BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

INCREMENTAL INDUCTIVE COVERABILITY FOR ALTERNATING FINITE AUTOMATA
INKREMENTÁLNÍ INDUKTIVNÍ POKRYTELNOST PRO ALTERNUJÍCÍ KONEČNÉ AUTOMATY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR PAVOL VARGOVČÍK
AUTOR PRÁCE

SUPERVISOR Mgr. LUKÁŠ HOLÍK, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2018

Zadání diplomové práce/19099/2017/xvargo01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Vargovčík Pavol, Bc.**
Obor: Matematické metody v informačních technologiích
Téma: **Inkrementální induktivní pokrytelnost pro alternující konečné automaty**
Incremental Inductive Coverability for Alternating Finite Automata
Kategorie: Teoretická informatika

Pokyny:

1. Seznamte se s problematikou testování prázdnosti jazyka alternujících automatů a nastudujte algoritmus IIC (Incremental Inductive Coverability) pro dobře strukturované systémy.
2. Navrhněte variaci algoritmu IIC pro konečné alternující automaty.
3. Implementujte navržený algoritmus.
4. Porovnejte výkonnost algoritmu s existujícími efektivními algoritmy (alespoň některý z protiřetězcových algoritmů) na vhodných datech, jako jsou instance problému získané z verifikačních úloh, z rozhodování logik, případně instance náhodně generované.

Literatura:

1. Johannes Kloos, Rupak Majumdar, Filip Niksic, Ruzica Piskac:Incremental, Inductive Coverability. CAV 2013: 158-173, Springer, LNCS, vol. 8044 (2013).
2. Parosh Aziz Abdulla, Yu-Fang Chen, Lukás Holík, Richard Mayr, Tomás Vojnar:When Simulation Meets Antichains. TACAS 2010: 158-174, LNCS, vol. 6015 (2010).
3. Pierre Ganty, Nicolas Maquet, Jean-François Raskin: Fixed point guided abstraction refinement for alternating automata. Theor. Comput. Sci. 411(38-39): 3444-3459 (2010).

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Holík Lukáš, Mgr., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

In this work, we propose a specialization of the inductive incremental coverability algorithm that solves alternating finite automata emptiness problem. We experiment with various design decisions, analyze them and prove their correctness. Even though the problem itself is PSpace-complete, we are focusing on making the decision of emptiness computationally feasible for some practical classes of applications. We have obtained interesting comparative results against state-of-the-art algorithms, especially in comparison with antichain-based algorithms.

Abstrakt

V tejto práci navrhujeme špecializáciu algoritmu *inductive incremental coverability*, ktorá rieši problém prázdnosti alternujúcich konečných automatov. Experimentujeme s rôznymi návrhovými rozhodnutiami, analyzujeme ich a dokazujeme ich korektnosť. Aj keď je známe, že problém je sám o sebe PSpace-ťažký, zameriavame sa na to, aby bolo rozhodovanie prázdnosti výpočetne prijateľné v niektorých triedach automatov s praktickým využitím. Dosiahli sme niekoľko zaujímavýcch výsledkov v porovnaní so špičkovými algoritmami, predovšetkým v porovnaní s algoritmami založenými na protireťazcoch.

Keywords

alternating finite automaton, incremental inductive coverability, well-structured transition system, emptiness, formal verification

Kľúčové slová

alternujúci konečný automat, inkrementálna induktívna pokryteľnosť, dobre štrukturovaný systém, prázdnosť, formálna verifikácia

Reference

VARGOVČÍK, Pavol. *Incremental Inductive Coverability for Alternating Finite Automata.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Lukáš Holík, Ph.D.

Incremental Inductive Coverability for Alternating Finite Automata

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mgr. Lukáš Holík, Ph. D. Additional information were given by Ing. Ondřej Lengál, Ph. D. and Ing. Petr Janků. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Pavol Vargovčík
May 22, 2018

Acknowledgements

I want to thank to my supervisor Lukáš Holík and Ondřej Lengál with whom I have had a lot of productive talks on this topic. I also want to thank to professor Bo-Cheng Lai who participated on the supervision of my work for two months during my internship at National Chiao Tung University in Taiwan.

Contents

1 Introduction **2**

2 Preliminaries **4**

2.1 Alternating Finite Automata 5

3 IIC for AFA Emptiness **14**

3.1 Basic AFA Representation and General AFA 14

3.2 Backward Transition Function 17

3.3 Reduction from AFA Emptiness to WSTS Covering 19

3.4 General IIC 20

3.5 IIC Implementation 24

3.6 AFA Representations 32

3.7 Backward Transition Function, Finalized 35

4 Antichain-based Algorithms **40**

5 Experimental Evaluation **43**

5.1 The Artificial Class of AFA: $Primes(n)$ 43

5.2 Real World Benchmarks 44

6 Conclusions and Future Work **52**

Bibliography **54**

A Implementation details **56**

A.1 Programming language choice 56

A.2 Input format 56

A.3 Translation of real world benchmarks 57

A.4 SAT solving 57

A.5 Visualization of algorithm state 58

Chapter 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 [5] and many practical classes of problems (WS1S or LTL formulae satisfiability [4, 10] 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, in the field of string analysis (analysis of programs with string variables).

Alternating finite automaton (AFA) is a deterministic finite automaton that is extended by the concept of existential and universal branching. Disjunction is implemented in constant time already with non-deterministic finite automata (NFA), using existential branching. By introducing the universal branching, it is easy to combine automata in constant time with conjunction. Negation can then be done in linear time too, simply by replacing existential branchings 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 [14], which is considered computationally infeasible in general. We however believe that it is possible to design algorithms able to avoid the high worst-case complexity for practical cases.

When accepting a word by an AFA, if a universal branching transition is performed, *all* of the states into which we branched, need to reach a final state by accepting the rest of the word. Comparing to NFA, a reachability graph of AFA therefore contains *cases*—sets of states—instead of single states.

The reachability graphs of alternating finite automata are monotone with respect to a **subsumption** relation. The subsumption relation indicates which parts of the reachability graphs can be discarded during the search for an accepting run. Small *cases* are more likely to reach final cases and their supersets are subsumed and can be pruned in a forward search for an accepting run. Similarly, big cases are more likely to be reachable from initial cases than their subsets. Therefore the paths via the subsets can be ignored in a backward search for an accepting run.

Algorithms using antichains have been studied in [9, 1]. These algorithms are basically simple state space explorations that benefit from the subsumption to reduce the number of nodes of the reachability graph that are needed to be explored. The antichain-based algorithms are currently considered as one of the best practical existing methods to check the AFA emptiness.

Another approach, used in software verification for simplification of the search space, is **abstraction refinement**. In this approach, the search space is oversimplified at first. The simplification then gets refined, typically by a use of counter-example analysis. The first algorithm for solving AFA emptiness with a use of abstraction refinement was an antichain-based approach that works on an abstract domain [11]. Other powerfull methods, based on abstraction, have recently appeared in the field of model checking, particularly the algorithms IC3/PDR [2, 12] (property driven reachability). In [13], the AFA have been translated to transition systems and their emptiness has been solved using IC3/PDR. The results achieved in [13] were interesting, however, the monotonicity of AFA disappears in the translation to transition systems, and is not utilised in the search. The IC3/PDR algorithms were later combined with subsumption pruning in IIC (*incremental inductive coverability*) [15], which is an adaptaion of IC3 to *well structured transitions systems*. The IIC algorithm has been applied in [15] for solving the coverability of Petri nets.

The main contribution of our work is adaptation of the IIC algoroithm to the problem of alternating automata emptiness—we show that the alternating finite automata are well structured transition systems and subsequently we specialize the IIC algorithm to solve their emptiness. The algorithm starts by over-approximating the space of cases that are reachable in one step, to set of all cases. Using counter-examples, it refines the space, by adding *blockers* to the first step. Blockers abstractly represent the over-approximation of the reachable cases—blockers are cases that are known to be unreachable in the given (first) step and subsume[1] their subsets. Bigger blockers subsume more of the search space, therefore, attempts are made to find as big blockers as efficiently possible. When the over-approximation of the cases reachable in one step is disjoint with the final cases, the algorithm progresses into the next step, over-approximating and refining it again in a similar manner (during the process of refinement of each step, also the preceding steps get usually refined more). The algorithm eventually discovers a valid counter-example (the AFA is then not empty), or detects a convergence (the AFA is then empty).

In the experimental evaluation, we compare efficiency of the IIC algorithm to backward and forward state space explorations using antichains. We have found an artificial class of AFA where IIC significantly outperforms both the forward and backward antichain algorithms. We have compared the three algorithms also on a set of real world benchmarks converted from the string program verification in [13]. Many of the AFA shared a similar simple structure that was ideal for the antichain algorithms and IIC could not outperform them. However, some of the real world benchmarks had more complex structures. They were problematic for antichains, on the bigger of them, antichain was timing out. Our implementation of IIC was consistently achieving better results on these benchmarks.

[1] The subsumption relation is a superset relation for the cases of AFA.

Chapter 2

Preliminaries

This chapter introduces mathematical structures and properties that will be used in the rest of the work. It also talks about the relationships between the structurs, e.g. conversions between various types of alternating automata.

Downward and upward closure Let $\preceq\, \subseteq U \times U$ be a preorder. *Downward closure* $X\!\downarrow$ of a set $X \subseteq U$ is a set of all elements smaller than elements from X, formally $X\!\downarrow\, = \{y \in U \mid \exists x \in X.\ y \preceq x\}$. Analogously for *upward closure*, $X\!\uparrow\, = \{y \in U \mid \exists x \in X.\ x \preceq y\}$. For any two sets $X, Y \subseteq U$, we say that X is a downward-closure or upward-closure *generator* of Y iff $X\!\downarrow\, = Y$, or $X\!\uparrow\, = Y$ respectively. We define downward and upward closure on a single element as $x\!\downarrow\, = \{x\}\!\downarrow$ and $x\!\uparrow\, = \{x\}\!\uparrow$. *Downward-closed* sets are those that equal their downward closures. Similarly, an *upward-closed* set equals its upward closures. We will denote the fact that a set is downward-closed or upward-closed by the corresponding arrow in the upper index (e.g. we may write $X^{\uparrow}$ if we know that $X = X\!\uparrow$). It holds that $X^{\downarrow} = X^{\downarrow}\!\downarrow$ and $Y^{\uparrow} = Y^{\uparrow}\!\uparrow$. It is known that the set of downward-closed sets is 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, and vice versa. 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.

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

$$\forall x_1, x_2 \in \Sigma.\ x_1 \to x_2 \implies \forall x_1' \in \Sigma.\ (x_1 \preceq x_1' \implies \exists x_2'.\ (x_2 \preceq x_2' \wedge x_1' \to x_2')) \tag{2.1}$$

The predecessor function $pre : 2^\Sigma \longrightarrow 2^\Sigma$ is defined the following way:

$$pre(X_2) = \{x_1 \in \Sigma \mid \exists x_2 \in X_2.\ x_1 \to x_2\} \tag{2.2}$$

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 $P^\downarrow$.

$$Covers(P^\downarrow, S) \overset{\text{def}}{\Leftrightarrow} \forall s \in I.\ \nexists s' \notin P^\downarrow.\ s \to^* s' \tag{2.3}$$

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

Replacement notation For a sequence $X \in \mathbb{X}^l$ of length l and a function $f : \mathbb{N} \to \mathbb{X}$, the notation $X[X_k \leftarrow f(k)]_{k=m}^n$, where $0 \leq m \leq n \leq l$, means that the elements $X_m X_{m+1} \cdots X_n$ are replaced by new elements $f(m)f(m+1)\cdots f(n)$ and the rest of X remains unchanged, formally

$$X[X_k \leftarrow f(k)]_{k=m}^n = X_0 \cdots X_{m-1} f(m) \cdots f(n) X_{n+1} \cdots X_l \tag{2.4}$$

2.1 Alternating Finite Automata

In this section we talk about various types of alternating automata, means of automata visualisation that will be used throughout the work, as well as conversions between the types of automata.

Alternating finite automaton (AFA, classical AFA) The concept of AFA has been introduced in [5]. It is formally defined in the following way. Let us fix the notation of an alternating finite automaton to the quintuple $M = (Q, \Sigma_M, I_M, \delta_M, F)$, where

- Q is a finite set of states. A subset of q is called *case*, cases will be denoted as ρ or $\boldsymbol{\varrho}$.
- 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*).
- $\delta_M : Q \times \Sigma_M \longrightarrow 2^{2^Q}$ is a transition function. If $\rho \in \delta_M(q, \sigma)$, we say that q leads to ρ by σ.
- $F : \mathbb{F}_Q^-$ is a negative boolean formula determining final cases. A case ρ is *final* iff $\rho \models F$.

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 $\rho_{i-1} \overset{\text{M}}{\underset{\sigma_i}{\to}} \rho i$ for every $0 < i \leq m$, where $\overset{\text{M}}{\underset{\sigma}{\to}} \subseteq 2^Q \times 2^Q$ is a *transition relation by the symbol* $\sigma \in \Sigma_M$ defined the following way.

The function *SuccsOfStates*(ρ_1, σ) returns for every q from ρ_1, the pairs (q, r_2), such that q leads to r_2 by σ. These r_2 are the possible choices of successors for the states q. The function *SuccsOfCase*(ρ_1, σ) then finds such subsets of *SuccsOfStates*(ρ_1, σ), which contain one choice of a successor r_2 for each state q. In the equation (2.5), we pick one of the subsets from *SuccsOfCase*(ρ_1, σ), unite the chosen successors r_2 into one case ρ_2.

$$\rho_1 \overset{\mathrm{M}}{\underset{\sigma}{\to}} \rho_2 \overset{\text{def}}{\Leftrightarrow} \exists \xi \in \mathit{SuccsOfCase}(\rho_1, \sigma).\ \rho_2 = \bigcup_{(q, r_2) \in \xi} r_2 \tag{2.5}$$

where

$$\mathit{SuccsOfCase}(\rho_1, \sigma) = \{\xi \subseteq \mathit{SuccsOfStates}(\rho_1, \sigma) \mid \forall q \in \rho_1\ \exists! r_2 \in 2^Q.\ (q, r_2) \in \xi\} \tag{2.6}$$

$$\mathit{SuccsOfStates}(\rho_1, \sigma) = \{(q, r_2) \in \rho_1 \times 2^Q \mid r_2 \in \delta_M(q, \sigma)\}. \tag{2.7}$$

If $\rho_1 \overset{\mathrm{M}}{\underset{\sigma}{\to}} \rho_2$, we say that ρ_1 is a *σ-predecessor* of ρ_2 and ρ_2 is a *σ-successor* of ρ_1. The AFA *transition* relation $\overset{\mathrm{M}}{\to} \subseteq 2^Q \times 2^Q$ is a transition relation by an arbitrary symbol:

$$\rho_1 \overset{\mathrm{M}}{\to} \rho_2 \overset{\text{def}}{\Leftrightarrow} \exists \sigma \in \Sigma_M.\ \rho_1 \overset{\mathrm{M}}{\underset{\sigma}{\to}} \rho_2 \tag{2.8}$$

If $\rho_1 \overset{\mathrm{M}}{\to} \rho_2$ then we say that ρ_1 is a *predecessor* of ρ_2 and ρ_2 is a *successor* of ρ_1. Similarly we can say that we can *transition forward* from ρ_1 to ρ_2, or *transition backward* from ρ_2 to ρ_1.

Let us define few properties of a run. A run is

- *terminating* iff $\rho_m \models F$,
- *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 will be visualised as a directed hypergraph $G = (V, \Sigma_M, E)$. Nodes V are the states Q of the AFA. For each transition $q \overset{\mathrm{M}}{\underset{\sigma}{\to}} \rho$, there is a hyperedge $(\{q\}, \sigma, \rho)$. If two hyperedges $(\{q\}, \sigma_1, \rho)$ and $(\{q\}, \sigma_2, \rho)$ differ only in the symbol σ, they are merged into one hyperedge labelled with σ_1, σ_2. The initial case is visualized as a hyperedge without label, leading to I_M: $(\emptyset, \emptyset, I_M)$. 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 2.1.

Inductive invariant The problem of proving AFA emptiness can be formulated as a problem of finding an *inductive invariant*. It is a set of cases J that is disjoint with final cases, includes all cases that are reachable from the initial case, and all the cases from J transition only into cases from J. If such an invariant is found, the AFA is empty. On the other hand, if it cannot be found because the reachable cases intersect final cases, the AFA is not empty.

In the IIC and antichain algorithms, the invariant is represented in an abstract way. In the sequel, the size of this abstract representation is called *size of inductive invariant*, and is used as a metric for comparison of the algorithms—a good invariant is the one that can be represented in a simple way, i.e. its size is small.

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

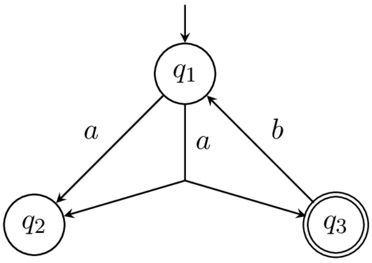


Figure 2.1: Visualisation of AFA

Symbolic AFA (sAFA) We base our definition on the concept of symbolic AFA discussed in [7]. Transitions of a symbolic AFA are tagged with symbolic formulae instead of bare symbols. Symbolic AFA is defined similarly to the classical AFA. It is a quintuple $M_s = (Q, V, I_M, \delta_s, F)$. In contrast to the classical AFA it includes

- a finite set of boolean variables V.
- a symbolic transition function $\delta_s : Q \times \mathbb{F}_V \longrightarrow 2^{2^Q}$.

Let $w = \varsigma_1 \ldots \varsigma_m, m \geq 0$ be a sequence of variable evaluations $\varsigma_i \in \mathcal{P}(V)$ for each $i \leq m$. A *run* of the sAFA M over w is a sequence $\rho = \rho_0 \varsigma_1 \rho_1 \ldots \varsigma_m \rho_m$ where $\rho_i \subseteq Q$ for each $0 \leq i \leq m$, and $\rho_{i-1} \overset{\text{M}}{\underset{\varsigma_i}{\to}} \rho_i$ for each $0 < i \leq m$, where $\overset{\text{s}}{\underset{\varsigma}{\to}} \subseteq 2^Q \times 2^Q$ is a *transition relation by the variable evaluation* $\varsigma \in \mathcal{P}(V)$. In comparison with the classical AFA, the only thing that has been changed in the definition of run is that symbols have been replaced by variable evaluations. As only the definition (2.7) depends directly on symbols, the rest of the definition of $\rho_1 \overset{\text{s}}{\underset{\varsigma}{\to}} \rho_2$ remains the same as in the definition of $\overset{\text{M}}{\underset{\sigma}{\to}}$. The definition (2.7) is replaced by the definition (2.9), which associates states q from ρ_1 with those cases r_2, to which q leads by formulae that are satisfied with the evaluation ς.

$$\mathit{SuccsOfStates}_s(\rho_1, \varsigma) = \{(q, r_2) \in \rho_1 \times 2^Q \mid \exists \phi \in \mathbb{F}_V.\ r_2 \in \delta_s(q, \phi) \wedge \varsigma \models \phi\}. \tag{2.9}$$

The properties *terminating*, *commencing*, *accepting*, as well as the *emptiness* property are defined the same way as for the classical AFA. The *transition* relation $\overset{\text{s}}{\to} \subseteq 2^Q \times 2^Q$ and visualisation are defined similarly to the classical AFA, only the symbols $\sigma \in \Sigma_M$ are changed to variable evaluations $\varsigma \in \mathcal{P}(V)$, or formulae $\phi \in \mathbb{F}_V$ respectively.

Minterms of formulae Let $\Phi = \{\phi_1, \cdots \phi_n\} \subseteq \mathbb{F}_X$ be a finite set of boolean formulae over variables X. A minterm is a minimal satisfiable Boolean combination of all formulae from Φ [8], formally

$$\begin{aligned} &\mathit{Minterms} : 2^{\mathbb{F}_X} \longrightarrow 2^{\mathbb{F}_X} \\ &\mathit{Minterms}(\Phi) = \left\{ \psi = \bigwedge_{i=1,\dots,n} \psi_i \;\middle|\; \forall i \in \{1, \cdots, n\}(\psi_i \in \{\phi_i, \neg\phi_i\}) \wedge \mathrm{SAT}(\psi) \right\} \end{aligned} \tag{2.10}$$

Number of minterms is exponential in the worst case [8]. An effective algorithm to extract the minterms from Φ is presented in [8]. Let $\equiv_\Phi \in \mathcal{P}(X)^2$ be an equivalence relation on evaluations of variables X.

$$\forall \varsigma_1, \varsigma_2 \in \mathcal{P}(X).\ \varsigma_1 \equiv_\Phi \varsigma_2 \overset{\text{def}}{\Leftrightarrow} \{\phi \in \Phi \mid \varsigma_1 \models \phi\} = \{\phi \in \Phi \mid \varsigma_2 \models \phi\} \tag{2.11}$$

Theorem 1. *For any finite set of boolean formulae $\Phi \subseteq \mathbb{F}_X$, the set $\mathit{Minterms}(\Phi)$ is isomorphic to the set of equivalence classes of $\equiv_\Phi$.*

sketch. A minterm ψ is a conjunction of all formulae $\phi \in \Phi$ in a positive (ϕ) or a negative form ($\neg\phi$). A change of a value of any $\phi \in \Phi$ for two $\varsigma_1, \varsigma_2 \in \mathcal{P}(X)$ would apparently result in $\varsigma_2 \not\models \psi$ if $\varsigma_1 \models \psi$. Furthermore, no pair of distinct minterms implies the same set of formulae from Φ. As the set of minterms covers all satisfiable Boolean combinations of the formulae from Φ, for each evaluation ς exists such a minterm ψ that $\varsigma \models \psi$. □

An example of minterm generation is provided in Figure 2.2.

$$X = \{a, b\}$$

$$\Phi \in \mathbb{F}_X = \{a, a \wedge b, a \vee a, \neg a\}$$

Minterm alias	$\mathit{Minterms}(\Phi)$	Simplified $\mathit{Minterms}(\Phi)$	Variable evaluations
ψ_1	$a \wedge (a \wedge b) \wedge (a \vee a) \wedge \neg(\neg a)$	$a \wedge b$	$\{a, b\}$
ψ_2	$a \wedge \neg(a \wedge b) \wedge (a \vee a) \wedge \neg(\neg a)$	$a \wedge \neg b$	$\{a, \neg b\}$
ψ_3	$\neg a \wedge \neg(a \wedge b) \wedge \neg(a \vee a) \wedge \neg a$	$\neg a$	$\{\neg a, b\}, \{\neg a, \neg b\}$

Figure 2.2: Example of *Minterms*(Φ)

Tagged minterms of formulae Let T be a set of tags. We can tag each formula in Φ with a tag $t \in T$. The minterm generation algorithm then associates a minterm ψ with the tags of all formulae that are in the positive form in ψ. Let $X = \{(\phi_1, t_1), \cdots, (\phi_n, t_n)\} \subseteq \mathbb{F}_V \times T$.

$$\begin{aligned}
&\mathit{TaggedMinterms}(X) \subseteq \mathbb{F}_V \times 2^T\\
&\mathit{TaggedMinterms}(X) = \left\{ (\psi, \tau) = (\bigwedge_{i=1,\ldots,n} \phi_i, \bigcup_{i=1,\ldots,n} \tau_i) \;\middle|\; \right.\\
&\qquad \forall i \in \{1, \cdots n\}. ((\phi_i, \tau_i) \in \{(\phi_i, \{t_i\}), (\neg\phi_i, \emptyset)\}) \wedge\\
&\qquad \left. SAT(\psi) \right\}
\end{aligned}$$

Minterms of sets Formula is just a symbolic representation of a set of variable evaluations. Therefore we can define the concept of minterms also for subsets of a finite universum U instead of formulae. Let $W = \{w_1, \cdots, w_n\} \subseteq 2^U$. A minterm is a minimal non-empty Boolean combination of all sets from W, formally

$$\begin{aligned}
&\mathit{Minterms} : 2^U \longrightarrow 2^U\\
&\mathit{Minterms}(W) = \left\{ u = \bigcap_{i=1,\ldots,n} u_i \;\middle|\; \forall i \in \{1, \cdots, n\} (u_i \in \{w_i, U \setminus w_i\}) \wedge u \neq \emptyset \right\}
\end{aligned} \tag{2.12}$$

The equivalence relation on the elements of U is defined also in a similar way:

$$\forall x_1, x_2 \in U.\ x_1 \equiv_W x_2 \overset{\text{def}}{\Leftrightarrow} \{w \in W \mid x_1 \in w\} = \{w \in W \mid x_2 \in w\} \tag{2.13}$$

The reasoning about the isomorphism of minterms to equivalence classes of $\equiv_W$ and the algorithm for computing minterms of sets is analogous (replacement of conjunction by intersection, negation by complement, satisfiability by comparison to empty set).

Tagged minterms for sets are defined also analogously. Let $X = \{(w_1, t_1), \cdots, (w_n, t_n)\} \subseteq 2^U \times T$.

$$\begin{aligned}
&\mathit{TaggedMinterms}(X) \subseteq 2^U \times 2^T\\
&\mathit{TaggedMinterms}(X) = \left\{ (u, \tau) = (\bigcap_{i=1,\ldots,n} u_i, \bigcup_{i=1,\ldots,n} \tau_i) \;\middle|\; \right.\\
&\qquad \forall i \in \{1, \cdots n\}. ((u_i, \tau_i) \in \{(w_i, \{t_i\}), (U \setminus w_i, \emptyset)\}) \wedge\\
&\qquad \left. u \neq \emptyset \right\}
\end{aligned}$$

Reductions between AFA and sAFA Conversion from a classical to a symbolic AFA is done by setting $V = \Sigma_M$ and

$$\forall q \in Q\ \forall \sigma \in \Sigma_M.\ \delta_s(q, \sigma \wedge \bigwedge_{v \in V \setminus \{\sigma\}} \neg v) := \delta_M(q, \sigma). \tag{2.14}$$

As the formula in the symbolic transition function encodes a singleton set of evaluation with $\sigma = 1$ and all other variables equaling to zero, it is apparent that this conversion does not break the emptiness property.

Reduction that preserves emptiness is possible also the other way. Recall the functions (2.7) and (2.9) that compute successors of states for classical and symbolic AFA, which is the only part of $\rightarrow$ (and $\rightarrow_s$) that depends directly on symbols (variable evaluations):

$$\mathit{SuccsOfStates}(\rho_1, \sigma) = \{(q, \boldsymbol{\varrho}) \in \rho_1 \times 2^Q \mid \boldsymbol{\varrho} \in \delta_M(q, \sigma)\}.$$

$$\mathit{SuccsOfStates}_s(\rho_1, \varsigma) = \{(q, \boldsymbol{\varrho}) \in \rho_1 \times 2^Q \mid \exists \phi \in \mathbb{F}_V.\ \boldsymbol{\varrho} \in \delta_s(q, \phi) \land \varsigma \models \phi\}.$$

Let Φ be a set of all formulae that occur on the transitions of an sAFA M_s. Note that we can replace $\mathbb{F}_V$ by Φ because for all other formulae $\phi \in \mathbb{F}_V \setminus \Phi$, the transition function $\delta_s(q, \phi)$ returns an empty set.

We need to choose such an alphabet Σ_M and such a transition function δ_M, for which $\mathit{SuccsOfStates}(\rho_1, \sigma)$ produces same sets of successors as $\mathit{SuccsOfStates}_s(\rho_1, \varsigma)$ for fixed ρ_1:

$$\begin{aligned} &\forall \rho_1 \subseteq Q\ \forall s \in q \times 2^Q. \\ &\quad \exists \varsigma \in \mathcal{P}(V).(s = \mathit{SuccsOfStates}_s(\rho_1, \varsigma)) \iff \exists \sigma \in \Sigma_M.(s = \mathit{SuccsOfStates}(\rho_1, \sigma)) \end{aligned} \tag{2.15}$$

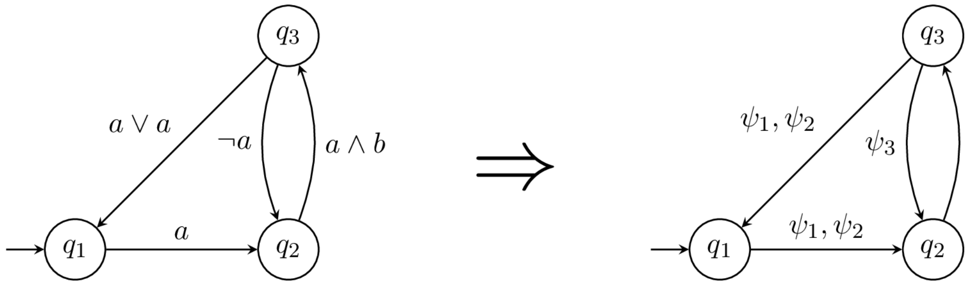
A simple solution would be to set $\Sigma_M = \mathcal{P}(V)$ and

$$\begin{aligned} &\forall q \in Q.\ \forall \varsigma \in \Sigma_M. \\ &\quad \exists \phi \in \Phi.(\boldsymbol{\varrho} \in \delta_s(q, \phi) \land \varsigma \models \phi) \iff \boldsymbol{\varrho} \in \delta_M(q, \varsigma) \end{aligned} \tag{2.16}$$

We however want to make the alphabet Σ_M as small as possible. As we know that minterms represent equivalence classes (by $\equiv_\Phi$) of $\mathcal{P}(V)$, where none of $\phi \in \Phi$ changes its value (theorem 1), we can set $\Sigma_M = \mathit{Minterms}(\Phi)$ and then

$$\begin{aligned} &\forall q \in Q.\ \forall \sigma \in \Sigma_M. \\ &\quad \exists \phi \in \Phi.(\boldsymbol{\varrho} \in \delta_s(q, \phi) \land \sigma \rightarrow \phi) \iff \boldsymbol{\varrho} \in \delta_M(q, \sigma) \end{aligned} \tag{2.17}$$

An example of conversion from sAFA to AFA is shown in Figure 2.3.



The minterm generation of ψ_1, ψ_2, ψ_3 for this example is shown in the figure 2.2.

Figure 2.3: Emptiness-preserving conversion from sAFA to AFA.

Succinct AFA (suAFA) In succinct AFA, successors of a state are defined by a positive boolean formula over states $\mathbb{F}_Q^+$ instead of a set of successor cases. Initial states are also defined by a positive boolean formula. A succinct afa $M_{\text{su}} = (Q, V, I_{\text{su}}, \delta_{\text{su}}, F)$ differs from classical AFA in the following fields:

- $I_{\text{su}} \in \mathbb{F}_Q^+$ is an initial state formula.
- $\delta_{\text{su}} : Q \times \Sigma_M \longrightarrow \mathbb{F}_Q^+$ is a succinct transition function.

$$\rho_1 \overset{\text{su}}{\underset{\sigma}{\rightarrow}} \rho_2 \overset{\text{def}}{\Leftrightarrow} \rho_2 \models \bigwedge_{q \in \rho_1} \delta_{\text{su}}(q, \sigma) \tag{2.18}$$

A succinct automaton can be visualized as a graph that contains state nodes, as well as nodes for operators in the succinct transition function, see Figure 2.4.

$$\begin{aligned} Q &= \{q_1, q_2, q_3\} \\ \Sigma_M &= \{a\} \\ I_M &= \{q_1\} \\ F &= \neg q_1 \wedge \neg q_2 \\ \delta_{\text{su}}(q, a) &= \begin{cases} q_2 \wedge (q_1 \vee q_3) & \text{for } q = q_1 \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

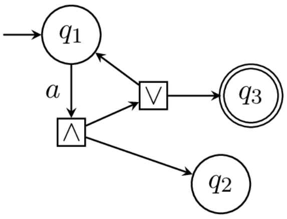


Figure 2.4: Visualisation of suAFA

Conversion from suAFA to AFA In the following text we describe a conversion algorithm rather informally and without a proof of preserving emptiness property. The idea of the conversion is however quite simple — replacing conjunctions by universal branchings and disjunctions by existential ones, inserting new states where needed while transitions from these new states consume no symbol of the original alphabet: a new symbol ε is created for these transitions and added to the alphabet of the classical AFA.

We may consider any positive boolean formula as a formula in a positive DNF, literals of which are states or positive boolean formulae. We want the positive DNF to be as big as possible, so if we have disjunctions deep in the formula but no conjunction is above them, e.g. $f = (f_1 \vee f_2) \vee f_3$, we pull the disjunction out to the top level of the formula: $f = \bigvee\{f_1, f_2, f_3\}$. We do the same operation also for the conjunctions in the DNF.

Example 1. When converting the positive boolean formula

$$f = q_1 \vee (q_2 \wedge (q_3 \wedge (q_4 \vee q_5))) \vee q_6$$

to the positive DNF, there is a disjunction $q_4 \vee q_5$ that cannot be pulled out because it is an argument of a conjunction. We consider $q_4 \vee q_5$ as a literal ϕ of the converted formula. All the other operators can be flattened. The following describes the result f' of the conversion.

$$\phi = q_4 \vee q_5$$

$$f' = \bigvee \left\{ \bigwedge\{q_1\}, \bigwedge\{q_2, q_3, \phi\}, \bigwedge\{q_6\} \right\}$$

For each $\sigma \in \Sigma_M, q \in Q$, we convert the formula $f = \delta_{\text{su}}(q, \sigma)$ to the positive DNF in the way shown in the example. If the literals of the positive DNF f' are only states, then the conjunctions in the DNF simply represent cases of $\delta_M(q, \sigma)$. If a conjunction contains a literal that is a formula ϕ, we create a new state q_ϕ and replace ϕ by q_ϕ in the disjunction. The literals of the positive DNF are now only states and we can compute $\delta_M(q, \sigma)$, by considering the conjunctions as cases. Then we recursively compute the transitions of all q_ϕ by the symbol ε: We convert the formula ϕ to a positive DNF, replace the literals that are formulae by new states, etc.

Example 2. This example shows overall conversion from a suAFA transition to AFA transitions. Let δ_{su} contain the transition

$$f = \delta_{\text{su}}(q_1, \sigma) = q_1 \vee (q_2 \wedge (q_3 \vee q_1) \wedge (q_4 \vee q_3))$$

First, we convert f to a positive DNF form, f'

$$f' = \bigvee \left\{ \bigwedge \{q_1\}, \bigwedge \{q_2, \phi_1, \phi_2\} \right\}$$

where

$$\phi_1 = q_3 \vee q_1$$
$$\phi_2 = q_4 \vee q_3$$

Now we create new states $Q := Q_{\text{su}} \cup \{q_{\phi_1}, q_{\phi_2}\}$ and replace the subformulae ϕ_1 and ϕ_2 with the new states in f'

$$f' = \bigvee \left\{ \bigwedge \{q_1\}, \bigwedge \{q_2, q_{\phi_1}, q_{\phi_2}\} \right\}$$

We create a transition for δ_M from the DNF f':

$$\delta_M(q_1, \sigma) := \{\{q_1\}, \{q_2, q_{\phi_1}, q_{\phi_2}\}\}$$

Then we recursively continue to compute the transitions $\delta_M(q_{\phi_1}, \varepsilon)$ and $\delta_M(q_{\phi_2}, \varepsilon)$.

$$\phi_1' = \bigvee \left\{ \bigwedge \{q_3\}, \bigwedge \{q_1\} \right\}$$
$$\delta_M(q_{\phi_1}, \varepsilon) := \{\{q_3\}, \{q_1\}\}$$
$$\phi_2' = \bigvee \left\{ \bigwedge \{q_4\}, \bigwedge \{q_3\} \right\}$$
$$\delta_M(q_{\phi_2}, \varepsilon) := \{\{q_4\}, \{q_3\}\}$$

In Figure 2.5, we have obtained an AFA with the structure and semantics resembling the original suAFA (for now, ignore the dashed transitions, they will be discussed subsequently):

Now, let us justify the presence of the self-loops by ε at the original states. From the case $\rho_1 = \{q_2, q_{\phi_1}, q_{\phi_2}\}\}$, we should be able to transition forward $\rho_2 = \{q_2, q_3, q_4\}$. However, we cannot because there is no transtion from b by ε and from the other two states there are only transitions by ε. Generally, from a case that contains original states of suAFA as well as states generated by the conversion, we must be able to transition forward by ε only from the generated states while staying in the original ones. For this reason we make reflexive transitions by ε (dashed ones) at each of the original states.

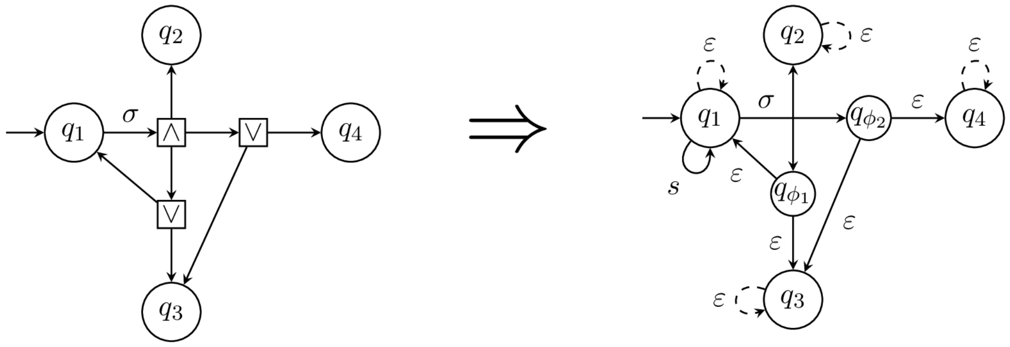


Figure 2.5: Conversion from suAFA to AFA

Succinct symbolic automata ssuAFA As the succinctness of automata defined in 2.1 has generally nothing to do with the input alphabet of the AFA, we can easily define the succinct symbolic automaton and its conversion to a symbolic automaton. The ssuAFA is a quintuple $M_{\text{ssu}} = (Q, V, I_{\text{ssu}}, \delta_{\text{ssu}}, F)$, where

- $\delta_{\text{ssu}} : Q \times \mathbb{F}_V \longrightarrow \mathbb{F}_Q^+$ is a succinct transition function.

The relation of *transition by variable evaluation* $\varsigma \in \mathcal{P}(V)$ is defined the following way:

$$\rho_1 \overset{\text{ssu}}{\underset{\varsigma}{\to}} \rho_2 \overset{\text{def}}{\Leftrightarrow} \exists \phi \in \mathbb{F}_V .\ (\sigma \models \phi) \wedge \rho_2 \models \bigwedge_{q \in \rho_1} \delta_{\text{ssu}}(q, \sigma) \tag{2.19}$$

The conversion from ssuAFA to sAFA works the same way as the conversion from suAFA to AFA, we only use formulae instead of symbols. Instead of creating the new symbol ε, we create a new variable v_ε, assign the formula v_ε to the transitions, where ε would be present in the suAFA-AFA conversion. To make the variable evaluations that model the transitions of the original suAFA exclusive with the evaluations that model the formula v_ε, we replace the formulae ϕ by $\neg v_\varepsilon \wedge \phi$ for each ϕ from the original transitions.

Chapter 3

IIC for AFA Emptiness

In this chapter, we present the main contribution of this thesis, which is an adaptation of the IIC algorithm to alternating automata. The adaptation was not trivial. Some of the solutions presented here are inspired by the Petri net coverability adaptation from [15] and some solutions are our original and specific for the case of alternating automata. We will explain our reasoning and show experimental results for multiple possible implementations of some components of IIC.

3.1 Basic AFA Representation and General AFA

Our implementation works with three different representations of AFA, each of which has some pros and cons. Understanding the details about all the three representations is not necessary to understand IIC. Now, we introduce only the simplest one of them, *per-symbol AFA*. Because the reason of existence of the other two is just optimization of the IIC, we do not want to introduce them before introducing IIC. The details about all the three representations are presented later—in the sections 3.6 and 3.7. After introducing the per-symbol AFA here, we introduce also a general AFA, which is an abstract AFA, which will hide the differences between the three representations. The IIC will be discussed in terms of the general AFA.

Per-symbol AFA This is a simple intuitive representation of the classical AFA. For each symbol σ, for each state q, for each r_2, such that $r_2 \in \delta_M(q, \sigma)$, it contains a triplet (q, σ, r_2):

$$\mathrm{Rep}_{\text{per-sym}} = \{(q, \sigma, \rho) \in Q \times \Sigma_M \times 2^Q \mid \rho \in \delta_M(q, \sigma)\}$$

General AFA This is an abstract representation of AFA, which will hide the differences between symbolic and classical AFA (and between the two representations of the classical AFA—the per-symbol one that has been already introduced and a symbol-set one that will be presented later in the section 3.6). In the following text we present only the relation of general AFA with the per-symbol AFA. The text is then repeated in a more verbose way in the section 3.6, where the real reason of talking about the general AFA will be more clear.

The general AFA $M_G = (Q, \Sigma_G, G, I_M, \overset{\mathrm{G}}{\to}, F)$. In the same manner as for the classical and symbolic automata, Q is a finite set of states, $I_M \subseteq Q$ is an initial case, $F \in \mathbb{F}_Q^-$ is a negative boolean formula determining final cases. Set of *guardable symbols* Σ_G and the set of guard G is defined differently for each of the three representations. Now we naturally introduce only the version for per-symbol AFA:

$$\Sigma_G := \Sigma_M$$

A *guard* γ from a set of guards G represents a set of guardable symbols, by each of which a transition can be performed. The transitions in the $\mathrm{Rep}_{\text{per-sym}}$ can be performed only by single symbols:

$$G := \Sigma_M$$

Let $\dashv \subseteq \Sigma_G \times G$ be a guard relation. We will say that a guard γ *guards* a guardable symbol g iff $g \dashv \gamma$. For per-symbol AFA, the guard relation is the equality:

$$g \dashv \gamma \Leftrightarrow g = \gamma$$

Similarly we define a general transition relation by a guardable symbol, denoted by $\overset{\mathrm{G}}{\underset{g}{\rightarrow}}$, for a guardable symbol $g \in \Sigma_G$. For per-symbol AFA, it is equivalent to the transition relation by symbol, as defined for the classical AFA:

$$\overset{\mathrm{G}}{\underset{g}{\rightarrow}} := \overset{\mathrm{M}}{\underset{g}{\rightarrow}}$$

Transition relation by any guardable symbol is denoted as $\overset{\mathrm{G}}{\rightarrow}$.

The following lemma shows that we can write $\{q\}\overset{\mathrm{G}}{\underset{g}{\rightarrow}}\rho$ instead of $\rho \in \delta_M(q, g)$ for per-symbol AFA. Note that the general AFA does not have any transition function, only the transition relation (because it is not possible to abstract the function of δ_M and δ_s into one function δ_G). For per-symbol AFA, the lemma does show that the transition function can be simply substituted with the transition relation. The lemma is a part of the lemma 10, which in addition discusses symbolic AFA.

Lemma 1. *A singleton case is a predecessor of r_2 iff its element leads to r_2.*

$$\begin{aligned} &\forall \sigma \in \Sigma_M \forall q \in Q.\ \forall \rho \subseteq Q. \\ &\quad \{q\}\overset{M}{\underset{\sigma}{\rightarrow}}\rho \Leftrightarrow \rho \in \delta_M(q, \sigma) \end{aligned}$$

Proof. From the definition of $\overset{\mathrm{M}}{\underset{\sigma}{\rightarrow}}$, *SuccsOfStates* produces all cases $\rho \in \delta_M(q, g)$ associated with q, *SuccsOfCase* produces singletons $\{(q, \rho)\}$ for each such ρ. In (2.5), the singleton $\{(q, \rho)\}$ is obviously converted to ρ. □

In case of per-symbol AFA, a general representation of AFA corresponds with the per-symbol representation:

$$\begin{aligned} &\mathrm{Rep}_G \subseteq Q \times G \times 2^Q \\ &\mathrm{Rep}_G = \mathrm{Rep}_{\text{per-sym}} \qquad \text{(for per-symbol AFA)} \end{aligned}$$

The last ingredient that will be needed in the description of IIC, is the predicate $\mathit{SymbolsNotCovered}(\gamma_q, \Gamma)$ where $\gamma_q \in G$ and $\Gamma \subseteq G$. The predicate says that some symbol g, which is guarded by γ_q, is not guarded by any $\gamma \in \Gamma$.

$$\mathit{SymbolsNotCovered}(\gamma_q, \Gamma) \overset{\text{def}}{\Leftrightarrow} \exists g \in \Sigma_G \dashv \gamma_q.\ \forall \gamma \in \Gamma.\ \neg(g \dashv \gamma) \tag{3.1}$$

The implementation for the per-symbol AFA is again simple, as the guards are actually the symbols:

$$\mathit{SymbolsNotCovered}(\gamma_q, \Gamma) \Leftrightarrow \gamma_q \notin \Gamma$$

Properties of the transition relation The lemmas introduced in this text define properties of the transition relation that will be used in the sequel. All the provided proofs are implied directly from the definition of the transition relation and are close to trivial.

The lemmas presented here already express only the properties of the general AFA. The proofs of the lemmas are also representation-agnostic. The only concept that is used in the proofs and is not taken from the definition of general AFA, is the function *SuccsOfCase* (2.6). Note that the definition of the function is shared among sAFA and AFA. Later, after reading the description of the representations in the section 3.6, one can easily check that it is similarly shared also among the three representations (two of which are just differently expressed classical and symbolic AFA, and the remaining one is something in between). The proofs will therefore not be repeated again after introducing all the three representations.

The following lemma will be later used e.g. for finding predecessors of a case (or its subsets): The predecessor can be composed by union of small singleton predecessors, which are easier to be found.

Lemma 2. *Predecessor can be composed of singleton predecessors: If all states from ρ_1 lead to a subset of $\boldsymbol{\varrho}_2$ by some fixed guardable symbol, then some subset of ρ_2 is a successor of ρ_1.*

$$\begin{aligned}&\forall g \in \Sigma_G .\ \forall \rho_1, \boldsymbol{\varrho}_2 \subseteq Q .\\ &\qquad \forall q \in \rho_1 .(\exists r_2 \subseteq \boldsymbol{\varrho}_2 .\ \{q\} \xrightarrow[g]{G} r_2) \Rightarrow \exists \rho_2 \in \boldsymbol{\varrho}_2 .\ \rho_1 \xrightarrow[g]{G} \rho_2\end{aligned} \tag{3.2}$$

Proof. Let $\rho_1 = \{q_1, \cdots, q_n\}$. We have a $r_{2,1}, \cdots, r_{2,n}$ such that $\{q_i\} \xrightarrow[g]{G} r_{2,i}$ for $i = 1, \cdots, n$. By the lemma 1, the $r_{2,i}$ are translated to the terms of δ_M. By the definition (2.6), it is one of the subsets of *SuccsOfCase*. The ρ_2 can be constructed as a union of $\boldsymbol{\varrho}_{2,1}, \cdots, \boldsymbol{\varrho}_{2,n}$, which conforms to the definition (2.5). □

The following lemma says that if a case ρ_1 is a g-predecessor of ρ_2, then all of its states lead to some subset of ρ_2 by g.

Lemma 3. *Each predecessor can be decomposed to singleton predecessors: If ρ_1 is a predecessor of ρ_2 by some guardable symbol, then all states from ρ_1 lead to a subset of ρ_2 by that symbol.*

$$\begin{aligned}&\forall g \in \Sigma_G .\ \forall \rho_1, \rho_2 \subseteq Q .\\ &\qquad \rho_1 \xrightarrow[g]{G} \rho_2 \Rightarrow \forall q \in \rho_1 .(\exists r_2 \subseteq \rho_2 .\ \{q\} \xrightarrow[g]{G} \rho_2)\end{aligned} \tag{3.3}$$

Proof. From the definition (2.5), ρ_2 is a union of the successors r_2 from ξ. The cases r_2 are therefore subsets of ρ_2. By the definition (2.6) of *SuccsOfCase*, each of the states in ρ_1 must have a successor in ξ. □

The lemma 3, together with the lemma 2 is used in the following lemma, to show that not only states, but also all subsets of ρ_1 are g-predecessors of subsets of ρ_2.

Lemma 4. *For each $g \in \Sigma_G$, the relation $\xrightarrow[g]{G}$ is monotone with respect to the superset relation $\supseteq$.*

$$\begin{aligned}&\forall g \in \Sigma_G .\ \forall \boldsymbol{\varrho}_1, \boldsymbol{\varrho}_2 \subseteq Q .\\ &\qquad \boldsymbol{\varrho}_1 \xrightarrow[g]{G} \boldsymbol{\varrho}_2 \implies \forall \rho_1 \subseteq \boldsymbol{\varrho}_1 .\ \exists \rho_2 \subseteq \boldsymbol{\varrho}_2 .\ \rho_1 \xrightarrow[g]{G} \rho_2\end{aligned} \tag{3.4}$$

Proof. By the lemma 3, the case $\boldsymbol{\varrho}_1$ can be decomposed to states that all lead to some subset of $\boldsymbol{\varrho}_2$ by g. The case ρ_1 is a subset of those states, and by the lemma 2, a $\rho_2 \subseteq \boldsymbol{\varrho}_2$ exists. □

Note that this trivially implies that the transition relation by any guardable symbol $\overset{G}{\to}$ is also monotone with respect to $\supseteq$.

3.2 Backward Transition Function

The backward transition function finds upward-closure generator of all predecessors of a given case $\boldsymbol{\varrho}_2$. Moreover, it associates the elements (cases) of the generator with the symbols, by which the subsets of the cases transition to $\boldsymbol{\varrho}_2$. The backward transition function will be later useful in some of the transition rules of IIC, as well as in the backward antichain algorithm.

Generators of a bare set of predecessors A *bare set of predecessors* $\overleftarrow{\delta}_\rho : 2^Q \longrightarrow 2^{2^Q}$ serves to find all predecessors ρ_1 of a given case ρ_2, or of any of its subsets.

$$\overleftarrow{\delta}_\rho(\boldsymbol{\varrho}_2) = \{\rho_1 \subseteq Q \mid \exists \rho_2 \subseteq \boldsymbol{\varrho}_2.\ \rho_1 \overset{G}{\to} \rho_2\} \tag{3.5}$$

Lemma 5. *The bare set of predecessors is upward closed with respect to the preorder* $\supseteq$*.*

$$\overleftarrow{\delta}_\rho(\boldsymbol{\varrho}_2) = \overleftarrow{\delta}_\rho(\boldsymbol{\varrho}_2)\Uparrow \tag{3.6}$$

Proof. If $\boldsymbol{\varrho}_1 \overset{G}{\to} \boldsymbol{\varrho}_2$, then by monotonicity of $\overset{G}{\to}$ with respect to $\supseteq$, any subset of $\boldsymbol{\varrho}_1$ has a successor that is a subset of $\boldsymbol{\varrho}_2$. □

Corrolary: As the bare set of predecessors $\overleftarrow{\delta}_\rho(\boldsymbol{\varrho}_2)$ is upward closed, we can represent it by an upward-closure generator $\overleftarrow{\delta}_\rho(\boldsymbol{\varrho}_2) = \{\boldsymbol{\varrho}_{1,1}, \cdots, \boldsymbol{\varrho}_{1,n}\}\Uparrow$. We will call the cases $\boldsymbol{\varrho}_{1,1}, \cdots, \boldsymbol{\varrho}_{1,n}$ *generator cases.*

Pairing of symbols with generator cases In the following discussion about the generators of $\overleftarrow{\delta}_\rho(\boldsymbol{\varrho}_2)$ (for a fixed $\boldsymbol{\varrho}_2$), we will associate such symbols with each generator case $\boldsymbol{\varrho}_1$, by which only the subsets of the generator case $\boldsymbol{\varrho}_1$ transition to $\boldsymbol{\varrho}_2$.

By the lemma 2, for each guardable symbol g_1, exists a singleton generator $\{\boldsymbol{\varrho}_{1,g_1}\}$ where $\boldsymbol{\varrho}_{1,g_1}$ is a set of all states q, such that $\exists r_2 \in \boldsymbol{\varrho}_2.\ \{q\} \overset{G}{\underset{g}{\to}} r_2$. Let us name this case $\boldsymbol{\varrho}_{1,g_1}$ a *generator case for g.*

However, other guardable symbols $g_2, \cdots, g_m$ may exist, generator cases for which are subsets of $\boldsymbol{\varrho}_{1,g_1}$ (we denote these generator cases $\boldsymbol{\varrho}_{1,g_2}, \cdots, \boldsymbol{\varrho}_{1,g_m}$). As there is a generator super-case $\boldsymbol{\varrho}_{1,g_1}$ for them, they can be omitted from the generator of the bare set of predecessors. We want to have as small generator as possible (therefore the generator cases must be big). As a huge number of guardable symbols may exist, we do not want to create one generator case for each guardable symbol and associate the symbol to the case[1]. We

[1] Actually, for per-symbol AFA, that is just what we do—we create one generator case for each guardable symbol; but for the other two representations, multiple symbols can be guarded by one guard and we can benefit from it. Unfortunately, we have still talked only about the per-symbol AFA and the real benefit is therefore not so obvious.

therefore need a better approach to find a generator and associate a *guard* γ (instead of a guardable symbol) with each generator case. For the generator case $\boldsymbol{\varrho}_{1,g_1}$ from the reasoning given above, the guard γ that is associated with $\boldsymbol{\varrho}_{1,g_1}$ would guard just all the guardable symbols $g_1, \cdots, g_m$.

Formally, a *backward transition function* is $\overleftarrow{\delta} : 2^Q \longrightarrow 2^{G \times 2^Q}$ is a function from $\rho_2 \in 2^Q$ to a subset of $G \times 2^Q$ that satisfies the following properties for any $\boldsymbol{\varrho}_2 \in 2^Q$:

- *Non-generator exclusion.* For all $(\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\boldsymbol{\varrho}_2)$, the case $\boldsymbol{\varrho}_1$ is a generator case of $\overleftarrow{\delta}_\rho(\boldsymbol{\varrho}_2)$—each subset of $\boldsymbol{\varrho}_1$ transitions to a subset of $\boldsymbol{\varrho}_2$:
$$\forall \rho_1 \subseteq \boldsymbol{\varrho}_1.\ \exists \rho_2 \subseteq \boldsymbol{\varrho}_2.\ \rho_1 \xrightarrow{\text{G}} \rho_2 \tag{3.7}$$
By the monotonicity of $\xrightarrow{\text{G}}$, the sufficient condition to supply this property is that all the cases $\boldsymbol{\varrho}_1$ from $\overleftarrow{\delta}(\boldsymbol{\varrho}_2)$ are predecessors of some subset of $\boldsymbol{\varrho}_2$.

- *Guard disjointness.* For each guardable symbol g, a unique $(\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\boldsymbol{\varrho}_2)$ exists, such that $g \dashv \gamma$:
$$\forall g \in \Sigma_G.\ \exists! (\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\boldsymbol{\varrho}_2).\ g \dashv \gamma \tag{3.8}$$

- *Covering of all possibilities.* If $\rho_1 \xrightarrow[g]{\text{G}} \rho_2$, then g with ρ_1 must be covered in the backward transition function.
$$\begin{gathered}\forall \rho_1 \subseteq Q.\ \forall \rho_2 \subseteq \boldsymbol{\varrho}_2.\ \forall g \in \Sigma_G. \\ \rho_1 \xrightarrow[g]{\text{G}} \rho_2 \implies \exists (\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\boldsymbol{\varrho}_2).\ g \dashv \gamma \wedge \rho_1 \subseteq \boldsymbol{\varrho}_1\end{gathered} \tag{3.9}$$

In the following text, we introduce the implementation of backward transition function for the per-symbol AFA representation. We will prove that the implementation is *valid*—it holds the three conditions stated above.

Backward transition for per-symbol AFA For the per-symbol AFA, the guards are only the symbols. We cannot guard multiple symbols, therefore the backward transition function only simply associates each symbol σ with the single generator case that is composed of all the states leading by σ to some subset of $\boldsymbol{\varrho}_2$:

$$\overleftarrow{\delta}_{\text{per-sym}}(\boldsymbol{\varrho}_2) = \left\{ (\sigma, \boldsymbol{\varrho}_1) \mid \sigma \in \Sigma_M \wedge \boldsymbol{\varrho}_1 = \{ q \in Q \mid \exists r_2 \subseteq \boldsymbol{\varrho}_2.\ (q, \sigma, r_2) \in \text{Rep}_{\text{per-sym}} \} \right\} \tag{3.10}$$

The proof of the validity of $\overleftarrow{\delta}_{\text{per-sym}}(\boldsymbol{\varrho}_2)$ is trivial. By definition, the cases $\boldsymbol{\varrho}_1$ are composed of the states that lead to some subset of $\boldsymbol{\varrho}_2$, therefore by the lemma 2, the $\overleftarrow{\delta}_{\text{per-sym}}(\boldsymbol{\varrho}_2)$ contains only generator cases and satisfies (3.7). The disjointness of the guards (3.8) is obviously satisfied. By the lemma 3, all states of each σ-predecessor ρ_1 must lead by σ to some subset of $\boldsymbol{\varrho}_2$. By definition, the generator case $\boldsymbol{\varrho}_1$ is composed of all such states. The condition (3.9) therefore holds.

Example 3. Given the AFA in Figure 3.1, the per-symbol backward transition function returns the set

$$\overleftarrow{\delta}_{\text{per-sym}}(\boldsymbol{\varrho}_2) = \{(a, \{q_2, q_3\}), (b, \{q_3, q_4\}), (c, \{q_1, q_2, q_3\}), (d, \{q_3, q_4\})\} \tag{3.11}$$

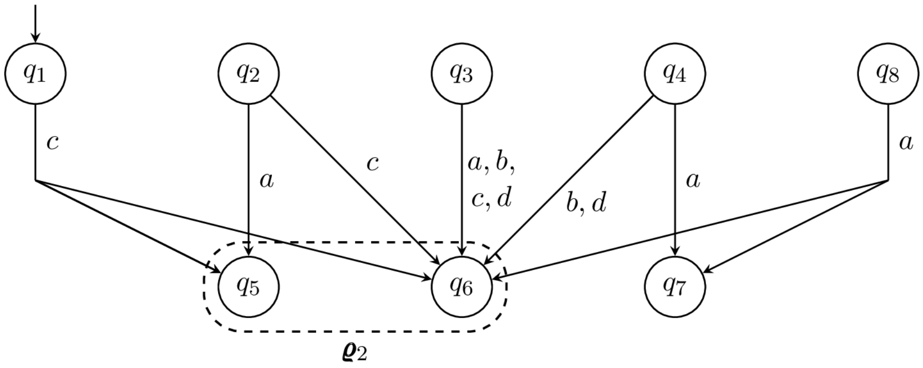


Figure 3.1: Backward transition function example

3.3 Reduction from AFA Emptiness to WSTS Covering

We convert the emptiness problem of an AFA $M_G = (Q, \Sigma_G, G, I_M, \overset{\mathrm{G}}{\rightarrow}, F)$ 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 a singleton with the initial case of AFA.
- $\rightarrow := \overset{\mathrm{G}}{\rightarrow}$ — transition function is the transition 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.
- $P^{\downarrow} := 2^Q \setminus \{\rho \in 2^Q \mid \rho \models F\}$ — bad states are all cases, for which F is true.

Lemma 6. *The system created the way stated above is a valid WSTS.*

To see that the lemma holds, notice that the $\preceq$ relation is a preorder on a finite domain. No infinite sequence that is purely non-increasing can exist. Therefore $\preceq$ is a well-quasi-order. The monotonicity of $\overset{\mathrm{G}}{\rightarrow}$ is proved in 4. □

Lemma 7. *The generated WSTS S is covered by $P^{\downarrow}$ iff the AFA M is empty.*

Proof. Recall the definition of covering:

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

As $\rightarrow = \overset{\mathrm{G}}{\rightarrow}$, the reflexive and transitive closure $\rho_0 \rightarrow^* \rho_m$ is apparently equivalent to the existence of a run $\rho = \rho_0 g_1 \rho_1 \ldots g_m \rho_m$. We have one initial WSTS state I_M, which is the initial case of AFA. For each WSTS state ρ', if $I_M \rightarrow^* \rho'$, then there exists a commencing run to ρ'. If ρ' is a bad state (final AFA case), the run is also terminating (and thus accepting). As $P^{\downarrow}$ is a complement of bad states, the ρ' is a bad state iff $\rho' \notin P^{\downarrow}$.

3.4 General IIC

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 sequence R of downward-closed sets of states $R_0^{\downarrow}R_1^{\downarrow}\dots R_N^{\downarrow}$ and a queue $\mathcal{Q}$ of counter-example candidates. We write $R|\mathcal{Q}$ to represent the algorithm state. Set $R_i^{\downarrow}$ is an over-approximated set of states that are reachable in i steps of WSTS. The number N is the currently analysed step of the system. It will increase during the algorithm (via the **Unfold** rule). Queue $\mathcal{Q}$ is a set of (κ, i) pairs where κ is an upward closed set of states from which we are sure that a bad state can be reached, and κ is reachable in i steps. The pairs from $\mathcal{Q}$ will be called *counter-example candidates*—if κ is reachable from an initial state, the set $P^{\downarrow}$ does not cover the WSTS. We write *min* $\mathcal{Q}$ to denote a set of pairs with minimal i. In addition, there is a special algorithm 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. When the algorithm terminates, the inductive invariant is the sequence of the over-approximated steps R, and the size of the invariant is the size of the representation of R (we will talk about the representation later).

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

Before introducing the particular transition rules, we quote that the algorithm is proved for soundness [15]. Moreover, if the WSTS is downward-finite, it is guaranteed to terminate [15]. The WSTS of AFA is downward-finite, because the state space of AFA is finite.

Transition rules The following discussion will be devoted to the transition rules of the IIC. The rules are presented in Figure 3.2, in the form:

$$\frac{C_1 \cdots C_k}{\sigma \mapsto \sigma'} \tag{3.12}$$

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.

We provide a brief description of the functionality of each of the transition rules. To be closer to the definition of the rules, we talk about them in terms of WSTS (note that in case of AFA, states of WSTS are cases), unless stated otherwise:

- **Initialize** — The rule initializes the algorithm state. Note that in terms of AFA, the downward closure (all supersets) of the initial case can be expressed as a complement of upward closure (all subsets) of all cases that are not supersets of I_M:

  $$I\!\downarrow = \{I_M\}\!\downarrow = 2^Q \setminus \{Q \setminus \{q\} \mid q \in I_M\}\!\uparrow$$

- **Valid** — The rule detects the convergence of the algorithm.

- **ModelSem** — A counter-example candidate, representing a run from an initial to a bad state, has been found.

- **ModelSyn** — A counter-example candidate, representing a run from the zero step (where only initial states are present) to a bad state, has been found.

- **Unfold** — No bad state can be reached in up to N steps. We start to analyze the step $N+1$. We begin with an over-approximating assumption that any state can be reached in the step $N+1$. This over-approximation then gets refined by the **Conflict** rule.
- **Candidate** — The currently analysed step N contains states out of $P^{\downarrow}$, we add one of them to the queue of counter-example candidates to check if it is really reachable, or is present in the step N only because of the over-approximation (in this case the step N is refined by the **Conflict** rule).
- **Conflict** — We have found out that the candidate $\kappa\uparrow$ cannot be reached from the step $i-1$. We therefore remove the candidate and refine the step i, by removing the upward closure of β from the step i. The generalization (more detail in 3.5) is a state β that is subsumed by κ (in terms of AFA, the case β is a superset of κ) and as well as κ, it cannot be reached from the state $i-1$. The states that subsume β are unreachable not only in the step i, but also in the steps $1, \cdots, i$ (see [15] for explanation and proof). We therefore remove them also from those steps.
- **Decide** — In contrast with the rule **Conflict**, we have found a predecessor of the candidate (κ, i) in the step $i-1$. We add the predecessor $(\kappa', i-1)$ to the queue of counter-example candidates.
- **Induction** — We have found out that a state $\beta_{i,j}$ (and its upward closure) that is unreachable in step i, is also unreachable in the step $i+1$. We remove the states also from the state $i+1$. Note that here we can apply the generalization too.

Valid

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

ModelSem

$$\frac{(\kappa, i) \in min\ \mathcal{Q} \quad I \cap \kappa\uparrow \neq \emptyset}{R|\mathcal{Q} \mapsto \mathsf{Unsafe}}$$

Decide

$$\frac{(\kappa, i) \in min\ \mathcal{Q} \quad i > 0 \quad \kappa' \in pre(\kappa\uparrow) \cap R_{i-1}^{\downarrow} \setminus \kappa\uparrow}{R|\mathcal{Q} \mapsto R|\mathcal{Q}.\textsc{Push}((\kappa', i-1))}$$

Initialize

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

Unfold

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

Conflict

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

Candidate

$$\frac{\kappa \in R_N^{\downarrow} \setminus P^{\downarrow}}{R|\emptyset \mapsto R|(\kappa, N)}$$

ModelSyn

$$\frac{(\kappa, 0) \in min\ \mathcal{Q}}{R|\mathcal{Q} \mapsto \mathsf{Unsafe}}$$

Induction

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

$$\beta \in Gen_i(\kappa) \overset{\text{def}}{\Leftrightarrow} \beta \preceq \kappa \wedge \beta\uparrow \cap I = \emptyset \wedge pre(\beta\uparrow) \cap R_i^{\downarrow} \setminus \beta\uparrow = \emptyset$$

Figure 3.2: IIC transition rules

To provide a better overview about how IIC works for AFA, we present an example of the overall run of IIC on a per-symbol AFA in the tables 3.1 and 3.2.

The analyzed per-symbol AFA is given by visualization, apparently $P^{\downarrow} = 2^Q \setminus \{\{q_4\}\}\Uparrow$

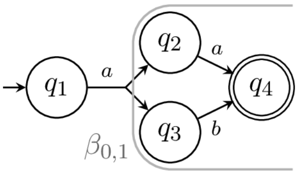
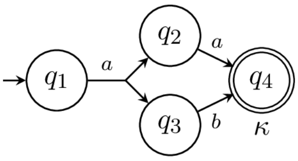
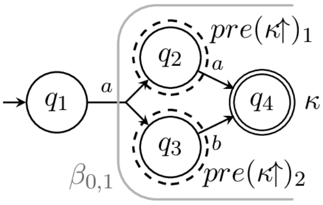
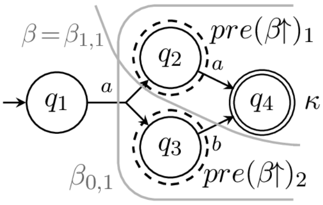
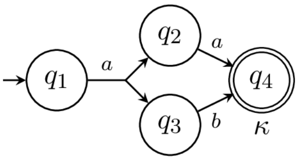
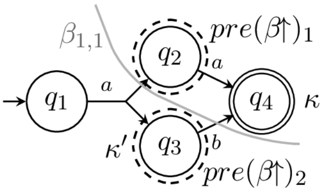
Application of rules	Visualization	Notes
Init $\implies$ **Initialize** $R_0 := 2^Q \setminus \{\{q_2, q_3, q_4\}\}\Uparrow$ $Q := \emptyset$		In the first step we cannot reach any subset of $\beta_{0,1} = \{q_2, q_3, q_4\}$. Note that we use the notation $\beta_{0,1}$, as in the **Induction** rule.
	N = 0	
$R_0^{\downarrow} \subseteq P^{\downarrow} \implies$ **Unfold** $R_1 := 2^Q$		We start to analyze the first step.
	N = 1	
$R_1^{\downarrow} \setminus P^{\downarrow} = \{\{q_4\}\}\Uparrow$ $\implies$ **Candidate**, $\kappa = \{q_4\}$ $Q := (\{q_4\}, 1)$		In the first step, a final case κ is reachable. We set up a new counter-example candidate.
$(\kappa, i) = (\{q_4\}, 1) \in min\ \mathcal{Q}$ $pre(\kappa\Uparrow) = \{\{q_2\}, \{q_3\}\}\Uparrow$ $R_0 = 2^Q \setminus \{\{q_2, q_3, q_4\}\}\Uparrow$ $pre(\kappa\Uparrow) \cap R_0 = \emptyset \implies$ **Conflict** $\beta := \{q_2, q_4\}$ $\beta \preceq \kappa : \{q_2, q_4\} \supseteq \{q_4\}$ $\beta\Uparrow \cap I = \emptyset :$ $\{\{q_2, q_4\}\}\Uparrow \cap \{\{q_1\}\} = \emptyset$ $pre(\beta\Uparrow) = \{\{q_2\}, \{q_3\}\}\Uparrow$ $pre(\beta\Uparrow) \cap R_0^{\downarrow} = \emptyset$ $\implies \beta \in Gen_0(\kappa)$ $R_1 := 2^Q \setminus \{\{q_2, q_4\}\}\Uparrow$ $\mathcal{Q}$.PopMin $(\mathcal{Q} = \emptyset)$	 q_1 q_2 q_3 q_4 a a b κ $pre(\kappa\Uparrow)_1$ $pre(\kappa\Uparrow)_2$ $\beta_{0,1}$  $\beta = \beta_{1,1}$ q_1 q_2 q_3 q_4 a a b κ $pre(\beta\Uparrow)_1$ $pre(\beta\Uparrow)_2$ $\beta_{0,1}$	We can see that no predecessor of the candidate $\kappa\Uparrow$ is reachable in the zero step. We apply the **Conflict** rule to refine the step R_1. By generalization, we find a superset of κ that is also unreachable, and remove its upward closure from R_1. Note that we could have picked also $\beta = \{q_3, q_4\}$ or $\beta = \{q_4\}$.
$R_1^{\downarrow} \subseteq P^{\downarrow} \implies$ **Unfold** $R_2 := 2^Q$		We start to analyze the second step.
	N = 2	
$R_2^{\downarrow} \setminus P^{\downarrow} = \{\{q_4\}\}\Uparrow$ $\implies$ **Candidate**, $\kappa = \{q_4\}$ $Q := (\{q_4\}, 2)$		In the second step, a final case κ is reachable. We set up a new counter-example candidate.
$(\kappa, i) = (\{q_4\}, 2) \in min\ \mathcal{Q}$ $pre(\kappa\Uparrow) = \{\{q_2\}, \{q_3\}\}\Uparrow$ $R_1 = 2^Q \setminus \{\{q_2, q_4\}\}\Uparrow$ $pre(\kappa\Uparrow) \cap R_1 \setminus \kappa\Uparrow = \{q_3\}\Uparrow$ $\implies$ **Decide**, $\kappa' = \{q_3\}$ $\mathcal{Q}$.Push$((\{q_3\}, 1))$ $(\mathcal{Q} = \{(\{q_3\}, 1), (\{q_4\}, 2)\})$	 $\beta_{1,1}$ q_1 q_2 q_3 q_4 a a b κ κ' $pre(\beta\Uparrow)_1$ $pre(\beta\Uparrow)_2$	We have found a predecessor of the counter-example candidate $\kappa\Uparrow$ in the first step. The predecessor is a new candidate $(\kappa', 1)$.

Table 3.1: Example of IIC, part 1

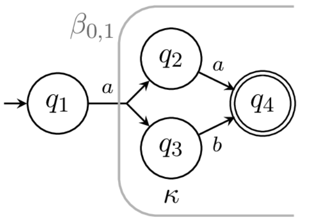
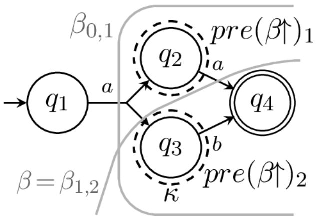
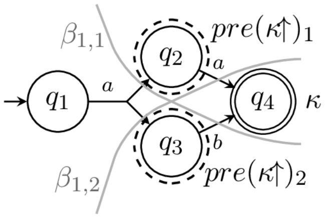
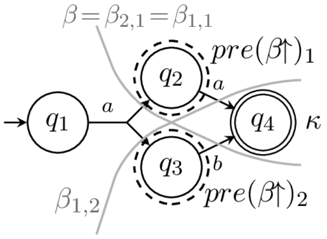
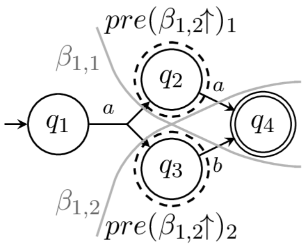
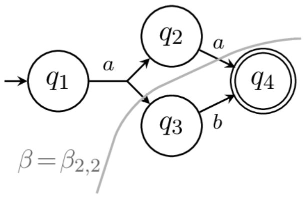
Application of rules	Visualization	Notes
$(\kappa, i) = (\{q_3\}, 1) \in min\ \mathcal{Q}$ $pre(\kappa\Uparrow) = \{\emptyset\}\Uparrow$ $R_0 = 2^Q \setminus \{\{q_2, q_3, q_4\}\}\Uparrow$ $pre(\kappa\Uparrow) \cap R_0 = \emptyset \implies$ **Conflict** $\beta := \{q_3, q_4\}$ $\beta \preceq \kappa : \{q_3, q_4\} \supseteq \{q_3\}$ $\beta\Uparrow \cap I = \emptyset :$ $\{\{q_3, q_4\}\}\Uparrow \cap \{\{q_1\}\} = \emptyset$ $pre(\beta\Uparrow) = \{\{q_2\}, \{q_3\}\}\Uparrow$ $pre(\beta\Uparrow) \cap R_0^{\downarrow} = \emptyset$ $\implies \beta \in Gen_0(\kappa)$ $R_1 := 2^Q \setminus \{\{q_2, q_4\}, \{q_3, q_4\}\}\Uparrow$ $\mathcal{Q}.\textsc{PopMin} \quad (\mathcal{Q} = \{(\{q_4\}, 2)\})$	 $\beta_{0,1}$ q_1 a q_2 a q_4 q_3 b κ  $\beta_{0,1}$ $pre(\beta\Uparrow)_1$ q_2 a q_1 a q_4 q_3 b $\beta = \beta_{1,2}$ κ $pre(\beta\Uparrow)_2$	We can see that no predecessor of the candidate $\kappa\Uparrow$ is reachable in the zero step. We apply the **Conflict** rule to refine the step R_1. By generalization, we find a superset of κ that is also unreachable, and remove its upward closure from R_1. Note that we could have picked also $\beta = \{q_3\}$.
$(\kappa, i) = (\{q_4\}, 2) \in min\ \mathcal{Q}$ $pre(\kappa\Uparrow) = \{\{q_2, q_3\}\}\Uparrow$ $R_1 = 2^Q \setminus \{\{q_2, q_4\}, \{q_3, q_4\}\}\Uparrow$ $pre(\kappa\Uparrow) \cap R_1 = \emptyset \implies$ **Conflict** $\beta := \{q_2, q_4\}$ $\beta \preceq \kappa : \{q_2, q_4\} \supseteq \{q_4\}$ $\beta\Uparrow \cap I = \emptyset :$ $\{\{q_2, q_4\}\}\Uparrow \cap \{\{q_1\}\} = \emptyset$ $pre(\beta\Uparrow) = \{\{q_2\}, \{q_3\}\}\Uparrow$ $pre(\beta\Uparrow) \cap R_1^{\downarrow} = \emptyset$ $\implies \beta \in Gen_1(\kappa)$ $R_2 := 2^Q \setminus \{\{q_2, q_4\}\}\Uparrow$ $\mathcal{Q}.\textsc{PopMin} \quad (\mathcal{Q} = \emptyset)$	 $\beta_{1,1}$ $pre(\kappa\Uparrow)_1$ q_2 a q_1 a q_4 κ q_3 b $\beta_{1,2}$ $pre(\kappa\Uparrow)_2$  $\beta = \beta_{2,1} = \beta_{1,1}$ $pre(\beta\Uparrow)_1$ q_2 a q_1 a q_4 κ q_3 b $\beta_{1,2}$ $pre(\beta\Uparrow)_2$	We can see that no predecessor of the candidate $\kappa\Uparrow$ is reachable in the first step. We apply the **Conflict** rule to refine the step R_2. By generalization, we find a superset of κ that is also unreachable, and remove its upward closure from R_2. Note that we could have picked also $\beta = \{q_3, q_4\}$ or $\beta = \{q_4\}$.
$i := 1$ $R_1 = 2^Q \setminus \{\{q_2, q_4\}, \{q_3, q_4\}\}\Uparrow$ $j := 2,\ \beta_{i,j} = \{q_3, q_4\}$ $\beta := \{q_3, q_4\}$ $\beta \preceq \beta_{i,j} : \{q_3, q_4\} \supseteq \{q_3, q_4\}$ $\beta\Uparrow \cap I = \emptyset :$ $\{\{q_3, q_4\}\}\Uparrow \cap \{\{q_1\}\} = \emptyset$ $pre(\beta\Uparrow) = \{\{q_2\}, \{q_3\}\}\Uparrow$ $pre(\beta\Uparrow) \cap R_1^{\downarrow} = \emptyset$ $\implies \beta \in Gen_1(\kappa)$ $\implies$ **Induction** $R_2 := 2^Q \setminus \{\{q_2, q_4\}, \{q_3, q_4\}\}\Uparrow$	 $pre(\beta_{1,2}\Uparrow)_1$ $\beta_{1,1}$ q_2 a q_1 a q_4 q_3 b $\beta_{1,2}$ $pre(\beta_{1,2}\Uparrow)_2$  q_2 a q_1 a q_4 q_3 b $\beta = \beta_{2,2}$	We have found out that none of the predecessors of $\beta_{1,2}\Uparrow$ is reachable in the first step, so $\beta_{1,2}\Uparrow$ is unreachable also in the second step. We can refine the second step. No bigger generalization of $\beta_{1,2}$ exists in this case. Note that the formal definition of this rule conveniently uses the existence of generalization to find out that $\beta_{1,2}$ is unreachable.
$R_1 = R_2 \implies$ **Valid**		IIC has converged.

Table 3.2: Example of IIC, part 2

3.5 IIC Implementation

Similarly to the Petri net coverability instance of IIC described in [15], 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 its upward closure is unreachable in the step $R_i^{\downarrow}$ (and also in the steps $R_0^{\downarrow} \cdots R_{i-1}^{\downarrow}$, as explained subsequently).

As proved in [15], the algorithm holds the invariant $R_i^{\downarrow} \subseteq R_{i+1}^{\downarrow}$, so if a blocker exists in the stage i, its upward closure (subsets) is unreachable also in the steps $0, \cdots, i-1$. We therefore do not need to have any of the blockers from $B_i{\uparrow}$ present in the stages $B_0, \cdots, B_{i-1}$. On the contrary, we assure in our implementation that such blockers are not present in the stages $B_0, \cdots, B_{i-1}$: if $j < i$, then $\nexists \beta_j \in B_j . \exists \beta_i \in B_i .\ \beta_i \preceq$. Similarly, if a blocker β_1 exists in B_i, all the other blockers $\beta_2 \in B_i$, such that $\beta_1 \preceq \beta_2$, would be redundant (β_1 already represents its upward closure), and we assure that they do not exist. Equivalence of any two successive steps $R_k^{\downarrow}$ and $R_{k+1}^{\downarrow}$ is then easily checked by $B_k = \emptyset$. The blockers in stages represent the inductive invariant and number of blockers is its size.

At any point of the IIC algorithm, stages $B_i \cup \cdots \cup B_N$ represent an under-approximation of cases that are not reachable in i or less steps of the AFA. Set $R_i^{\downarrow} = \Sigma \setminus B_i{\uparrow} \cup \cdots \cup B_N{\uparrow}$ is then an over-approximation of the reachable cases.

As for the implementation of the queue $\mathcal{Q}$, the new counter-example candidates are added by transition rules **Candidate** and **Decide**. Before applying the former rule, the queue $\mathcal{Q}$ is empty. The latter rule adds a candidate with a step index $i-1$. It is smaller than the actual minimal step index i, so in both cases the added candidate will be the new *min* $\mathcal{Q}$. The rule **Conflict** is the only rule that removes candidates. It removes only the candidates *min* $\mathcal{Q}$. The queue of counter-example candidates $\mathcal{Q}$ can be apparently conveniently implemented as a stack, where *min* $\mathcal{Q}$ is always at the root of the stack.

The following paragraphs talk about the actual implementation of each of the IIC transition rules, which are summarized in Figure 3.2. For the reader's convenience, we present the formal definition of the general transition rules again with each discussion about the rule. In the description of each rule, the text above the formal definition is a note only about the general version of the rule (more verbose than the note in the introductory section 3.4) while the text below the formal definition is a description of the implementation.

Initialize The rule initializes the state of the algorithm. We start with an empty queue $\mathcal{Q}$ and the zero step is initialized with downward closure of the initial states of WSTS.

$$\frac{}{\mathsf{Init} \mapsto I{\downarrow}|\emptyset} \tag{3.13}$$

In the WSTS of AFA, we have one initial state, which is the initial case I_M. We want to represent $I_M{\downarrow}$ as a set of blockers B_0, such that

$$I_M{\downarrow} = \Sigma \setminus B_0{\uparrow} = 2^Q \setminus B_0{\uparrow}$$

In terms of AFA, the upward closure $B_0{\uparrow}$ contains all cases, that do not include I_M — all cases, from which any state from I_M is missing. The generator of those cases is apparently

$$B_0 = \{Q \setminus \{q\} \mid q \in I_M\} \tag{3.14}$$

The index of the actually analyzed step is set to $N := 0$.

Valid This rule checks for convergence of IIC. By [15], if any two consecutive steps are equal (the first of any two consecutive stages has no blockers), the algorithm converges and the emptiness of AFA is proved.

$$\frac{\exists i < N.\ R_i^{\downarrow} = R_{i+1}^{\downarrow}}{R|\mathcal{Q} \mapsto \mathsf{Safe}} \tag{3.15}$$

As noted in the introduction to the section 3.5, equivalence of any two successive steps $R_i^{\downarrow}$ and $R_{i+1}^{\downarrow}$ is easily checked by $B_k = \emptyset$. As advised in [15], we call this rule any time it is applicable—after every change to the stages (after application of the rules **Conflict** and **Induction**).

Unfold If the candidate queue is empty, it means that we have proved that no previously queued counter-example candidate is reachable from any initial case. As $R_N^{\downarrow} \subseteq P^{\downarrow}$, we cannot apply the rule **Candidate** to add a new one — we have proved so far that no bad state is reachable in N steps. We start to explore a new step. We begin with an over-approximating assumption that in the new step, all WSTS states are reachable.

$$\frac{R_N^{\downarrow} \subseteq P^{\downarrow}}{R|\emptyset \mapsto R \cdot \Sigma|\emptyset} \tag{3.16}$$

We adjust the problem of inclusion to an equivalent one

$$R_N^{\downarrow} \subseteq P^{\downarrow} \Leftrightarrow \nexists \rho \in R_N^{\downarrow} \cap \Sigma \setminus P^{\downarrow} \tag{3.17}$$

The set $P^{\downarrow}$ is defined by means of the negative formula F that determines final cases of AFA, and its complement is:

$$\Sigma \setminus P^{\downarrow} := \{\rho \in 2^Q \mid \rho \models F\}$$

As the cardinality of this complement could be exponential to the size of the formula F, we avoid evaluating it. Instead, by the following adjustments, we construct a positive formula $H \subseteq \mathbb{F}_Q^+$ that determines presence of a case ρ in $R_N^{\downarrow}$, to be able to check the inclusion by a SAT solver.

The actually analyzed step $R_N^{\downarrow}$ is represented by a stage $B_N \subseteq 2^Q$ as

$$R_N^{\downarrow} = 2^Q \setminus B_N{\uparrow} = \{x \in \Sigma \mid \nexists \beta \in B_N.\ \beta \preceq x\} = \{\rho \in 2^Q \mid \nexists \beta \in B_N.\ \rho \subseteq \beta\}$$

By words, $R_N^{\downarrow}$ contains all those cases that are not subsets of any $\beta \in B_N$. For given $\beta \in 2^Q$, the cases that are not included in β are those that contain a state out of β:

$$R_N^{\downarrow} = \left\{ \rho \in 2^Q \;\middle|\; \forall \beta \in B_N.\ \bigvee_{q \notin \beta} q \in \rho \right\} = \left\{ \rho \in 2^Q \;\middle|\; \rho \models \bigwedge_{\beta \in B_N} \bigvee_{q \notin \beta} q \right\}$$

By this adjustment, we have obtained the formula $H = \bigwedge_{\beta \in B_N} \bigvee_{q \notin \beta} q$ and we can finally rewrite the inclusion (3.17) as

$$R_N^{\downarrow} \subseteq P^{\downarrow} \Leftrightarrow \nexists \rho \in 2^Q.\ \rho \models F \wedge H \Leftrightarrow \neg SAT\left(F \wedge \bigwedge_{\beta \in B_N} \bigvee_{q \notin \beta} q\right) \tag{3.18}$$

By use of a SAT solver, we find out whether $R_N^{\downarrow} \subseteq P^{\downarrow}$. If so, a new stage $B_{N+1} = \emptyset$ is added. It is empty because of the over-approximation — no unreachable states are assumed to exist. The index of the actually analyzed step N is then increased to $N := N + 1$.

Candidate The candidate queue is empty, it means that we have proved that no previously queued counter-example candidate is reachable from any initial state. If the currently analyzed step $R_N^{\downarrow}$ still contains bad states, we have to add one of those bad states as a candidate into the queue. Note that this rule is applied if the condition $R_N^{\downarrow} \subseteq P^{\downarrow}$ of the **Unfold** rule is not satisfied.

$$\frac{\kappa \in R_N^{\downarrow} \setminus P^{\downarrow}}{R|\emptyset \mapsto R|(\kappa, N)} \tag{3.19}$$

We obtain the candidate κ as a witness of the SAT, solved in the **Unfold** rule. Then we push it to the empty stack $\mathcal{Q}$.

Observing the behaviour of existing SAT solvers, we have found out that the solvers mostly return witnesses κ that are small in cardinality. This is not very convenient for IIC because from small candidates, smaller blockers are created by the **Conflict** rule, and the refinement is thus slower. We overcome this problem by negating the literals in F and H. Then the witness is a complement of κ that is small in cardinality, and κ is therefore big.

ModelSem A counter-example (κ, i) witnesses the existence of a path to a bad state—from all states that subsume κ, a bad state can be reached. If an initial state exists that subsumes κ (the state is in the $\kappa\!\uparrow$), a bad state can be reached from the initial state. The counter-example therefore witnesses the non-emptiness of AFA.

$$\frac{(\kappa, i) \in min\ \mathcal{Q} \quad I \cap \kappa\!\uparrow \neq \emptyset}{R|\mathcal{Q} \mapsto \mathsf{Unsafe}} \tag{3.20}$$

As we have only one initial state—the case I_M, the condition can be checked as $(\kappa, i) \in min\ \mathcal{Q} \quad I_M \subseteq \kappa$.

This rule is checked after adding a candidate into the queue $\mathcal{Q}$. This happens in the **Conflict** and **Candidate** rules. For the **Candidate** rule, it is sufficient to check $I_M \not\models F$ only once, at the beginning.

For the case of $i = 0$, we can apply a simpler **ModelSyn** rule instead.

ModelSyn A counter-example (κ, i) witnesses the existence of a path to a bad state—from all states that subsume κ, a bad state can be reached. The zero step contains upward closure of $I\!\uparrow$. If κ is reachable in the zero step, it means that some initial state subsumes κ and a bad state can be reached from the initial state. The counter-example κ therefore witnesses the non-emptiness of AFA.

$$\frac{(a, 0) \in min\ \mathcal{Q}}{R|\mathcal{Q} \mapsto \mathsf{Unsafe}} \tag{3.21}$$

Checking $i = 0$ is much less computationally expensive than checking the condition of the **ModelSem** rule. That is probably the purpose of this rule in IIC.

Decide This transition rule serves as an induction step to prove the reachability of bad states. We have a counter-example candidate (κ, i). *Induction hypothesis:* We know that from all states that subsume κ, a bad state is reachable. We also know that no initial state subsumes κ. *Induction step:* The candidate κ is over-approximatingly assumed to be reachable from an initial state in i steps. We want to prove (or contradict; see the **Conflict** rule) this assumption.

Apparently, to be reachable from an initial state, some state of $\kappa' \in \mathit{pre}(\kappa\uparrow)$ must exist in the over-approximation $R^{\downarrow}_{i-1}$ of states reachable in $i-1$ steps. Moreover, as justified by the subsequent reasoning, we are not interested in states κ' that subsume κ. By the lemma 8 it holds that if only states that subsume κ would be predecessors of κ, then their predecessors would subsume κ too, etc. No state in the chain of predecessors would be therefore subsumed by initial states I. By monotonicity, κ is then not reachable. Note that to keep the description simple, this is more of intuition than formal proof. For more precise explanation (with use of *relative inductivity*), please refer to [15].

Lemma 8. $\forall X \subseteq \Sigma.\ \mathit{pre}(X) \subseteq X \implies \mathit{pre}(\mathit{pre}(\cdots \mathit{pre}(X))) \subseteq X$

sketch. By definition,

$$\mathit{pre}(X) = \{x_1 \in \Sigma \mid \exists x_2 \in X.\ x_1 \to x_2\}$$

By substition,

$$\mathit{pre}(\mathit{pre}(X)) = \{x_1 \in \Sigma \mid \exists x_2 \in \mathit{pre}(X) \subseteq X.\ x_1 \to x_2\} \qquad \square$$

etc.

Apparently, $\mathit{pre}(\mathit{pre}(X)) \subseteq \mathit{pre}(X) \subseteq X$.

If find a κ' that conforms the conditions stated above, we perform the induction step by adding a new candidate $(\kappa', i-1)$ into the queue $\mathcal{Q}$. Reachability of $(\kappa', i-1)$ will be thus decided afterwards: If κ' is reachable (**ModelSem**), then a bad state is reachable. Otherwise, the step $R^{\downarrow}_{i-1}$ will have been refined by the **Conflict** rule, before the candidate (κ, i) will be checked again with the rules **Decide**/**Conflict**.

We recall the transition rule here:

$$\frac{(\kappa, i) \in \mathit{min}\ \mathcal{Q} \quad i > 0 \quad \kappa' \in \mathit{pre}(\kappa\uparrow) \cap R^{\downarrow}_{i-1} \setminus \kappa\uparrow}{R|\mathcal{Q} \mapsto R|\mathcal{Q}.\textsc{Push}((\kappa', i-1))} \tag{3.22}$$

In terms of AFA, the function pre finds all predecessors of subsets of the case κ. This problem has been already expressed in (3.5) by the function $\overleftarrow{\delta}_{\rho}$. And we have shown in the lemma 5 that the function $\overleftarrow{\delta}_{\rho}$ returns an upward-closed set of cases. We have described in 3.2 the approaches we use to find a generator of the upward closed set $\mathit{pre}(\kappa\uparrow)^{\uparrow}$ by the function $\overleftarrow{\delta}(\kappa)$. By the condition (3.7), the function $\overleftarrow{\delta}(\kappa)$ finds only the predecessors of κ and by the condition (3.9), it finds all of the predecessors.

From the definition of the rule, the case $\kappa' \in \mathit{pre}(\kappa\uparrow)$ must not be a subset of κ, and it must be contained in $R^{\downarrow}_{i-1}$. As $R^{\downarrow}_{i-1} = 2^Q \setminus B_{i-1}\uparrow \cup \cdots \cup B_N\uparrow$, this can be reformulated in a way that the case κ' must be neither a subset of κ, nor a subset of any blocker $\beta \in B_{i-1} \cup \cdots \cup B_N\uparrow$. By the property (3.9), such a κ' has some superset $\boldsymbol{\varrho}_1$ in $\overleftarrow{\delta}(\kappa)$. As the generator case $\boldsymbol{\varrho}_1$ is a superset of κ', it is as well as κ', neither a subset of κ, nor of any blocker. By (3.7), $\boldsymbol{\varrho}_1 \in \mathit{pre}(\kappa\uparrow)$.

We can therefore pick the new candidate κ' only from generator cases returned by $\overleftarrow{\delta}(\kappa)$:

$$(\gamma, \kappa') \in \{(\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\kappa) \mid \boldsymbol{\varrho}_1 \not\subseteq B_{i-1} \cup \cdots \cup B_N \cup \{\kappa\}\} \tag{3.23}$$

If such a κ' exists, we push it to the stack $\mathcal{Q}$.

Conflict In contrast with the **Decide** rule, we have found out that the over-approximation of the step $R^{\downarrow}_{i-1}$ does not contain any predecessor of $\kappa\uparrow$, or all the predecessors of $\kappa\uparrow$ subsume κ. We thus know that $\kappa\uparrow$ is not reachable in i steps — it did represent a spurious counter-example. We therefore remove it from the candidate queue and refine the step $R^{\downarrow}_{i}$, so that it would not contain $\kappa\uparrow$. Rather than removing $\kappa\uparrow$ from the step $R^{\downarrow}_{i}$, we remove its generalization — upward closure of a state β that is subsumed by κ and, as well as κ, it is unreachable in i steps. The β is unreachable not only in the step $R^{\downarrow}_{i}$, but in all steps $R^{\downarrow}_{1} \cdots R^{\downarrow}_{i}$, so we remove it from all those steps, so that the algorithm would hold the invariant $R^{\downarrow}_{k} \subseteq R^{\downarrow}_{k+1}$ for all $k < N$ (for more details, see [15]).

$$\textbf{Conflict}: \frac{(\kappa, i) \in min\ \mathcal{Q} \quad i > 0 \quad pre(\kappa\uparrow) \cap R^{\downarrow}_{i-1} \setminus \kappa\uparrow = \emptyset \quad \beta \in Gen_{i-1}(\kappa)}{R|\mathcal{Q} \mapsto R[R^{\downarrow}_{k} \leftarrow R^{\downarrow}_{k} \setminus \beta\uparrow]^{i}_{k=1}|\mathcal{Q}.\textsc{PopMin}} \tag{3.24}$$

This rule is applied if no κ' is found in the rule **Conflict**, such that would satisfy the equation (3.23). It means that we could find blockers $b \in B_{i-1} \cup \cdots \cup B_N$ that subsume all the predecessors of κ. By use of those blockers, we compute a generalization β (described subsequently) of the candidate κ. The generalization β will be a new blocker for the stage B_i. Before adding β to B_i, as noted in the beginning of the section 3.5, we must assure that no redundant blocker exists. Since each step k is represented by stages as $R^{\downarrow}_{k} = 2^{Q} \setminus B_k\uparrow \cdot \cdots \cdot B_N\uparrow$, the new blocker β applies to all steps $k = 1, \cdots, i$. We therefore remove all the blockers that are subsets of β from the stages $j = 1, \cdots, i$, because they are redundant. Then we add β to the stage B_i.

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 the specialization of generalization for AFA.

We apply generalization $Gen_{i-1}(\kappa)$ to cheaply create a blocker β that is subsumed by κ. The blocker β is then going to be added to the stage i. The new blocker β should hold some properties (that are known to be held by κ): It should not subsume any initial state and it should not be reachable in $i - 1$ steps.

$$\beta \in Gen_{i-1}(\kappa) \overset{\text{def}}{\Leftrightarrow} \beta \preceq \kappa \wedge \beta\uparrow \cap I = \emptyset \wedge pre(\beta\uparrow) \cap R^{\downarrow}_{i-1} \setminus \beta\uparrow = \emptyset \tag{3.25}$$

We break the condition for valid generalizations into these three restrictions (the first and the second one are conveniently translated into terms of AFA):

$$\beta \supseteq \kappa \tag{3.26a}$$

$$\beta \not\supseteq I_M \tag{3.26b}$$

$$pre(\beta\uparrow) \cap R^{\downarrow}_{i-1} \setminus \beta\uparrow = \emptyset \tag{3.26c}$$

We know that $\kappa\uparrow$ does not have predecessors in the step $R^{\downarrow}_{i-1}$. It means that we have found a blocker b for each pair $(\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\kappa)$, such that $\boldsymbol{\varrho}_1 \subseteq b$—this fact is formulated in the lemma 9. We associate the corresponding guards γ with the blockers b, one blocker for each pair $(\gamma, \boldsymbol{\varrho}_1)$:

$$\text{Б} \subseteq \{(b,\gamma) \mid b \in B_{i-1} \cup \cdots \cup B_N \cup \{\kappa\} \wedge (\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\kappa) \wedge \boldsymbol{\varrho}_1 \subseteq b\}$$
$$\forall (\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\kappa).\ \exists!(b,\gamma') \in \text{Б}.\ \gamma = \gamma' \tag{3.27}$$

Lemma 9. *If the precondition for **Conflict** is satisfied, then for each pair* $(\gamma, \boldsymbol{\varrho}_1) \in \overleftarrow{\delta}(\kappa)$*, exists a pair* $(b,\gamma) \in \text{Б}$ *such that* $\boldsymbol{\varrho}_1 \subseteq b$*.*

The generalization is computed by a use of Б in the following way:

$$\beta := Q \setminus (\mathit{MinimumHittingSet} \circ \mathit{AssureKappaDisjoint} \circ \mathit{ForbiddenCases})(\text{Б}) \tag{3.28}$$

Overview: First of all, we create a set of forbidden cases that are not allowed to subsume the blocker β (to be included in it), otherwise, the condition (3.26b) or (3.26c) would be violated. Then, to satisfy the condition (3.26a), we assure that the forbidden cases do not intersect κ. If some forbidden case intersected κ, the approximative minimum hitting set (that is created afterwards) could contain a state $q \in \kappa$, and after complementation, $\beta \not\supseteq \kappa$. In the end, we try to find a blocker β that subsumes (includes) many cases, but does not subsume (include) any of the forbidden ones—we prefer blockers β with bigger cardinality. We apply an approximative *greedy algorithm* for solving *minimum hitting set* problem [6] (which is dual to the *set cover* problem), to find a case that intersects each forbidden case. The generalization β is then obtained by complementation of this case. We provide a visual example in Figure 3.3.

The function $\mathit{AssureKappaDisjoint} : 2^{2^Q} \longrightarrow 2^{2^Q}$ assures in a simple way that the forbidden cases do not intersect κ:

$$\mathit{AssureKappaDisjoint}(P) = \{\rho \setminus \kappa \mid \rho \in P\} \tag{3.29}$$

The most complex is the function $\mathit{ForbiddenCases} : 2^{2^Q \times G} \longrightarrow 2^{2^Q}$. The following text describes the intuition behind its construction, afterwards, it is defined in (3.30) and proved for correctness in the theorems 2 and 3. By (3.26b), the generalization should not include the initial case, I_M is therefore one of the forbidden cases. Now, let us assure the condition (3.26c). When talking about blockers, we will use the term *to block*, that means "to be subsumed by" or, in terms of AFA, "to include". By each symbol $g \in \Sigma_G$, all predecessors of $\beta\upharpoonleft$ should be blocked by some $b \in B_{i-1}\upharpoonleft \cup \cdots \cup B_N\upharpoonleft \cup \beta\upharpoonleft$. To simplify the explanation, let us temporarily assume that only one symbol $g \in \Sigma_G$ exists. For the given g, let us choose a single arbitrary blocker $b \in B_{i-1}\upharpoonleft \cup \cdots \cup B_N\upharpoonleft \cup \kappa\upharpoonleft$ that blocks all g-predecessors of $\kappa\upharpoonleft$—we pick such a blocker that $(b,\gamma) \in \text{Б} \wedge g \dashv \gamma$. Then, only the g-successors of states $q \notin b$ are forbidden cases, because g-predecessors of the g-succesors of the states in b are blocked by b, and therefore they are not reachable in $i-1$ steps. Moreover, we know that the forbidden cases are not subsets of κ, because states out of b do not lead to subsets of κ, as all g-predecessors of $\kappa\upharpoonleft$ are blocked by b. Therefore, the resulting generalization β will be able include κ and none of the forbidden cases.

Now let us forget the simplification that Σ_G is singleton. The transitions of AFA are represented as $(q, \gamma_q, r_2) \in \mathrm{Rep}_G$. The successor r_2 is a forbidden case, if the guard γ_q guards a symbol, which is not guarded by any guard associated with a blocker that blocks q.

The forbidden cases are then computed in the following way

$$\begin{aligned}&\mathit{ForbiddenCases}(\text{Б}) = \{I_M\} \cup \{r_2 \subseteq Q \mid \\ &\quad (q, \gamma_q, r_2) \in \mathrm{Rep}_G \wedge \\ &\quad \mathit{SymbolsNotCovered}(\gamma_q, \{\gamma \mid (b, \gamma) \in \text{Б} \wedge q \in b\})\}\end{aligned} \tag{3.30}$$

Finally, let us prove that the function *ForbiddenCases* is correct: By proving the theorem 2, we will show that the generalization can be computed. The subsequent theorem 3 shows that the generalization satisfies also the condition (3.26c).

Theorem 2. *A hitting set for* $\mathit{AssureKappaDisjoint}(\mathit{ForbiddenCases}(\text{Б}))$ *exists.*

Proof. A hitting set exists, if $\mathit{AssureKappaDisjoint}(\mathit{ForbiddenCases}(\text{Б}))$ does not contain empty set. The empty set could obviously appear iff a forbidden case was a subset of κ (by the set difference in *AssureKappaDisjoint*). We can therefore prove that $\nexists r_2 \in \mathit{ForbiddenCases}(\text{Б}).\ r_2 \subseteq \kappa$.

One of the forbidden cases is I_M. As κ is checked by the **ModelSem** rule, the initial case I_M cannot be its subset.

For the other forbidden cases r_2, we show by contradiction that $r_2 \not\subseteq \kappa$. Let us assume that exists such a $r_2 \in \mathit{ForbiddenCases}(\text{Б})$ that is a subset of κ. The case r_2 is included in the representation Rep_G as (q, γ_q, r_2). By the definition of the AFA representation and by the definition of *SymbolsNotCovered* (3.39),

$$\exists g \dashv \gamma_q.\ \{q\} \overset{\mathrm{G}}{\underset{g}{\rightarrow}} r_2 \wedge \forall (b, \gamma) \in \text{Б}.\ q \notin b \vee \neg(g \dashv \gamma)$$

We show that exists a pair $(b, \gamma) \in \text{Б}$, such that $q \in b \wedge g \dashv \gamma$, by which we contradict the proof assumption.

As $\{q\} \overset{\mathrm{G}}{\underset{g}{\rightarrow}} r_2$ and $r_2 \subseteq \kappa$, from the condition (3.9), exists a pair $(\varrho_1, \gamma) \in \overleftarrow{\delta}(\kappa)$, such that $g \dashv \gamma$ and $\{q_1\} \subseteq \boldsymbol{\varrho}_1$. From the lemma 9, exists a $(b, \gamma) \in \text{Б}$, such that $\rho_1 \subseteq b$. Therefore, $q \in b$. □

The following theorem shows that the forbidden cases are complete—each case $\beta \supseteq \kappa$, that does not include any of the forbidden cases, has all its predecessors (and the predecessors of its subsets) blocked in the step R_{i-1}. The condition (3.26c) is thence satisfied.

Theorem 3. *The forbidden cases are complete.*

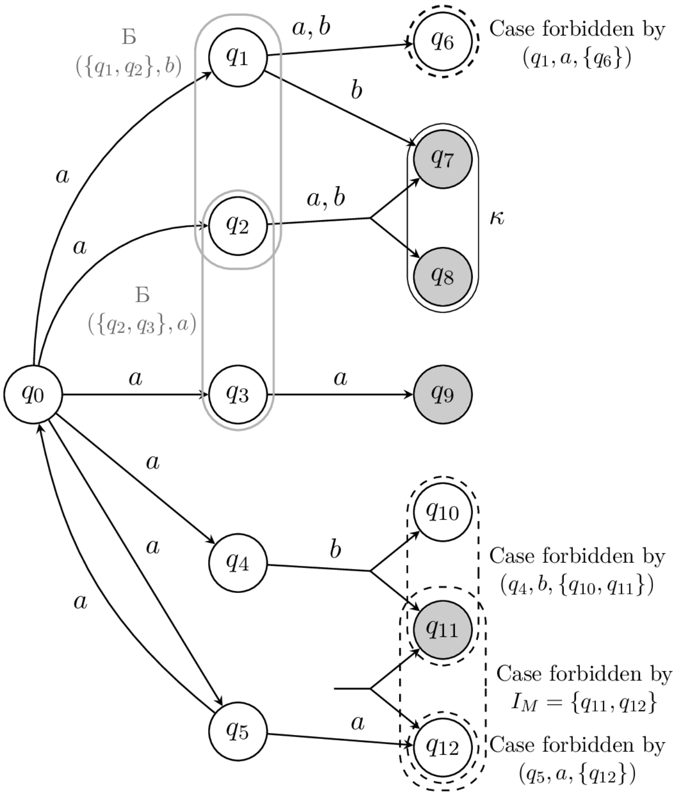
$$\forall \beta \supseteq \kappa.\ \mathit{ForbiddenCases}(\text{Б}) \cap \beta{\uparrow} = \emptyset \implies \mathit{pre}(\beta) \cap R^{\downarrow}_{i-1} \setminus \beta = \emptyset \tag{3.31}$$

Proof. First we rewrite the condition (3.31) to a more verbose form. For all $\beta \supseteq \kappa$,

$$\begin{aligned}&(\nexists r_f \in \mathit{ForbiddenCases}(\text{Б}).\ r_f \subseteq \beta) \implies \\ &\qquad\qquad \forall \rho_1 \in \mathit{pre}(\beta{\uparrow}).\ \exists b \in B_{i-1} \cup \dots \cup B_N \cup \{\beta\}.\ \rho_1 \subseteq b\end{aligned}$$

We will prove by contradiction that the condition is satisfied. Let us assume that the negation holds: Exists a $\beta \supseteq \kappa$, such that

$$\begin{aligned}&(\nexists r_f \in \mathit{ForbiddenCases}(\text{Б}).\ r_f \subseteq \beta) \wedge \\ &\qquad\qquad \exists \rho_1 \in \mathit{pre}(\beta{\uparrow}).\ \nexists b \in B_{i-1} \cup \dots \cup B_N \cup \{\beta\}.\ \rho_1 \subseteq b\end{aligned}$$



The figure visualizes a part of a per-symbol AFA that is interesting for generalization in the step $i-1$. In the relevant stages $B_{i-1}\cup\cdots\cup B_N$, the relevant blockers are $\{\{q_1,q_2\},\{q_2,q_3\}\}$. The states $q_0,\cdots,q_5$ are all in singleton forbidden cases (not visualized). Other forbidden cases, as well as blockers and the generalized candidate κ are visible. States in the minimum hitting set are white. States in the resulting generalization β are grey. Note that the case $\{q_9\}$ is not forbidden, because the state q_3 is in the blocker with $\gamma=a$. The case $\{q_6\}$ is forbidden, because the blocker of state q_1 lacks the symbol a.

Figure 3.3: Example of Generalization

The condition $\rho_1 \in pre(\beta{\twoheaduparrow})$ can be unwound to

$$\exists g \in \Sigma_G.\ \forall q \in \rho_1.\ \exists r_2 \subseteq \beta.\ \{q\}\xrightarrow[g]{G} r_2 \tag{3.32}$$

As the guards in $\overleftarrow{\delta}(\kappa)$ are disjoint (3.8), the guards in Б are disjoint too (by the restriction in (3.27)). For the g from (3.32) therefore exists a unique $(b_g,\gamma_g)\in$ Б, such that $g \dashv \gamma_g$. We know that ρ_1 is not blocked by any $b \in B_{i-1}\cup\cdots\cup B_N\cup\{\beta\}$. As $\beta \supseteq \kappa$, the case ρ_1 is nor blocked by any $b \in B_{i-1}\cup\cdots\cup B_N\cup\{\kappa\}$ and therefore it is not blocked by b_g.

It means that exists a $q\in\rho_1$ such that $q\notin b_g$. Obviously, exists also $(q,\gamma_q,r_2)\in \mathrm{Rep}_G$, such that $q \notin b_g$, $g\dashv\gamma_q$ and $r_2\subseteq\beta$. The symbols of γ_q are not covered by the guards $\{\gamma \mid (b,\gamma)\in Б \wedge q\in b\}$, because for the symbol g, the γ_g is the only guard from Б that guards g and the guard γ_g is not present in the set of guards, because $q\notin b$. The case $r_f = r_2 \subseteq \beta$ is therefore present in the *ForbiddenCases*. □

Induction The rule serves for pushing blockers forward from steps i to $i+1$. If there is a blocker $\beta\in B_i$ such that its generalization β' exists from the step i, we can push it to the step $i+1$. It maybe needs some more explanation — generalization of β from the step i exists iff all of its predecessors are unreachable in i steps and therefore it is a valid blocker for $i+1$.

$$\frac{R_i^{\twoheaddownarrow} = \Sigma\setminus\{\beta_{i,1},\ldots,\beta_{i,m}\}{\twoheaduparrow} \quad \beta\in Gen_i(\beta_{i,j}) \text{ for some } 1\le j\le m}{R|\emptyset \mapsto R[R_k^{\twoheaddownarrow} \leftarrow R_k^{\twoheaddownarrow}\setminus\beta{\twoheaduparrow}]_{k=1}^{i+1}|\emptyset} \tag{3.33}$$

Induction is implemented the following way: for each step i, we iterate through all $\beta \in B_i$ and check using the backward transition function $\overleftarrow{\delta}$, similarly to the **Decide**/**Conflict** rules, whether the blocker β has a predecessor reachable in i steps. If such a predecessor does not exist, Generalization is applied. The result of the generalization is added into the stage B_{i+1}, after removing redundant blockers the same way as in the **Conflict** rule.

Future Induction. In the **Induction** rule, we use an optimization noted in [15]. The induction can be used also for the step $R_N^{\downarrow}$, but no step R_{N+1} (nor stage B_{N+1}) then exists where the new blocker β could be added. We therefore create a temporary stage B'_{N+1} and add the blocker β there. Right after applying the next **Unfold** rule, we set all the blockers in B'_{N+1} as the blockers of the newly created step. Until then, we consider the blockers in the temporary stage as if they applied to all the stages $i = 1, \cdots, N$: In the equations (3.23) and (3.27), the sets of blockers for the step $i - 1$ will be computed as $B_{i-1} \cup \cdots \cup B_N \cup B'_{N+1} \cup \{\kappa\}$.

3.6 AFA Representations

To get closer to the real-world applications, we take a succinct symbolic automaton as the input for our implementation. We however never solve it directly. We convert it to an sAFA, which we solve per-se, or convert it further to a classical AFA. The second conversion uses minterm generation, which may create many minterms for each formula from a transition of sAFA. One transition of sAFA may therefore produce many AFA transitions that differ only in their symbol. We may represent each such class of transitions $\{\rho_1 \xrightarrow[\sigma_1]{\mathrm{M}} \rho_2, \cdots, \rho_1 \xrightarrow[\sigma_n]{\mathrm{M}} \rho_2\}$ as a single transition with the set of symbols $\rho_1 \xrightarrow[S]{\mathrm{M}} \rho_2$ where $S = \{\sigma_1, \cdots, \sigma_n\}$.

To sum up, we work with three different representations of AFA, each of which has its pros and cons. The following list is a preliminary comparison of the three representations, as for memory efficiency of the representation and time efficiency of computing backward and forward transitions. A detailed description of computing the backward transition will be described in the section 3.2. The computation of forward transition in the Generalization component of IIC is different from the one in the forward antichain algorithm from the chapter 4, and the detailed description can be found in the referenced text.

Per-symbol AFA A representation of classical AFA. For each symbol $\sigma \in \Sigma_M$, for each state $q \in Q$, we have a set of all $\rho \subseteq Q$, such that q leads to ρ by σ. Formally, the representation can be defined as

$$\mathrm{Rep}_{\text{per-sym}} = \{(q, \sigma, \rho) \in Q \times \Sigma_M \times 2^Q \mid \rho \in \delta_M(q, \sigma)\} \tag{3.34}$$

The above mentioned set of ρ is hidden in the formal definition. For given q and σ, we can extract it as $\{\rho \in 2^Q \mid (q, \sigma, \rho) \in \mathrm{Rep}_{\text{per-sym}}\}$.

Its advantage is that it is very simple and no minterm generation is needed to perform forward and backward transitions. Its disadvantage is that if there is a lot of symbols in the alphabet, we just have to iterate through all of them to compute the successors or predecessors of a case, even if there are big equivalence classes of symbols that would produce the same successors, or predecessors respectively. Another disadvantage is memory inefficiency in comparison to the other two representations.

Symbol-set AFA It is another representation of classical AFA. Symbol-set AFA is similar to per-symbol AFA, but for the same $(q, \rho) \in Q \times 2^Q$, symbols by which q leads to ρ are grouped together into one set. Formally,

$$\mathrm{Rep}_{\text{sym-set}} = \left\{(q, S, \rho) \in Q \times 2^{\Sigma_M} \times 2^Q \mid S \neq \emptyset \wedge S = \{\sigma \in \Sigma_M \mid \rho \in \delta_M(q, \sigma)\}\right\} \qquad (3.35)$$

In comparison with the per-symbol AFA, the representation is apparently smaller in size. In addition, we can easily (by minterm generation) find the equivalence classes of symbols that would produce the same successors or predecessors for the forward and backward transitions. Minterm generation is however an expensive operation, exponential to the number of transitions.

Symbolic AFA For each state $q \in Q$, we have all combinations $\phi \times \rho \in \mathbb{F}_V \times 2^Q$, such that $\rho \in \delta_s(q, \phi)$. Formally

$$\mathrm{Rep}_{\text{symbolic}} = \{(q, \phi, \rho) \in Q \times \mathbb{F}_V \times 2^Q \mid \rho \in \delta_s(q, \phi)\} \qquad (3.36)$$

This representation is the most brief one, as for size. Obtaining predecessors and successors of a case is similar to the symbol-set AFA, but instead of the set operations in the minterm generation, we combine formulae and call SAT, which makes it much more expensive in terms of time. Despite of this big disadvantage, it has an advantage that no conversion from sAFA to AFA is needed.

The necessity of a conversion from sAFA to AFA is a big disadvantage for the per-symbol and symbol-set representations. During the conversion, we generate minterms of all formulae that are present on the transitions of sAFA. As the blow-up of minterms can be up to exponential, it is sometimes computationally infeasible to handle the conversion, or to load the converted AFA to the computer's memory. The algorithms working with symbolic AFA on the other hand need to perform a lot of SAT solving, which makes them slow (the minterm generation in the backward transition function 3.2, SAT solving in the equations (3.39) and (4.3)).

General AFA To be able to talk about the three representations of AFA at once, we define an abstract, general automaton $M_G = (Q, \Sigma_G, G, I_M, \overset{G}{\rightarrow}, F)$, that will stand for any of the three representations. In the same manner as for the classical and symbolic automata, Q is a finite set of states, $I_M \subseteq Q$ is an initial case, $F \in \mathbb{F}_Q^-$ is a negative boolean formula determining final cases. Set of *guardable symbols* Σ_G is defined as:

- $\Sigma_G := \Sigma_M$ for per-symbol AFA,
- $\Sigma_G := \Sigma_M$ for symbol-set AFA,
- $\Sigma_G := \mathcal{P}(V)$ for symbolic AFA.

A *guard* γ from a set of guards G represents a set of guardable symbols, by each of which a transition can be performed.

- $G := \Sigma_M$ for per-symbol AFA (it represents a singleton set),
- $G := 2^{\Sigma_M}$ for symbol-set AFA,
- $G := \mathbb{F}_V$ for symbolic AFA.

Let $\dashv \subseteq \Sigma_G \times G$ be a guard relation. We will say that a guard γ *guards* a guardable symbol g iff $g \dashv \gamma$.

- $g \dashv \gamma \Leftrightarrow g = \gamma$ for per-symbol AFA,
- $g \dashv \gamma \Leftrightarrow g \in \gamma$ for symbol-set AFA,
- $g \dashv \gamma \Leftrightarrow g \models \gamma$ for symbolic AFA.

Similarly we define a general transition relation by a guardable symbol, denoted by $\xrightarrow[g]{\mathrm{G}}$, for a guardable symbol $g \in \Sigma_G$. The relation refers to the corresponding transition relation depending on the representation of automaton, we are talking about:

- $\xrightarrow[g]{\mathrm{G}} := \xrightarrow[g]{\mathrm{M}}$ for per-symbol AFA,
- $\rho_1 \xrightarrow[g]{\mathrm{G}} := \xrightarrow[g]{\mathrm{M}}$ for symbol-set AFA,
- $\xrightarrow[g]{\mathrm{G}} := \xrightarrow[g]{\mathrm{s}}$ for symbolic AFA,

Transition relation by any guardable symbol is denoted as $\xrightarrow{\mathrm{G}}$.

The following lemma shows that we can write $\{q\}\xrightarrow[g]{\mathrm{G}}\rho$ instead of $\rho \in \delta_M(q,g)$ for per-symbol and symbol-set automata, and instead of $\exists \phi.\ g \models \phi \wedge \rho \in \delta_s(q,\phi)$ for symbolic automata.

Lemma 10. *A singleton case is a predecessor of r_2 iff its element leads to r_2.*

$$\forall \sigma \in \Sigma_M \forall q \in Q.\ \forall \rho \subseteq Q.\ \{q\}\xrightarrow[\sigma]{M}\rho \Leftrightarrow \rho \in \delta_M(q,\sigma) \tag{3.37}$$

$$\forall \varsigma \in \mathcal{P}(V) \forall q \in Q.\ \forall \rho \subseteq Q.\ \{q\}\xrightarrow[\varsigma]{\mathrm{s}}\rho \Leftrightarrow \exists \phi \in \mathbb{F}_V.\ (\varsigma \models \phi) \wedge \rho \in \delta_s(q,\phi) \tag{3.38}$$

Proof. From the definition of $\xrightarrow[\sigma]{\mathrm{M}}$, *SuccsOfStates* produces all cases $\rho \in \delta_M(q,g)$ associated with q, *SuccsOfCase* produces singletons $\{(q,\rho)\}$ for each such ρ. In (2.5), the singleton $\{(q,\rho)\}$ is obviously converted to ρ. Analogously for symbolic AFA. □

A general representation of AFA corresponds with the particular representations of per-symbol, symbol-set and symbolic AFA.

$$\mathrm{Rep}_G \subseteq Q \times G \times 2^Q$$

- $\mathrm{Rep}_G := \mathrm{Rep}_{\text{per-sym}}$ for per-symbol AFA,
- $\mathrm{Rep}_G := \mathrm{Rep}_{\text{sym-set}}$ for symbol-set AFA,
- $\mathrm{Rep}_G := \mathrm{Rep}_{\text{symbolic}}$ for symbolic AFA,

The last ingredient that will be needed in the description of IIC, is the predicate *SymbolsNotCovered*(γ_q,Γ) where $\gamma_q \in G$ and $\Gamma \subseteq G$. The predicate says that some symbol g, which is guarded by γ_q, is not guarded by any $\gamma \in \Gamma$.

$$\mathit{SymbolsNotCovered}(\gamma_q,\Gamma) \overset{\text{def}}{\Leftrightarrow} \exists g \in \Sigma_G \dashv \gamma_q.\ \forall \gamma \in \Gamma.\ \neg(g \dashv \gamma) \tag{3.39}$$

The predicate *SymbolsNotCovered* is implemented differently for each AFA representation. The following definitions obviously satisfy the property (3.39).

- *SymbolsNotCovered*$(\gamma_q,\Gamma) \Leftrightarrow \gamma_q \notin \Gamma$ for per-symbol AFA,
- *SymbolsNotCovered*$(\gamma_q,\Gamma) \Leftrightarrow \emptyset \neq \gamma \setminus \bigcup \Gamma$ for symbol-set AFA,
- *SymbolsNotCovered*$(\gamma_q,\Gamma) \Leftrightarrow \neg SAT(\gamma \wedge \neg \bigvee \Gamma)$ for symbolic AFA.

3.7 Backward Transition Function, Finalized

In the following text, we introduce the implementations of backward transition functions for the symbol-set and symbolic AFA. We will prove that each of the implementations is *valid*—it holds the three conditions for the backward transition functions: non-generator exclusion (3.7), guard disjointness (3.8) and covering all possibilities (3.9).

Backward transition for symbol-set AFA – naive implementation Each state q leads to a subset of $\boldsymbol{\varrho}_2$ by the set of symbols

$$\mathrm{SymbolsTo}_q(\boldsymbol{\varrho}_2) = \bigcup \{S \subseteq \Sigma_M \mid \exists r_2 \subseteq \boldsymbol{\varrho}_2.\ (q, S, r_2) \in \mathrm{Rep}_{\text{sym-set}}\} \tag{3.40}$$

In the naive implementation we find all cases of ρ_1 that are predecessors of some subset of $\boldsymbol{\varrho}_2$. We naturally look for equivalence classes of symbols where the same set of states leads to some subset of $\boldsymbol{\varrho}_2$ by all symbols in the class. We therefore associate all states q which lead to a subset of $\boldsymbol{\varrho}_2$ with the respective $\mathrm{SymbolsTo}_q(\boldsymbol{\varrho}_2)$, and compute the tagged minterms.

$$\mathrm{PredecessorStates}(\boldsymbol{\varrho}_2) = \{(s, q) \in 2^{\Sigma_M} \times Q \mid s = \mathrm{SymbolsTo}_q(\boldsymbol{\varrho}_2) \wedge s \neq \emptyset\} \tag{3.41}$$

$$\delta_{\text{sym-set}}(\boldsymbol{\varrho}_2) = \mathit{TaggedMinterms}(\mathrm{PredecessorStates}(\boldsymbol{\varrho}_2)) \tag{3.42}$$

Recall the *TaggedMinterms* that are defined in (2.1). For a set of tagged sets $X = \{(s_1, q_1), \cdots, (s_n, q_n)\} \subseteq 2^{\Sigma_M} \times Q$, the tagged minterms are defined as

$$\mathit{TaggedMinterms}(X) \subseteq 2^{\Sigma_M} \times 2^Q$$

$$\mathit{TaggedMinterms}(X) = \left\{ (\gamma, \rho_1) = (\bigcap_{i=1,\dots,n} \gamma_i, \bigcup_{i=1,\dots,n} r_i) \;\middle|\; \forall i \in \{1, \cdots n\}.\ ((\gamma_i, r_i) \in \{(s_i, \{q_i\}), (\Sigma_M \setminus s_i, \emptyset)\}) \wedge \gamma \neq \emptyset \right\}$$

This paragraph shows that the function $\delta_{\text{sym-set}}$ is a valid backward transition function. The tagged minterms are all Boolean combinations γ of the sets of symbols s_i. Moreover, each combination γ is associated with a set ρ_1 of those q_i, for which $\gamma \subseteq s_i$ (as s_i is one of the operands of the intersection $\gamma = \bigcap \gamma_i$ iff $q_i \in \rho_1$). Since s_i is a set of symbols by which q_i leads to subsets of $\boldsymbol{\varrho}_2$, the case ρ_1 contains just the states that lead to subsets of $\boldsymbol{\varrho}_2$ by all symbols from γ, and by the composition lemma 2, all ρ_1 are predecessors of some subsets of $\boldsymbol{\varrho}_2$. The condition (3.7) therefore holds. By the decomposition lemma 3, all states of all g-predecessors of $\boldsymbol{\varrho}_2$ lead by g to subsets of $\boldsymbol{\varrho}_2$. As the minterms represent equivalence classes of symbols, they are disjoint and (3.8) is thus satisfied. The condition (3.9) also holds, since the minterms contain *all* the Boolean combinations. □

Example 4. Given the AFA in Figure 3.1, the naive symbol-set backward transition would be computed the following way

$$\mathrm{PredecessorStates} = \{(\{c\}, q_1), (\{a, c\}, q_2), (\{a, b, c, d\}, q_3), (\{b, d\}, q_4)\}$$

$$\delta_{\text{sym-set}}(\boldsymbol{\varrho}_2) = \{(\{c\}, \{q_1, q_2, q_3\}), (\{a\}, \{q_2, q_3\}), (\{b, d\}, \{q_3, q_4\})\}$$

We can see in the above example that two generator cases $\{q_1, q_2, q_3\}$ and $\{q_2, q_3\}$ can be merged because the latter one is a subset of the former one. In the following paragraph, an effective subset detection, based on minterm tree analysis, will be included into the minterm generation algorithm.

Backward transition for symbol-set AFA – improved implementation In the following text, we present an improved backward transition function that effectively detects existence of subsets among the generator cases. Introducing this subset detection into the backward transition function of symbol-set (and symbolic) AFA has brought a significant speedup of the IIC algorithm.

For the sets of symbols $W = \{s_1, \cdots, s_n\}$. The algorithm for minterm generation proposed in [8] builds a binary tree of depth n where the root node contains the universum Σ_M and nodes at level i contain all minterms of $\{s_1, \cdots, s_i\}$, for $1 \leq i \leq n$. A node μ_l at level i is a left child of node μ at level $i-1$ iff $\mathit{SymbolsOf}(\mu_l) = \mathit{SymbolsOf}(\mu) \cap s_i$. A node μ_r at level i is a right child of node μ at level $i-1$ iff $\mathit{SymbolsOf}(\mu_r) = \mathit{SymbolsOf}(\mu) \setminus s_i$.

Example 5. Continuing from the example 4, let $W = \{\{c\}, \{a, c\}, \{a, b, c, d\}, \{b, d\}\}$. The tree in Figure 3.4 is the tree generated from W. The minterms (denoted as $\mathit{SymbolsOf}(\mu)$ are in the first row of the nodes, the rest of the data in the nodes can be ignored for now—it concerns the algorithm for subset detection (we will talk about it subsequently). Note that e.g. the node μ_3 is the right child of the node μ_1, so it contains the symbol $\mathit{SymbolsOf}(\mu_3) = \mathit{SymbolsOf}(\mu_1) \setminus s_1 = \{a, b, c, d\} \setminus \{c\} = \{a, b, d\}$, which is one of the minterms in $\mathit{Minterms}(\{s_1\})$.

Our algorithm for subset detection processes the tree from leaves to the root and accumulates some additional data at each node. The additional data at node μ, which is at level i, is a set $\mathit{AdData}(\mu) \subseteq 2^{\Sigma_M} \times 2^Q$. The additional data $\mathit{AdData}(\mu)$ will be constructed in a way that it would hold modified conditions (3.7), (3.8) and (3.9). The first condition restricts the transitions that must exist from subsets of ρ_1 to any subset of ρ_2 to the transitions by symbols from $\mathit{SymbolsOf}(\mu)$. The second condition expresses that the guards on the same level of the minterm tree are disjoint. The third condition is relaxed in a way that it must hold only for the alphabet $\mathit{SymbolsOf}(\mu)$ instead of Σ_M, and only for the predecessors ρ_1 that are subsets of $\{q_{i+1}, \cdots, q_n\}$.

- *Non-generator exclusion (restricted).* For all $(\gamma, \boldsymbol{\varrho}_1) \in \mathit{AdData}(\mu)$, the case $\boldsymbol{\varrho}_1$ is a generator case of $\overleftarrow{\delta}_\rho(\boldsymbol{\varrho}_2)$:

$$\forall \rho_1 \subseteq \boldsymbol{\varrho}_1.\ \exists \rho_2 \subseteq \boldsymbol{\varrho}_2.\ \exists \sigma \in \mathit{SymbolsOf}(\mu).\ \rho_1 \xrightarrow[\sigma]{\mathrm{G}} \rho_2 \tag{3.43}$$

- *Guard disjointness (on the same level of the minterm tree).* For each level $i = 1, \cdots, n$, let $\boldsymbol{\mu}_i$ be a set of nodes on the level i of the minterm tree.

$$\forall g \in \Sigma_G.\ \exists! \mu \in \boldsymbol{\mu}_i.\ \exists! (\gamma, \boldsymbol{\varrho}_1) \in \mathit{AdData}(\mu).\ g \dashv \gamma \tag{3.44}$$

- *Covering of all possibilities (relaxed).* If $\rho_1 \xrightarrow[g]{\mathrm{G}} \rho_2$, then g with ρ_1 must be covered in the backward transition function.

$$\begin{gathered} \forall \rho_1 \subseteq \{q_{i+1}, \cdots q_n\}.\ \forall \rho_2 \subseteq \boldsymbol{\varrho}_2.\ \forall \sigma \in \mathit{SymbolsOf}(\mu). \\ \rho_1 \xrightarrow[\sigma]{\mathrm{M}} \rho_2 \implies \exists (\gamma, \boldsymbol{\varrho}_1) \in \mathit{AdData}(\mu).\ \sigma \in \gamma \wedge \rho_1 \subseteq \boldsymbol{\varrho}_1 \end{gathered} \tag{3.45}$$

If μ is the root node ($i = 0$), then we can set $\delta_{\text{sym-set}}(\boldsymbol{\varrho}_2) = \mathit{AdData}(\mu)$ (predecessors are then subsets of $\{q_1, \cdots, q_n\}$ and $\mathit{SymbolsOf}(\mu)$ is the universum Σ_M).

For the leaf nodes μ, the additional data contain the only element, which is $\mathit{AdData}(\mu) = \{(\mathit{minterm}(\mu), \emptyset)\}$. As there are no generator cases, the condition (3.43) is trivially satisfied. The minterms form equivalence classes, the condition (3.44) therefore holds. Finally, as ρ_1 in (3.45) is empty and the reduced alphabet contains only the symbols $\mathit{SymbolsOf}(\mu)$, the additional data $\mathit{AdData}(\mu)$ satisfies also the relaxed condition (3.45).

At each level i of the tree, the algorithm processes the nodes μ from left to right and performs the following operation where μ_l is a left child of μ and μ_r is its right child (if the child does not exist, its AdData is empty). To satisfy the guard disjointness (3.44) condition, each guard of a child node must be merged just once to the guards of the parent node. For this purpose we first define a unique assignment: If for an AdData element of the right child, an AdData element of the left child exists with a bigger $\boldsymbol{\varrho}_1$, then just one such element of the left child is assigned to the element of the right child.

$$
\begin{aligned}
&\mathit{UniqueAssignment} \subseteq \mathit{AdData}(\mu_l) \times \mathit{AdData}(\mu_r)\\
&\forall(\gamma_r, \boldsymbol{\varrho}_{1,r}) \in \mathit{AdData}(\mu_r).((\exists(\gamma_l, \boldsymbol{\varrho}_{1,l}) \in \mathit{AdData}(\mu_l).\ \boldsymbol{\varrho}_r \subseteq \boldsymbol{\varrho}_l) \implies\\
&\qquad\qquad \exists!(p, (\gamma_l, \boldsymbol{\varrho}_{1,l})) \in \mathit{UniqueAssignment}.\ p = (\gamma_r, \boldsymbol{\varrho}_{1,r}) \land \boldsymbol{\varrho}_{1,r} \subseteq \boldsymbol{\varrho}_{1,l})
\end{aligned}
$$

Now we can continue with the contstruction of AdData:

$$
\begin{aligned}
\mathit{AdData}(\mu) = \{&(\gamma, \boldsymbol{\varrho}_1) \in 2^{\Sigma_M} \times 2^Q \mid\\
&\quad (\gamma_l, \boldsymbol{\varrho}_{1,l}) \in \mathit{AdData}(\mu_l) \land\\
&\quad \boldsymbol{\varrho}_1 = \{q_{i+1}\} \cup \boldsymbol{\varrho}_{1,l} \land\\
&\quad \gamma = \gamma_l \cup \bigcup\{\gamma_r \mid (\gamma_r, \boldsymbol{\varrho}_{1,r}) \in \mathit{AdData}(\mu_r) \land \boldsymbol{\varrho}_{1,r} \subseteq \boldsymbol{\varrho}_{1,l} \land\\
&\qquad\qquad ((\gamma_l, \boldsymbol{\varrho}_{1,l}), (\gamma_r, \boldsymbol{\varrho}_{1,r})) \in \mathit{UniqueAssignment}\}\}\\
\cup&\\
\{&(\gamma, \boldsymbol{\varrho}_1) \in 2^{\Sigma_M} \times 2^Q \mid\\
&\quad (\gamma_r, \boldsymbol{\varrho}_{1,r}) \in \mathit{AdData}(\mu_r) \land\\
&\quad \boldsymbol{\varrho}_1 = \boldsymbol{\varrho}_{1,r} \land \gamma = \gamma_r \land\\
&\quad \nexists(\gamma_l, \boldsymbol{\varrho}_{1,l}) \in \mathit{AdData}(\mu_l).\ \boldsymbol{\varrho}_{1,r} \subseteq \boldsymbol{\varrho}_{1,l}\}
\end{aligned}
\tag{3.46}
$$

The definition of AdData is a union of two sets. In the following text, the sets will be denoted as the components of the definition of AdData.

We will prove the induction step: If the additional data of μ_l and μ_r satisfy the conditions (3.43), (3.44) and (3.45), their parent node μ satisfies the three conditions as well.

Theorem 4. *Let $\boldsymbol{\varrho}_2$ be a case, μ be a non-leaf node of the minterm tree of tagged sets*

$$\{(s_1, q_1), \cdots, (s_n, q_n)\} = \mathit{PredecessorStates}(\boldsymbol{\varrho}_2)$$

and $0 \leq i < n$ be the level at which μ resides in the tree. Let μ_l and μ_r be left and right child nodes of μ, the AdData of which satisfy (3.43). Then for each $(\gamma, \rho_1) \in \mathit{AdData}(\mu)$, the condition (3.43) is satisfied.

Proof. The $\mathit{AdData}(\mu)$ is a union of two sets. The latter one contains only the generators from the right child $\boldsymbol{\varrho}_{1,r}$, which satisfy (3.7) by the induction hypothesis. From the definition of minterm tree, $\mathit{SymbolsOf}(\mu_l) = \mathit{SymbolsOf}(\mu) \cap s_{i+1}$ and from the definition of minterms, $\mathit{SymbolsOf}(\mu_l) \neq \emptyset$. The additional data of the left child $\mathit{AdData}(\mu_l)$ contains only generator cases $\boldsymbol{\varrho}_{1,l}$ composed of states that lead to $\boldsymbol{\varrho}_2$ by symbols from $\mathit{SymbolsOf}(\mu_l)$. The state q_{i+1} leads to subsets of $\boldsymbol{\varrho}_2$ by symbols from s_{i+1}. The set of symbols of the left child node, $\mathit{SymbolsOf}(\mu_l) = \mathit{SymbolsOf}(\mu) \cap s_{i+1}$, contains no other symbols. Then by composition, the new cases $\{q_{i+1}\} \cup \boldsymbol{\varrho}_{1,l}$ are predecessors of subsets of $\boldsymbol{\varrho}_2$ by symbols from $\mathit{SymbolsOf}(\mu_l) \subseteq \mathit{SymbolsOf}(\mu)$. □

Now let us prove the induction step for (3.44) as well.

Theorem 5. *Let $\boldsymbol{\varrho}_2$ be a case. For a tree of tagged sets*

$$\{(s_1, q_1), \cdots, (s_n, q_n)\} = \mathit{PredecessorStates}(\boldsymbol{\varrho}_2),$$

let i be a level of the tree, $0 \leq i < n$. Let $\boldsymbol{\mu}_i$ be all nodes at the level i and $\boldsymbol{\mu}_{i+1}$ be the nodes at the level $i+1$. If the condition (3.44) is satisfied for the level $i+1$ (the guards of all nodes in the level $i+1$ are disjoint), then it holds also for the level i.

Proof. If a node $\mu_l \in \boldsymbol{\mu}_{i+1}$ is a left child of a node $\mu \in \boldsymbol{\mu}_i$, all of its guards γ_l are obviously merged with just one γ of μ (in the first component of the AdData definition). If a node $\mu_r \in \boldsymbol{\mu}_{i+1}$ is a right child of a node $\mu \in \boldsymbol{\mu}_i$, then its guard γ_r appers just once even in the second component of the AdData definition, or, if a left child μ_l exists and $\exists(\gamma_l, \boldsymbol{\varrho}_l) \in \mathit{AdData}(\mu_l)$. $\boldsymbol{\varrho}_r \subseteq \boldsymbol{\varrho}_l$, it appears in the first component in the union with the γ_l, which is uniquely assigned to the γ_r. Just one such assignment exists, therefore symbols of γ_r appear also just once in the AdData of the parent.

Each node in $\boldsymbol{\mu}_{i+1}$ has just one parent in the tree, each guard from the nodes of $\boldsymbol{\mu}_{i+1}$ is merged with just one guard of the parent node. As each guardable symbol was guarded just by one guard in the level $i+1$, it is therefore guarded just once in the level i. □

The following theorem proves the induction step for (3.45).

Theorem 6. *Let $\boldsymbol{\varrho}_2$ be a case, μ be a non-leaf node of the minterm tree of tagged sets*

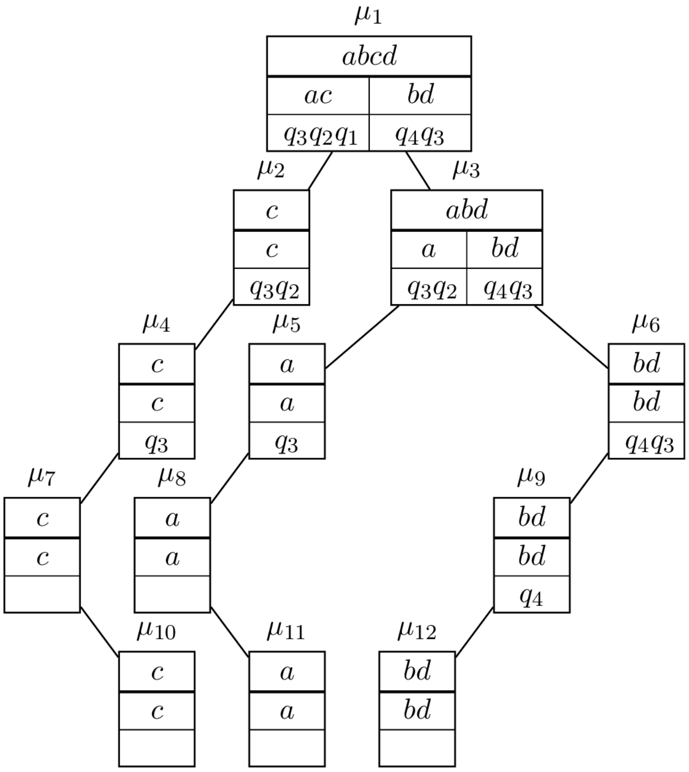
$$\{(s_1, q_1), \cdots, (s_n, q_n)\} = \mathit{PredecessorStates}(\boldsymbol{\varrho}_2)$$

and $0 \leq i < n$ be the level at which μ resides in the tree. Let μ_l and μ_r be left and right child nodes of μ, the AdData of which satisfy (3.45). Then the condition (3.45) is satisfied.

Proof. We divide the proof in two parts. At first, we prove the condition (3.45) for $\rho_1 \subseteq \{q_{i+2}, \cdots, q_n\}$. Then we prove by contradiction that for $\rho_1 \subseteq \{q_{i+1}, \cdots, q_n\}$, such that $q_{i+1} \in \rho_1$, the condition is still satisfied.

In comparision with the condition for $\mathit{AdData}(\mu_l)$, the alphabet $\mathit{SymbolsOf}(\mu)$ is enlarged by $\mathit{SymbolsOf}(\mu) \setminus s_{i+1}$. Those $\rho_{1,r}$ that transition to $\boldsymbol{\varrho}_2$ via $\mathit{SymbolsOf}(\mu) \setminus s_{i+1}$ are covered by some $(\gamma_r, \boldsymbol{\varrho}_{1,r}) \in \mathit{AdData}(\mu_r)$. The pair $(\gamma_r, \boldsymbol{\varrho}_{1,r})$ is added to $\mathit{AdData}(\mu)$ in the second part of (3.46), or if $\exists(\gamma_l, \boldsymbol{\varrho}_{1,l}) \in \mathit{AdData}(\mu_l)$. $\boldsymbol{\varrho}_{1,r} \subseteq \boldsymbol{\varrho}_{1,l}$, then the first component of (3.46) contains $\{q_{i+1}\} \cup \boldsymbol{\varrho}_{1,l} \supseteq \rho_{1,r}$ with a γ that includes γ_r.

In comparison with the condition for $\mathit{AdData}(\mu_r)$, the alphabet $\mathit{SymbolsOf}(\mu)$ is enlarged by $\mathit{SymbolsOf}(\mu) \cap s_{i+1}$. Those $\rho_{1,l}$ that transition to $\boldsymbol{\varrho}_2$ via $\mathit{SymbolsOf}(\mu) \cap s_{i+1}$ are covered by some $(\gamma_l, \boldsymbol{\varrho}_{1,l}) \in \mathit{AdData}(\mu_l)$. The first part of (3.46) contains for each such pair $(\gamma_l, \boldsymbol{\varrho}_{1,l})$, a pair $(\gamma, \boldsymbol{\varrho}_1)$ where $\boldsymbol{\varrho}_1 \supseteq \boldsymbol{\varrho}_{1,l} \supseteq \rho_{1,l}$ and $\gamma \supseteq \gamma_l$.



To save space, the set $\{x_1, x_2, \cdots, x_n\}$ is denoted as $x_1x_2\cdots x_n$.

Figure 3.4: Minterm tree with *AdData*

By definition (3.46), for all $(\gamma_l, \boldsymbol{\varrho}_{1,l}) \in \mathit{AdData}(\mu_l)$, the $\mathit{AdData}(\mu)$ contains a pair $(\gamma, \boldsymbol{\varrho}_1)$, such that $\boldsymbol{\varrho}_1 = \{q_{i+1}\} \cup \boldsymbol{\varrho}_{1,l}$ and $\gamma \supseteq \gamma_l$. We will show by contradiction that for each ρ_1 that contains q_{i+1}, the guard γ_l covers all symbols by which ρ_1 leads to $\boldsymbol{\varrho}_2$, and $\{q_{i+1}\} \cup \boldsymbol{\varrho}_{1,l}$ includes ρ_1. Let us suppose that

$$\exists \sigma \in \mathit{SymbolsOf}(\mu).\ \exists \rho_1 \subseteq \{q_{i+1}, \cdots, q_n\}.\ \exists \rho_2 \subseteq \boldsymbol{\varrho}_2.$$
$$q_{i+1} \in \rho_1 \wedge \rho_1 \overset{\mathrm{M}}{\underset{\sigma}{\rightarrow}} \rho_2 \wedge \forall (\gamma_l, \boldsymbol{\varrho}_{1,l}) \in \mathit{AdData}(\mu_l).\ \rho_1 \not\subseteq \{q_{i+1}\} \cup \boldsymbol{\varrho}_{1,l} \vee \sigma \notin \gamma_l$$

The state q_{i+1} leads to subsets of $\boldsymbol{\varrho}_2$ only by symbols from s_{i+1}. Then by decomposition, $\sigma \in \mathit{SymbolsOf}(\mu) \cap s_{i+1} = \mathit{SymbolsOf}(\mu_l)$. Let $\rho_{1,l} = \rho_1 \setminus \{q_{i+1}\}$. By monotonicity of $\overset{\mathrm{M}}{\underset{\sigma}{\rightarrow}}$, the case $\rho_{1,l}$ is a σ-predecessor of some subset of $\boldsymbol{\varrho}_2$. As $\rho_{1,l} \subseteq \{q_{i+2}, \cdots, q_n\}$, we can say that

$$\exists \sigma \in \mathit{SymbolsOf}(\mu_l).\ \exists \rho_1 \subseteq \{q_{i+2}, \cdots, q_n\}.\ \exists \rho_2 \subseteq \boldsymbol{\varrho}_2.$$
$$\rho_1 \overset{\mathrm{M}}{\underset{\sigma}{\rightarrow}} \rho_2 \wedge \forall (\gamma_l, \boldsymbol{\varrho}_{1,l}) \in \mathit{AdData}(\mu_l).\ \rho_{1,l} \not\subseteq \boldsymbol{\varrho}_{1,l} \vee \sigma \notin \gamma_l$$

which is a contradiction with the induction hypothesis, particularly with the condition (3.45) for μ_l. □

Corrolary: The algorithm for computing the backward transition function with subset detection is valid.

Backward transition for symbolic AFA Analogous as for the symbol-set AFA, only the minterm generation for formulae is used, and union of γ in the improved implementation is replaced by disjunction.

Chapter 4

Antichain-based Algorithms

Forward and backward algorithms for deciding AFA emptiness, based on a computation of the least fixpoint of a monotone function have been introduced in [9]. In this chapter we describe our implementation of the algorithms. The description is much more brief, because for this work, antichain is only a comparison reference algorithm.

The following are the definitions from [9]. A *chain* K over Q is such a set of cases $K = \{\rho_1, \cdots, \rho_n\} \subseteq 2^Q$, for which $\rho_1 \subset \cdots \subset \rho_n$ and $n > 1$. An *antichain* over Q is a set $A \subseteq 2^Q$ that does not include chains, i.e. $\forall \boldsymbol{\varrho}, \boldsymbol{\varrho}' \in A.\ \boldsymbol{\varrho} \not\subset \boldsymbol{\varrho}'$. Set of antichains over Q will be denoted $\mathbb{A}$. For two antichains $A, A' \in \mathbb{A}$, we say that $A \sqsubseteq A'$, iff all cases of A are included in cases of A', formally $\forall \boldsymbol{\varrho} \in A.\ \exists \boldsymbol{\varrho}' \in A'.\ \boldsymbol{\varrho} \subseteq \boldsymbol{\varrho}'$. For two antichains $A, A' \in \mathbb{A}$, we say that $A \mathrel{\widetilde{\sqsubseteq}} A'$ if all cases from A' have a subset in A, formally $\forall \boldsymbol{\varrho} \in A'.\ \exists \boldsymbol{\varrho}' \in A.\ \boldsymbol{\varrho} \subseteq \boldsymbol{\varrho}'$[1]. For an arbitrary set of cases X, a case $\rho \in X$ is *maximal*, iff no other $\rho' \in X$ exists, such that $\rho \subset \rho'$. Similarly $\rho \in X$ is *minimal*, iff $\nexists \rho' \in X.\ \rho' \subset \rho$. The set of maximal cases of X is denoted as $\lceil X \rceil$ and the set of minimal cases of X is denoted as $\lfloor X \rfloor$. Given two antichains $A, A' \in \mathbb{A}$, the $\sqsubseteq$-lub (least upper bound) of A and A' is the antichain $A \sqcup A' = \lceil A \cup A' \rceil$ and the $\widetilde{\sqsubseteq}$-glb (greatest lower bound) of A and A' is the antichain $A \mathbin{\widetilde{\sqcap}} A' = \lfloor A \cup A' \rfloor$.

Let $\mathit{Models}(F)$ be a set of cases that satisfy F, formally $\mathit{Models}(F) = \{\rho \subseteq Q \mid \rho \models F\}$. Let $\mathit{Pre}(X)$ be a set of predecessors of all cases from X. Then $\lceil \mathit{Pre}(X) \rceil$ can be apparently computed by the backward transition function, introduced in the section 3.2, as

$$\lceil \mathit{Pre}(X) \rceil = \lceil \{\rho_1 \mid \rho_2 \in X \land (\gamma, \rho_1) \in \overleftarrow{\delta}(\rho_2)\} \rceil$$

Let $\mathit{Post}(X)$ be a set of successors of all cases from X. Similarly as Pre, it can be computed by unifying the successors of all cases from X. A set of successors of a case ρ_1 is denoted as $\mathit{post}(\rho_1)$. By the decomposition lemma 3, for some symbol $g \in \Sigma_G$, minimal g-successors of a case ρ_1 are the combinations of minimal g-successors r of states $q \in X$. For a given $\rho_1 = \{q_1, \cdots, q_n\}$, we compute $\mathit{post}(\rho_1)$ in the following way

[1] The tuples $\langle \mathbb{A}, \sqsupseteq \rangle$ and $\langle \mathbb{A}, \widetilde{\sqsupseteq} \rangle$ are complete lattices

- For per-symbol AFA

$$post(\rho_1) = \left\{ \bigcup_{i=1\ldots n} r_i \;\middle|\; \sigma \in \Sigma_M \wedge \forall i \in \{1, \cdots, n\}.\ (q_i, \sigma, r_i) \in \mathrm{Rep}_{\text{per-sym}} \right\} \tag{4.1}$$

- For symbol-set AFA

$$post(\rho_1) = \left\{ \bigcup_{i=1\ldots n} r_i \;\middle|\; \forall i \in \{1, \cdots, n\}.\ (q_i, S_i, r_i) \in \mathrm{Rep}_{\text{per-sym}} \wedge \bigcap_{i=1\ldots n} S_i \neq \emptyset \right\} \tag{4.2}$$

- For symbolic AFA

$$post(\rho_1) = \left\{ \bigcup_{i=1\ldots n} r_i \;\middle|\; \forall i \in \{1, \cdots, n\}.\ (q_i, \phi_i, r_i) \in \mathrm{Rep}_{\text{per-sym}} \wedge \mathrm{SAT}(\bigwedge_{i=1\ldots n} \phi_i) \right\} \tag{4.3}$$

Then the minimum cases of $Post(X)$ are computed as

$$\lfloor Post(X) \rfloor = \left\lfloor \bigcup_{\rho_1 \in X} post(\rho_1) \right\rfloor$$

Now, we present the forward and backward antichain-based algorithms from [9], using minimum fixed point search μ. The following three statements are equivalent:

- The AFA is not empty
- $\{I_M\} \sqsubseteq (\mu X.\ \lceil Pre(X) \rceil \sqcup \lceil Models(F) \rceil)$ (backward antichain)
- $(\mu X.\ \lfloor Post(X) \rfloor \,\widetilde{\sqcap}\, \{I_M\}) \cap Models(F) \neq \emptyset$ (forward antichain).

The following is the pseudocode for the basic antichain, derived from the pseudocode from [9].

Algorithm 1 Backward antichain algorithm for solving AFA emptiness

```
procedure AntB($M_G = (Q, \Sigma_G, G, I_M, \overset{G}{\rightarrow}, F)$)
    Start ← $\{I_M\}$;
    Frontier ← Done ← $\lceil Models(F) \rceil$;
    while Frontier $\neq \emptyset \wedge$ Start $\not\sqsubseteq$ Frontier do
        Frontier ← $\{\rho \in Pre($Frontier$) \mid \rho \not\sqsubseteq$ Done$\}$;
        Done ← Done $\sqcup$ Frontier;
    return (Start $\not\sqsubseteq$ Frontier);
```

After convergence, the antichain Done represents the cases that are not reachable from the initial case and can reach final cases. Its complement is the inductive invariant. Its cardinality is the size of the inductive invariant.

In our case, this implementation has a drawback: the size of $Models(F)$ can be exponential to the input size. We want to minimize the consequences of this exponential boom, and therefore we do a semi-depth-first search instead of the breadth-first search: In the beginning, we add just one model of F into the $\mathsf{Frontier}$ (the witness returned by the SAT solver), then we resolve whole $\mathsf{Frontier}$ as in 1. When the $\mathsf{Frontier}$ is empty, if we still did not find an accepting run, we check if some $\rho \subseteq Q$ exists, such that $\rho \in \mathit{Models}(F) \cap \mathsf{Done}$ (we find such ρ in a similar manner as we find κ in the **Candidate** rule of IIC). If we find such a ρ, we add it to the $\mathsf{Frontier}$, otherwise, we have converged and the AFA is empty. We have also experimented with a complete depth-first search. In this case, the $\mathsf{Frontier}$ is actually a stack. In the line 5 of the algorithm, we only pop one ρ from the stack (such that $\rho \not\sqsubseteq \mathsf{Done}$, compute its predecessors, and push them to the stack. In the line 6, we extend the antichain Done only with ρ instead of whole $\mathsf{Frontier}$.

The forward antichain has been implemented in a manner analoguous to 1. The antichain Done represents the inductive invariant directly, its cardinality is the size of the inductive invariant. As we start with $\{I_M\}$, there is no problem with the exponential boom, the only problem is that we need to solve SAT after each addition of a case to Done. Complete depth-first search could however be beneficial for some AFA.

The experiments in the chapter 5 have been done with depth-first search for both forward and backward antichains. On the benchmarks where the antichains had some problems (the non-flagellar ones), we have tried also the semi-depth-first search and the original breadth-first search versions, their efficiencies were worse. On the flagellar benchmarks, the efficiency was more or less indepent of the type of the search.

Chapter 5

Experimental Evaluation

The experiments were performed on a machine with Intel Core i7 processor with two cores and 16GB of RAM. The choice of the language—Python—has also big impact on the measured times, but it should have no impact on the comparison of antichain and IIC, which are both implemented in Python. We expect that a similar implementation in C++ would be about 60 times faster (for more details, see the appendix A).

5.1 The Artificial Class of AFA: $Primes(n)$

First, we demonstrate on an artificial class of AFA that IIC has a potential to converge faster if a simple inductive invariant exists. Let us introduce a class of AFA $Branch(m)$ by the diagram in Figure 5.1. The guards on the transitions are omitted from the diagram—all of them are equal[1].

We define the $Primes(n)$ class of AFA in the following way. For a given n, let $\pi = 2, 3, 5, 7, 11, \cdots$ be a sequence of the first n prime numbers. The AFA $Primes(n)$ is then conjunction of automata $Branch(\pi_1), \cdots, Branch(\pi_n)$ where q_4 of the kth branch (we can choose k randomly) branches is not a final state. A simple inductive invariant can be found in the AFA—no case that contains states from the kth branch is final and each reachable case contains a state from the kth branch. The antichain algorithms are unable to benefit from this simple invariant. They need to explore all the $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdots \pi_n$ variations of states in the cycles of each branch, to prove that the AFA is empty. On the contrary, the comparison in Figure 5.2 shows that IIC does benefit from the existence of a simple invariant and converges much faster.

[1] Choice of the AFA representations is therefore of no importance—we choose per-symbol AFA (with singleton alphabet).

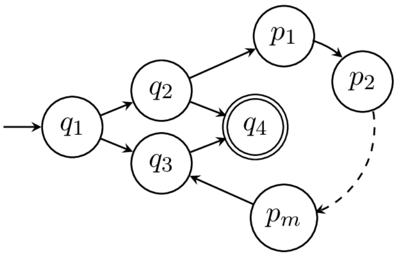


Figure 5.1: The $Branch(m)$ AFA

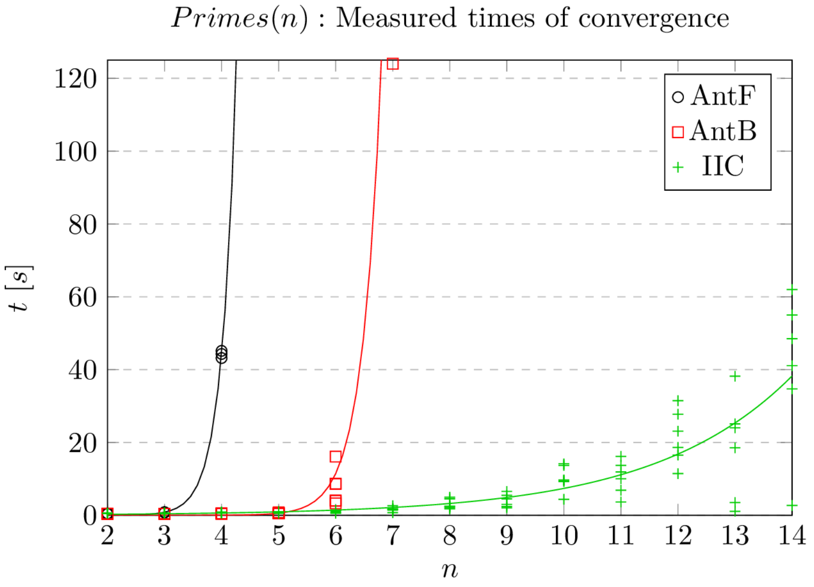


Figure 5.2: Times of convergence for the $Primes(n)$ instances of AFA

The biggest instance of $Primes(n)$ where forward antichain solves this problem in a reasonable time (43 seconds) is for $n = 4$. Backward antichain is better: the maximal n where it reliably does not time out (with a four minute timeout) is $n = 6$, with the times varying in up to 16 seconds. Dominance of IIC on this class of AFA is evident.

5.2 Real World Benchmarks

We have also experimented with real world benchmarks. The authors of the Sloth string solver [13] provided for us a set of succinct AFA used in their experiments[2]. We have converted a random selection of 874 benchmarks to all the three representations and measured the efficiency of IIC and antichain algorithms on deciding their emptiness. We cannot fairly compare our current implementations of IIC and antichains with Sloth, because our implementations are still just unoptimized prototypes, written in Python and not well optimized. We will however provide some interesting results also from this comparison.

We set a two minute timeout for the conversion, as well as for the solving. The conversion to classical AFA (per-symbol and symbol-set representations) performs minterm generation, which can take up to exponential amount of time comparing to the number of transitions in the suAFA. In 238 out of the 874 benchmarks, the minterm

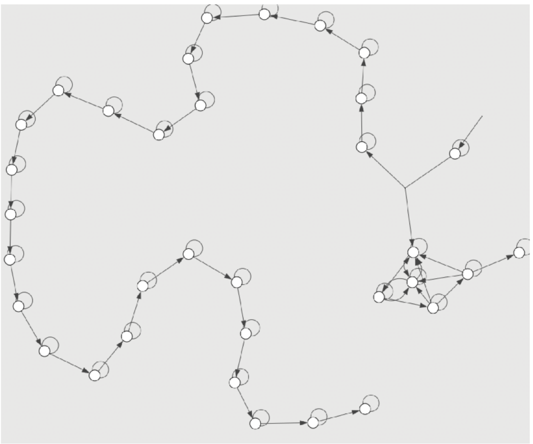


Figure 5.3: Structure of a flagellar AFA
(The symbols are not visualized.)

[2] The AFA extracted from the Stranger project [17], using the *replace* operation

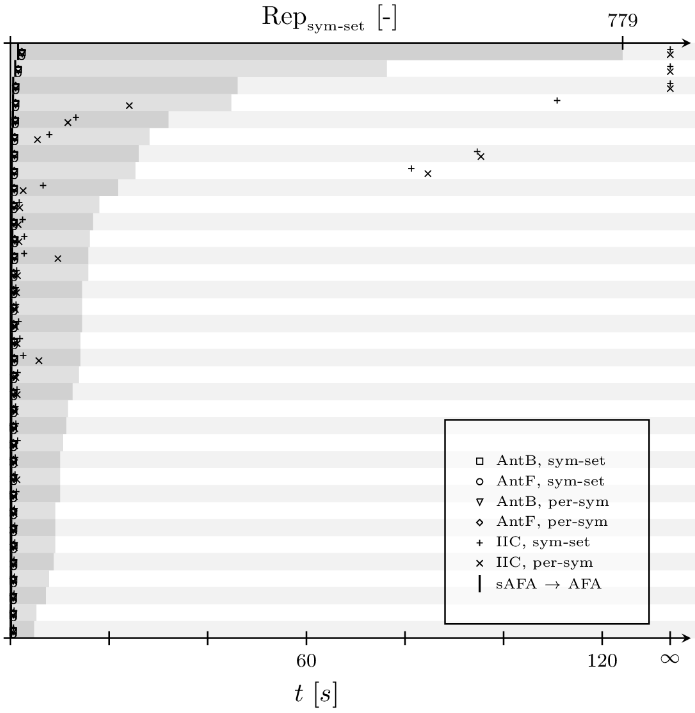
generation timed out. The only possibility would thus be to solve these cases with the symbolic AFA representation. Unfortunately, each of these 238 benchmarks was timing out also when solving using the symbolic AFA (we will see later in the measurements that the performance of solving symbolic AFA is very poor in comparison with symbol-set and per-symbol equivalents). From the remaining 636 cases, the antichain algorithms performed very well in 601 of the benchmarks—all the four versions (forward/backward antichain on per-symbol/symbol-set AFA) terminated in less than 2 seconds. An overwhelming majority of these cases shared a similar structure, outlined in Figure 5.3. In the sequel, let us call them *flagellar* AFA, due to their visual resemblance to flagellata (the microorganisms). It can be seen that the initial case has two states that are parts of two disjoint branches—a long one, with no cycles, and a short, more complex one, but small. The language of all flagellar AFA from the benchmarks we have is empty.

The structure of flagellar AFA is very convenient for the antichain algorithms and IIC thus cannot win there. We have even found four instances of flagellar AFA where SLOTH timed out while antichains did solve them smoothly. As for IIC, the efficiency depends on the result of the first generalization. There is some randomization[3] in our minimum hitting set greedy algorithm. Sometimes the first generalization leads the algorithm to a "good way"—the first added blocker is $Q \setminus \{q\}$ where q is the first state from the long branch. Other time, the first blocker lacks the first state from the short branch and the IIC then progresses in a non-optimal way (the whole short branch needs to be solved before each proceeding one state further in the long branch). A simple heuristic could be implemented to force the IIC to pick the good way, but actually, flagellar automata were not so interesting for us because they can be already efficiently solved by antichain algorithms, hence we rather concentrate on other classes of examples. We can observe on the bigger benchmarks in Figure 5.4 that some of the IIC times are much higher than the others—this is caused by the random pick of the first blocker.

Figure 5.4 shows the measured times for antichains (forward, ANTF, and backward, ANTB) and IIC, on a random selection of 35 of the flagellar benchmarks. Each row of the graph represents one benchmark. The gray bar indicates one of the metrics of AFA that can be considered to represent the AFA "size"—the number of transitions (cardinality) in the symbol-set (or smybolic, it is the same) AFA representation. Other metrics are presented by numbers to the right of the table: number of AFA states $|Q|$, number of minterms obtained in the conversion to classical AFA (the alphabet size $|\Sigma_M|$), number of variables in the formulae of the symbolic AFA $|V|$, and again the cardinality of the symbolic (or symbol-set) representation $|\text{Rep}|$ (it is the same value as is visualized by the gray bar). The data points show the measured times for individual algorithms and AFA representations.

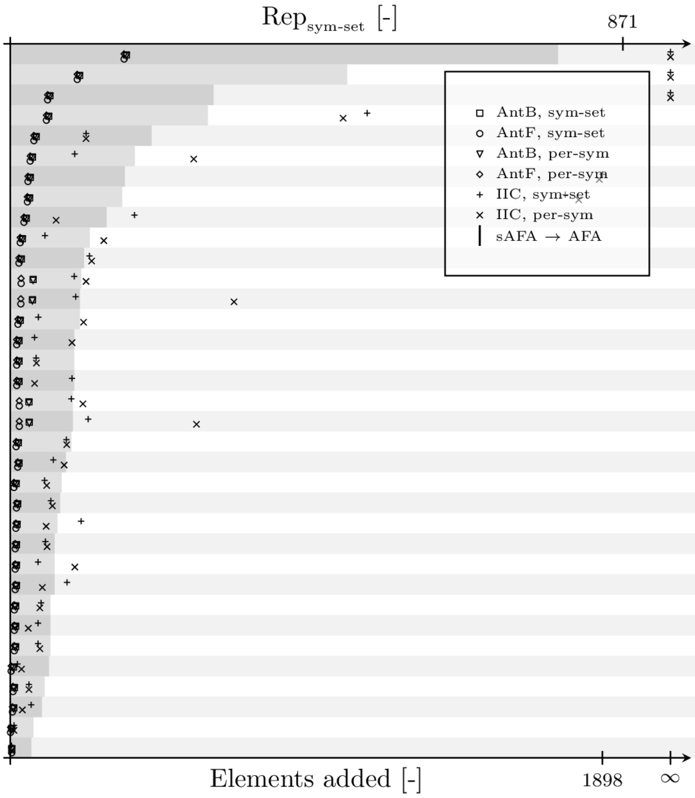
Figure 5.5 shows the numbers of additions of a case into the antichain, or for IIC, the number of blocker additions—it shows the number of changes of the main part of the algorithm state. Figure 5.6 shows the size of the antichain (or the number of blockers, respectively) in the end of the algorithm—the size of the discovered inductive invariant. In all the charts, the antichain-based algorithms consistently outperform IIC. Note that we have omitted symbolic representations of AFA from the measurements—the algorithms ran many times slower than the ones for the classical AFA because of the expensive multiple calls of the SAT solver at each minterm generation in the computation of *pre*.

[3] At each stage, if more than one state hits the maximal number of cases, a random one of them is added to the minimum hitting set.



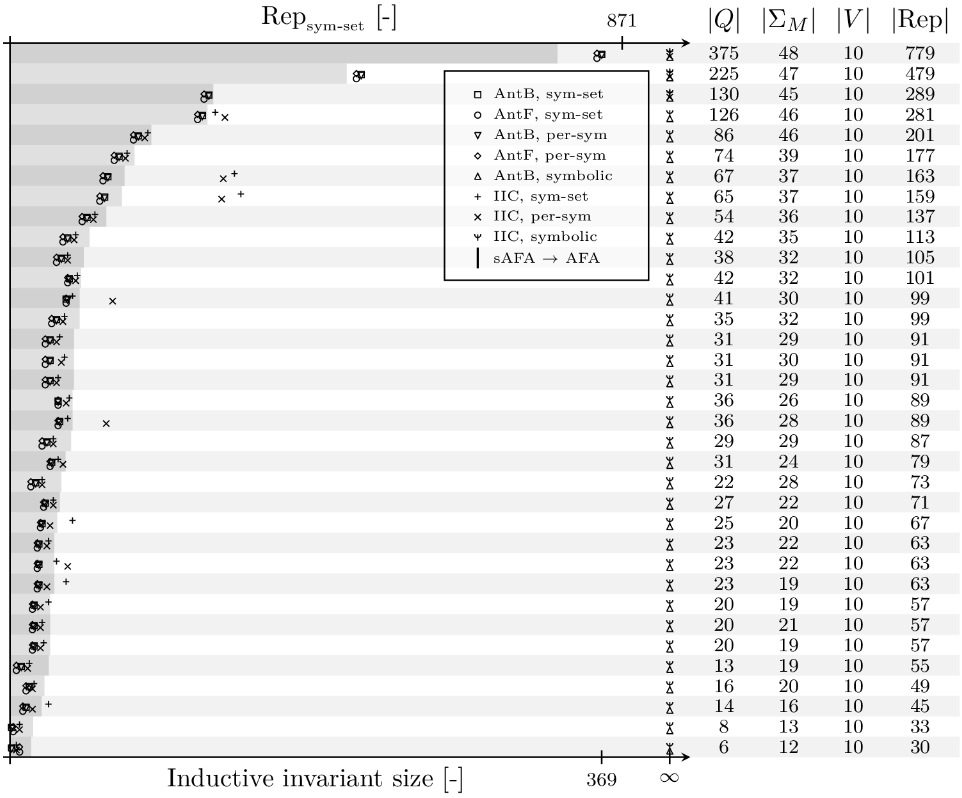
$\lvert Q\rvert$	$\lvert\Sigma_M\rvert$	$\lvert V\rvert$	$\lvert\text{Rep}\rvert$
375	48	10	779
225	47	10	479
130	45	10	289
126	46	10	281
86	46	10	201
74	39	10	177
67	37	10	163
65	37	10	159
54	36	10	137
42	35	10	113
38	32	10	105
42	32	10	101
41	30	10	99
35	32	10	99
31	29	10	91
31	30	10	91
31	29	10	91
36	26	10	89
36	28	10	89
29	29	10	87
31	24	10	79
22	28	10	73
27	22	10	71
25	20	10	67
23	22	10	63
23	22	10	63
23	19	10	63
20	19	10	57
20	21	10	57
20	19	10	57
13	19	10	55
16	20	10	49
14	16	10	45
8	13	10	33
6	12	10	30

Figure 5.4: Time efficiency of the algorithms for flagellar AFA



$\lvert Q\rvert$	$\lvert\Sigma_M\rvert$	$\lvert V\rvert$	$\lvert\text{Rep}\rvert$
375	48	10	779
225	47	10	479
130	45	10	289
126	46	10	281
86	46	10	201
74	39	10	177
67	37	10	163
65	37	10	159
54	36	10	137
42	35	10	113
38	32	10	105
42	32	10	101
41	30	10	99
35	32	10	99
31	29	10	91
31	30	10	91
31	29	10	91
36	26	10	89
36	28	10	89
29	29	10	87
31	24	10	79
22	28	10	73
27	22	10	71
25	20	10	67
23	22	10	63
23	22	10	63
23	19	10	63
20	19	10	57
20	21	10	57
20	19	10	57
13	19	10	55
16	20	10	49
14	16	10	45
8	13	10	33
6	12	10	30

Figure 5.5: Number of blockers (or antichain cases) added, for flagellar AFA



$\lvert Q\rvert$	$\lvert\Sigma_M\rvert$	$\lvert V\rvert$	$\lvert\text{Rep}\rvert$
375	48	10	779
225	47	10	479
130	45	10	289
126	46	10	281
86	46	10	201
74	39	10	177
67	37	10	163
65	37	10	159
54	36	10	137
42	35	10	113
38	32	10	105
42	32	10	101
41	30	10	99
35	32	10	99
31	29	10	91
31	30	10	91
31	29	10	91
36	26	10	89
36	28	10	89
29	29	10	87
31	24	10	79
22	28	10	73
27	22	10	71
25	20	10	67
23	22	10	63
23	22	10	63
23	19	10	63
20	19	10	57
20	21	10	57
20	19	10	57
13	19	10	55
16	20	10	49
14	16	10	45
8	13	10	33
6	12	10	30

Figure 5.6: Size of the inductive invariant, for flagellar AFA

In addition to the 601 benchmarks that were trivial for antichains, we have found a set of 35 benchmarks, where the global minterm generation did not time out, but antichains did not solve them in 2 seconds. Three of these automata were actually flagellar (the measurements of the tree can be seen on the top of Figure 5.8), they only were too big to be solved by antichains in less than 2 seconds. The rest had more varying and interesting structures, e.g. see Figure 5.7. All of the *non-flagellar* AFA were non-empty, therefore we could not measure times of convergence of the algorithms. It can be observed in Figure 5.8, that the symbol-set IIC wins on a majority of these examples, and if it does not win, it is not far behind the winner. This is an interesting result that shows the practical potential of IIC in the problem of deciding the AFA emptiness.

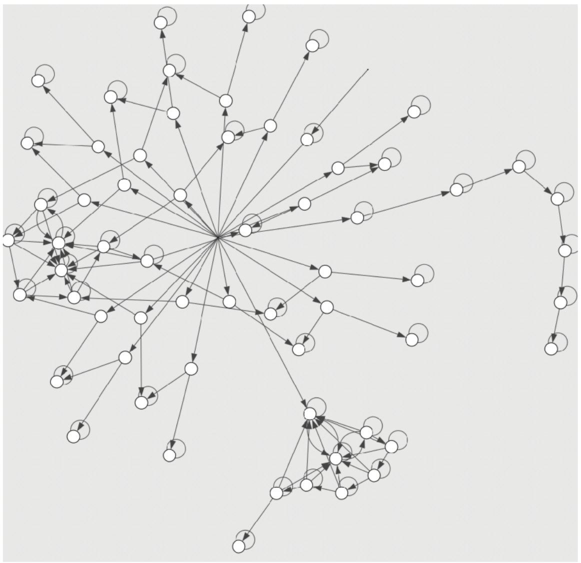
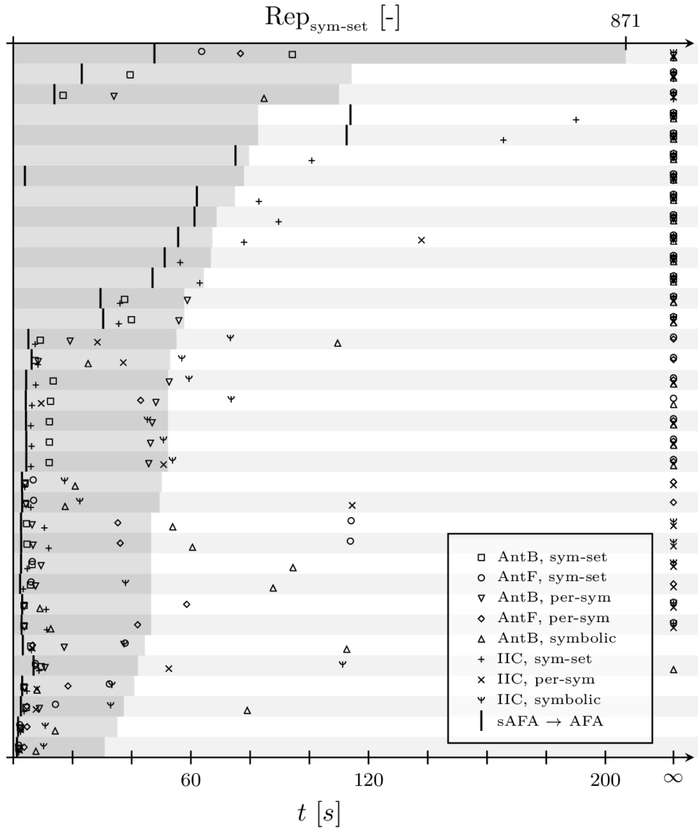


Figure 5.7: Non-flagellar AFA

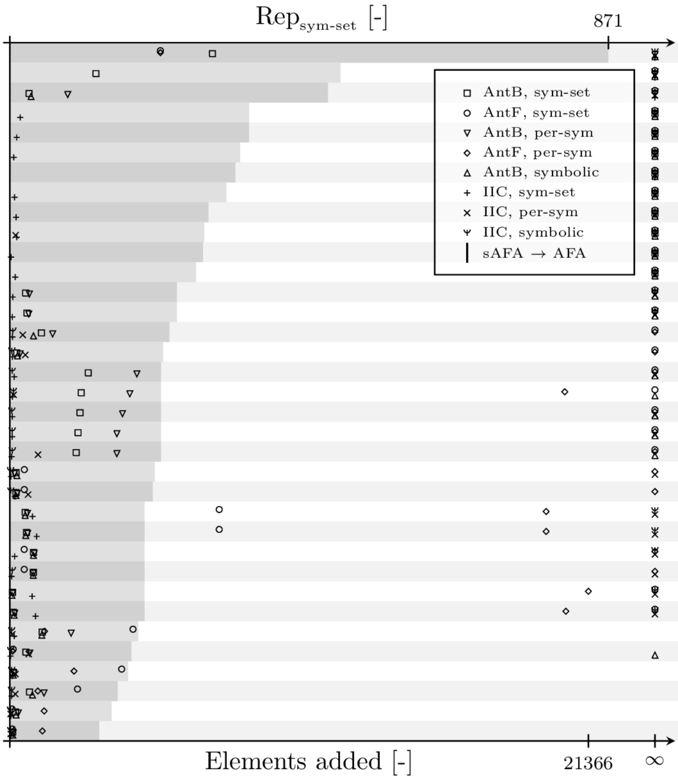
The overall numbers of algorithm state modifications (blocker additions for IIC, case additions for antichain) is visible in Figure 5.9. The size of the discovered inductive invariant (number of blockers/antichain cases in the end of the algorithms) is shown in Figure 5.10. The gap between the symbol-set IIC and antichains is even more apparent on these two metrics.

Note that the implementations working with symbolic AFA are much slower than the symbol-set implementations. When using the per-symbol AFA representation, due to its simplicity, the efficiency is sometimes better than for the symbol-set one, usually if analyzing small AFA.



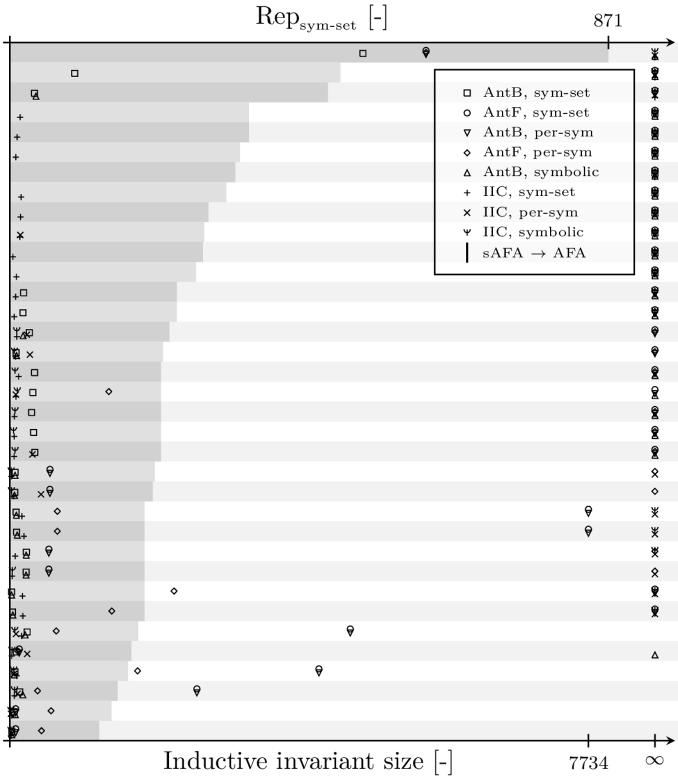
$\lvert Q\rvert$	$\lvert\Sigma_M\rvert$	$\lvert V\rvert$	$\lvert\text{Rep}\rvert$
216	817	28	871
153	868	19	481
147	579	19	463
103	6878	28	348
103	6878	28	348
98	5617	28	335
104	341	19	328
93	4993	28	315
89	4369	28	289
87	3953	28	283
86	3953	28	281
83	3758	28	271
75	2705	28	243
75	2705	28	243
70	392	19	232
67	409	19	223
66	358	19	220
66	358	19	220
66	358	19	220
66	358	19	220
66	358	19	220
63	375	19	211
62	358	19	208
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
59	287	19	187
35	951	37	177
54	274	19	172
49	261	19	157
46	222	19	148
40	196	19	130

Figure 5.8: Time efficiency of the algorithms for non-flagellar AFA



$\lvert Q\rvert$	$\lvert\Sigma_M\rvert$	$\lvert V\rvert$	$\lvert\mathrm{Rep}\rvert$
216	817	28	871
153	868	19	481
147	579	19	463
103	6878	28	348
103	6878	28	348
98	5617	28	335
104	341	19	328
93	4993	28	315
89	4369	28	289
87	3953	28	283
86	3953	28	281
83	3758	28	271
75	2705	28	243
75	2705	28	243
70	392	19	232
67	409	19	223
66	358	19	220
66	358	19	220
66	358	19	220
66	358	19	220
66	358	19	220
63	375	19	211
62	358	19	208
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
59	287	19	187
35	951	37	177
54	274	19	172
49	261	19	157
46	222	19	148
40	196	19	130

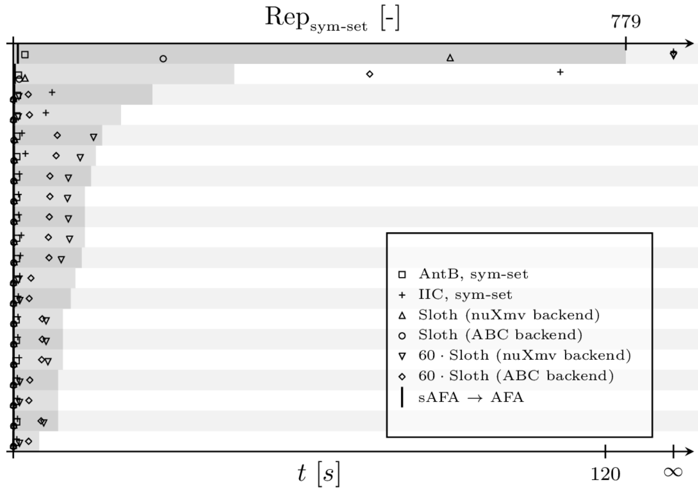
Figure 5.9: Number of blockers (or antichain cases) added, for non-flagellar AFA



$\lvert Q\rvert$	$\lvert\Sigma_M\rvert$	$\lvert V\rvert$	$\lvert\mathrm{Rep}\rvert$
216	817	28	871
153	868	19	481
147	579	19	463
103	6878	28	348
103	6878	28	348
98	5617	28	335
104	341	19	328
93	4993	28	315
89	4369	28	289
87	3953	28	283
86	3953	28	281
83	3758	28	271
75	2705	28	243
75	2705	28	243
70	392	19	232
67	409	19	223
66	358	19	220
66	358	19	220
66	358	19	220
66	358	19	220
66	358	19	220
63	375	19	211
62	358	19	208
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
59	287	19	187
35	951	37	177
54	274	19	172
49	261	19	157
46	222	19	148
40	196	19	130

Figure 5.10: Size of the inductive invariant, for non-flagellar AFA

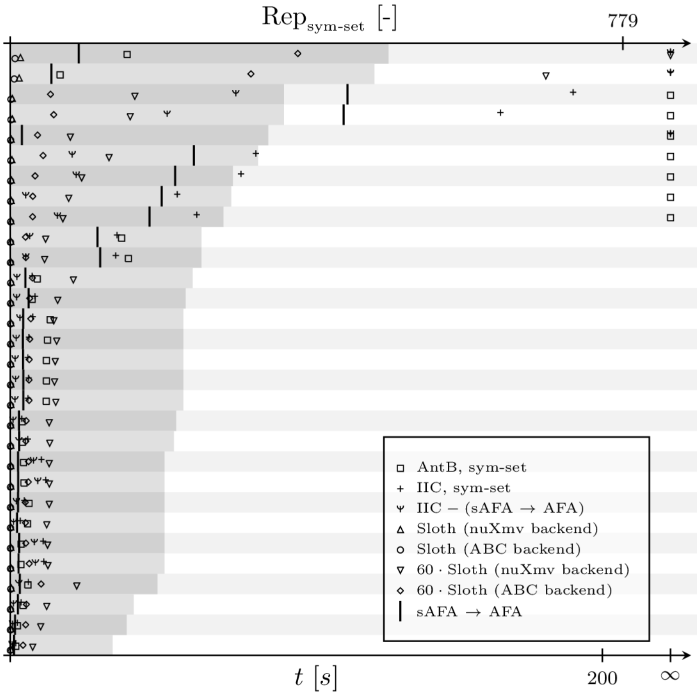
Finally, we present the comparison with the times from the Sloth project. The comparison could not be done in a fair way because our implementation of IIC is still an unoptimized prototype written in Python while Sloth is built over a fine tuned industrial strength tool ABC [3] written in C. Based on the measurement discussed in the appendix A, we expect that a similar implementation in C++ would be about 60 times faster. The comparison therefore includes also times for Sloth, multiplied by 60. These times will be denoted as *artificially slowed-down*. Obviously, this is not fair too, but it gives us a more realistic image about the comparison. In Figure 5.11, we can see that antichain-based algorithms are sometimes better than Sloth in case of flagellar AFA, even if implemented in Python. The IIC is mostly better than the artificially slowed-down Sloth on the flagellar AFA.



$\lvert Q\rvert$	$\lvert\Sigma_M\rvert$	$\lvert V\rvert$	$\lvert\mathrm{Rep}\rvert$
375	48	10	779
126	46	10	281
74	39	10	177
54	36	10	137
42	35	10	113
38	32	10	105
35	32	10	99
31	29	10	91
31	30	10	91
31	29	10	91
29	29	10	87
31	24	10	79
22	28	10	73
23	22	10	63
23	22	10	63
23	19	10	63
20	19	10	57
20	21	10	57
20	19	10	57
8	13	10	33

Figure 5.11: Time comparison of antichain and IIC with Sloth for flagellar AFA

Figure 5.12 is more interesting for us. Unfortunately, the measured instances of non-flagellar AFA appear to be quite easily solved by Sloth. As for the bigger instances of the non-flagellar benchmarks (which were the most interesting ones in comparison with the antichain), our implementation of IIC loses even with the artificially slowed-down Sloth. Again, this may or may not be a fair result. We believe that it is possible to work more effectively with symbolic AFA by a use of caching of SAT results and by reusing similar minterm trees from older minterm generations, instead of computing minterms from scratch in each backward transition function. If we achieved similar efficiency as for symbol-set AFA, the conversion from sAFA to AFA could be avoided. We therefore present also the times without the conversion, which are already comparable with the artificially slowed-down Sloth. Anyway, the most interesting comparison would be on benchmarks that are challenging for both antichain and Sloth. Such instances are unfortunately not present in our set of benchmarks so far.



$\lvert Q\rvert$	$\lvert\Sigma_M\rvert$	$\lvert V\rvert$	$\lvert\mathrm{Rep}\rvert$
153	868	19	481
147	579	19	463
103	6878	28	348
103	6878	28	348
104	341	19	328
93	4993	28	315
87	3953	28	283
86	3953	28	281
83	3758	28	271
75	2705	28	243
75	2705	28	243
70	392	19	232
67	409	19	223
66	358	19	220
66	358	19	220
66	358	19	220
66	358	19	220
66	358	19	220
63	375	19	211
62	358	19	208
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
58	307	19	196
59	287	19	187
49	261	19	157
46	222	19	148
40	196	19	130

Figure 5.12: Time efficiency of antichain and IIC with Sloth for non-flagellar AFA

Chapter 6

Conclusions and Future Work

We have successfully specialized the IIC algorithm for AFA emptiness problem and created a working implementation of the specialized algorithm. We have observed the impact of various design decisions and optimization on the performance of the algorithm, namely symbolic representation of the symbols on the transitions (symbolic AFA), set representation of the symbols on the transitions (symbol-set AFA, in contrast with the simple per-symbol AFA, which has one symbol at each transition), generalization, future induction, various implementations of the backward transition function. For the practical cases, the most performing version of IIC was the version with the set representation of the symbols on transitions (symbol-set AFA) that used generalization, future induction and the improved implementation of the backward transition function (proposed in the section 3.7).

The real-world benchmarks were provided to us in the form of succinct AFA, which we were translating to symbolic AFA (the translation algorithm is our contribution as well). We could work with the symbolic AFA directly, or translate them further to classical AFA for a faster analysis with minimal number of calls of a SAT solver (it is called only for discerning final cases). The versions of IIC implementation which worked with the classical AFA (symbol-set and per-symbol AFA) performed much better than the version working with the symbolic AFA. On the other hand, the translation from symbolic AFA was a big performance bottleneck, which disallowed these versions to even start for a significant number of the benchmarks.

One of our main goals was to compare efficiency of our specialization of the IIC algorithm with the antichain-based algorithms [9]. We have implemented the forward and backward antichain algorithms and found a class of AFA with a simple inductive invariant where IIC performed much better than the two antichain algorithms. From the results of measurements on the real-world benchmarks, it is apparent that IIC often outperforms antichains if the structure of the AFA is complex.

However our implementations of the algorithms are still unoptimized prototypes, we have made an efficiency comparison with the Sloth implementation, which is described in [13]. The results of the comparison itself were not very favourable to IIC. However, after some hypothetical adjustments of the results, based on simple measurements of how well could an optimized version of our IIC implementation perform if written in C++, we have obtained results that are comparable with Sloth. We know that no real conclusions can be based on these optimistic adjustments, but it gives some light into the research, and primarily, it gives us some indications about which way to continue the work. For example, we can see that we should concentrate on the symbolic AFA analysis because the translation from sAFA to AFA is too expensive.

We have some ideas for the nearest future goals. First of all, it would be favourable to gather more benchmarks from various application domains. The real-world benchmarks on which we have performed our comparison were all extracted from the same class of the problem and were very easy even for the antichains or for Sloth. The IIC thus did not have chance to significantly win over the other algorithms. The improvement of the symbolic AFA analysis would be also beneficial for this reason because more benchmarks could be analyzed—it would be not necessary to rely on the feasibility of the conversion from sAFA to AFA. Various heuristics can be invented and added, which could help in some practical cases. For example, we have implemented a heuristic that leads the generalization in a way that causes faster convergence of the IIC for flagellar automata. It was however not worth mentioning in the description of the experimental evaluation as the IIC was still much slower than antichain. Symbolic representation of the state space could be also an interesting topic for research. Finally, to be able to fairly compare the IIC with other algorithms, it is necessary to optimize the implementation and primarily, reimplement it in a language that is designed to compile to a high-performance machine code.

Bibliography

[1] Abdulla, P. A.; Chen, Y.-F.; Holík, L.; et al.: When Simulation Meets Antichains. In *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2010. pp. 158–174.

[2] Bradley, A. R.; Manna, Z.: Checking Safety by Inductive Generalization of Counterexamples to Induction. In *Formal Methods in Computer Aided Design, 2007. FMCAD '07*. Nov 2007. pp. 173–180.

[3] Brayton, R.; Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2010. pp. 24–40.

[4] Büchi, J. R.: *Weak Second-Order Arithmetic and Finite Automata*. New York, NY: Springer New York. 1990. pp. 398–424.

[5] Chandra, A. K.; Stockmeyer, L. J.: Alternation. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 1976. ISSN 0272-5428. pp. 98–108.

[6] Chvátal, V.: A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*. vol. 4, no. 3. 1979: pp. 233–235.

[7] D'Antoni, L.; Kincaid, Z.; Wang, F.: A Symbolic Decision Procedure for Symbolic Alternating Finite Automata. *ArXiv e-prints*. October 2016. 1610.01722.

[8] D'Antoni, L.; Veanes, M.: Minimization of Symbolic Automata. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM. 2014. pp. 541–553.

[9] De Wulf, M.; Doyen, L.; Henzinger, T. A.; et al.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2006. pp. 17–30.

[10] De Wulf, M.; Doyen, L.; Maquet, N.; et al.: Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking. In *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2008. pp. 63–77.

[11] Ganty, P.; Maquet, N.; Raskin, J.-F.: Fixed point guided abstraction refinement for alternating automata. *Theoretical Computer Science*. vol. 411, no. 38. 2010: pp. 3444 – 3459. implementation and Application of Automata (CIAA 2009).

[12] Hoder, K.; Bjørner, N.: Generalized Property Directed Reachability. In *Theory and Applications of Satisfiability Testing – SAT 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2012. pp. 157–171.

[13] Holík, L.; Janků, P.; Lin, A. W.; et al.: String constraints with concatenation and transducers solved efficiently. In *Proceedings of the ACM on Programming Languages, Article 4*, vol. 2. POPL. 2018.

[14] Jančar, P.; Sawa, Z.: A note on emptiness for alternating finite automata with a one-letter alphabet. *Information Processing Letters*. vol. 104, no. 5. 2007: pp. 164 – 167.

[15] Kloos, J.; Majumdar, R.; Niksic, F.; et al.: Incremental, Inductive Coverability. In *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2013. pp. 158–173.

[16] de Moura, L.; Bjørner, N.: Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2008. pp. 337–340.

[17] Yu, F.; Alkhalaf, M.; Bultan, T.: STRANGER: An Automata-based String Analysis Tool for PHP. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'10. Berlin, Heidelberg: Springer-Verlag. 2010. pp. 154–157.

Appendix A

Implementation details

This appendix talks about implementation details of the IIC and antichain algorithms, choice of SAT solver, and input format of the implementation.

A.1 Programming language choice

The implementation language of the IIC and antichain algorithms was Python, due to its flexibility and easiness of fast development. To increase performance, we have rewritten an early version of the implementation to C++. We have measured that the C++ implementation was about 60 times faster than the corresponding version of the Python implementation. It was using per-symbol representation of AFA and it supported the description of final cases only by a purely conjunctive formula. Therefore, no SAT solving had to be performed. There were many other differences with the implementation described in this work, but we do not consider them relevant for the proportion between C++ and Python performances. From that time, we have abandoned the C++ implementation, as it was hard to maintain. However, we can roughly expect that the performance of the current version could be 60 times better, after reimplementing to C++. Obviously, since the main performance bottleneck of the symbolic AFA is SAT solving, we can expect this acceleration only for the symbol-set and per-symbol AFA.

A.2 Input format

The input for the algorithm is given directly as a Python code. We represent cases as bitvectors (we have also tried using the Python's built-in `set`, but the difference in performance was negligible, so we did not continue to maintain this representation). The format is dependent on the representation of AFA. To give an image, we provide a simple example of the input for symbol-set AFA. As explained in the end of the description of the **Candidate** transition rule, the literals in the formula F are negated. The following code represents the AFA from the example 3.1.

```
from z3 import And, Bool

N = 6 # Number of states.
I = 0b0001 # Initial case.
N_MTS = 2 # Size of the alphabet.

qs = tuple(Bool(f'q{i}') for i in range(N)) # Comprehension of all states.

# Definition of final states, with negated literals - this one represents
# F = ¬q0 ∧ ¬q1 ∧ ¬q2
F = And(qs[0], qs[1], qs[2])

# For each state q, we have tuples of symbol set S and cases rho, such that
# (q, S, rho) in Rep.
POST = [
    [ ({0}, [0b0110]) ],
    [ ({0}, [0b1000]) ],
    [ ({1}, [0b1000]) ],
    [],
]
```

A.3 Translation of real world benchmarks

The authors of Sloth [13] has provided us benchmarks they used for the comparison with CVC4 and S3P tools. We have used the Haskell language to implement the parser of these benchmarks, as well as translators to the three of our representations of AFA (including the suAFA to sAFA/AFA conversion). The performance bottleneck of this translation is again in the SAT solving, which is used for the translation to per-symbol and symbol-set AFA. Minterms for bigger benchmarks could not be generated in time (2 minutes), so they were analysed only as symbolic AFA. None of them could be solved in time neither by the algorithms using symbolic AFA.

A.4 SAT solving

We use the Z3 as a SAT solver [16], which appeared to be the best choice after some recherche and experimentation. Its application in our implementation takes place in translation (section A.3), final case detection (the **Unfold**/**Candidate** transition rules) and minterm generation in 3.7.

A.5 Visualization of algorithm state

To help us analyze IIC and antichains, we have implemented a simple GUI that shows the AFA and displays the various types of cases in the algorithm state (blockers, candidates, *pre*, forbidden cases, antichain elements . . .) in various colors. We have used GTK and the *igraph* library to display the AFA. The GUI helped us find out that many of the benchmarks share very similar structure. For example, the figure A.1 shows the GUI with the *Primes*(5) AFA displayed in a certain state of the IIC. At some point of the run of the IIC, it shows a check that a predecessor (green) of a candidate (orange) is blocked by a blocker (pink).

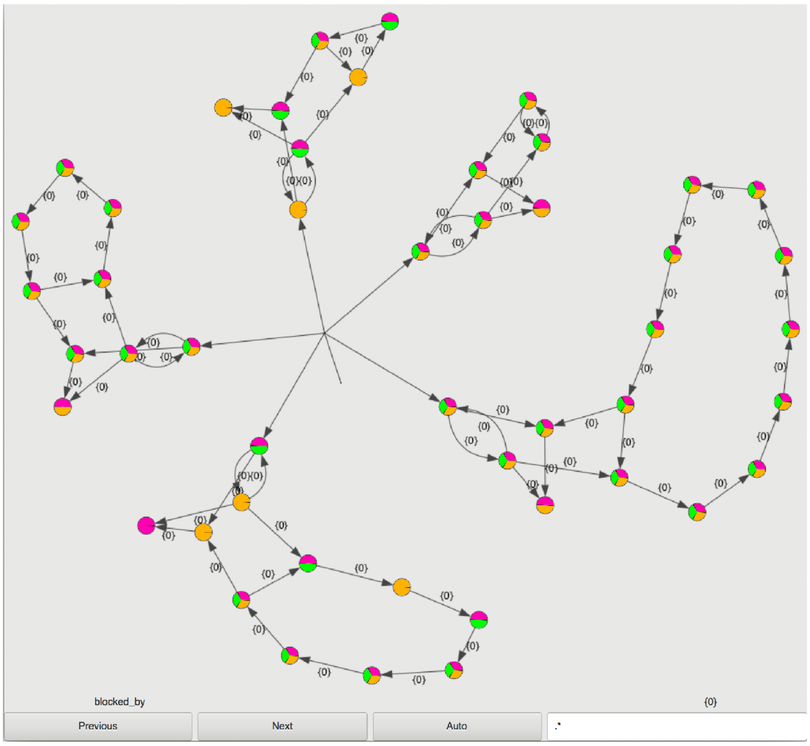


Figure A.1: Visualisation GUI (*Primes*(5))