Finite Automata and Their Applications in Formal Verification

Theory of Programming Languages, 2018/2019

Pavol Vargovčík
ivargovcik@fit.vutbr.cz

Abstract

This recherche summarizes various forms of finite automata, which leverage succinctness and applicability of finite automata, in comparison with the explicit deterministic finite automata. Discussed are common difficulties that arise when using the succinct forms instead of the explicit one along with state-of-the-art methods that overcome the difficulties on many practical instances of automata. Finally, modern practical applications in network security, verification and logics are presented.

1 Introduction

Finite automata are one of the fundamental concepts of the automata theory, which has already evolved for more than 60 years. The finite automata have found most of their early interest and practical use in lexical analysis (used most notably in compilers) and later in verification (LTL model checking, WS1S model checking).

Common practice is to compress the automata in a way that is convenient for ameliorating the performance of logical operations on the languages of the automata, resulting in more natural representation and better time and space complexity of automaton synthesis for certain classes of applications. Several forms of finite automata have been investigated that have equal expressive power but differ in succinctness, most notable of which are deterministic, nondeterministic, alternating automata, automata with finite counters and Boolean transition systems. These succinct forms enable a number of real-world problems to be modelled by finite automata. However, cost for these benefits is considerable — theoretical bounds of computational complexity for deciding classical problems (e.g. language containment, emptiness and universality) is increased, often superpolynomially.

Despite the theoretical bounds of the automata analysis, new techniques are being investigated, making compromises between succinctness and simulation time. A significant amount of research is also dedicated to techniques that work effectively for certain fragments of the problems, which opens space for practical applicability of finite automata in e.g. xml compression [29], business process analysis [63], network intrusion detection [6, 39], DNA synthesis [40], string program verification [32].

In addition to the applications resulting from introduction of the succinct forms, the finite automata theory is further exploited in new related models of automata: finite weighted [24], asynchronous [66], probabilistic [51], timed [4] or tree [44] automata, research of which

is driven by new practical applications, e.g. image compression [19], synthesis of distributed algorithms [61], cursive handwriting recognition [56], verification of systems with microcontrollers [23], shape analysis [17].

2 Models of Finite Automata

Automata has been studied for a long time. Initial and most simple definition is what we nowadays call deterministic finite automata. Due to explosion of automaton state space for even small practical problems, other more succinct forms with equal expressive power exist. This section introduces the concept of deterministic, nondeterministic, alternating and symbolic automata, which all have equal expressive power (they are reducible among each other), along with short discussion about their applications. All of these forms of automata process words (strings of symbols from a finite alphabet Σ) and decide containment of the processed word in a regular language given by the automaton. They are therefore also called *word* automata. A word automaton represents a language — a possibly infinite set of words. It can be viewed as a complete function with signature $\Sigma^* \to \{0, 1\}$.

In the end of the section, the concept of tree automata is shown, which have found their application e.g. in shape analysis. Tree automata are a generalization of word automata, in a way that they process trees instead of words (words are isomorphic to unary trees).

A comprehensive reference, which discusses these forms (and also some applications from the section 4) in further detail, is [25].

2.1 Deterministic Finite Automata

Deterministic finite automaton (DFA) is defined as a tuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of DFA *states*;
- Σ is a finite *alphabet*;
- $\delta : Q \times \Sigma \to Q$ is a deterministic *transition function*;
- $q_0 \in Q$ is an *initial state*;
- $F \subseteq Q$ is a set of *final states*.

A *configuration* of a DFA is a tuple of a state and a word that is left to be processed by the automaton $\kappa \in Q \times \Sigma^*$. A *transition* exists between two configurations (denoted as $\kappa \mapsto \kappa'$) iff $\kappa = (q, aw)$, $\kappa' = (q', w)$ and $q' = \delta(q, a)$. In other words, the automaton $\mathcal{M}$ processes the first symbol of the input word and transitions from its current state q to the next state q' according to its transition function. For simplicity, let us assume that the automaton is *complete*, i.e. its transition function is complete[1].

A *run* of a DFA is a sequence of configurations, transitions between which exist $\kappa_1 \mapsto \kappa_2 \mapsto \cdots \mapsto \kappa_n$. Iff $\kappa_n = (q_F, \epsilon)$, such that $q_F \in F$, the run is *finalizing*. Iff $\kappa_1 = (q_0, w)$, the run is *initial*. A run that is both initial and finalizing is *accepting*. It is known that for a given deterministic automaton $\mathcal{M}$ and an arbitrary word w, only one run exists, starting

[1] Definitions of incomplete automata exist but they are trivially convertible to the complete ones. They are therefore omitted from this text

with $(q0, w)$. If the run is accepting, we say that $\mathcal{M}$ accepts w, otherwise we say that $\mathcal{M}$ refuses w.

A language of an automaton $\mathcal{L}(\mathcal{M})$ is a set of all words accepted by the automaton $\mathcal{M}$:

$$\mathcal{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \mathcal{M} \text{ accepts } w\}$$

Deterministic finite automaton has no memory for the computation history — it acts depending only on the current state and the current symbol on the input. If we wanted to simulate the computation of a standard computer, we would need to encode the actual state of the computer's memory into the state of DFA. Even with small memories, the number of states would be unfeasibly huge – 2^n, where n is the number of bits in the memory. Therefore, the practical use of DFA is targeted at other, specialized domains. The domains where DFA find their usability include among others regular expressions (usually in combination with NFA) [6, 39], modelling some of the structural constraints in DNA [40], image compression [19] (with weighted DFA) or texture generation [53].

The advantage of using DFA is fast simulation, language complementation and analysis of language emptiness and universality. DFA however suffers from some major deficits. Firstly, they cannot model systems with infinite number of states — models like pushdown automata or turing machines are used for this purpose, which are out of the scope of this text. Secondly, the number of states rapidly grows when trying to apply set operations (union, intersection, complement) on languages of multiple automata, or trying to model some basic concepts that are nondeterministic by their nature, e.g. iteration in string theory.

2.2 Nondeterministic Finite Automata

Nondeterministic finite automaton (NFA) is defined as a pentuple $\mathcal{M} = (Q, \Sigma, \delta, Q_0, F)$ where

- Q is a finite set of states;
- Σ is a finite alphabet;
- $\delta : Q \times \Sigma \to 2^Q$ is a nondeterministic transition function;
- $Q_0 \subseteq Q$ is a set of initial states;
- $F \subseteq Q$ is a set of final states.

The definition differs from DFA in the definition of transition function and set of initial states.

A *configuration* of a NFA is defined in the same way as for DFA. A *transition* exists between two configurations (denoted as $\kappa \mapsto \kappa'$) iff $\kappa = (q, aw)$, $\kappa' = (q', w)$ and $q' \in \delta(q, a)$. The difference from the DFA is that being in a state q, processing the symbol a, there are more possibilities of a next state q'. This is the first (and the main) source of nondeterminism.

A *run* of a NFA is a sequence of configurations, transitions between which exist $\kappa_1 \mapsto \kappa_2 \mapsto \cdots \mapsto \kappa_n$. The run is initial iff $\kappa_1 = (q_0, w)$, such that $q_0 \in Q_0$. The conditions for finalizing and accepting run remain the same as for DFA. The second source of nondeterminism is visible here — there are multiple initial states q_0. For an arbitrary fixed nondeterministic automaton $\mathcal{M}$ and an arbitrary fixed word w, multiple runs may exist,

some of which may be accepting and the other ones non-accepting (this property is called *ambivalence*). If any of these runs is accepting, we say that $\mathcal{M}$ accepts w, otherwise $\mathcal{M}$ refuses w.

We say that a transition from a configuration $\kappa = (q, aw)$ is *existential* — to claim that a run from κ is accepting a state $q' \in \delta(q, a)$ must *exist*, such that a run from $\delta(q', w)$ is finalizing.

Language of a nondeterministic automaton is defined analogically to the definition for DFA.

Each DFA can be trivially converted to language-equivalent NFA by setting $Q_0 = \{q_0\}$ and setting the resulting values of the nondeterministic transition function to singletons of the deterministic transition's resulting values. The inverse conversion is not so easy: well-known *subset construction* algorithm is used, which produces DFA with the number of states upper-bounded by 2^n, where n is the number of NFA states. This blow-up of states is called *exponential boom*.

NFA are better at modelling algorithms with nondeterministic nature, e.g. pattern matching of regular expressions such as `"prefix.*some.*thing.*suffix"` or deciding subset containment in a set-trie [57]. Moreover, NFA accepting union of languages of two NFA can be constructed in linear time to the size of the two NFA accepting the operands of the union. Language emptiness is, as well as for DFA, decided in linear time to the size of the automaton — it is sufficient to checggk existence of a path in the automaton's graph from an initial to a final state (using e.g. depth-first search algorithm from initial states). Language universality is however PSpace-complete.

Language complement is still an issue — it is proved [34] that no algorithm can be found, computational complexity of which is generally better than determinization and subsequent DFA complementation. Therefore, the complement is $O(2^n)$ in space and time. Problematic is also language intersection. The intersection automaton is created using a so-called cross-product construction and its time and space complexity is $O(n_1 \cdot n_2 \cdot \cdots \cdot n_m)$, i.e. for m automata which are roughly of the same size n, it is $O(n^m)$. Solving the containment problem may also suffer from performance issues, as nondeterministic transition does not lead to a single state, but instead a track of multiple states must be maintained and when performing a transition, transitions from all of the maintained states must be performed. As a compromise between the NFA, which is small in size but slow in containment solving, and DFA, which is big but decides the containment fast, several approaches for combining the NFA and DFA have been proposed [6, 39], successfully applied in network intrusion detection.

2.3 Alternating Finite Automata

Alternating finite automata (AFA) introduce the concept of *universal* transitions in addition to the NFA's *existential* transitions. With transitions in AFA, we are able not only to tell that one of the successor configurations must have a finalizing run but also that *all* subconfigurations in some of the successor configurations must have finalizing runs. The purpose of AFA is allowing intersection to be expressed, which will also enables cheap complement to be performed.

To be able to introduce alternating finite automata, let us first define $\mathcal{B}^+(S)$, the set of *positive Boolean formulae* [27] over S, which are the Boolean formulae of the form $\psi ::= s \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2$, where $s \in S$. The set of valuations that satisfy ψ is denoted as $[\![\psi]\!] \subseteq 2^S$.

As ψ is positive, $[\![\psi]\!]$ is an *upward closed* set — if a valuation X is contained in $[\![\psi]\!]$, then also all supersets of X are contained.

Similarly we can define *negative Boolean formulae* $\mathcal{B}^-(S)$, which have the form $\psi ::= \neg s \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2$. For a negative formula ψ, the set of valuation is *downward closed* — if a valuation X is contained in $[\![\psi]\!]$, then also all subsets of X are contained.

Alternating finite automaton [18, 27] is then defined as a pentuple $\mathcal{M} = (Q, \Sigma, \delta, \iota, \phi)$ where

- Q is a finite set of states;
- Σ is a finite alphabet;
- $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is an alternating transition function;
- $\iota \in \mathcal{B}^+(Q)$ is a positive formula determining the initial states;
- $\phi \in \mathcal{B}^-(Q)$ is a negative formula determining the final states.

A configuration κ is from a domain $\kappa \in 2^Q \times \Sigma^*$, the first component of the configuration is a *cell* (a set of states) instead of a state. A transition $\kappa \mapsto \kappa'$ exists iff $\kappa = (c, aw)$, $\kappa' = (c', w)$ and for each $q \in c$, a valuation $v \in [\![\delta(q, a)]\!]$ exists, such that $v \subseteq c'$.

A *run* of an AFA is a sequence of configurations, transitions between which exist $\kappa_1 \mapsto \kappa_2 \mapsto \cdots \mapsto \kappa_n$. The run is initial iff $\kappa_1 = (c_0, w)$, such that $\iota(c_0)$. The run is finalizing if $\kappa_n = (c_n, \epsilon)$ and $\phi(c_n)$. A run is accepting iff it is initial and finalizing. If any accepting run exists with an initial configuration (c_0, w), we say that $\mathcal{M}$ accepts w, otherwise $\mathcal{M}$ refuses w.

Alternating finite automata have found their use in formal verification, as they can cheaply express language intersection and inclusion. Specifically, applications in LTL model checking and string solvers are striking novadays. The ease of performing complement and other logical operations on AFA enables advances in pattern matching and its applications not only for text parsers but also for DNA synthesis [40].

By further leveraging the level of automaton succinctness the efficiency of simulation suffers again. As well as for NFA, multiple configurations may be achieved at the same time for a given prefix of the input word. Moreover, a configuration contains a set of states instead of a single state, as in the case of NFA and DFA. There is an exponential number of upward-closed subsets of Q and in the worst case all of them must be analysed.

Deciding language emptiness as well as universality of alternating finite automata is PSpace-complete [33]. Deciding language emptiness will be discussed in detail in the section 3. Universality can be checked by language complement (linear time) and subsequent emptiness check.

2.4 Symbolic Finite Automata

In automata that model practical applications (e.g. regular expression matching), it is a very common pattern that between some states q and q', transitions over a huge number of symbols exist. For example, a comprehensive representation of transition relation of the minimal DFA for a simple regular expression `a.b` would contain 28 items for an alphabet of

all ASCII lowercase latin symbols:

$$
\begin{aligned}
&\mathcal{M} = (\{q_0, q_1, q_2, q_3\}, \Sigma, \delta, q_0, \{q_3\}) \\
&\Sigma = \{a, b, c, d, \ldots, x, y, z\} \\
&\delta = \{ \\
&\qquad (q_0, a, q_1), \\
&\qquad (q_2, b, q_3), \\
&\qquad (q_1, a, q_2), \\
&\qquad (q_1, b, q_2), \\
&\qquad (q_1, c, q_2), \\
&\qquad (q_1, d, q_2), \\
&\qquad \ldots, \\
&\qquad (q_1, z, q_2), \\
&\}
\end{aligned}
$$

With the growth of the alphabet, automata for such patterns would obviously grow too. UTF8 can include about 2^21 characters, so this growth is inacceptable. The use of nondeterministic or alternating automata would obviously not be a solution.

The formalism of symbolic automata has been introduced for this purpose. Any of the DFA, NFA or AFA definition could be modified to be symbolic by replacing the alphabet Σ in the domain of the transition function by $\mathcal{B}(\Sigma)$, where $\mathcal{B}(S)$ is the set of all boolean formulae over variables S. This way, the set Σ is no longer interpreted as a set of symbols, but instead as a set of variables. The specific symbols are now actually valuations of Σ, which is reflected in the following definition of configuration and transition relation. A configuration of a symbolic DFA is a tuple $\kappa \in Q \times (2^{\Sigma})^*$. A transition $\kappa \mapsto \kappa'$ exists iff $\kappa = (q, aw)$, $\kappa' = (q', w)$ and $\exists (q, \sigma, q') \in \delta.\ \sigma(a)$.

To give an example of automaton accepting pattern `[a-z]` in the ASCII alphabet, we map the variables Σ to the eight bits of ASCII: $\Sigma = \{b_0, b_1, \ldots, b_7\}$. Then, the only transition in the pattern's automaton would be guarded by the formula

$$\neg b_7 \wedge b_6 \wedge b_5 \wedge (\neg b_4 \vee \neg b_3 \vee (\neg b_2 \wedge (\neg b_1 \vee \neg b_0))).$$

2.5 Transducers, Counting Automata and Tree Automata

The following text briefly introduces other models closely related to word finite automata, which will be mentioned in the section 4.

Transducers: Nondeterministic finite transducer (NFT) over an input alphabet Σ_I and an output alphabet Σ_O is an NFA with the alphabet of the tuples $\Sigma = \Sigma_I \times \Sigma_O$. If a NFT accepts a word $(a_1, b_1) \ldots (a_n, b_n)$, we say that it *transduces* the input word $a_1 \ldots a_n$ to the output word $b_1 \ldots b_n$. Note that a given input word can be transduced by given NFT to multiple output words. With a fixed transducer, we define a transducer relation $\tau : \Sigma_I^* \times \Sigma_O^*$ in a way that $w_I \tau w_O$ iff w_I is transduced to w_O. Similarly, deterministic finite transducers (DFT) are DFA over the alphabet $\Sigma = \Sigma_I \times \Sigma_O$. Moreover, the transducer relation of DFT is a function. Each NFT with input alphabet Σ_I and output alphabet Σ_O can be transformed

to a DFT with the same input alphabet Σ_I and the output alphabet 2^{Σ_O}, so that $w_I \tau' w_O$ iff $w_O = \{w \in \Sigma_O^* \mid w_I \tau w\}$ and $w_O \neq \emptyset$ where τ is the transducer relation of the NFT and τ' is the transducer relation (function) of the DFT. The construction is called transducer determinization and is described in detail in [15].

Counting Automata: Counting automata [58] are a sextuple $(Q, V, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0, F have the same meaning as for the finite automata, V is a set of integer variables and the transitions are guarded not only by symbols of alphabet but also by a formula in Pressburger logic over the variables $\{v, v' \mid v \in V\}$. A configuration of a counting automaton is a tuple (q, β) where q is a state and β is a valuation of the variables V. Initial configuration is $(q_0, \lambda x.\ 0)$. A transition guarded by a symbol a and a fromula ϕ from (q, β) to (q', β') can be performed if the symbol on the input tape equals a and ϕ is satisfied by the valuations β, β'. The Pressburger formulae therefore serve to both enable transitions and modify the valuations of V. A special type of counting automata exists — bounded counting automata, integer variables of which are bounded. Bounded counting automata have equal power to finite automata and are used for succinctness.

Tree Automata: Tree automata (TA) are a generalization of finite automata. They define languages of trees instead of languages of words; note that words can be represented as a special class of trees — linked lists. we will discuss only the basic notion of tree automata, automata for ranked trees, although there are several other definitions [44], e.g. hedge automata or tree-walking automata.

A *ranked alphabet* Σ is a finite set of symbols together with an arity $|a| \in \mathbb{N}$ for each $a \in \Sigma$. A set of finite trees $\mathcal{T}_\Sigma$ is the least set containing for each $a \in \Sigma$ and each $t_1, \ldots, t_{|a|} \in \mathcal{T}_\Sigma$ also $a(t_1, \ldots, t_{|a|})$. Note that Σ must contain at least one nullary symbol for $\mathcal{T}_\Sigma$ to be non-empty. The nullary symbols are the leaves of the trees.

A nondeterministic finite tree automaton (NFTA) is a quadruple TA $= (Q, \Sigma, \delta, F)$, where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is a ranked alphabet and δ is a transition relation $\delta \subseteq \bigcup_{i=0}^{k} Q^i \times \Sigma \times Q$, such that for each $(\mathbf{q}, a, q') \in \delta$, $|\mathbf{q}| = |a|$. When processing a tree $T \in \mathcal{T}_\Sigma$, the nodes of T get nondeterministically marked by states of the TA in the following way. Whenever all child nodes of some node a are marked with the states $(q_1, \ldots, q_{|a|})$ and $((q_1, \ldots q_{|a|}), a, q') \in \delta$[2], the node a is marked with the state q'. Note that the leaf nodes of T are marked right away (if a transition $((), a, q') \in \delta$ exists), since they have no child nodes. The marking then continues in a bottom-up fashion. Whenever the root node of T is marked with a final state, the automaton TA accepts the tree T. A language of trees $\mathcal{L}(\text{TA})$ over Σ is a subset of $\mathcal{T}_\Sigma$, accepted by the tree automaton TA.

3 Language Emptiness Analysis

Particularly important question in formal verification is, whether the investigated system can reach a *bad* state, i.e. a state in which the given property does not hold. If the relevant state space of the system can be encoded as a finite automaton where the final states are the *bad* states determined by the condition, it is a well known fact that the verification question is decidable. The existence of an accepting path (path from an initial to a final/bad state) can be reformulated as existence of a word that is accepted by the language of the automaton

[2]Note that more than one such transition may exist, hence the nondeterminism.

and therefore as non-emptiness of the language. The verification question that has been posed is thus equivalent to the language emptiness problem.

As has been already discussed in the previous section, the language emptiness problem can be solved in linear time for DFA and NFA — by using the depth-first search from the initial states. For the alternating finite automata, however, the problem is PSpace-complete [33]. The exponential blow-up of conversion from AFA to NFA is in many cases inacceptable, therefore a considerable amount of research has been done [22, 27, 32, 20] to overcome the theoretical bound for real-world instances of the AFA emptiness problem. As the emptiness checking of AFA is the most interesting problem, the rest of the section will be dedicated to AFA. We will therefore use the terms *initial cells* and *bad cells* instead of initial states and bad states, as the transition relation of AFA depend on cells (set of states) instead of states (as it was for DFA and NFA).

The emptiness-checking algorithms proposed by the aforementioned research are based on modern formal verification techniques (e.g. model-checking, abstraction refinement), benefitting in addition from the monotonicity property of the AFA transition relation.

Monotonicity: Let us define the *subsumption* order $\kappa_1 \preceq \kappa_2$ of configurations (we say that κ_1 subsumes κ_2):

$$(c_1, w_1) \preceq (c_2, w_2) \text{ iff } w_1 = w_2 \text{ and } c_1 \subseteq c_2.$$

The positiveness of formulae in the AFA transition function yields the monotonicity of the transition relation: If a configuration $\kappa = (c, aw)$ transitions into the configuration $\kappa' = (c', w)$, then all configurations subsumed by κ transition into configurations subsumed by κ'. Formally,

$$\forall \kappa_1, \kappa_1', \kappa_2.\ \kappa_1 \to \kappa_1' \wedge \kappa_1 \preceq \kappa_2 \implies \exists \kappa_2'.\ \kappa_2 \to \kappa_2' \wedge \kappa_1' \preceq \kappa_2'.$$

The monotonicity property also holds for the transitive closure of the transition relation and therefore can be used for induction: If a cell c is reachable from an initial cell, then all supercells of c are reachable and therefore the reachability of the supercells need not be analysed. Similarly, if a bad cell is reachable from a cell c, then a bad cell is reachable also from the subcells of c and the bad cell reachability from the subcells again need not be checked.

3.1 Antichain-based state space exploration

The antichain-based approaches were first introduced in [22] and are based on the idea of regular model checking. They resemble in a large extent the emptiness checking of NFA. For NFA, the state space is searched step-by-step from initial states to all reachable states in a DFS or BFS manner. If a final state is reached during the search, the language is not empty, otherwise, if all the reachable state space has been already visited, the language is empty. Another possibility is to search backwards — starting in the final (bad) states and performing backward transitions to states that have not yet been visited. The language is not empty if the search reaches the initial state. For AFA, the forward/backward-reachable cells are analysed instead of states, abusing in addition the monotonicity property of $\to$ to omit the subsumed cells from the search. We say that a cell c_1 subsumes the cell c_2 iff $c_1 \subseteq c_2$ for the forward search and iff $c_1 \supseteq c_2$ for the backward search.

The forward antichain algorithm holds a *frontier* T (implemented as a stack for a DFS of the state space or a queue for a BFS) of cells $c \in T$ that are queued for checking if a bad state can be reached from c. In addition, it maintains a set of cells D, analysis of which has been already launched. The algorithm starts with $T := \lfloor [\![\iota]\!] \rfloor$ and $D := \emptyset$, where $\lfloor X \rfloor$ is the set of $\subseteq$-minimal elements from X (note that the ones that are not minimal need not be analysed due to the monotonicity of $\rightarrow$). Then, the main loop follows, in each iteration of which a cell $c \in T$ is moved to D, and replaced in T by its $\rightarrow$-successors S. Making use of monotonicity, before the replacement, the elements of D, T and S are adjusted, so that $D \cup T \cup S = \lfloor D \cup T \cup S \rfloor$[3]. Specifically, all cells in $D \cup T$ that are supercells of some cell from S are removed from the corresponding set (D or T) and vice versa, all cells in S that are supercells of some cell in $D \cup T$ are removed. The sets D and T contain cells that are known to be reachable from an initial cell. Whenever a bad cell is added to T, the language is claimed to be non-empty. The emptiness is claimed when the frontier T is emptied.

The backward antichain algorithm works (and performs better in many cases) in a similar fashion, only the frontier is initialized to $T := \lceil [\![\phi]\!] \rceil$ ($\lceil X \rceil$ containing the $\subseteq$-maximal elements of X), the maximal cells are maintained instead of minimal ones in the set $D \cup T \cup S$, the set S contains $\rightarrow$-predecessors of c and finally, the language non-emptiness is claimed if $\iota(c)$ for some c being added to T.

Formally, and hiding the implementation details, the two algorithms can be very succinctly defined (and proved for emptiness) using complete lattices and fixpoints. We reference the interested reader to [27].

The set of $\rightarrow$-successors or $\rightarrow$-predecessors can be computed for non-symbolic automata using a nested `for` loop, iterating whole alphabet, yielding a symbol a, in the outer loop, iterating the states of the cell c in the inner-loop and unioning their a-successors (a-predecessors respectively). For symbolic automata, SAT solvers are used for this purpose.

3.2 Combinational synthesis using And-Inverter Graphs

And-Inverter Graph (AIG) [37] is an acyclic directed graph with three types of nodes, each of which has exact number of incoming edges (fanins) and arbitrary number of outgoing edges (fanouts): *input nodes* and *constant zero nodes* have zero fanins, *NOT gates* have one fanin, *AND gates* have two fanins. Let (x, y) be an edge in the graph from the node x to the node y. We say that x is the *fanin child* of y. For a node x, let x^1 denote its first fanin and x^2 the second one (if they are defined). Let $C(x)$ represent the set of all fanin childs of x and let the *fanin cone* $C*(x)$ be the transitive closure of the function C applied to x.

Let us discuss the semantics of AIG. A unique variable is assigned to each input node. Let x be an input node, then $v(x)$ denotes its assigned Boolean variable. Each node of the graph represents a formula in the Boolean logic over the variables assigned to the input nodes. For a node x, its formula is denoted as $\langle x \rangle$ and defined as $v(x)$ if x is an input node, 0 if x is a constant zero node, $\neg\langle x^1 \rangle$ if x is a NOT gate and $\langle x^1 \rangle \wedge \langle x^2 \rangle$ if x is an AND gate. As the set of connectives $\{\wedge, \neg\}$ is functionally complete, AIG can express any Boolean formula. W.l.o.g., the constant zero node is always unique.

There are two main benefits of representing formulae in AIG:

- Sharing of fanin cones — Representation of two formulae f and g by the AIG nodes x_f and x_g can share some part of the structure, e.g. for $f = v_1 \wedge v_2 \wedge v_3 \wedge v_4 \wedge v_5$ and

[3] As only the minimal elements are maintained, the set itself is an antichain and hence the name of the algorithm.

$g = v_1 \wedge v_2 \wedge v_3 \wedge v_4 \wedge v_6$, there can be a node x_1 representing the common part of the two formulae. The node x_1 will be the common first child of x_f and x_g and the second child will be the input representing v_5 or v_6 respectively.

- Simplicity — The representation is often very natural. Moreover, there are only three types of nodes with fixed numbers of fanins, allowing for fast pattern detection in local structural synthesis and fast and simple SAT solver implementations (which benefit also from the aforementioned sharing).

Local structural synthesis: The set of all possible structurally different rooted AIG (structural patterns) with the depth D or less is finite (e.g. for the most usual $D = 2$, there are 235 structural possibilities). The depth D is fixed (usually to 2 [14, 37]) and each of the structural patterns is assigned a restructuring procedure (e.g. $\neg\neg x \rightarrow x$). Whenever a node is added to the graph, it first is checked using hashing if there is another with the same type and the same set of childs. If so, the two nodes represent the same formulae and are merged — one of them is deleted and its fanout is unified with the fanout of the other node. Otherwise, the node's transitive childs up to the depth D are matched to one of the structural patterns and restructured. New nodes are created during the restructuring process. The local structural synthesis is applied to them.

SAT sweeping: Not all functional equivalences in the graph are detected using the local structural synthesis. At certain intervals of AIG construction, the more expensive SAT sweeping [38, 65] is applied to merge the rest of the functionally equivalent nodes. The basic idea is as follows. First, a number of random simulations are performed, in each of which a random Boolean value is assigned to each variable and the functional value of the formula at each node is computed. The nodes are then grouped to equivalence classes of the relation $\sim$, defined in a way that two nodes are $\sim$-equivalent if the functional values of their formulae were the same for each of the random simulations. The nodes in the same class are good candidates to be functionally equivalent. In the same class, the nodes are checked pairwise for functional equivalence using a SAT solver: AIG representing the XOR formula is built upon the two nodes and its output is checked with an AIG-specialized SAT solver (e.g. MiniSat [60]). If the result of the check is UNSAT, the two nodes are functionally equivalent and can be merged.

Similar approach using BDD was formerly introduced in [37] but due to the BDD explosion, it is reported that the SAT sweeping mostly performs much better in practical cases [14].

3.3 IC3

The abbreviation IC3 stands for *Incremental Construction of Inductive Clauses for Indubitable Correctness*. This model-checking algorithm is also known under its alias *Property Driven Reachability, PDR*. It has been introduced by Bradley [13] and gained a lot of attention in the community of formal verification researchers. The general idea, based on abstraction refinement via counter-example analysis, is outlined in the following text for the domain of *transition systems*.

Let $\mathbb{L}$ be a logic and X be a finite set of variables. Then, $\mathbb{L}(X)$ denotes all formulae of the logic $\mathbb{L}$ over the variables X. Further, let $[\![\mathbb{L}(X)]\!]$ denote the set of all valuations of

variables X in the logic $\mathbb{L}$ (e.g. if the logic is Boolean, $[\![\mathbb{B}(X)]\!] = 2^X$). Let $\psi \in \mathbb{L}(X)$ be a formula. Then, $[\![\psi]\!] \subseteq [\![\mathbb{L}(X)]\!]$ is the set of all valuations satisfying the formula ψ. Let $\top$ be a formula, such that $[\![\top]\!] = [\![\mathbb{L}(X)]\!]$. For a valuation $V \in [\![\mathbb{L}(X)]\!]$, let $[V] \in \mathbb{L}(X)$ denote a formula that is satisfied just by the valuation V — e.g. for Boolean logic, the formula $[V]$ contains either a positive or a negative literal for each variable in X, positive iff $v \in V$.

The apostrophe operator applies to variables (then it is actually not an operator but x' is a new variable, read as *striped* x), sets of variables (then, $X' = \{x' \mid x \in X\}$) and formulae — then, $' : \mathbb{L}(X) \to \mathbb{L}(X')$ denotes substituing all variables in the formula for their striped counterparts.

Transition system (TS) over a logic $\mathbb{L}$ is a quadruple $\mathrm{TS}_\mathbb{L} = (U, I, T, P)$, where U is a set of variables and I, T, P are predicates over the variables, given by formulae of the logic $\mathbb{L}$:

- $I \in \mathbb{L}(U)$ delimits *initial valuations*,
- $T \in \mathbb{L}(U \cup U')$ is a *transition relation* — a formula over the variables from U and their striped counterparts (representing the successor valuations),
- $P \in \mathbb{L}(U))$ is the property, invariance of which we check.

Valuation V_r of the variables U is *reachable* iff a sequence of valuations (*run*) $V_1 V_2 \cdots V_n$ exists, such that

$$V_r = V_n \wedge V_1 \in [\![I]\!] \wedge \forall i \in \{2, \cdots, n\}.\ T(V_{i-1}, V_i).$$

The invariance problem is to check if P holds for all reachable valuations.

Similarity of AFA and TS is apparent. The AFA language emptiness problem can be reduced into the invariance problem of the transition system over the Boolean logic. For a symbolic AFA $M = (Q, \Sigma, \delta, \iota, \phi)$, let us define the reduction to TS $= (U, I, T, P)$. Let *invalid transition* be a relation between two cells c_1 and c_2, such that no symbol $a \in 2^\Sigma$ exist, such that $(c_1, aw) \to (c_2, w)$ for any w. The reduction introduces a sink variable q_s, which gets set whenever an invalid transition is made and never gets cleared (therefore whenever an invalid transition is in a run, the run is non-accepting).

- $U = Q \cup \{q_s\} \cup \Sigma$ — The variables of the U are the states of the AFA, the additional sink state (which indicates that an invalid transition has been performed somewhere in the run) and the alphabet.
- $I = \iota$ — The initial valuations are directly inherited — note that the alphabet variables are unconstrained.
- $P = \neg(\phi \wedge \neg q_s)$ — The investigated property is "not reaching final valuations in a valid way". If this property is invariant, the language of the AFA is obviously empty.
- $T = q'_s \Leftrightarrow \left(q_s \vee \left(\bigvee_{(q,\sigma,\psi)\ \in \delta} \neg(q \wedge \sigma \Rightarrow \psi')\right)\right)$ — The next sink state defines what are invalid transitions. Invalid are all transitions from valuations that are already invalid. Invalid are also transitions, where for some tuple of the automaton's transition function, the condition $q \wedge \sigma$ is met but the resulting positive formula is not satisfied by the next states. Note that there are no restrictions on how the Σ-part of the next valuation should look — the input symbols are arbitrary.

Having the reduction of AFA emptiness to TS, let us proceed to the definition of IC3. The state of the algorithm is a sequence of frames $F_0 F_1 \cdots F_k$. For all $i \in \{0, \cdots, k\}$, the frame $F_i \in \mathbb{L}(U)$ is a formula that over-approximates the valuations reachable in i or less steps. The valuations of the frames naturally (as the valuations of F_i include valuations reachable in $0, 1, \cdots$ and i steps) form a chain $[\![F_0]\!] \subseteq [\![F_1]\!] \subseteq \cdots \subseteq [\![F_k]\!]$. The zero frame is precise, not an approximation, and does not change during the algorithm run: $F_0 = I$. The other frames get added (i.e. k increments) and refined during the algorithm, so that 1) they would not contain any bad valuations (i.e. $F_i \implies P$ for all $i <= k$) and 2) F_{i+1} over-approximated the successors of F_i, formally $F_i \wedge T \implies F'_{i+1}$ for all $i < k$.

Top level of IC3: Starting with $k := 0$ and $F_0 := I$, the algorithm works as follows:

1. Containment of any bad valuation in F_k is checked using a SAT solver[4] with the query SAT($\neg(F_k \implies P)$). If a bad valuation V exists in F_k, it is returned as a witness of the SAT. The *blocking phase* (see the algorithm 1) is invoked as ProveReachabilityOrBlock(V, k), in which the reachability of V in k or less steps is proved or disproved. The frames $F_1 \cdots F_k$ get refined during the blocking phase. The reachability is disproved when the frame F_k gets refined in a way that it no more contains the valuation V. This step is repeated until no bad valuation is found in F_k (or until the reachability of a bad valuation is proved).

2. At this point, all the aforementioned conditions for the frames are satisfied. To speed-up the convergence of the algorithm, the *propagation phase* is applied, which tries to further refine the frames $F_1 \cdots F_k$ using a cheap propagation technique.

3. The algorithm converges (and the invariance is proved) if $F_i = F_{i+1}$ for some $i < k$. Otherwise, k is incremented and the new frame F_k is initialized to cover all valuations of U: $F_k := \top$. Afterwards, the control flow returns to the point 1.

Blocking phase: Let us define the blocking phase in a recursive way in the algorithm 1. It takes a valuation V as its argument, along with the index i of the frame that contains V. The procedure proves or disproves reachability of V in i or less steps. If $i = 0$, the reachability is trivially proven. Otherwise, using a SAT solver, the procedure decides if a transition exists from any valuation V_{pre} in the frame F_{i-1} to the valuation V. If the SAT solver proves unsatisfiability, V is not reachable in i or less steps because F_{i-1} includes all valuations reachable in $i-1$ or less steps and the transition from any of them to V does not exist. In this case the F_i is refined so that it would not contain the valuation V. Also all $F_1 \ldots F_{i-1}$ are refined so that they would not contain V because V is unreachable in i or *less* steps. Otherwise, if the SAT solver has proved that V has a $\rightarrow$-predecessor V_{pre} in the frame F_{i-1}, V_{pre} is returned as a witness and the blocking phase procedure is recursively called for V_{pre} and F_{i-1} to prove or disprove the reachability of V_{pre}.

Let us analyze the formula $\psi = F_{i-1} \wedge \neg[V] \wedge T \wedge [V]'$ from the algorithm 1. Satisfying are all valuations $V_p re, V'_p re$ of the variables $U \cup U'$ (where $V_p re$ is the valuation of U and $V'_p re$ is the valuation of U'), for which

[4]More generally, an SMT solver is used if the logic $\mathbb{L}$ is not the Boolean logic. This is applied to all notions of SAT further in the text.

Algorithm 1 Blocking phase

```
1:  function ProveReachabilityOrBlock(V, i)
2:      if I(V) then                                              ▷ Reachable!
3:          Reconstruct and print the counter-example and terminate the IC3.
4:      repeat
5:          let ψ = F_{i−1} ∧ ¬[V] ∧ T ∧ [V]′
6:          V_pre, V′_pre := SAT(ψ)
7:          if the above was UNSAT then
8:              F_j := F_j ∧ ¬[V]    for j ∈ 1, ..., i
9:          else
10:             ProveReachabilityOrBlock(V_pre, i − 1)
11:     until witness is null
```

1. $[V]'$ is satisfied — it means that the $V_p're = V'$, the valuation $V_p're$ is therefore not interesting and will be ignored in the sequel (it is actually interesting for counter-example reconstruction when using generalization but it is out of the scope of the present text),
2. $F_{i-1}(V_{pre})$ and a transition exists from V_{pre} to V'_{pre},
3. V_{pre} is not V' — note that if the only predecessor of V would be V, the only predecessor of the predecessor could be again only V, etc. V therefore would not be reachable. The algorithm would work without this enhancement, but it would perform worse.

Generalization: A drawback of the present algorithm 1 is that it analyses only a single valuation at a time. In the logics with infinite valuations, the algorithm would not be guaranteed to terminate. Even in the finite logics (as is the logic of our interest — the Boolean logic), the algorithm would not perform well in time as well as memory. Introducing the generalization into the algorithm, it is guaranteed to terminate for some classes of infinite-state systems (e.g. for well-structured transition systems [36]), while also enhancing the analysis also for finite-state transition systems.

The implementation of the generalization procedure itself is specific for the given logic $\mathbb{L}$ but its application can be incorporated into the general IC3 algorithm. We define two types of generalization:

- Witness weakening $\mathrm{Gen}_w : [\![\mathbb{L}(U)]\!] \times \mathbb{L}(U) \to \mathbb{L}(U)$ — This generalization is applied whenever the SAT solver returns a witness V of satisfiability of a formula ψ. The generalization uses the witness to cheaply find a weaker formula than $[V]$ (or $[V]$ in the worst case), such that $\mathrm{Gen}_w(\psi, V) \implies \psi$. The formula $\mathrm{Gen}_w(\psi, V)$ should not be much more structurally complex than $[V]$: e.g. if we chose $\mathrm{Gen}_w(\psi, V) = \psi$, the SAT solving in the IC3 algorithm would become heavy and the overall computation would resemble the model-checking approaches based on incremental unrolling (e.g. see the section 3.4). We will show a typical implementation of Gen_w for Boolean logic later in the paragraph **Generalization for Boolean logic**.
- Blocker weakening $\mathrm{Gen}_b : \mathbb{L}(U) \times \mathbb{L}(U) \to \mathbb{L}(U)$ — This generalization $\mathrm{Gen}_b(\nu, \psi)$ is applied whenever a formula ψ of the form $F\neg\nu\wedge T\wedge nu'$ ends up with the *unsatisfiability*

result. The generalization tries to find a formula ν_g that is weaker than ν (or ν in the worst case), such that $F \wedge \neg\nu_g \wedge T \wedge \nu'_g$ would remain unsatisfiable *and* $I \wedge \nu_g$ would be unsatisfiable. The second condition ensures that the valuations of the generalization do not overlap with initial states. Note that in the algorithm 2, the generalization would be removed from the frames $F_1 \dots F_i$. If it overlapped the initial states, the constraint $\forall i \in 1, \dots, k.\ I \implies F_i$ would be thus violated.

Good generalization can result in better refinement of frames and therefore faster convergence. Implementing Gen_b, one should follow similar advices concerning structural complexity as for Gen_w.

The algorithm 1 can be rewritten using generalization, resulting in the algorithm 2. The type of the procedure's first argument changes from a valuation to a formula. The top-level invocation (discussed formerly in the paragraph **Top level of IC3**) must be also changed to ProveReachabilityOrBlock($\mathrm{Gen}_w(V, \neg(F_k \implies P)), k$).

Algorithm 2 Blocking phase with generalization

```
1:  function ProveReachabilityOrBlock(ν, i)
2:      if SAT(I ∧ ν) then                                   ▷ Reachable!
3:          Reconstruct and print the counter-example and terminate the IC3.
4:      repeat
5:          let ψ = F_{i−1} ∧ ¬ν ∧ T ∧ ν'
6:          V_pre, V'_pre := SAT(ψ)
7:          if the above was UNSAT then
8:              F_j := F_j ∧ ¬Gen_b(ν, ψ)    for j ∈ 1, ..., i
9:          else
10:             ProveReachabilityOrBlock(Gen_w(V_pre, ψ), i − 1)
11:     until witness is null
```

Propagation phase: The propagation phase serves to cheaply refine the frames for faster convergence of the algorithm. By construction, each frame is a conjunction of subformulae — see the line no. 8 in the algorithm 2 where the frames are refined by adding subformulae $\neg\mathrm{Gen}_b(\nu, \psi)$. For each $i = 1, \dots, k$, let us decompose $F_i = \top \wedge \neg\xi_1 \wedge \dots \wedge \neg\xi_n$ and let us take an arbitrary subformula ξ_j. If all predecessors of all valuations of ξ_j lay out of F_i, the valuations are unreachable not only in i or less steps but also in $i+1$ steps. The subformula ξ_j can be therefore added to F_{i+1}, refining the $(i+1)$-th frame. Moreover, the blocker weakening generalization can be applied here.

Generalization for Boolean logic: The generalization implementation is dependent on the logic of the TS (and even in the same logic, multiple generalization function can be defined). We show an example of a simple and popular generalization approach for the Boolean logic, which is used e.g. in the ABC tool [14]. The approach is also implicitly capable to detect and make use of the monotonicity of the transition relation.

- Witness weakening — We have a formula ψ and a witness of its satisfiability — a valuation V. We set the intermediate result $g := [V]$. We iterate through the literals

Algorithm 3 Propagation phase

function PROPAGATESUBFORMULAE
  for $i \in 1, \ldots, k-1$ **do**
    let $F_i = \top \wedge \neg\xi_1 \wedge \cdots \wedge \neg\xi_n$
    for $j \in 1, \ldots, n$ **do**
      let $\psi := F_i \wedge \neg\xi_j \wedge T \wedge \xi'_j$
      if UNSAT(ξ) **then**
        $F_j := F_j \wedge \neg\mathrm{Gen}_b(\xi_j, \psi)$ **for** $j \in 1, \ldots, i+1$

of g and try to omit the currently iterated literal in each iteration, checking if the new g still implies ψ. If the check fails, the last omission is reverted. The check can be efficiently done using ternary simulation. After the loop, the formula g is the result of $Gen_w(V, \psi)$.

- Blocker weakening — We have two formulae: ν and an unsatisfiable ψ, which is in the form $\psi = F \wedge \neg\nu \wedge T \wedge \nu'$. The formula ν was obtained by the aforementioned witness weakening and therefore it is a conjunction of literals. Initially, we set the intermediate result $g := \nu$. We iterate through the literals of g and try to omit the currently iterated literal in each iteration, checking if the two conditions for the blocker weakening still hold (both the formulae $F \wedge \neg g \wedge T \wedge g'$ and $I \wedge g$ should be unsatisfiable). If the check fails, the last omission is reverted. Unfortunately, the ternary simulation is of no help in blocker weakening and SAT solving must be performed in the check.

Optimized implementations of IC3 for the Boolean logic TS: The two most notable implementations of the IC3 algorithm for the Boolean logic are in the tools ABC [14] and nuXmv [16]. The authors of NUXMV admit in [16] that ABC performs better, which is also indicated by the results of the measurements in [32] where the AFA emptiness problem is solved with these tools. We will therefore focus on ABC in the sequel.

The ABC tool uses AIG for representing the Boolean formulae (frames, transition relation, delimitation of initial states, the property P and the temporary formulae in the blocking and propagation phases). Therefore all the techniques for combinational synthesis discussed in 3.2 are used to simplify the representations. However always applicable, the expensive SAT sweeping is not used for the temporary formulae. The implementation uses the MiniSAT [60] solver adjusted for AIG. In addition to the techniques noted in this recherche, other enhancing ideas have been implemented to ABC, most notable of which are

- **Gate-level abstraction** [48] — Using the AIG representation of the formulae, when solving satisfiability, a part of the AIG is abstracted out. The logical values in the abstract part are unknown and its analysis is ignored — the abstracted part is opted out and the interface between the abstract and non-abstract part is represented as primary inputs (where arbitrary values are allowed). If the formula is unsatisfiable using the abstraction or the unabstracted part of the graph is sufficient to imply satisfiability, the SAT result is ready-made. Otherwise, the abstraction gets refined and the SAT procedure continues. This technique is applied not only in IC3 but also in the sequential synthesis 3.4.

- **Localization abstraction** [55] — Recall the witness weakening of a valuation V

which satisfies ψ. The literals of the formula g (initialized to $[V]$) are tried to be omitted and the check of $g \implies \psi$ is done by ternary simulation. The literals of variables which are not present in ψ can be omitted in the beginning without any check. Therefore some of the computation of the ternary simulation is spared.

- **Analysis of Counter-examples To Generalization** (CTG) [30] — Recall the blocker weakening: after omission of a literal, the formula g is checked to be unsatisfiable. If it is satisfiable instead, the omission is reverted and we iterate to the next literal. With the CTG analysis, before reverting the change, reachability of the witness of the satisfiability (the predecessor of some valuation of g) is analysed and if the witness is itself unreachable, it can be opted out from the previous frame (along with its generalization) and the unsatisfiability check is performed again. The reachability of the witness is analysed only up to a predefined depth (the depth 1 has experimentally shown best results), otherwise, the algorithm tends to analyse and refine many irrelevant parts of the state space.

Another abstraction-based algorithm, which is specialized for AFA, has been pioneered in [27] but instead of abstracting the reachable state space, it groups the states of AFA into a set of partitions, each of which represents a state of an *abstract AFA*. This abstract AFA is checked for language emptiness using antichain algorithms and if a counter-example is found, the partitions are refined if it is spurious or the non-emptiness is claimed otherwise.

3.4 Sequential Synthesis

Sequential synthesis is an older (comparing to IC3) TS verification approach. Similarly to IC3, the sequential synthesis also makes use of SAT solving but a smaller number of complex and heavyweight SAT instances is solved in contrast with the bigger number of simple SAT instances, as was the case for IC3. The sequential synthesis comes under the category of symbolic model checking algorithms.

Each transition system can be transformed (using a construction similar to the reduction from AFA to TS [32, 47]) to a transition system, for which

- $U = L \cup J$ where L is a set of *latches* and J is a set of *inputs*,
- $I = \bigwedge_{l \in L} \neg l$ — all latches are initialized to logical zeroes,
- $T = T_L \cup T_J$ where T_L is a total function $(L \cup J) \to L'$ and $T_J = 2^(L \cup J) \times 2^{J'}$, i.e. the valuation of inputs J is arbitrary and independent on the previous valuations,
- let $B = \neg P$ be a *bad output* formula, which represents the valuations that do not hold the property P; B contains only the variables L.

The following text informally outlines the verification using sequential synthesis [47], which is implemented in the ABC tool [14]. For brevity, the operations are not formally defined and the text often only references the figures 1 and 2 to illustrate the operations and notation. The transition relation T and the bad output B can be implemented with AIG, input nodes of which correspond to the variables $L \cup J$. The figure 1a illustrates the construction — the nodes $L_1 \dots L_n J_1 \dots J_n$ are the input nodes of the AIG. Two or more (the number is specified by the parameter j) successive applications of the transition function can be represented by *unwinding* the AIG, as illustrated in the figure 1b. Note that

the additional input nodes $J_{21} \dots J_{2n}$ have been created and the resulting AIG is denoted G.

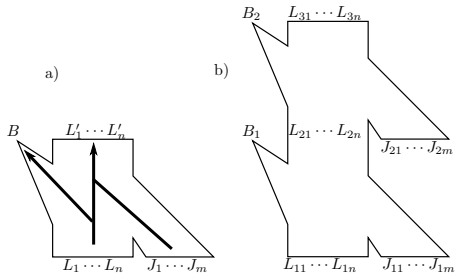

Figure 1: a) AIG representation of (T, B), b) Unwinded AIG G for $j = 2$

The following algorithm works with the parameters j (the number of T unwindings) and i (the number of G unwindings), which are initially set to some small natural numbers (e.g. $i := 2$ and $j := 2$) and increased over time if the verification fails to decide the reachability of bad valuations:

1. Unwind the transition relation j times as illustrated in the figure 1b to create the AIG G. Then unwind the G graph i times, as illustrated in the figure 2a, creating the AIG H.

2. Let us define the equivalence $\sim_i$ over gates of G (illustrated in the figure 2b): Assuming the input nodes of H which correspond to the latch variables of the TS are replaced with constant zero nodes, the two nodes x and y are $\sim_i$-equivalent iff they are functionally equivalent in each of the unwindings $G_1, G_2, \dots G_i$. The $\sim_i$-equivalent nodes are guaranteed to have the same values in the first i valuations of any TS run. The $\sim_i$-equivalence classes can be detected using SAT sweeping. After determining the equivalence classes, we replace the latch inputs of H back from constant zero nodes to input nodes.

3. If any of the bad outputs $B_{11} \dots B_{ij}$ is not $\sim_i$-equivalent with constant zero, a bad valuation is reachable in i or less steps. Otherwise, we will try to prove in the step 4 that the bad valuations are unreachable in any number of steps. We set a new equivalence $\sim_* := \sim_i$ for the first iteration of the step 4.

4. Consequently, we try to prove that the $\sim_*$-equivalent gates have the same values in any number of steps. First, we unwind G an additional time, adding G_{i+1} to H. Then, assuming[5] that the $\sim_i$-equivalent gates in $G_1 \dots G_i$ are functionally equivalent, we check using SAT sweeping if the equivalence classes are broken or not in the unwinding G_{i+1} (they are broken if the $\sim_*$-equivalent nodes are functionally inequivalent in G_{i+1}), see figure 2c. If some of them are broken into smaller classes, we refine the equivalence $\sim_*$ accordingly and repeat the step 4. Otherwise, two situations may arise:

   - The bad outputs $B_{11}, \dots, B_{ij}$ are still in the same class with the constant zero node — the algorithm successfully terminates, claiming the invariance of the property P.

[5]The assumption is made using a technique called *speculative reduction* [49, 46]

- Some of the bad outputs is not $\sim_*$-equivalent to constant zero — one or both of the parameters i and j are increased (e.g. doubled) and the algorithm is repeated from the step 1.

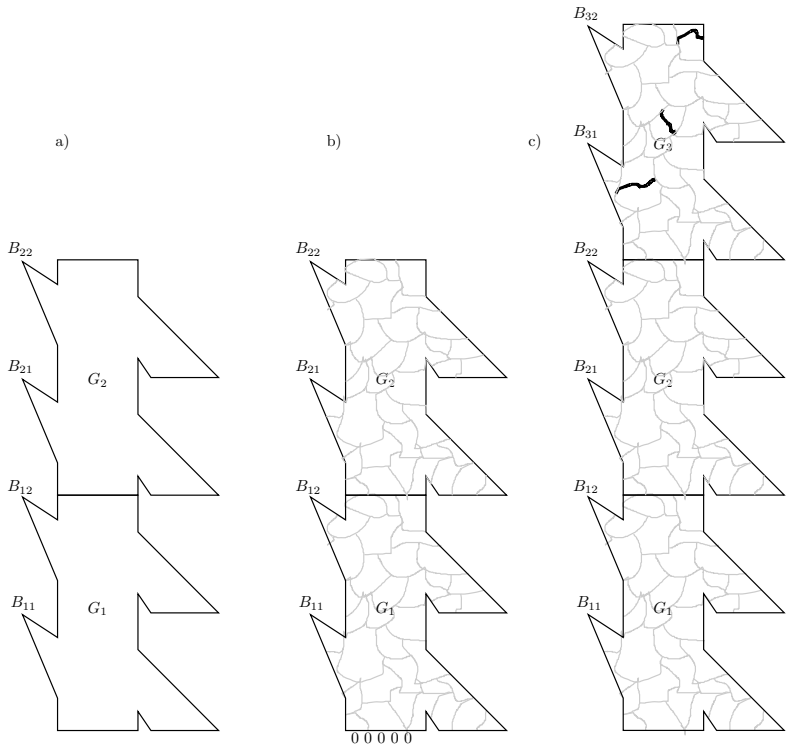

Figure 2: a) The unwinded G for $i = 2$ b) $\sim_i$-equivalence classes, c) Refinement of the $\sim_i$-equivalence classes using additional unwinding without the initial latch restriction.

4 Finite Automata Applications

A typical application of finite automata is regular expression matching or its reverse — generation of words from a regular language, which has found its use e.g. in texture generation [53], or decompression of images [19]. Advanced high-speed regular expression matching has been used for network intrusion detection [6, 39]. In verification, well-known are reductions from satisfiability of formulae in LTL or WS1S logic to AFA. The former reduction is polynomial, the latter is elementary to the number of alternations of negation and quantification [25]. Other fresh applications in the formal verification are actively investigated, including string analysis and shape analysis, which are discussed in the rest of the section.

4.1 String Analysis

With the rapid expansion of webs and application of scripting languages, analysis of string-manipulating programs, i.e. the programs, control flow of which is dependent on strings,

has gained a significant interest. A popular approach of analysing how strings are being manipulated is *symbolic execution* [7, 45, 35, 59], which are based on constraint solvers over the string domain [3, 5, 26, 42]. The general string solving with naive use of concatenation, string-length comparisons and finite-state transducers yields undecidability [43] but researchers actively focus this area to discover useful decidable fragments of the theory. In the present recherche, we will focus on the techniques applied in the string solver SLOTH [32], which was the first practically usable tool able to solve *acyclic* and *straight-line* fragments of the SMT (*Satisfiability Modulo Theory* [21]) over the string logic constraints. The formal definition of the two fragments is out of the scope of the present text, for more details, we refer the interested reader to [32].

The symbolic execution engine takes a program as its input. The string manipulations in the program (e.g. concatenation, replacement of a regular expression match by a given string, or extracting the length of a string) are encoded as finite transducers (which are in the alternating form for succinctness). Assertions in the form of pattern matches, i.e. `assert` `¬(x ∼ /regex/)`, are encoded as alternating finite automata. The overall program is then expressed as a conjunction of the constraints expressed by these automata and transducers. For the conjunctions that hold the acyclicity or straight-lineness, Holík et al. [32] provides recipes for combining the present automata and transducers into one multitrack alternating finite automaton, language of which is empty iff all the assertions are invariantly satisfied. Otherwise, a descriptive counter-example can be extracted from the accepting run of the AFA.

To solve the AFA emptiness problem, Holík et al. [32] uses the state-of-the-art AFA analysis approaches and tools (described above, in the section 3), namely ABC and NUXMV.

4.2 Shape Analysis

The term *shape analysis* denotes verification of programs using heap-allocated dynamic data structures. Distinctive features of such programs are:

- pointer variables which reference memory locations in the heap;
- statements for allocating memory locations in the heap and storing pointer to the location in a pointer variable;
- statements for freeing memory locations referenced by given pointers.

Typical properties of interest concerning the heap-manipulating programs are existence of *garbage* (memory that is allocated but no longer referenceable; memory leak), dereferencing of **nil** pointers or reading uninitialized memory. Shape invariants of dynamic data structure (so-called *heap graph*) can be also verified, e.g. questions can be posed, such as "Is the memory location referenced by the pointer p a root of a tree? Is it a root of a DAG?".

Undecidability of shape analysis has been shown in [41, 52]. Several approaches however exist that are specialized to a decidable fragment of shape analysis or admit the general undecidability but use over-approximation, providing imprecise, yet conservative results.

Regular model checking: In the automata-based regular model checking approaches, the program state is encoded as a word (or a tree) that includes information particularly about the program location, the program stack variable valuations (uninitialized/nil/reference to a heap graph node) and *heap graph*, nodes of which represent allocated memory locations,

edges represent the pointers between the locations. A nondeterministic transducer is generated from the program. If considering the simplest case, 1-selector-linked structures (linear and cyclic lists), the size of the program location and variable valuations is bounded, therefore encodable into the transducer's state[6]. The transducer non-deterministically guesses the next program location and variable valuations after reading the two portions of the current program state. Afterwards, when processing and rewriting the heap graph, the guess is checked for consistency with the current data in the heap. If inconsistent, a transition to a nonaccepting sink state is performed. The transducer is capable of detecting nil pointer dereferencing, read from uninitialized memory or violating the shape invariants. The information about the error is then written to the output tape as an invalid state and propagated through any number transducer applications (tape rewrites).

Recall the notation of a transducer relation $\tau : \Sigma^* \times \Sigma^*$. Problematic (and undecidable) is the computation of the transitive closure (fixpoint) $\tau^*(I)$ of the transducer relation when applied to the set of initial program states I. For very special classes of programs, the transitive closure can be computed precisely [1, 2]. In general cases, over-approximating conservative techniques are being used, based on abstraction refinement [9, 11], extrapolation and widening [62, 9, 8], regular grammar inference [28], none of which is guaranteed to terminate if demanding precise results. A related approach using counting automata [58] abstraction is presented in [10]. The use of counters in automata allows to represent the heap graph in a bounded way (for 1-selector-linked structures), thus enabling whole program state to be encoded in the finite transducer's memory. Although the memory location reachability problem is still undecidable in general, properties of some special classes of programs can be proven with counting automata.

Automata based Hoare-style reasoning The reasoning with the Hoare rules is an approach with theorem-proving flavour. Two predominant paradigms for defining heap semantics exist:

- *Store-based semantics* represent the heap as a collection of memory locations, pointer variables and a mapping from memory locations to memory locations, which describes edges of the heap graph [64, 50].
- In *storeless semantics*, every memory location is identified with the set of paths that lead to the corresponding node in the heap graph. A natural representation of the set of paths is using regular languages and finite automata [31, 12]. Therefore, this paradigm is our matter of interest.

The mix of store-based and storeless semantics has been introduced and applied in [54].

Chakraborty [17] presents the basic principle of the approaches based on storeless semantics paradigm by defining a minimalistic *simplified alias logic (SAL)* and Hoare inference rules for the logic. Presenting the logic and the rules is beyond the scope of the present recherche. As Chakraborty [17] states, the Hoare triple is derivable by repeated applications of given inference rules but the pitfall of the approach is in undecidability of satisfiability checking in SAL. The technique is however effectively applicable on several interesting programs.

[6] if considering general graph structures, tree transducers are used and the reader is referenced to [11]

References

[1] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and effcient. In *International Conference on Concurrency Theory*, pages 116–131. Springer, 2002.

[2] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Algorithmic improvements in regular model checking. In *International Conference on Computer Aided Verification*, pages 236–248. Springer, 2003.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *International Conference on Computer Aided Verification*, pages 150–166. Springer, 2014.

[4] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2): 183–235, 1994.

[5] Clark Barrett, Cesare Tinelli, Morgan Deters, Tianyi Liang, Andrew Reynolds, and Nestan Tsiskaridze. Efficient solving of string constraints for security analysis. In *Proceedings of the Symposium and Bootcamp on the Science of Security*, pages 4–6. ACM, 2016.

[6] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, page 1. ACM, 2007.

[7] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 307–321. Springer, 2009.

[8] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large. In *International Conference on Computer Aided Verification*, pages 223–235. Springer, 2003.

[9] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract regular model checking. In *International Conference on Computer Aided Verification*, pages 372–386. Springer, 2004.

[10] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomáš Vojnar. Programs with lists are counter automata. In *International Conference on Computer Aided Verification*, pages 517–531. Springer, 2006.

[11] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *International Static Analysis Symposium*, pages 52–70. Springer, 2006.

[12] Marius Bozga, Radu Iosif, and Yassine Lakhnech. On logics of aliasing. In *International Static Analysis Symposium*, pages 344–360. Springer, 2004.

[13] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-18275-4.

[14] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14295-6.

[15] Marie-Pierre Béal and Olivier Carton. Determinization of transducers over finite and infinite words. *Theoretical Computer Science*, 289(1):225 – 251, 2002. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(01)00271-7. URL `http://www.sciencedirect.com/science/article/pii/S0304397501002717`.

[16] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08867-9.

[17] Supratik Chakraborty. Reasoning about heap manipulating programs using automata techniques. In *Modern Applications of Automata Theory*, pages 193–228. World Scientific, 2012.

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

[19] Karel Culik II and Jarkko Kari. Image compression using weighted finite automata. *Computers & Graphics*, 17(3):305–313, 1993.

[20] Loris D'Antoni, Zachary Kincaid, and Fang Wang. A symbolic decision procedure for symbolic alternating finite automata. *Electronic Notes in Theoretical Computer Science*, 336:79–99, 2018.

[21] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[22] Martin De Wulf, Laurent Doyen, Thomas A Henzinger, and J-F Raskin. Antichains: A new algorithm for checking universality of finite automata. In *International Conference on Computer Aided Verification*, pages 17–30. Springer, 2006.

[23] Henning Dierks. Specification and verification of polling real-time systems. In *Ausgezeichnete Informatikdissertationen 1999*, pages 32–41. Springer, 2000.

[24] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata*. Springer Science & Business Media, 2009.

[25] Deepak D'souza and Priti Shankar. *Modern Applications of Automata Theory*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2011. ISBN 9789814271042, 9814271047.

[26] Loris D'Antoni and Margus Veanes. Static analysis of string encoders and decoders. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 209–228. Springer, 2013.

[27] Pierre Ganty, Nicolas Maquet, and Jean-François Raskin. Fixed point guided abstraction refinement for alternating automata. *Theoretical Computer Science*, 411(38):3444 – 3459, 2010. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2010.05.037. URL `http://www.sciencedirect.com/science/article/pii/S0304397510003294`. Implementation and Application of Automata (CIAA 2009).

[28] Peter Habermehl and Tomáš Vojnar. Regular model checking using inference of regular languages. *Electronic Notes in Theoretical Computer Science*, 138(3):21–36, 2005.

[29] Subramanian Hariharan and Priti Shankar. Compressing xml documents with finite state automata. In *Proc. of the Int'l Conference on Implementation and Application of Automata*, 2005.

[30] Zyad Hassan, Aaron R Bradley, and Fabio Somenzi. Better generalization in ic3. In *2013 Formal Methods in Computer-Aided Design*, pages 157–164. IEEE, 2013.

[31] Charles Antony Richard Hoare and He Jifeng. A trace model for pointers and objects. In *European Conference on Object-Oriented Programming*, pages 1–18. Springer, 1999.

[32] Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.*, 2(POPL):4:1–4:32, December 2017. ISSN 2475-1421. doi: 10.1145/3158092. URL `http://doi.acm.org/10.1145/3158092`.

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

[34] Galina Jirásková. State complexity of some operations on binary regular languages. *Theoretical Computer Science*, 330(2):287–298, 2005.

[35] Scott Kausler and Elena Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 259–270. ACM, 2014.

[36] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 158–173, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8.

[37] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, Dec 2002. ISSN 0278-0070. doi: 10.1109/TCAD.2002.804386.

[38] Andreas Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 50–57. IEEE Computer Society, 2004.

[39] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 155–164. ACM, 2007.

[40] Christophe Ladroue and Sara Kalvala. Constraint-based genetic compilation. In Adrian-Horia Dediu, Francisco Hernández-Quiroz, Carlos Martín-Vide, and David A. Rosenblueth, editors, *Algorithms for Computational Biology*, pages 25–38, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21233-3.

[41] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[42] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. An efficient smt solver for string constraints. *Formal Methods in System Design*, 48(3):206–234, 2016.

[43] Anthony W Lin and Pablo Barceló. String solving with word equations and transducers: towards a logic for analysing mutation xss. In *ACM SIGPLAN Notices*, volume 51, pages 123–136. ACM, 2016.

[44] Christof Löding. Basics on tree automata. In *Modern Applications of Automata Theory*, pages 79–109. World Scientific, 2012.

[45] Blake Loring, Duncan Mitchell, and Johannes Kinder. Expose: practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 196–199. ACM, 2017.

[46] Feng Lu and K-T Cheng. Ichecker: An efficient checker for inductive invariants. In *2006 IEEE International High Level Design Validation and Test Workshop*, pages 176–180. IEEE, 2006.

[47] A. Mishchenko, M. Case, R. Brayton, and S. Jang. Scalable and scalably-verifiable sequential synthesis. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 234–241, Nov 2008. doi: 10.1109/ICCAD.2008.4681580.

[48] Alan Mishchenko, Niklas Een, Robert Brayton, Jason Baumgartner, Hari Mony, and Pradeep Nalla. Gla: Gate-level abstraction revisited. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1399–1404. EDA Consortium, 2013.

[49] Hari Mony, Jason Baumgartner, Viresh Paruthi, and Robert Kanzelman. Exploiting suspected redundancy without proving it. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 463–466. IEEE, 2005.

[50] Andreas Podelski and Thomas Wies. Boolean heaps. In *International Static Analysis Symposium*, pages 268–283. Springer, 2005.

[51] Michael O Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.

[52] Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[53] Clifford A Reiter. Synthetic coloring of fractal terrains and triangular automata. *The Visual Computer*, 11(6):313–318, 1995.

[54] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *ACM SIGPLAN Notices*, volume 40, pages 296–309. ACM, 2005.

[55] Yen-Sheng Ho Alan Mishchenko Robert and Brayton Niklas Een. Enhancing pdr/ic3 with localization abstraction.

[56] Dana Ron, Yoram Singer, and Naftali Tishby. On the learnability and usage of acyclic probabilistic finite automata. *Journal of Computer and System Sciences*, 56(2):133–152, 1998.

[57] Iztok Savnik. Index data structure for fast subset and superset queries. In *International Conference on Availability, Reliability, and Security*, pages 134–148. Springer, 2013.

[58] M. P. Schützenberger. Finite counting automata. *Information and Control*, 5:91–107, 1962.

[59] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

[60] Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *SAT*, page 31, 2009.

[61] Alin Ştefănescu, Javier Esparza, and Anca Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *International Conference on Concurrency Theory*, pages 27–41. Springer, 2003.

[62] Tayssir Touili. Regular model checking using widening techniques. *Electronic Notes in Theoretical Computer Science*, 50(4):342–356, 2001.

[63] Wil MP van der Aalst, HT De Beer, and Boudewijn F van Dongen. Process mining and verification of properties: An approach based on temporal logic. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 130–147. Springer, 2005.

[64] Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. *The Journal of Logic and Algebraic Programming*, 73 (1-2):111–142, 2007.

[65] Qi Zhu, Nathan B Kitchen, Andreas Kuehlmann, and Alberto Sangiovanni-Vincentelli. Sat sweeping with local observability don't-cares. In *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, pages 129–148. Springer, 2011.

[66] Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO-Theoretical Informatics and Applications*, 21(2):99–135, 1987.