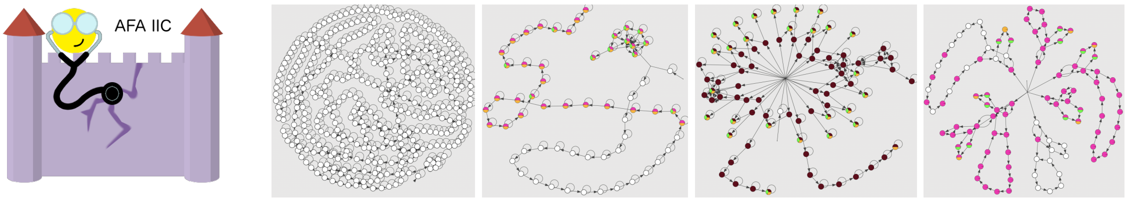
BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Incremental Inductive Coverability for Alternating Finite Automata

Pavol Vargovčík*



Abstract

In this paper we propose a specialization of the inductive incremental coverability algorithm that solves alternating finite automata emptiness problem. We analyse and experiment with various design decisions, add heuristics to guide the algorithm towards the solution as fast as possible. Even though the problem itself is proved to be PSPACE-complete, we are focusing on making the decision of emptiness computationally feasible for some practical classes of applications. So far, we have obtained some interesting preliminary results in comparison with antichain-based algorithms.

Keywords: alternating finite automaton — emptiness — incremental inductive coverability — well-structured transition system — verification — safety

Supplementary Material: *N/A*

*xvargo01@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Finite automata are one of the core concepts of computer science. Alternation in automata theory has already been studied for a long time [1] and many practical classes of problems (WSTS or LTL formulae satsfiability and many more) can be efficiently reduced in polynomial time to the problem of alternating automata emptiness. We are particularly motivated by the applications of alternating automata in software verification and in string analysis [2], where they can be used to detect ways to break their safety or, if no ways are detected, to formally prove that the program is safe.

Alternating finite automaton (AFA) is a deterministic finite automaton that is extended by the concept existential transitions and universal transitions. Disjunction is implemented in constant time already with non-deterministic finite automata, using existential transitions. By introducing the universal transition it is easy to combine automata in constant time with conjunction. Negation is done in linear time simply by replacing existential transitions with universal ones and vice versa, and by swapping final and non-final states. Although these operations are efficient, checking of alternating automata emptiness (i.e. checking whether a given automaton accepts the empty language) is unfortunately PSPACE-complete [3], which is considered computationally infeasible. We however believe that it is possible to design algorithms able to avoid the high worst-case complexity in practical cases.

Simple state space explorations using antichains [4] to subsume states and to reduce the number of states that is needed to be explored, are currently considered as one of the best existing methods to check the emptiness. On the other hand, a popular model-checking algorithm IC3 has been adapted for well-structured transition systems. The adapted algorithm is named IIC [5] (*incremental inductive converability*) and in contrast to IC3, it benefits from subsumption.

The main contribution of our research is design and implementation of a new algorithm—we show that the alternating finite automata are *well structured transition systems* and subsequently we specialize the IIC algorithm to solve their emptiness. The IIC algorithm implicitly uses subsumption on states, and counter-example guided abstraction by under-approximating the reachable state space. It is progressing in incremental steps until convergence, or until a valid counter-example is found. We compare efficiency of the IIC algorithm to backward and forward antichain-based state space explorations.

We have found some artificial classes of AFA where IIC significantly outperforms forward antichain-based exploration algorithm and one class where IIC outperforms both forward and backward one. We have compared the three algorithms also on real-world benchmarks extracted from the string program verification. For most of the cases, maximal and final cardinality of state space representation (number of blockers for IIC, number of antichain elements) were very similar. As for time, IIC was converging more slowly for bigger benchmarks because altering the sets of blockers for IIC is quite expensive. For benchmarks with non-empty automata, IIC was sometimes much better than antichain in all metrics, or antichain was much better than IIC, but it was probably only by chance—the winning approach has just luckily got faster on the right way to bad states; these measurement results were not reproducible. Most of the benchmarks were very similar and the other ones were more complex and the measurements were often timing out on them. The implementation was written in the *Python* language and we are planning to make a more efficient one, to get more interesting results. We also need to determine properties of the input AFA that are significant for IIC performance, add optimizations and heuristics to operations and decisions where they would be effective, and suppress them in situations where they just waste the computing power.

2. Preliminaries

Downward and upward closure Let $\preceq\, \subseteq U \times U$ be a preorder. *Downward closure* $X\!\downarrow$ of a set $X \in U$ is a set of all elements lesser than some element from X, formally: $X\!\downarrow = \{y \mid y \in U \;\wedge\; \exists x \in X.\, y \preceq x\}$. Analogous for *upward closure*: $X\!\uparrow = \{y \mid y \in U \wedge \exists x \in X.\, x \preceq y\}$. We define downward and upward closure on a single element as $x\!\downarrow = \{x\}\!\downarrow$ and $x\!\uparrow = \{x\}\!\uparrow$. *Downward-closed* and *upward-closed* sets are those that already contain all the lesser elements from U, or greater respectively. We will designate the fact that a set is downward-closed or upward-closed by the corresponding arrow in the upper index. Intuitively it holds that $X^{\downarrow} = X^{\downarrow}\!\downarrow$ and $Y^{\uparrow} = Y^{\uparrow}\!\uparrow$. It is known that the set of downward-closed sets are closed under union and intersection, same for the set of upward-closed sets. Furthermore, if we complement a downward-closed set, we get an upward-closed one, similarly for the opposite. A system of an universum and a preorder $(U, \preceq)$ is downward-finite if every set $X \subseteq U$ has a finite downward closure.

Well-quasi-order A preorder $\preceq\, \subseteq U \times U$ is a *well-quasi-order*, if each infinite sequence of elements $x_0, x_1, \cdots$ from U contains an increasing pair $x_i \preceq x_j$ for some $i < j$.

Well-structured transition system (WSTS) Let us fix the notation of a well-structured transition system to the quadruple $S = (\Sigma, I, \rightarrow, \preceq)$, where

- Σ is a set of states.
- $I \subseteq \Sigma$ is a set of initial states.
- $\rightarrow\, \subseteq \Sigma \times \Sigma$ is a *transition* relation, with a reflexive and transitive closure $\rightarrow^*$. We say that s' is reachable from s if $s \rightarrow^* s'$.
- $\preceq\, \subseteq \Sigma \times \Sigma$ is a relation. We will call it *subsumption* relation, and if $a \preceq b$, we will say that b subsumes a. relation.

A system S is a WSTS iff the subsumption relation is a well-quasi-order and the *monotonicity* property holds:

$$\forall s_1 \forall s_1'.\, s_1 \rightarrow^* s_1' \implies \forall s_2.\, (s_1 \preceq s_2 \implies \exists s_2'.\, (s_2 \preceq s_2' \wedge s_2 \rightarrow^* s_2')) \quad (1)$$

The functions $pre : \Sigma \longrightarrow 2^{\Sigma}$ and $post : \Sigma \longrightarrow 2^{\Sigma}$ are defined the following way: $pre(s') = \{s \in \Sigma \mid s \rightarrow s'\}$, similarly $post(s) = \{s' \in \Sigma \mid s \rightarrow s'\}$.

Covering We say that a downward-closed set of states $P^{\downarrow}$ *covers* a WSTS S iff the set of states that are reachable from initial states of S is included in $P^{\downarrow}$.

$$Covers(P^{\downarrow}, S) \overset{\text{def}}{\Leftrightarrow} \forall s \in I.\, \nexists s' \notin P^{\downarrow}.\, s \rightarrow^* s' \quad (2)$$

We will use the term *bad states* for the complement $\Sigma \setminus P^{\downarrow}$.

Alternating finite automaton (AFA) Let us fix the notation of an alternating finite automaton to the quintuple $M = (Q, \Sigma_M, I_M, \delta, F)$, where

- Q is a finite set of states. A subset of q is called *case*, cases will be denoted as ρ or ϱ.

- Σ_M is a finite set of symbols — an input alphabet.
- $I_M \subseteq Q$ is an initial set of states (also called *initial case*).
- $\delta : Q \times \Sigma_M \longrightarrow 2^{2^Q}$ is a transition function.
- $F : \mathbb{F}_Q^-$ is a negative boolean formula determining final cases. A case ρ is *final* if $F \wedge \bigwedge_{q \notin \rho} \neg q$ is satisfiable.

Let $w = \sigma_1 \ldots \sigma_m, m \geq 0$ be a sequence of symbols $\sigma_i \in \Sigma_M$ for every $i \leq m$. A *run* of the AFA M over w is a sequence $\rho = \rho_0 \sigma_1 \rho_1 \ldots \sigma_m \rho_m$ where $\rho_i \subseteq Q$ for every $0 \leq i \leq m$, and $\rho_{i-1} \underset{\sigma_i}{\rightarrow} \rho_i$ for every $0 < i \leq m$, where $\underset{\sigma}{\rightarrow} \subseteq 2^Q \times 2^Q$ is a *transition relation by symbol* $\sigma \in \Sigma_M$ defined the following way:

$$\rho_1 \underset{\sigma}{\rightarrow} \rho_2 \overset{\text{def}}{\Leftrightarrow}$$
$$\exists \xi \in Z.\ \rho_2 = \bigcup \{\varrho \in 2^Q \mid \exists q \in Q.\ (q, \varrho) \in \xi\} \quad (3)$$

where

$$Z = \{\xi \subseteq \Xi \mid \forall q \in \rho_1\ \exists!\ \varrho \in 2^Q.\ (q, \varrho) \in \xi\} \quad (4)$$

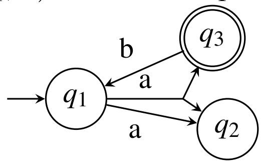
$$\Xi = \{(q, \varrho) \in \rho_1 \times 2^Q \mid \varrho \in \delta(q, \sigma)\}. \quad (5)$$

The AFA *transition* relation $\rightarrow_M \subseteq 2^Q \times 2^Q$ is a transition relation by an arbitrary symbol:

$$\rho_1 \rightarrow_M \rho_2 \overset{\text{def}}{\Leftrightarrow} \exists \sigma \in \Sigma_M.\ \rho_1 \underset{\sigma}{\rightarrow} \rho_2 \quad (6)$$

Let us define few properties of a run. It is *terminating* iff $rho_m \models F.\rho_m \subseteq f$, *commencing* iff $I_M \subseteq \rho_0$, *accepting* iff it is terminating and commencing. An AFA M is *empty* if none of the runs over M is accepting. This *emptiness* property is denoted as $\mathrm{Empty}(M)$.

AFA can be visualized similarly to NFA as a directed graph, universal transitions are visualized as a forking arrow. More detail is described in the appendix, here we provide an example of visualization for $Q = \{q_1, q_2, q_3\}$, $\Sigma_M = \{a, b\}$, $I_M = \{q_1\}$, $F = \neg q_1 \wedge \neg q_2$, $\delta(q_1, a) = \{\{q_2\}, \{q_2, q_3\}\}$, $\delta(q_3, b) = \{\{q_1\}\}$ and $\delta(q, \sigma) = \emptyset$ for other q and a.



3. IIC for AFA emptiness

We have instantiated the general IIC algorithm from [5] for deciding the emptiness problem of alternating finite automata. Some decisions about the reduction were straightforward, some of them were inspired by the Petri net coverability instance and some of the decisions were done by us. We will explain our reasoning and show experimental results for multiple possible implementations of some parts of IIC.

3.1 General IIC state

The IIC algorithm decides whether a downward-closed set $P^{\downarrow}$ covers all reachable states of a well-structured transition system.

State of the IIC algorithm consists of a vector R of downward-closed sets of states $R_0^{\downarrow} R_1^{\downarrow} \ldots R_N^{\downarrow}$ and a queue Q of counter-example candidates. We write $R|Q$ to represent the algorithm state. Set $R_i^{\downarrow}$ is an over-approximated set of states that are reachable in i steps of WSTS. N is the currently analysed step of the system. Queue Q is a set of (a, i) pairs, where a is an upward closed set of states from which we are sure that a bad state can be reached in i steps. We write *min* Q to denote a pair with minimal i. In addition there is a special initial state Init and two terminating states: Unsafe means that we proved that a state out of $P^{\downarrow}$ is reachable, Safe is a proof that $P^{\downarrow}$ contains all reachable states.

The state of the algorithm is modified by application of transition rules, until the Safe or Unsafe state is reached.

For practical reasons we postpone the description of the particular transition rules. Prior to that, we will start with convertion of the algorithm state for solving the AFA emptiness problem. Then we will introduce all the general transition rules along with their instantiation and notes about actual implementation. The algorithm is proved for soundness and if the WSTS is downward-finite, it is guaranteed to terminate [5], what holds for AFA due to finiteness of the state space.

3.2 State of the IIC for AFA

We convert the emptiness problem of an AFA $M = (Q, \Sigma_M, I_M, \delta, F)$ to the coverability problem of a downward-finite WSTS $S = (\Sigma, I, \rightarrow, \preceq)$ in a way that:

- $\Sigma := C$ — states of WSTS are the cases of AFA.
- $I := \{I_M\}$ — initial states of WSTS are singleton with the initial case of AFA.
- $\rightarrow := \rightarrow_M$ — transition function is the step function of AFA.
- $\preceq := \supseteq$ — well-quasi-ordering relation is the superset relation, therefore, downward closure $X\downarrow$ is the set of all supersets of X and upward closure $X\uparrow$ is the set of all subsets of X respectively.
- $P^{\downarrow} := \Sigma \setminus F\uparrow$ — bad states are all subsets of F (to avoid the confusion we will call them bad

cases).

The following two theorems are proved in the appendix.

Theorem 1 *The system created the way stated above is a valid WSTS.*

Theorem 2 *The generated WSTS S is covered by $P^\downarrow$ iff the AFA M is empty.*

Similarly to the Petri net coverability instance of IIC described in [5], we represent the steps $R_1^\downarrow, \cdots, R_N^\downarrow$ as so-called *stages* $B_1, \cdots, B_N$, which are themselves sets of blockers β. Blocker β is a case about which we are sure that it is not reachable in i or less steps. The algorithm holds the invariant $R_i^\downarrow \subseteq R_{i+1}^\downarrow$, so if a blocker exists in stage i, its effect applies to the steps $0 \cdots i$. We assure that if $i < j$ then $\nexists b \in B_i .\ b\!\uparrow \cap B_j$. Equivalence of two successive stages is then easily checked by $B_k = B_{k+1}$. Blockers are upward-closed, i.e. even no subset of a blocker is reachable. At any point of the IIC algorithm, a stage B_i represents an under-approximation of cases that are not reachable in i or less steps of the AFA. Set $R_i^\downarrow = \Sigma \setminus \bigcup B_i\!\uparrow$ is then an over-approximation of the reachable cases. The exception is R_0, which is represented directly by the initial case I_M. We will thus always handle the zero step in a special way.

The queue of counter-example candidates Q is implemented as stack. Since current implementation performs depth-first search, the peak of the stack is always *min* Q.

3.3 Forward and backward transitions

In this section we introduce the forward and backward transition functions, which describe the internal representation of the transition function of AFA. We benefit from the backward transition function $\delta_\sigma^\leftarrow$ when computing predecessors of case for the transition rules **Decide/Conflict**. The forward transition function $\delta_\sigma^\rightarrow$ is useful in computing Generalization.

Without the loss of generality we assume that the states of the input automaton are integers in range $1, \cdots, |Q|$. Same for the input alphabet symbols, they are integers in range $1, \cdots, |\Sigma_M|$. For simplicity of explanation will assume that a case of the AFA is represented as a set of states, however there is also an implementation with bit vectors that is more efficient for small state spaces. Initial states are represented as a vector of cases.

Finally we have a vector of relations $\delta_1, \cdots, \delta_{|\Sigma_M|}$, that are themselves vectors, containing for every state $q = 1, \cdots, |Q|$ a set of all cases $q' \in 2^{|Q|}$ such that $q \delta_\sigma q'$. Formally we may consider this forward $\delta_\sigma^\rightarrow$ as a function of type $Q \longrightarrow 2^{2^{|Q|}}$, because it maps states (integers) to sets of successor cases.

In IIC we have to do also backward transitions, so in preprocessing we convert this representation to the opposite one: δ_σ is a mapping from cases that can be successors q' of any state q to all their predecessors. Backward $\delta_\sigma^\leftarrow$ has type $2^{|Q|} \longrightarrow 2^{|Q|}$: it maps cases to sets of predecessor states.

3.4 Transition rules

The most of the talk will be devoted to the transition rules of the IIC. The rules of the algorithm for general WSTS are presented in the figure 1 and are of the form

$$\frac{C_1 \cdots C_k}{\sigma \mapsto \sigma'} \tag{7}$$

We can apply a rule if the algorithm is in the state σ and conditions $C_1 \cdots C_k$ are met, σ' is then a new state. We will introduce the rules, specialize them for our instance of the IIC and explain their purpose.

3.4.1 Initialize

In contrast with the general IIC, we start the algorithm with an empty vector of stages because zero step that contains downward closure of initial cases is handled in a special way (we do not store it in our stage vector). The candidate queue is initialized to an empty stack.

3.4.2 Valid

This rule checks for convergence of IIC. If any two consecutive steps are equal (the first of the two equal stages has no blockers), we have proved the emptiness of AFA.

3.4.3 Unfold

If the candidate queue is empty (we have proved that no candidate is reachable from any initial case) and we have not yet converged, we start to explore new step: we start with the over-approximating assumption that in the new step we can reach all cases (the stage has no blockers).

3.4.4 Candidate

If we have empty candidate queue and the last step $R_N^\downarrow$ is intersecting bad cases, add one of the bad cases from the intersection into the queue. If the last applied rule was **Unfold**, all the bad states are elements of $R_N^\downarrow$. We can therefore choose F as the new candidate. Then, as the candidates are upward-closed (all the bad cases are included in the F candidate), if F is eventually blocked, we are sure that $R_N^\downarrow$ is not overlapping bad cases anymore. The implementation of this rule is then very simple: we apply it right after the **Unfold** rule and we just add the case F into the candidate queue.

3.4.5 ModelSem

If a counter-example candidate includes some initial case, we know that a bad case can be reached from the initial case. The counter-example is thus valid and the AFA is not empty.

3.4.6 Decide/Conflict

These two rules are tightly connected together and are the main part of the IIC. The rule **Decide** performs a backward transition *pre* from the counter-example candidate. If the rule *fails to transition back* from a candidate (a, i) (i.e. no predecessor α, for which $a \npreceq \alpha$, is in the previous step), we know that the candidate is spurious and the **Conflict** rule is applied. The **Conflict** rule removes the candidate and refines the step i by adding a new blocker β to the stage B_i that is a *generalization* (see below) of the candidate a.

We see that a predecessor can be blocked not only by a blocker form the previous step but also by the candidate. If it is a subsumption of the candidate, we know that its predecessor will be again only a subset of itself and we thus never reach any initial case.

If we fail to transition back (the **Conflict** rule is being applied), we add a new blocker to the stage i and it will affect all the steps $k = 1, \dots, i$. The blocker could be the candidate a itself but we can often cheaply find something better than a by so-called generalization $Gen_{i-1}(a)$. Generalization is an arbitrary function that efficiently finds some β subsumed by the blocked candidate a, such that β shares some common properties with a: the predecessors of β are blocked in the step $i-1$ and β does not contain any of the initial cases[1]. The two properties ensure that β is a valid blocker in the step i.

We implement these rules in the following way. For each symbol σ in the AFA alphabet Σ_M we iterate through the AFA's backward transition function $\delta_\sigma^{\leftarrow}$ (see 3.3). We accumulate predecessors of cases ρ that are subsets of the candidate a.

$$\alpha_\sigma = \{q \in Q \mid \rho \subseteq a \wedge q \in \delta_\sigma^{\leftarrow}(\rho)\} \tag{8}$$

We try to find a blocker $\beta_\sigma \in \bigcup_{k=i}^{N} B_k$ that includes α_σ. If such a blocker β_σ does not exist, then α_σ is a non-blocked predecessor in the step $i-1$ and we can enqueue it as a candidate for $i-1$ (**Decide**, $\alpha := \alpha_\sigma$). Before enqueueing, we check it with the **ModelSem** rule to ensure that the candidate does not include the initial case.

We represent the zero step in a special way (as a set of initial cases instead of a set of blockers). Thus, if $i = 1$, we only check if α_σ includes some of the initial cases. If so, we have found a valid predecessor in the zero step and by the **ModelSyn** rule we know that the counter-example is finished (then the IIC ends up in the Unsafe state). For the Generalization purposes we need to find the blockers β_σ — cases that are not reachable in the zero step (do not include I) and do include α_σ. For each σ, we find some q such that $q \in I \wedge q \notin \alpha_\sigma$ and then $\beta_\sigma = Q \setminus q$; .

The candidate is spurious if it is blocked for all symbols of the alphabet (**Conflict**). We compute a generalization β (described soon) of the candidate and add it as a blocker for step i. As the new blocker applies to all steps $j = 1, \cdots i$, we remove all blockers that subsume β from those stages.

Generalization Generalization is an important component of the rules **Conflict** and **Induction**. It is the most vaguely defined part of the IIC and a big part of our contribution is specialization of generalization for AFA.

We apply generalization $Gen_i(a)$ to cheaply create a blocker b that is subsumed by a. The blocker b is then going to be added to the step $i+1$. The new blocker b should hold some properties (that are held by a). It should not subsume any initial state and all of its predecessors should be blocked in the step i.

$$Gen_i(a) := \{b \mid b \preceq a \wedge b{\Uparrow} \cap I = \emptyset \wedge pre(b{\Uparrow}) \cap R_i^{\downarrow} \setminus b{\Uparrow} = \emptyset\} \tag{9}$$

We break the condition for valid generalizations into these three restrictions:

$$b \preceq a \tag{10a}$$

$$b{\Uparrow} \cap I = \emptyset \tag{10b}$$

$$pre(b{\Uparrow}) \cap R_i^{\downarrow} \setminus b{\Uparrow} = \emptyset \tag{10c}$$

First of all we create a set of forbidden WSTS states (AFA cases) C that are not allowed to subsume the blocker b (to be included in it). Otherwise the condition 10b or 10c would be violated. We try to find a blocker b that blocks (is subsumed by) many WSTS states (AFA cases) but is not subsumed by any of the forbidden ones. So we apply approximative *greedy algorithm* for solving *minimum hitting set* problem [6] (which is dual to the *set cover* problem) to find a set of AFA states D that intersects each case from C. If D does not intersect b, then b is guaranteed not to include any of the forbidden WSTS states from C. The new blocker b is therefore obtained as a complement of D: $b = Q \setminus D$.

If some case from C intersected a, the approximative minimum hitting set D could contain a state from

[1] We are sure that a does not contain any initial case because we check every added candidate by the **ModelSem** rule

Valid
$$\frac{\exists i < N.\ R_i^{\downarrow} = R_{i+1}^{\downarrow}}{R|Q \mapsto \mathsf{Safe}}$$

ModelSem
$$\frac{\mathit{min}\ Q = (a,i) \quad I \cap a{\uparrow} \neq \emptyset}{R|Q \mapsto \mathsf{Unsafe}}$$

Decide
$$\frac{\mathit{min}\ Q = (a,i) \quad i > 0 \quad \alpha \in \mathit{pre}(a{\uparrow}) \cap R_{i-1}^{\downarrow} \setminus a{\uparrow}}{R|Q \mapsto R|Q.\textsc{Push}((\alpha, i-1))}$$

Initialize
$$\frac{}{\mathsf{Init} \mapsto [I{\downarrow}]|\emptyset}$$

Unfold
$$\frac{R_N^{\downarrow} \subseteq P^{\downarrow}}{R|\emptyset \mapsto R \cdot \Sigma|\emptyset}$$

Conflict
$$\frac{\mathit{min}\ Q = (a,i) \quad i > 0 \quad \mathit{pre}(a{\uparrow}) \cap R_{i-1}^{\downarrow} \setminus a{\uparrow} = \emptyset \quad \beta \in \mathit{Gen}_{i-1}(a)}{R|Q \mapsto R[R_k^{\downarrow} \leftarrow R_k^{\downarrow} \setminus \beta{\uparrow}]_{k=1}^{i}|Q.\textsc{PopMin}}$$

Candidate
$$\frac{a \in R_N^{\downarrow} \setminus P^{\downarrow}}{R|\emptyset \mapsto R|[(a,N)]}$$

ModelSyn
$$\frac{\mathit{min}\ Q = (a,0)}{R|Q \mapsto \mathsf{Unsafe}}$$

Induction
$$\frac{R_i^{\downarrow} = \Sigma \setminus \{r_{i,1}, \ldots, r_{i,m}\}{\uparrow} \quad b \in \mathit{Gen}_i(r_{i,j}) \text{ for some } 1 \leq j \leq m}{R|\emptyset \mapsto R[R_k^{\downarrow} \leftarrow R_k^{\downarrow} \setminus b{\uparrow}]_{k=1}^{i+1}|\emptyset}$$

Figure 1. IIC transition rules

a. Then b would not be a superset of a what breaks 10a. To ensure 10a we will obtain C from an intermediate set C_0 (that ensures 10b and 10c and is defined subsequently) following way: $C = \{c \setminus a \mid c \in C_0\}$.

To ensure the restriction 10b, the set C_0 simply contains all initial cases. To break the restriction 10c, there must exist a symbol σ, for which a predecessor of some subset of b is not blocked in the step i. As a is a valid blocker in the step $i+1$, for each σ, all of its σ-predecessors are blocked in i by some blocker β_σ (we know β_σ from **Decide/Conflict**). The blocker b is not blocked by β_σ in the step i iff a state $q \notin \beta_\sigma$ exists, for which $\exists c.\ q\delta_\sigma c \wedge c \subseteq a$. As a conclusion of these intuitions, C_0 also contains the successors of all states out of β_σ. We get these successors by using the forward transition function (see 3.3).

$$C_0 = I \cup \{\rho \mid \sigma \in \Sigma_M \ \wedge\ q \in Q \setminus \beta_\sigma \ \wedge\ \rho \in \delta_\sigma^{\rightarrow}(q)\} \tag{11}$$

3.4.7 ModelSyn

A counter-example candidate can obtain zero step index only if it was created by the **Decide** rule, which ensures inclusion in the zero step. If a counter-example is included in the zero step, it means it is a superset of some initial case s (no blockers can be added to the zero step by any rule). As the candidate is upward closed, it represents also the initial case s. All the candidates lead to bad states, and thus the counter-example is valid and the AFA is not empty.

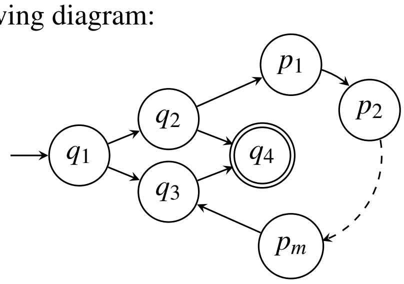
3.4.8 Induction

The **Induction** transition rule serves for pushing blockers forward from steps i to $i+1$. We can see that luckily the general definition of the rule represents the step i with the concept of blockers, same as our implementation. We can therefore directly translate it: If there is a blocker $\beta \in B_i$ such that a generalization for it exists from the step i, we can push it to the step $i+1$. It needs some more explanation — generalization of β from the step i exists iff all of its predecessors are blocked in the step i and therefore it is a valid blocker for $i+1$.

Induction is implemented like this: for every step i we iterate through all $b_{ij} \in B_i$ and check if there is for every symbol of alphabet $\sigma \in \Sigma_M$, in the same step i, a blocker β_σ that includes the predecessors of β. If so, we generalize the β (we use the corresponding β_σ to compute the generalization) and add the result, in the same way as in the rule **Decide**, into the step $i+1$.

4. Experimental Evaluation

IIC has much better results than antichains on a class of AFA $\mathit{Primes}(n)$. Let us define a class of AFA $\mathit{Branch}(m)$, with a single-symbol alphabet, by the following diagram:



With given n, let $\pi = 2,3,5,7,11,\cdots$ be a sequence of first n prime numbers. The AFA $\mathit{Primes}(n)$ is then conjunction of automata $\mathit{Branch}(\pi_1),\cdots,\mathit{Branch}(\pi_n)$, where q_4 of one (e.g. random) of the branches is not a final state. Since one of the branches lacks a final state, the automaton $\mathit{Primes}(n)$ is empty. The biggest instance of $\mathit{Primes}(n)$ where forward antichain solves this problem in a reasonable time (15 seconds) is for $n = 4$. For $n = 5$ it is more than two minutes. Backward antichain is better: the maximal n where it does not time out is $n = 6$, with 41 seconds. IIC converges in reasonable time for bigger n, e.g. for $n = 15$ it is still 40 seconds.

Evidently, this particular class of AFA can be easily detected in preprocessing, but it is interesting that

IIC can efficiently solve it implicitly. It may indicate that IIC can efficiently decide emptiness for some other, more interesting classes of AFA.

We have experimented also with practical benchmarks extracted from [2]. The antichain-based algorithms performed objectively better. A possible reason for this is that vast majority of simple benchmarks was very similar—conjunction of one simple long chain of states and one very branch that is a bit more interesting but very small. These benchmarks were very easy for antichains and as IIC is much more complex, it could not break through with its power. There are also benchmarks that are more interesting but too complex and if the automata are empty, none of the three approaches converges. If an interesting automaton is not empty, IIC or antichains sometimes discover the bad state "by chance", other time they time out. As all the three algorithms are implemented in *Python*, the poor performance on the interesting examples can be still significantly ameliorated by rewriting the code to a more performing language.

Among 179 benchmarks, we considered 58 of them interesting, i.e. antichain did not solve them in all five runs (3 backward and 2 forward runs) in less than 2 seconds. 50 of the interesting benchmarks timed out (with 2 minute timeout) for both antichain and IIC, from which 44 were non-empty (we know it from the results of experiment in [2]) and 6 were with unknown result (the solver in [2] timed out). From the other 8 benchmarks, 2 were solved significantly[2] better by IIC concerning the final and maximal number of blockers (versus the final and maximal number of antichain elements). They were both non-empty. One of them had also significantly better time. From the same 8 benchmarks that did not time out, 5 were significantly better solved by backward antichain (which was always better than the forward one) concerning time, 3 of them were significantly better concerning the final and maximal number of antichain elements. One of them was empty, but after visual analysis we have found out that the benchmark is similar to the non-interesting ones, just much bigger.

The experiments were performed on a machine with Intel Core i7 processor with two cores and 16GB of RAM.

5. Conclusions

We have specialized IIC for AFA emptiness problem, implemented it and compared the implementation to antichain-based algorithms on artificial and practical benchmarks. We were able to find an artificial class of AFA where IIC performed much better, e.g. for the instance *Primes*(6) the IIC terminated in 7 seconds, backward antichain in 41 seconds, and forward antichain timed out (terminated in more than two minutes). For bigger instances of *Primes* the difference was increasing exponentially. Research still needs to be done to find more artificial classes, to determine the properties of AFA which affect the efficiency of IIC, and also practical benchmarks that dispose of these properties. We have shown that IIC has some potential for the AFA problems and shared our experience and thoughts concerning design and implementation decisions as well as experimentation.

[2] at least two times; applies to the rest of the text

Acknowledgements

I would like to thank my supervisor Mgr. Lukáš Holík, Ph.D. for his help.

References

[1] A. K. Chandra and L. J. Stockmeyer. Alternation. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 98–108, 1976.

[2] Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. String constraints with concatenation and transducers solved efficiently. In *Proceedings of the ACM on Programming Languages*, volume 2 of *Article 4*. POPL, 2018.

[3] Petr Jančar and Zdeněk Sawa. A note on emptiness for alternating finite automata with a one-letter alphabet. *Information Processing Letters*, 104(5):164 – 167, 2007.

[4] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–174, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 158–173, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Vašek Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

Appendix A - Visualization

Forking graph For visualisation purposes we define a graph with forking edges, as a tuple $G = (V, L, E)$, where V is a finite set of nodes, L is a set of labels and $E \subseteq V \times L \times 2^V$ is a set of forking edges. Each node $x \in V$ of a graph is visualized as a circle labelled with v. Each edge $(n, l, W) \in E$ is visualized as a point p, a line labelled with l connecting x and p, and for each node $m \in W$, an arrow that leads from p to m. If there are multiple edges that differ only in labels, they can be all visualized as a single edge with all the labels separated by comma. As an example, we present a graph $G = (\{a,b,c\}, \{l_1, l_2\}, \{(a, l_1, \{b,c\}), (a, l_2, \{b,c\})\})$ in figure 2.

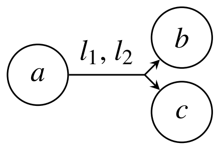


Figure 2. Forking graph

Visualization of AFA AFA will be visualised as a forking graph $G = (V, \Sigma_M, E)$, where $V = Q$ and $E = \{(q, \sigma, \rho) \in Q \times \Sigma_M \times 2^Q \mid \rho \in \delta_M(q, \sigma)\}$. The initial case is visualized as a hanging forking edge leading to I_M. The final cases are visualized only if F is of form $F = \bigwedge_{q \in Q \setminus Q_F} \neg q$, where $Q_F \subseteq Q$ is a set of final states. Then the nodes in Q_F are demarked with double borders.

As an example, we show a visualization of an automaton in figure 3.

$$Q = \{q_1, q_2, q_3\}$$
$$\Sigma_M = \{a, b\}$$
$$I_M = \{q_1\}$$
$$F = \neg q_1 \wedge \neg q_2$$
$$\delta_M(q, \sigma) =$$
$$\begin{cases} \{\{q_2\}, \{q_2, q_3\}\} & \text{for } q = q_1 \wedge \sigma = a \\ \{\{q_1\}\} & \text{for } q = q_3 \wedge \sigma = b \\ \emptyset & \text{otherwise} \end{cases}$$

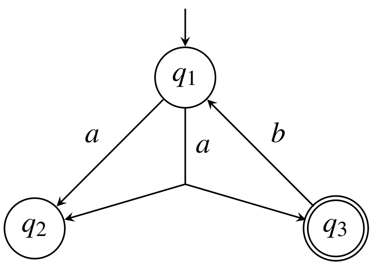


Figure 3. Visualization of AFA

Appendix B - Correctness of reduction

Lemma 2.1 *Monotonicity of $\preceq$ relative to $\rightarrow$ is satisfied.*

Proof 2.1 *If $\rho_1 \rightarrow \rho_1'$ then there exists a σ and a q_i' for every $q_i \in c$ that satisfies $q\delta_\sigma q'$ and ρ_1' is the union of those q_i'. We know that $\rho_2 \subseteq \rho_1$, so we obtain it by removing some of the q_i states. We then get ρ_2' by removing the corresponding q_i' cases from the union operands. Clearly $\rho_2' \subseteq \rho_1'$, as we obtained it by unifying smaller set of cases.*

Now we have to prove soundness of the reduction: the generated WSTS S is coverable by $P^{\downarrow}$ iff the AFA M is empty.

Proof 1 *The monotonicity property is satisfied and as $\preceq$ is a preorder on a finite domain, no infinite sequence that is purely increasing can exist, therefore $\preceq$ is a well-quasi-order.*