraints system produces an automaton and in view of Lemma 2, we can fix the values of obviously different variables.

Lemma 4 (Fix different values). *Let $S_{u_1}, \ldots, S_{u_M}$ be M obviously different variables. Then $\mathrm{CSP}(O)$ is solvable iff $\mathrm{CSP}(O) \cup \{S_{u_i} = i \mid i \in [M]\}$ is solvable.*

The simple observation stated in the previous lemma improves the solution of a corresponding SAT problem defined below significantly, as described in Section 5.

Given a table $T : (U \cup U\Sigma) \times V \to \{+, -, ?\}$, we can easily approximate obviously different states: For $u, u' \in (U \cup U\Sigma)$. States S_u and $S_{u'}$ are obviously different, if the rows $T(u)$ and $T(u')$ are obviously different.

3.4 Translation of CSP to SAT

We would like to efficiently solve the CSP problem presented above. Such a solution is proposed in [OS01]. We follow, for reasons of simplicity, a different yet efficient approach. In order to solve the CSP problem, we translate it to an equivalent propositional-logic satisfiability problem in conjunctive normal form (CNF). Therefore, we need to represent the constraints formulated above in terms of equalities and inequalities as well as the possible assignments to values from $[N]$ in CNF form. More specifically, we have to encode in CNF constraints of the following form.

1. $S_u \in [N]$
2. $S_u \neq S_{u'}$
3. $S_u = S_{u'} \implies S_{ua} = S_{u'a}$
4. $S_u = i$ for some $i \in [N]$.

Namely, every constraint should be a conjunction of disjunctions of literals, where every literal is either a proposition or its negation. We propose two different encodings: *binary* and *unary*. While the first is more compact for representing large numbers, it turns out that the unary encoding speeds-up solving the resulting SAT problem.

Binary encoding We show how to encode the constraints by using binary encoding for numbers. In order to encode the restriction that a variable S_u takes a value in $[N]$ (case 1), we encode the value of S_u by m propositional variables $S_u^1 \ldots S_u^m$, where $m := \lceil \log_2 N \rceil$. Intuitively, the assignment to $S_u^1 \ldots S_u^m$ is the binary encoding of $S_u - 1$. Thus, we allocate m propositional variables for every string in the domain of O. Furthermore, we limit the range to exactly N, involving up-to $(\log N)^2$ clauses, unless the value of S_u is fixed (case 4).

In order to encode the restriction that $S_u \neq S_{u'}$ (case 2) we do the following. We have $S_u \neq S_{u'}$ iff there is a distinguishing bit in their binary representation. Thus, $S_u \neq S_{u'}$ iff $\bigvee_{k \in \{1,\ldots,m\}} S_u^k \neq S_{u'}^k$ is satisfiable, which reads in CNF as

$$\begin{array}{rl}
\varphi = & (\ S_u^1 \vee S_{u'}^1 \vee \quad \ldots \quad \vee S_u^m \vee S_{u'}^m\ ) \wedge \\
& (\ S_u^1 \vee S_{u'}^1 \vee \quad \ldots \quad \vee \neg S_u^m \vee \neg S_{u'}^m\ ) \wedge \\
& (\ S_u^1 \vee S_{u'}^1 \vee \ldots \vee \neg S_u^{m-1} \vee \neg S_{u'}^{m-1} \vee S_u^m \vee S_{u'}^m\ ) \wedge \\
& (\ S_u^1 \vee S_{u'}^1 \vee \ldots \vee \neg S_u^{m-1} \vee \neg S_{u'}^{m-1} \vee \neg S_u^m \vee \neg S_{u'}^m\ ) \wedge \\
& \qquad\qquad \vdots \\
& (\ \neg S_u^1 \vee \neg S_{u'}^1 \vee \quad \ldots \quad \vee \neg S_u^m \vee \neg S_{u'}^m\ )
\end{array}$$

Thus, each inequality is encoded by 2^m clauses. Recall that m is logarithmic in the number of states of the prospective automaton. It follows that the number of clauses is linear with respect to the number of states of the target automaton.

In order to encode the restriction that $S_u = S_{u'} \rightarrow S_{ua} = S_{u'a}$ (case 3) we do the following. Clearly, $S_u = S_{u'} \rightarrow S_{ua} = S_{u'a}$ is equivalent to $(S_u \neq S_{u'} \vee S_{ua} = S_{u'a})$. We encode this restriction in CNF using the same scheme as for cases 1 and 2, except that we add clauses for $S_u = S_{u'}$. We obtain clauses of the form

$$\begin{array}{l}
(\ \varphi \vee \ S_{ua}^1 \ \vee \neg S_{u'a}^1\ ) \wedge \\
(\ \varphi \vee \neg S_{ua}^1 \vee \ S_{u'a}^1\ ) \wedge \\
\qquad \vdots \\
(\ \varphi \vee \ S_{ua}^m \ \vee \neg S_{u'a}^m\ ) \wedge \\
(\ \varphi \vee \neg S_{ua}^m \vee \ S_{u'a}^m\ )
\end{array}$$

where φ is as defined above. This can be easily translated to CNF. Thus, every such constraint yields $2^{m+1}m$ CNF clauses.

The restriction that $S_u = i$ (case 4) is encoded by requiring that the corresponding bits of the binary representation of i are set or unset. Thus, we get m clauses for every such constraint.

Let n be the number of strings in $\mathcal{D}(O)$ and N be the size of the automaton in question. Then $\mathrm{CSP}(O)$ has $\mathcal{O}(n^2)$ constraints. Thus, the binary SAT encoding yields $\mathcal{O}(n^2 N \log N)$ clauses over $\mathcal{O}(n \log N)$ variables.

Unary encoding Surely, we can translate the CSP problem to an equivalent SAT problem using a unary encoding for values. While in general, a unary encoding uses exponentially more propositional variables, we obtain a similar number of clauses, since the constraints can be encoded using less clauses. Furthermore, for the problem sizes in question this exponential blow-up seems to be admissible. In fact, it turns out, that the employed SAT solver performs much better with the unary encoding than with the binary encoding.

In order to encode the restriction that a variable S_u takes a value in $[N]$ (case 1), we allocate N propositional variables $S_u^1, \ldots, S_u^N$ and require that $S_u^j = 1$ implies $\bigwedge_{k \neq j} S_u^k = 0$. Hence, N^2 clauses are used for all these constraints.

In order to encode the restriction that $S_u \neq S_{u'}$ (case 2) we do the following. We have $S_u \neq S_{u'}$ iff $(\neg S^1_u \vee \neg S^1_{u'}) \wedge (\neg S^2_u \vee \neg S^2_{u'}) \wedge \cdots \wedge (\neg S^N_u \vee \neg S^N_{u'})$. Thus, we need N clauses for each inequality.

The restriction $S_u = S_{u'} \rightarrow S_{ua} = S_{u'a}$ (case 3) is encoded by clauses of the following form.

$$
\begin{array}{c}
(\ \neg S^1_u \vee \neg S^1_{u'} \vee S^1_{ua} \vee \neg S^1_{u'a}\ ) \wedge \\
\vdots \\
(\ \neg S^1_u \vee \neg S^1_{u'} \vee S^N_{ua} \vee \neg S^N_{u'a}\ ) \wedge \\
(\ \neg S^2_u \vee \neg S^2_{u'} \vee S^1_{ua} \vee \neg S^1_{u'a}\ ) \wedge \\
\vdots \\
(\ \neg S^2_u \vee \neg S^2_{u'} \vee S^N_{ua} \vee \neg S^N_{u'a}\ ) \wedge \\
\vdots \\
(\ \neg S^N_u \vee \neg S^N_{u'} \vee S^N_{ua} \vee \neg S^N_{u'a}\ )
\end{array}
$$

Thus, we require N^2 CNF clauses for each equation. Finally, the restriction $S_u = i$ (case 4) is trivial to represent.

Let n be the number of strings in $\mathcal{D}(O)$ and N the size of the target automaton. Totally, the unary encoding has $\mathcal{O}(n^2N^2)$ clauses with $\mathcal{O}(nN)$ variables.

4 Inference of Network Invariants

We now describe how to compute a proper network invariant in the case that one exists. For the rest of this section, we fix systems S, P, and a property automaton φ.

We start with an informal explanation. We are using an unbounded number of *students* whose job it is to suggest possible invariants, one *teaching assistant* (TA) whose job is to answer queries by the students, and one *supervisor* whose job is to control the search process for a proper invariant. The search starts by the supervisor instructing one student to look for a proper invariant.

Like in Angluin's algorithm, every active student maintains a table (using $+$, $-$, and ?) and makes it weakly closed and weakly consistent by asking the TA membership queries. The TA answers with either $+$, $-$, or ?, as described below. When the table is weakly closed and consistent, the student translates the table to a sample O and this to a CSP problem. He solves the CSP problem using the SAT encoding. The solution with minimum range is used to form an automaton I that is proposed to the supervisor. The supervisor now checks whether I is indeed a proper invariant by checking (P), (I1), and (I2). If yes, the supervisor has found a proper invariant. If not, one of the following holds.

1. There is a string w such that $w \in \mathcal{L}(S \parallel I)$ but $w \notin \mathcal{L}(\varphi)$,
2. There is a string w such that $w \in \mathcal{L}(P)$ but $w \notin \mathcal{L}(I)$,
3. There a string w such that $w \in \mathcal{L}(P \parallel I)$ but $w \notin \mathcal{L}(I)$.

In the first case, the projection $pr^I_{S\parallel I}(w)$ should be removed from I. In the second case, the string w should be added to I. In these cases, the supervisor returns the appropriate string with the appropriate acceptance information to the student, who continues in the same manner as before.

In the last case, it is not clear, whether w should be added to I or removed from $P \parallel I$. For the latter, we have to remove the projection $pr^I_{P\parallel I}(w)$ from I. Unless w is listed negatively or $pr^I_{P\parallel I}(w)$ is listed positively in the table, both possibilities are meaningful. Therefore, the supervisor has to follow both tracks. She copies the table of the current student, acquires another student, and asks the current student to continue with w in I and the new student to continue with $pr^I_{P\parallel I}(w)$ not in I.

In order to give answers, the teaching assistant uses the same methods as the supervisor, however, whenever a choice is possible she just says ?.

Choices can sometimes yield conflicts that are observed later in the procedure. For example, w might be added to I and the new automaton I' proposed by the student together with S does not satisfy φ with w' as counter example. It is then possible that $pr^{I'}_{S\parallel I'}(w') = w$ requesting to set w's entry to $-$. Such a case reveals a conflicting assumption and requires the student to retire. If no working student is left, no proper invariant exists.

Clearly, the procedure sketched above finds a proper invariant if one exists. However, it consumes a lot of resources and may yield a proper invariant that is not minimal. We show how to adapt the supervisor so that it uses only one student at a given time and it stops with a minimal proper invariant. Intuitively, the supervisor keeps track of the active students as well as the sizes of recently suggested automata. Whenever a student proposes a new automaton of size N, the supervisor computes the appropriate answer, which is either a change of the student's table or the answer *proper invariant found*. However, she postpones answering the student (or stopping the algorithm), gives the student priority N, and puts the student on hold. Then the supervisor takes a student that is on hold with minimal priority and sends the pre-computed instrumentation to the corresponding student. In case the student's instrumentation was tagged *proper invariant found* the procedure stops by printing the final proper invariant. Note that students always propose automata of at least the same size as before since the learning algorithm returns a *minimal* automaton conforming to the sample. Thus, whenever a proper invariant is found, it is guaranteed that the proper invariant is eventually reported by the algorithm, unless a smaller proper invariant is found before.

To be a little more precise, consider the pseudo code for the supervisor given in Algorithm 1. The supervisor maintains a working set (set of students on hold) that contains triplets of the form $(n, table, automaton)$. Such a triplet consists of a table, a lower bound on the minimal-size of an automaton consistent with the table, and if the table yields a proper invariant a pointer to an automaton.

In line 2, the working set is initialized with the following triplet: size 1, empty table, no invariant. Then, we enter a loop in which a triplet with minimal number of states n is taken out of the working set. If the pointer to the automaton exists, then we have a proper invariant. Since the number of states is minimal, it is indeed a minimal proper invariant—and the algorithm terminates. If not, we ask the student to make the table closed and consistent and to propose a new automaton (based on SAT solving) and also list its number of states (line 8). The supervisor continuous by checking whether the proposed automaton is indeed an invariant. If a counter example is obtained by checking (P) or (I1), this counter example is added to the table, the automaton pointer is set to NULL (proposed automaton is not proper invariant), and the triplet is added

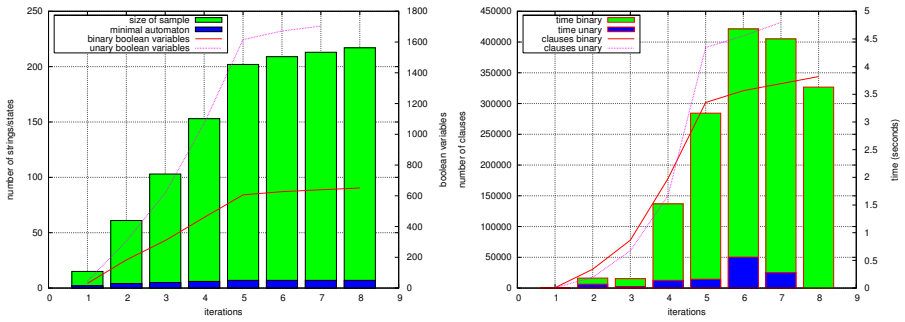
Algorithm 1 Pseudo code for the supervisor

```
1  Function supervisor ()
2    wset = { (1, getEmptyTable(),NULL) };
3    do
4      (wset, (n, table, automaton)) = takeOutWithMin_n(wset);
5      if (automaton ≠ NULL) then
6        print (automaton); stop (); // a previously found invariant is proved minimal
7      print ("Considered all automata up−to size ", n−1);
8      (n, table, automaton) = student ( table );
9      cex = checkP(automaton);
10     if cex ≠ NULL then
11       wset = wset ∪ addition( (n, table, NULL), (pr(cex), −));
12       continue_while;
13     cex = checkI1(automaton);
14     if cex ≠ NULL then
15       wset = wset ∪ addition( (n, table, NULL), (cex, +));
16       continue_while;
17     cex = checkI2(automaton);
18     if cex = NULL then // we have an invariant, store it
19       wset = wset ∪ (n, table, automaton);
20     else
21       wset = wset ∪ addition( (n, table, NULL), (pr(cex), −));
22                   ∪ addition( (n, table, NULL), (cex, +));
23   while wset ≠ ∅;
24   print ("No invariant exists ");
```

to the working set (lines 11,15). If the information that should be added to the table conflicts with the information already stored there, the `addition` function just returns the empty set, stopping further treatment of this triplet. If a counter example is found for case (I2), the supervisor tries to add both possibilities, possibly enlarging the working set (lines 21–22). If no counter example is obtained, we have a proper invariant and store it in the working set (line 19). Unless no smaller invariant is found, it is printed later in line 6. If we reach line 24, all possible invariants have been ruled out and no proper invariant exists. Overall, the procedure guarantees that at most one student is working and that the final proper invariant is indeed minimal.

Theorem 2. *For systems S, P and a safety property φ in $\mathfrak{D}$, the procedure outlined above computes minimal proper network invariant for (S, P, φ), if one exists.*

Proof (sketch). Clearly, whenever an automaton is printed, it is a proper invariant. Minimality follows from the way the supervisor searches for invariants as well as from the property that invariants proposed by the student are minimal. It remains to show that indeed one proper invariant is found, if it exists. The only way to fail this property would be to stay forever in the while loop, without examining new possible proper invariants. This is only possible if the triplets in the working set do not increase in the number of states n. However, the combined learning algorithm proposes always a new automaton



(a) sample and automaton

(b) unary versus binary

Fig. 1. Finding a proper invariant based on SAT solving

whenever a new string based on a counter example is added to the table. As there are only finitely many automata of a fixed size, we obtain the desired contradiction.

5 Experimental results

To validate our approach in practice, we have engineered the approach described in the previous sections for the setting of *transition structures*, which have been used extensively in the context of network invariants [KP00].

We have implemented the procedure of finding a proper invariant, if one exists, based on the verification tool TLV [PS96], which we employ for checking the abstraction relation and checking the safety property ((P), (I1), (I2)) and on the SAT solver ZCHAFF [MMZ$^+$01] used for computing prospective proper invariants. Using TLV and ZCHAFF, the remaining effort was to come-up with code implementing the completion of tables, and gluing the three tools together.

We used the well-studied example of a simple mutual exclusion protocol taken from [KP00]. We were interested in a proper invariant showing that the safety property *no two processes are in the critical section simultaneously* holds.

In first experiments we skipped the step of making the table complete and consistent, thus using only Biermann's algorithm. In total, we examined 78 possible invariants before finding the minimal one with 7 states after approximately 20 minutes, solving 78 SAT problems using the binary encoding.

To reduce the number of SAT instances, we have experimented with rounding off the information in the table before applying Biermann's algorithm. Interestingly, this idea alone fails. Extending the table yields less but larger SAT problems. With the binary encoding, one of the SAT problems alone took about 30 minutes. Using the unary encoding, solving the SAT instances turns out to be much faster though the overall approach is still not satisfactory. Only by combining the optimization of fixing values for obviously different variables we convert the approach to a working method.

In conclusion, it is the combination of Angluin's and Biermann's algorithms, reduced to SAT solving based on unary encoding and fixing the variables of obviously different variables that yields best results. Figure 1 reports the values of this combination for our example. We needed up-to 217 entries in a sample yielding minimal automata of up-to 7 states (Figure 1(a)). Figure 1(b) shows the speed-up using the unary encoding for this setting. Intuitively, although the unary encoding might be slightly bigger than the binary one, the information encoded in the SAT problem is less "packed" allowing a SAT solver to perform more optimizations.

Overall, we needed seven iterations taking roughly two seconds to come up with the proper invariant, which promises also successful results for a setting with larger proper invariants.

6 Conclusion

In this paper, we presented a procedure searching for a proper network invariant based on learning techniques. To this end, we developed a learning procedure combining ideas of Angluin's L^* and Biermann's inference algorithm. Moreover, we have shown that the resulting learning algorithm allows an efficient implementation via a reduction to SAT and using existing SAT solvers. The search for a proper invariant terminates with an invariant with a minimal number of states, provided one exists, and might otherwise not terminate. Since the studied problem is undecidable, this cannot be avoided.

While we have experienced that learning techniques do not scale easily to large systems [BJLS03], our approach should be understood as an alternative to finding invariants manually. For this, it has to be competitive for systems of sizes that could alternatively be handled by hand.

On the same line, it is important to note that our search procedure iteratively considers larger and larger prospective proper invariants. This implies that it can be used to decide the question whether a proper network invariant with up-to n states exists, for every fixed natural number n. Practically, it means that the algorithm is able to continuously report on its progress, i.e., the size of the current prospective invariant. Thus, even if no proper invariant is found after a while, a user can learn that no invariant with size up-to the one currently studied exists.

It would be interesting to combine learning methods for ω-automata in the search for network invariants. This would allow us to handle also more complex properties of the parameterized system in question.

References

[AK86] Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 22(6):307–309, 1986.

[AMN05] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *17th CAV*, LNCS 3576, pages 548–562. 2005.

[Ang87] Dana Angluin. Learning regaular sets from queries and counterexamples. *IC*, 75:87–106, 1987.

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behaviour. *IEEE TOC*, 21:592–597, 1972.

[BJLS03] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin's learning. TR 2003-039, Uppsala University, 2003.

[CCST05] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarrantee reasoning for simulation conformance. In 17^{th} *CAV*, LNCS 3576, pages 534–547. 2005.

[CTTV04] E. M. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In 15^{th} *Concur*, LNCS 3170, pages 276–291. 2004.

[CTV06] E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In 7^{th} *VMCAI*, LNCS 3855, 126–141. 2006.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In 17^{th} *CADE*, LNCS 1831, pages 236–254, 2000.

[EN95] E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *POPL*, 1995.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *IC*, 37(3):302–320, 1978.

[HV05] Peter Habermehl and Tomás Vojnar. Regular model checking using inference of regular languages. *ENTCS*, 138(3):21–36, 2005.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In 6^{th} *TACAS*, LNCS 1785, 2000.

[KM95] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *IC*, 117(1):1–11, 1995.

[KP00] Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *STTT*, 2(4):328–342, 2000.

[KPP05] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *IC*, 200(1):35–61, 2005.

[KPSZ02] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In 13^{th} *Concur*, LNCS 2421, 2002.

[Lan92] Kevin J. Lang. Random dfa's can be approximately learned from sparse uniform examples. In *COLT*, pages 45–52, 1992.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In 24^{th} *POPL*, 1997.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, 530–535. ACM, 2001.

[OG92] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, *Series in Machine Perception and AI* 1, pages 49–61. World Scientific, 1992.

[OS01] Arlindo L. Oliveira and João P. Marques Silva. Efficient algorithms for the inference of minimum size dfas. *Machine Learning*, 44(1/2):93–119, 2001.

[PO98] Jorge M. Pena and Arlindo L. Oliveira. A new algorithm for the reduction of incompletely specified finite state machines. In *ICCAD*, 482–489, 1998.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *8th CAV*, pages 184–195, 1996.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In 12^{th} *CAV*, LNCS 1855, pages 328–343. Springer Verlag, 2000.

[VSVA04a] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for fifo automata. In *FSTTCS*, LNCS 3328, 494–505. 2004.

[VSVA04b] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *ICFEM*, LNCS 3308, 274–289. 2004.

[VSVA05] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In *TACAS*, LNCS 3440, 2005.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, LNCS 407, 1989.