Antichain with SAT and Tries

Lukáš Holík
Faculty of Information Technology, Brno University of Technology, Czech Republic

Pavol Vargovčík
Faculty of Information Technology, Brno University of Technology, Czech Republic

Abstract

We introduce a SAT-enabled version of an antichain algorithm for checking language emptiness of alternating finite automata (AFA) with complex transition relations encoded as compact logical formulae. The SAT solver is used to compute predecessors of AFA configurations, and at the same time, to evaluate the subsumption of newly found configurations in the antichain of the previously found ones. The algorithm could be naively implemented by an incremental SAT solver where the growing antichain is represented by adding new clauses. To make it efficient, we 1) force the SAT solver to prioritize largest/subsumption-strongest predecessors (so that weaker configurations are not even generated), and 2) store the antichain clauses in a special variant of a trie that allows fast subsumption testing. The experimental results suggest that the resulting emptiness checker is very efficient compared to the state of the art and that our techniques improve the performance of the SAT solver.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Regular languages

Keywords and phrases SAT, Trie, Antichain, Alternating automata, Subset query

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.15

Supplementary Material *Software (Source Code)*: `https://github.com/p4l1ly/antisat/tree/5901055e699e7b95ba3dec63d11445fcd791f5de` [50]
archived at `swh:1:rev:5901055e699e7b95ba3dec63d11445fcd791f5de`

Funding This work was supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908 and the FIT BUT internal project FIT-S-23-8151.

1 Introduction

This paper presents a modification of a known antichain algorithm for deciding *language emptiness of alternating finite automata* (*AFA* and *AFA emptiness* for short) that utilizes SAT solving to handle complex transition relations over large structured alphabets.

AFA add conjunctive branching to nondeterminism. That is, a transition may require the rest of the word is accepted not by one successor state, but by each state in a set of successor states. This makes them exponentially more succinct than non-deterministic automata [12, 36]. The succinctness of AFA is paid for by the PSPACE-completeness of the language emptiness test. The test is based on a *de-alternation* that converts the AFA to a nondeterministic finite automaton (NFA), of at most exponential size, and checks the reachability of a final state from an initial one in the NFA. The de-alternation constructs the NFA by making explicit reachable *configurations* of the AFA and transitions between them, where a configuration is a set of all conjunctive branches possibly reached after reading some word. Efficient AFA emptiness testing algorithms mitigate the worst-case complexity by focusing on the reachability question while avoiding generating the entire NFA. They exploit the structure of the configurations to define effective state space pruning and abstraction techniques. The search is done on-the-fly, while constructing the NFA, the NFA transitions stay implicit.

AFA can be practical in applications where automata are combined with Boolean and similar operations, as demonstrated in works on string solving [1, 51], LTL model checking such as [48, 19, 26], analyzing regular expressions [15, 14], and deciding logics as WS1S [47, 25] and have a strong potential to improve efficiency in regular model checking [7, 2, 53] or deciding linear arithmetics [6, 52]. These applications indeed generate AFA with complex transition relations over large alphabets (often in the form of bit-vectors). To keep them manageable, it is necessary to work with transition relations represented symbolically.

We present a variation on the *antichain AFA emptiness test* of [22] that targets this scenario. The original antichain algorithm fights the NFA state-space explosion by pruning the reached configurations that are smaller than others (we say they are *subsumed*). This is guaranteed to preserve the answer to the reachability query. We use the variant with the backward exploration, found superior in [15, 24]. We assume that complex transition relations are represented symbolically as boolean formulae over states and symbol bits. In this case, the main bottlenecks are the computation of predecessors of already reached configurations and the pruning of predecessors that are subsumed by other already reached configurations.

Our main contribution is a way to address both these bottlenecks simultaneously and efficiently using a SAT solver. We use the SAT solver as a white box, giving us a direct access to its internal data structures and routines and to leverage their efficiency optimally. Specifically, (1) we use SAT solving with preferences [11, 21] and force the SAT solver to give priority to solutions that correspond to subsumption stronger configurations. We also encode the subsumption testing against reached configurations into the SAT query, and we boost its performance by equipping the SAT solver with a specialised data structure, based on a trie, for storing the antichain of reached configurations. We introduce novel techniques that optimize the trie. Mainly, we allow elements of every branch of the trie to be sorted in its unique way that maximizes prefix sharing (the standard trie uses a global fixed order), and we integrate into it a technique of double clause watches to facilitate fast unit propagation.

We compared an implementation of our AFA emptiness check against the best AFA emptiness checkers on a comprehensive benchmark from [24] (which compares accessible AFA checkers). The results confirm a positive impact of our new techniques and show that our AFA checker outperforms the state of the art on a significant portion of the benchmark, such as examples coming from boolean combinations of regular properties, and has a strongly orthogonal performance to the other checkers.

2 Related Work

Existing approaches to AFA emptiness and related problems [15, 34, 51, 28] adapt model-checking algorithms such as IC3/PDR [8, 33, 9] or Impact [39], or use purely automata approaches, antichain algorithms [54, 22], on which we build here, possibly enhanced with the abstraction [29] or the up-to congruence techniques [16]. Available existing implementations were recently compared in [24], showing complementary results of different tools and algorithms. We note that despite being presented as a modification of the antichain algorithm, our techniques could be applied to most of the above algorithms, and to any algorithm that generates subset-maximal predecessors/successors under a transition relation formula and prunes away those subsumed by existing configurations (also e.g. antichain algorithms for inclusion checking of NFA, tree automata, context-free grammars).

Many approaches and tools use symbolic representations of transitions, BDDs, logical formulae, or intervals of numbers [41, 35, 32, 37]. The concept of symbolic automata encapsulates a class of symbolic representations generically [17, 16]. The works [15, 34, 51] end up translating the transition relations to and-inverter graphs [40] and delegating the decision problem to a model checker. Our representation of AFA, with the transition relation described by boolean formulae over states and symbol bits, is one of the most permissive and hence compact and flexible (the other approaches require for instance that the formula is a conjunction of a part talking only about symbols and a part talking only about states, which by itself may lead to verbosity).

Among many existing SAT solvers, MiniSAT [23], on which we build, is relatively old, but still performing well on a number of real-world problems. It is a textbook example of a modern SAT solver based on the CDCL algorithm (conflict-driven clause learning) which is simple and well-documented. All modern SAT solvers are based on CDCL (except few based on deep learning [43, 3]) and share many similarities with MiniSAT, some of them [4, 44] are even founded on its source code. They differ mostly in heuristics, preprocessing, inprocessing (i.e. simplification of the problem during solving), and specialized features (like XOR support in Cryptominisat [44]). The SAT Competition 2023 was dominated by CaDiCaL and Kissat [5], which are focused on advanced heuristics and inprocessing, fine-tuned for typical SAT benchmark classes.

SATO solver [55] is the only SAT solver that used a trie-based data structure and achieved moderate speed-ups in general SAT solving. Unlike us, they did not specialise in AFA emptiness, trie in SATO were not combined with clauses and CDCL was hence harder to implement efficiently (SATO has dropped its use of tries before implementing CDCL), they also did not consider the mixed-order trie. Our implementation of watched literals also allows more utilization of the prefix sharing in the trie. The trie was also used in the CNF preprocessor of Alembic [31], to achieve reusing the preprocessor state on shared prefixes of clauses.

We use the trie primarily for the *subset query*: testing whether a given set has a superset among sets stored in a data structure. There are alternative data structures for that. We have made a preliminary comparison of the applicable data structures: LVBDDs [30], covering sharing trees [20], set tries [27], and SAT, which has indicated that the trie has the best performance and is simpler than the covering sharing trees that performed similarly.

3 Preliminaries

Boolean formulas and SAT. A *Boolean formula* ψ over a set of *variables* X is generated by the grammar $\psi ::= 0 \mid 1 \mid x \mid \psi \land \psi \mid \psi \lor \psi \mid \neg\psi$ where $x \in X$. We use $\mathcal{B}(X)$ to denote the set of all Boolean formulae over X. A *positive Boolean formula* is a Boolean formula without negation and 0. The set of all positive Boolean formulae over X is denoted $\mathcal{B}^+(X)$. A *literal* is a variable or a negated variable. $m \models \varphi$ denotes that the *(boolean) assignment* $m : X \to \{0, 1\}$ *satisfies* φ (it is a *model/solution*). We will often abuse the notation and treat a boolean assignment as the set of variables assigned 1 in it (we write $x \in m$ to denote $m(x) = 1$ and relate assignments using $\subseteq$). φ is *satisfiable* iff it has a model. A formula in *CNF (conjunctive normal form)* is a conjunction of *clauses*, each being a disjunction of literals.

An instance of the *satisfiability problem (SAT)* is a CNF, and its solution is a model or the answer UNSAT if no model exists. For simplicity, we will present our algorithms in the context of the well-known Davis–Putnam–Logemann–Loveland (DPLL) method [18], even though most of the modern solvers (including ours) are based on its successor, conflict-driven clause learning (CDCL) [38, 42]. Our novel techniques are not affected by the CDCL extension.

Algorithm 1 DPLL augmented for computing preferred models [11].

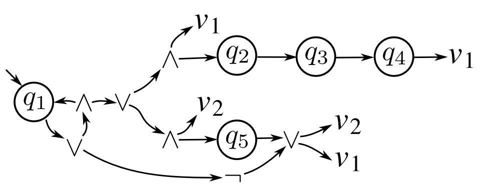
```
Input: $m$: a partial model, $P$: preferred literals, $\varphi$: immutable CNF, $\varphi_{\mathrm{mut}}$: mutable
       CNF
Output: a set $M$ of preferred models (w.r.t. $P$) of $\varphi \wedge \varphi_{\mathrm{mut}}$
1  Function SAT-PREF-REC($m$, $P$, $\varphi$, var $\varphi_{mut}$):
       // $m'$ is an updated (partial) model or CONFLICT, resp. UNSAT
2      $m' \leftarrow$ unit-propagate($m$, $\varphi \wedge \varphi_{mut}$);
3      if $m' \neq$ CONFLICT then
4          if $m'$ has an unassigned preferred literal $p \in P$ then
5              return SAT-PREF-REC($m' \cup \{p\}$, $P$, $\varphi$, var $\varphi_{mut}$) $\cup$
6                      SAT-PREF-REC($m' \cup \{\neg p\}$, $P$, $\varphi$, var $\varphi_{mut}$);
7          else
8              $m' \leftarrow$ SAT($m'$, $\varphi \wedge \varphi_{mut}$);
9              if $m' \neq$ UNSAT then
10                 $\varphi_{\mathrm{mut}} \leftarrow \varphi_{\mathrm{mut}} \wedge \bigvee_{\lambda \in P \wedge m' \models \neg\lambda} \lambda$;
11                 return $\{m'\}$;
12     return $\emptyset$;
```

SAT with preferences. We consider an extension of DPLL introduced in [11] that adds a set of *preferred literals* P as an additional input to the problem. *SAT with preferences* is a procedure `SAT-PREF(`P,φ, φ_{mut}`)` that, given a set of *preferred literals* P, the *immutable part* φ of the input formula, and the *mutable part* φ_{mut}, returns the set of *preferred models* M of the conjunction $\varphi \wedge \varphi_{\mathrm{mut}}$. The set of the preferred models consists of exactly one representative per every class of models of $\varphi \wedge \varphi_{\mathrm{mut}}$ satisfying some maximal subset of P (maximal wrt. $\subseteq$). The input formula is split into two parts because they are internally processed differently. The mutable part is iteratively updated and a specialised data structure for it is a major part of our contribution. Since it is relevant to the technicalities of our work, we recall in more detail its implementation through a recursive procedure `SAT-PREF-REC` in Algorithm 1 (it is initially called with the empty partial model m). It is a basic DPLL scheme with two differences:

1. On lines 4-6, the splitting heuristic chooses from the unassigned preferred literals first or continues with the normal SAT procedure if all preferred literals are assigned.
2. If a model is found (at line 8), it is not returned immediately. Instead, the model is added to the set of results and a clause is added (line 10) that prunes models with a subset of positive preferred literals from the further search.

Alternating automata. Let $V = \{v_1, v_2, \ldots\}$ be a set of *symbol-variables*. An assignment $a : V \to \{0, 1\}$ is a *symbol* and Σ is an *alphabet*, the set of all symbols. A *word* is a sequence of symbols and Σ^* is the set of all (finite) words.

We consider alternating automata in a form used in [49]. Symbols are encoded as bit-vectors and the transition relation for each state is given as a Boolean formula over states and symbol-bits. The solutions of the transition formula encode both the symbol as well as the set of states into which the state can transition. The formula allows to mix symbol-variables and states almost arbitrarily, the only restriction is that states do not appear under negation. This is more permissive and succinct than in other works, which typically require a separation of a formula talking about symbols from the one talking about states, and sometimes even the state formula in DNF.



$Q = \{q_1 \cdots q_5\}, \quad V = \{v_1, v_2\}, \quad \iota = q_1, \quad F = \emptyset$
$\delta(q_1) = \neg(v_1 \vee v_2) \vee (q_1 \wedge ((v_1 \wedge q_2) \vee (v_2 \wedge q_5)))$
$\delta(q_2) = q_3$
$\delta(q_3) = q_4$
$\delta(q_4) = v_1$
$\delta(q_5) = v_2 \vee v_1$

Figure 1 AFA generated from an LTL formula $((v_1XXXv_1) \vee (v_2X(v_2 \vee v_1)))U\neg(v_1 \vee v_2)$. Even though its set of final states F is empty, its language is not empty. A short accepting run exists, namely $\{q_1\}[v_1{=}0, v_2{=}0]\emptyset$, as the transition to the empty configuration is enabled, as the model $a \cup c_2 = \emptyset \cup \emptyset = [v_1{=}0, v_2{=}0, q_1{=}0, \cdots, q_5{=}0]$ satisfies the first operand of the top-level disjunction in $\delta(q_1)$. An example of a longer accepting run would be $\{q_1\}[v_1{=}v_2{=}1]\{q_1, q_2\}[v_1{=}v_2{=}0]\{q_3\}[v_1{=}v_2{=}0]\{q_4\}[v_1{=}v_2{=}1]\emptyset$.

Formally, an *alternating finite automaton (AFA)* is a quintuple $\mathcal{M} = (Q, V, \delta, \iota, F)$ where 1) Q is a finite set of *states*; 2) V is a finite set of *symbol-variables*; 3) $\delta : Q \to \mathcal{B}^+(Q \cup \mathcal{B}(V))$ is a *transition function*; 4) $\iota \in Q$ is the *initial state*; and 5) $F \subseteq Q$ is the set of *final states*. A *configuration* of $\mathcal{M}$ is a boolean assignment to states $c : Q \to \{0,1\}$. The automaton has the single *initial configuration* $\{\iota\}$ and the subsets of F are *final configurations*. A configuration k *transitions* into a configuration ℓ via a symbol a, denoted as $k \xrightarrow{a} \ell$, whenever $a \cup \ell \models \bigwedge_{q \in k} \delta(q)$. We write $k \longrightarrow \ell$ if there exists a symbol a such that $k \xrightarrow{a} \ell$ and say that ℓ is a *successor* of k and that k is a *predecessor* of ℓ. We will usually use c, d for configurations in a general context and k, ℓ when speaking about a pair of a predecessor and a successor. We write $\mathsf{Pre}(a, \ell)$ and $\mathsf{Pre}(\ell)$ to denote the sets $\{k \mid k \xrightarrow{a} \ell\}$ and $\{k \mid k \longrightarrow \ell\}$ of predecessors of ℓ via the symbol a or via any symbol, respectively. A *run* of $\mathcal{M}$ over a word $a_1 \cdots a_n \in \Sigma^*$ is a sequence $c_0, a_1, c_1, a_2, \ldots, a_n, c_n$ of configurations interleaved with symbols such that for all $i : 0 \leq i < n$, $c_i \xrightarrow{a_{i+1}} c_{i+1}$. The run is *accepting* if $c_0 = \{\iota\}$ and $c_n \subseteq F$. The *language* of $\mathcal{M}$ is the set $\mathcal{L}(\mathcal{M})$ of all words with an accepting run. The *AFA emptiness problem*, that we are concerned with in this paper, is to decide whether $\mathcal{L}(M) = \emptyset$. Figure 1 has an example of an automaton with a non-empty language and its accepting run. In the rest of the paper, we will consider a fixed AFA $\mathcal{M} = (Q, V, \delta, \iota, F)$.

Antichain algorithm for AFA emptiness. In this paragraph, we will recall the backward antichain algorithm for AFA emptiness which we optimize in this paper. AFA emptiness is well known to be a PSPACE-complete problem. It requires deciding the reachability of a final configuration from the initial one in the graph of the relation $\longrightarrow$. A *naive forward algorithm* explores the graph from the initial configuration in a search for a final one, the *naive backward algorithm* starts from the final configurations and searches for the initial configuration following $\longrightarrow$ backwards. An obvious problem is the possible exponential explosion of the search space of configurations. The antichain algorithms, first proposed in [22], introduce a search space pruning strategy based on subsumption of configurations. Antichain algorithms exist in several variants, from which we use the backward antichain. It is based on the backward naive algorithm and was found empirically superior over the forward variant in [15, 24].

Let us first outline the essence of the backward antichain informally. The *subsumption* relation used to prune the space of configuration is, in its basic variant which we use here, simply the set containement: a configuration c' *subsumes* a configuration c iff $c \subseteq c'$. The

Algorithm 2 Basic antichain algorithm.

```
Input: AFA $\mathcal{M} = (Q, V, \delta, \iota, F)$
Output: Is the language of $\mathcal{M}$ empty?
1  if $\iota \in F$ then return NOT EMPTY;
2  $Visited \leftarrow \{F\}$;
3  $W \leftarrow \{F\}$;
4  while $W \neq \emptyset$ do
5      $\ell \leftarrow \mathsf{pop}(W)$;
6      for $a \in \Sigma$ do
7          for $k \in \mathsf{Pre}(a, \ell)$ do
8              if not $k \Subset Visited$ then
9                  if $\iota \in k$ then return NOT EMPTY;
10                 $Visited \leftarrow \lceil Visited \cup \{k\} \rceil$;
11                 $\mathsf{push}(W, k)$;
12 return EMPTY
```

key property that allows to discard the configurations subsumed by other already reached configurations in antichain algorithms is that the subsumption is a simulation relation on the (reversed) relation $\longrightarrow$. The property that holds with our definition of AFA is actually even stronger: If $k \longrightarrow \ell$, $k' \subseteq k$, and $\ell \subseteq \ell'$, then $k' \longrightarrow \ell'$. This can be shown to justify the correctness of an optimized AFA emptiness test that: (1) starts the backward exploration of configuration space from only the maximal accepting configuration F, (2) keeps only the *antichain* (a set of incomparable elements) of the subsumption-maximal configurations through the search, and (3) concludes reaching the target on finding any configuration that includes ι.

Let us now recall the backward antichain algorithm for testing AFA emptiness more formally. For a set/configuration c and a set of configurations/sets A, the *subset query* of c towards A, denoted $c \Subset A$, stands for $\exists d \in A. c \subseteq d$. An *antichain* is a set of elements incomparable wrt some preorder, here a set of configurations ordered by $\subseteq$. For any set of configurations A, $\lceil A \rceil = \{c \in A \mid \neg\exists d \in A.\ c \subset d\}$ is the set of its inclusion-maximal elements.

The pseudocode of the backward antichain algorithm is in Algorithm 2. In addition to the antichain of the inclusion-maximal set of visited configurations *Visited*, it maintains a worklist W of configurations to be explored, where exploring a configuration means finding all its predecessors on lines 6-7, filtering out the subsumed ones on line 8, checking for the initialness on line 9, and updating the antichain and worklist on lines 10-11. In the paper, we focus on scenarios where the transition relation is large as a consequence of a large alphabet Σ. In such cases, Algorithm 2 has two bottlenecks.

First, the algorithm spends most of its time on line 8 with the subset query, where, for every predecessor of a visited configuration ℓ, it checks that it is not subsumed by other visited configurations stored in the antichain *Visited*. The efficiency of this subset query is obviously the primary concern since the antichain may grow exponentially large. A similar and also expensive operation is line 10, pruning the antichain from configurations subsumed by k. Line 10 is however called only a fraction of times line 8 is called (only when the test on line 8 passes), so it is much cheaper overall.

The second bottleneck is the iteration through all predecessors, $\mathsf{Pre}(\ell)$, on lines 6 and 7. We call it *predecessor iteration*. When the alphabet is large, it is often the case that $\mathsf{Pre}(a, \ell)$ and $\mathsf{Pre}(b, \ell)$ have a non-empty intersection, and the inner for-loop body is thus called many times to generate the same predecessor k.

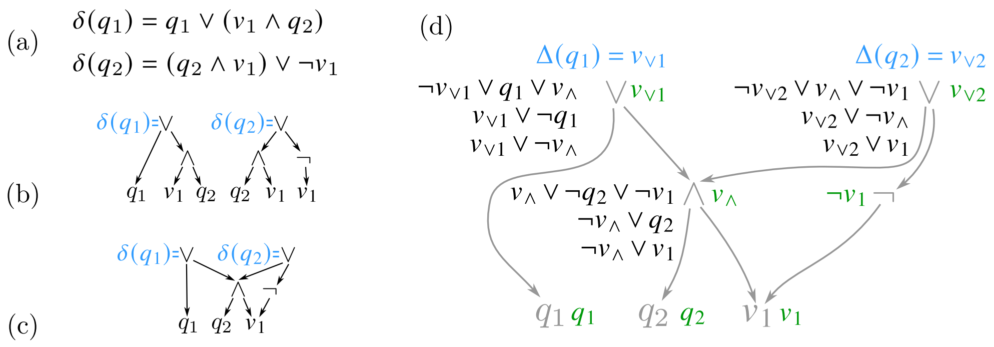


Figure 2 Representation of δ and its transformation to CNF. The transition formulae in (a) correspond to the trees in (b) where isomorphic sub-trees can be shared as in (c) and the CNF obtained by Tseyitin transformation is in (d). The links connecting states to their transition formula in (b) and (c) are shown in blue. In (d), the green literals are the names assigned by the Tseyitin transformation to the original sub-formulae (shown in grey), Δ is shown in blue, and original sub-formulae are also annotated with their equivalents produced by Tseyitin transformation. The Tseytin variables are indexed by the top-most operator of the sub-formulae they represent plus a number to distinguish between sub-formulae with the same top-most operator. Δ is depicted in blue.

This paper addresses the two bottlenecks, the subset query and the predecessor iteration, via the use of a succinct symbolic representation of the transition function and SAT solvers.

4 Symbolic Transition Function as Formula DAG

Various tools use a symbolic representation of large transition relations with large alphabets, using BDDs, intervals, formulae, effective boolean algebras, and implement the predecessor iteration efficiently in a manner that does not unfold the succinct representation. We use an alternative that is general and often very succinct. The transitions from a state are represented as a state-positive boolean formula in a compact DAG form (each single sub-formula is represented only once). The transition function δ particularly maps states to boolean formulae. The forest of abstract syntax trees of all formulae in the image of δ is a directed acyclic graph (DAG). Nodes are labelled by the function symbols $\mathtt{0}, \mathtt{1}, q \in Q, v \in V, \wedge, \vee, \neg$ and represent subformulae, and each state q in the domain of δ is *linked* to the node representing $\delta(q)$. See Figure 2b, where links from the domain are blue. Structurally equivalent nodes are merged, see Figure 2c.

Since we are going to process the formulae in a SAT solver, we transform the forest of transition formulae into CNF using the Tseytin transformation. It is almost the standard Tseytin transformation, up to that it does not generate a Tseytin variable for the entire formula. We briefly detail it below. It generates the CNF formula $\mathsf{CNF}(\delta)$. First, the *name* of a formula φ, $n(\varphi)$, is defined as φ itself if φ is a variable, as $\neg(n(\psi))$ if φ is $\neg\psi$, and as the *Tseytin variable* v_φ if $\varphi = \psi \oplus \psi', \oplus \in \{\wedge, \vee\}$. In the last case, we also define a *naming formula* of φ as the CNF version of the equivalence $v_\varphi \leftrightarrow n(\psi) \oplus n(\psi')$. Finally, $\mathsf{CNF}(\delta)$ is the conjunction of all naming formulae of (con/dis)junctions that are sub-formulae in the image of δ.

The representation of δ that we will use then consists of 1) the formula $\mathsf{CNF}(\delta)$, 2) the mapping $\Delta : Q \to V_T \cup V \cup Q \cup \{\neg v \mid v \in V_T \cup V\}$ that maps each $q \in Q$ to the name of $\delta(q)$, the literal $n(\delta(q))$. An example is in Figure 2d.

5 Predecessor-subset Query with SAT Solvers

We will now show how SAT solving with preferences can be used in AFA emptiness check in a way that makes good use of the fast internal data structures of the solver, and how to further improve this by specialised data structures.

We combine the subset query and the predecessor iteration into a *predecessor-subset query*. Given a successor ℓ and an antichain of configurations *Visited*, the predecessor-subset query returns the antichain of maximal predecessors of ℓ that are not subsumed by *Visited*, that is, the set

$$\mathsf{newpre}(\ell, \mathit{Visited}) = \lceil \mathsf{Pre}(\ell) \cup \mathit{Visited} \rceil \setminus \mathit{Visited}$$

This is done by calling the procedure $\texttt{SAT-PREF}(P := \Delta(Q), \varphi, \varphi_{\mathit{Visited}})$ from Algorithm 1 on the formula that consists of the immutable part φ that represents the AFA transition relation and ℓ, and the mutable part $\varphi_{\mathit{Visited}}$ that represents the antichain *Visited*. The set of preferred literals is specified as $P := \Delta(Q)$, the variables that correspond to the predecessor states in the CNF formula $\mathsf{CNF}(\delta)$ from Section 4 representing the transition relation.

The immutable part φ is

$$\varphi := \mathsf{CNF}(\delta) \wedge \varphi_\ell \quad \text{where} \quad \varphi_\ell = \bigwedge_{q \in Q \setminus \ell} \neg q$$

After the transition relation in the first conjunct, the second conjunct φ_ℓ expresses that we want a predecessor of particularly ℓ. It asserts that after taking the transition from the predecessor, no state outside ℓ should be touched.

The mutable formula $\varphi_{\mathit{Visited}}$, used to prune the solutions subsumed by the antichain *Visited*, is

$$\varphi_{\mathit{Visited}} := \bigwedge_{c \in \mathit{Visited}} \bigvee_{q \in Q \setminus c} \Delta(q)$$

Here, a clause for $c \in \mathit{Visited}$ says that some state not in c must appear in a new predecessor clause, a solution to the SAT problem. The new predecessor cannot be a subset of c. The antichain *Visited* is initialised with $\mathit{Visited} := \{F\}$.

The predecessor will be read from each model m_k from the returned set of preferred models as

$$k(m_k) := \{q \in Q \mid m_k \models \Delta(q)\}$$

Note that, by the construction of the input formulae, m_k encodes precisely one transition $k \xrightarrow{a} \ell$. We extract the states of k by taking the Δ-variables, representing the pre-image, that are set to `1`. The procedure `SAT-PREF` guarantees that it will be a maximal predecessor of ℓ not subsumed by *Visited*.

The emptiness test needs to add the newly generated predecessor to the antichain. In `SAT-PREF`, this is done automatically because of the choice of $\varphi_{\mathit{Visited}}$ as the mutable part and of $\Delta(Q)$ as the preferred literals. Indeed, adding a configuration c to the antichain *Visited* represented by $\varphi_{\mathit{Visited}}$ means adding a clause with the literal $\Delta(q)$ for every q not contained in c. Let $c = k(m_k)$. Since the states q that are not contained in c are just the ones for which $\Delta(q)$ is not satisfied by m_k, adding the predecessor $k(m_k)$ to the antichain can be described by the following assignment:

$$\varphi_{\mathit{Visited}} \leftarrow \varphi_{\mathit{Visited}} \wedge \bigvee_{q \in Q \,\wedge\, m_k \models \neg\Delta(q)} \Delta(q)$$

Algorithm 3 Clausal antichain, antichain algorithm with combined subset-predecessor query.

```
  Input: AFA $\mathcal{M} = (Q, V, \delta, \iota, F)$
  Output: Is the language of $\mathcal{M}$ empty?
1 if $\iota \in F$ then return $NOT\ EMPTY$;
2 $\varphi_{Visited} \leftarrow \bigvee_{q \in Q \setminus F} \Delta(q)$;
3 $W \leftarrow \{F\}$;
4 while $W \neq \emptyset$ do
5     $\ell \leftarrow \mathsf{popMaxSize}^{1}(W)$ ;
6     for $m_k \in \texttt{SAT-PREF}(\Delta(Q),\ \mathsf{CNF}(\delta) \wedge \varphi_\ell,\ var\ \varphi_{Visited})$ do
7         if $\iota \in k(m_k)$ then return $NOT\ EMPTY$;
8         $\mathsf{push}(W, k(m_k))$;
9 return $EMPTY$
```

Since we call SAT-PREF with $\varphi_{\text{mut}} := \varphi_{Visited}$ and $P := \Delta(Q)$, this assignment corresponds to the update of φ_{mut} at line 10 of Algorithm 1, namely $\varphi_{\text{mut}} \leftarrow \varphi_{\text{mut}} \wedge \bigvee_{\lambda \in P\ \wedge\ m_k \models \neg\lambda} \lambda$.

The full SAT-enhanced antichain algorithm is summarized in Algorithm 3, where the predecessor-subset query covers the entire loop on line 6 of Algorithm 2, except worklist update and the initialness check. Note that SAT-PREF here indeed simultaneously finds newpre(ℓ, *Visited*), i.e., solves the subset query and the maximal predecessor iteration, and updates the antichain represented by $\varphi_{Visited}$.

We call Algorithm 3 the *clausal antichain.* A brief experiment has shown that its maximisation of predecessors via preferences is fundamental for the performance of the antichain algorithm, hence we use the feature by default in all our experiments. The lack of support for preferences in incremental SAT solvers is a major reason why one cannot just seamlessly exchange out-of-the-shelf solvers in the clausal antichain algorithm. The ease of implementing modifications such as this is one of the reasons why we are using MiniSAT.

6 Trie for Storing the Antichain

The antichain is the largest data structure that can grow exponentially in $|Q|$ and the subsumption queries over it are the most costly operations of the emptiness check. We have therefore introduced a specialised data structure based on trie for it. It replaces the internal SAT solver structures for the mutable formula, that represents the antichain. The clausal antichain equipped with a trie will be called *trie antichain.* Our trie data structure is designed to fit the following requirements:

1. a compact representation of the antichain *Visited*,
2. a fast insertion of the newly found predecessors into *Visited*, and
3. a fast subset query that integrates well with the routines and data structures of the SAT solver.

Our choice to use the trie was based on a small experiment that we did with known data structures for storing sets of sets, specialized for fast subset queries. The experiment was running unoptimized implementations of lattice-valued BDD (LVBDD) [30], set tries [27],

[1] popMaxSize is a simple heuristic, implemented previously e.g. in VATA [37], that takes a configuration with the greatest cardinality first, in a belief that its predecessors could prune a large part of the search space (our short experiment has indeed shown a performance improvement over FIFO/LIFO).

(a)

$$\begin{aligned}(v_1 \lor v_4)&\land\\(v_2 \lor v_4)&\land\\(v_3 \lor \neg v_5 \lor v_6)&\land\\(v_3 \lor \neg v_5 \lor v_7)&\end{aligned}$$

(b)

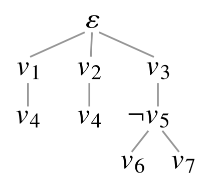


Figure 3 Clauses (a) and the corresponding trie (b) with the fixed ordering $v_1 < \cdots < v_7$.

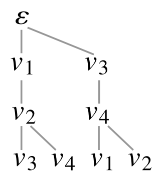


Figure 4 Trie that is smaller with a nonfixed ordering than with any fixed one.

covering sharing trees [20], and SAT unit propagation for positive boolean formulae (AND-OR graphs) with various approaches to formula simplification and sharing detection. The algorithms were solving a number of subset queries $x \subseteq Y$ with random sets. We were counting the numbers of reads and writes to the internal data structures, to get a picture of how could the optimized counterparts perform. LVBDDs solved the subset query very efficiently but their construction was unbearably prone to explosion. Set tries performed the best, closely followed by covering sharing trees (which are harder to implement). AND-OR graphs performed much worse than set tries and covering-sharing trees.

Standard trie. *Trie* is a data structure that was originally introduced in [27] for a compact representation of sets of strings. It is an ordered tree with a single root on top, denoted ε, and edges pointing downwards. The other nodes are labelled with alphabet symbols. Immediate successors of a *parent* node are its *children* and nodes without children are *leaves*. A *branch* of a trie node n is a path from n to a leaf, excluding n. A trie represents a set of strings, which are obtained by concatenating node labels of branches of ε, denoted *root branches*. Trie has been used to represent sets of sets of totally ordered elements, where a single set corresponds to a sorted string of its elements. It can thus be used to represent a CNF, viewed as a set of sets of literals. We use it to represent the antichain clauses of $\varphi_{Visited}$. An example of a trie representing the clauses from Figure 3a is shown in Figure 3b.

Mixed order trie. Sorting each branch using the same (fixed) order does not give optimal sharing of subsets as prefixes, as illustrated in Figure 4, where *no* fixed order would create as small trie as the one shown in the figure. Even if a good fixed order existed, it would be hard to guess it in advance. We therefore introduce a *mixed-order trie* where we do not fix the ordering. Allowing every branch to be sorted differently does indeed make our trie more compact. A new set x is added to the trie by finding a root branch y with a longest prefix that is a subset of x, and appending a new branch with the elements $x \setminus y$ after its prefix with the elements $x \cap y$ (with elements sorted arbitrarily). This would normally make the addition more expensive, as it would involve a search for a root branch with the largest prefix included in x. However, as we will explain in Section 6.2.2, in the context of AFA emptiness and generation of newpre(ℓ, *Visited*), the search can be completely avoided.

6.1 Clausal Watches

We adapt the well known double-watch literals scheme to our trie data structure. The scheme has been introduced in [55] and is consistently used in modern SAT solvers where it is essential for the efficiency of unit propagation and detection of conflicts. Let us recapitulate it briefly. The solver constantly watches two unassigned literals at each clause that is not yet known to be satisfied.[2] When one of the watched literals is assigned `0` and all the other unwatched literals are `0` as well, then unit propagation is triggered and assigns `1` to the other watched literal because it is the only way to satisfy the clause. When `0` gets assigned to both watched literals and all other literals in the clause are assigned `0`, then a conflict is detected. Otherwise, if other literals are unassigned or `1` on assigning `0` to a watched literal, the watch is simply moved to one of them. No action is required during backtracking where some of the watched variables may change values from `1` to `X`, keeping the only invariant that two distinct non-`0` literals are watched.

6.1.1 Double-Watch Scheme in Trie

We will detail our adaptation of the double-watch literals scheme to the clauses stored in the trie. It concerns clause prefix sharing, avoidance of repeated scanning of clauses on triggering a watch, and handling of backtracking. We will also show in Section 6.2 how our technique can work in synergy with the SAT search for preferred models, especially when it is used to generate newpre(c, *Visited*).

The double-watch technique is implemented on tries using two sets of *guards*, the *Front* and the *Rear* guards, that are essentially watches that can be assigned to nodes (and store some additional data, e.g., a pointer from *Front* to *Rear*). The SAT solver will start with one *Front* and one *Rear* at the root of the tree, and they will descend down the tree, *Rear* following *Front*. The following invariant will hold after the initialisation (described below) and after every unit propagation phase. For every root branch of the trie (clause):

1. If the root branch is not yet satisfied (does not have a `1` valued node), then it contains a single *Front* and a single *Rear*. In satisfied root branches, *Front* may be missing.
2. The *Rear* is never below the *Front*.
3. The *Rear* is at the first node valued `X` or `1` (`X` means no value assigned yet).
4. If the *Rear* is valued `X`, then the *Front* is on the second highest node that is `X` or `1`.

Initially, we have a single *Front* and a single *Rear* at the root of the trie. Note that the root is a special node without a literal. We work with it as it had the constant value `0`. The invariant (points 1-4 above) is established by *initialisation*, by calling functions `move-front-down` and then `move-rear-down` on the root. The functions implement a recursive descent of the guard watches through the tree and spreading of the guard watches to the branches, and are used to reestablish the invariant after a decision or unit propagation. The descend functions and the handlers of assigning `0` to literals are given in pseudocode in Algorithm 4.

The `move-front-down` procedure searches for a new place for *Front* in the branches of the node on which it was called. In every branch, it either

[2] The MiniSAT solver provides a watch interface where handlers can be registered to a literal. When the SAT solver valuates the watched literal with `0`, its registered handlers get triggered.

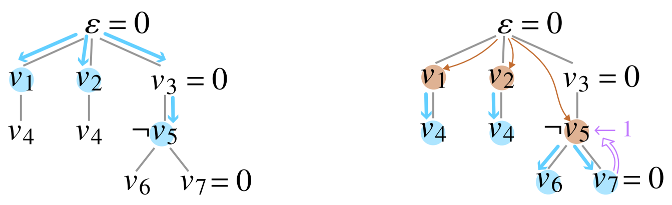


Figure 5 Initial movement of front (blue) and rear (tan) guards; unit propagation is violet.

1. places a watched copy of the *Front* at the highest node valued `X` (unassigned value) (lines 7-9), or,
2. on arrival at the leaf without seeing `X` on the way, it calls unit propagation (line 12) that assigns `1` to the literal guarded by the *Rear* above or triggers a conflict if the *Rear*'s literal is already `0`.

The `move-rear-down` procedure called on a node with a *Rear* guard works analogously, but

1. The *Front* is also sent downwards, in order to stay below the *Rear* (lines 19-22).
2. *Rear* never triggers unit propagation at leaves, as that is done when handling *Front*.

It is important to note that both `move-front-down` and `move-rear-down` are always called from a node where the respective guard is no longer present/watched. Either they are called from the watch handler, that calls line 1 or 13 and removes the watch automatically, or from line 22, which is preceded by the removal of the watch on line 21.

There are also two interesting implementation details important for efficiency. First, nodes keep a flag allowing to detect that the guard *Front* is watched on the node (on line 19), and the flag is kept up to date on lines 8 and 4. Second, a fast unit propagation on line 12 uses instant access from the *Front* to its *Rear*. The instant access is used also at line 2, to check if the root branch is not already satisfied, in which case `move-front-down` would be useless. This is facilitated by the pointer *front.rear*, which is up-to-date initially and is updated on line 20, where the *Rear* catches up with the front. Updating only at this point is indeed enough. Notice that when *Rear* moves, the connection from the *Front* guards to its copies created on line 8 will not be correctly established. It will however be reestablished in time on line 20. Indeed, the *Front* must be below the new *Rear*, hence all the literals on the way to it are valued `0`, since *Front* moves down only via `0` valued nodes. Therefore, the descend of *Rear*, which also continues uninterrupted on `0` valued nodes, will reach the *Front*, where the connection is reestablished by line 20. When *Front* is moving down, it and its copies are keeping the connection to the right *Rear* (the function `new-front` copies the pointer).

We note that in our implementation of Algorithm 4, every active *Rear* guard has a list of locations of the *Front* guards below and every *Front* guard has a pointer to its *Rear* above. This is used when *Rear* is assigned `0`, to immediately "jump" to the *Front* guards.

▶ **Example 1.** Guard movement from their initial position at ε, is shown in Figure 5. At first, *Front* guards perform a series of `move-front-down` from all nodes that have value `0` (including ε). Then, *Rear* guards are similarly transported from the root node to the nodes guarded by *Front* guards, triggering their `move-front-down` again. The node v_7, on which a front guard lands, is valuated to zero but as it is a leaf node, instead of a downward move, a unit propagation of the literal under its rear guard ($\neg v_5$) is triggered.

Algorithm 4 Implementation of Assignment Watches in a Trie.

```
Function front-watch-handler(front):
    if value (front.rear.literal) ≠ 1 then move-front-down(front) ;
Function move-front-down(front):
    front.node.front ← NULL;
    if front.node is not a leaf then
        foreach child of front.node do
            if value (child.literal) = X then
                child.front ← new-front(front, child);
                watch(¬child.literal, front-watch-handler, child.front);
            else if value (child.literal) = 0 then
                move-front-down(new-front(front, child));
    else unit-propagate(front.rear.literal, front.node);       // front may cause
     conflict
Function rear-watch-handler(rear):
    move-rear-down(rear);
Function move-rear-down(rear):
    foreach child of rear.node do
        childRear ← new-rear(rear, child);
        if value (child.literal) ≠ 1 then
            if child.front then
                child.front.rear ← childRear;
                unwatch(child.front);
                move-front-down(child.front);
            if value (child.literal) = X then
                watch(¬child.literal, rear-watch-handler, childRear);
            else move-rear-down(childRear) ;
```

6.1.2 Backtracking in Trie with Double Watches

In the case without the trie, there is no need for backtracking a clause state because clauses have no state. A clause is just scanned from left to right whenever its watched literal is valuated. The case with the trie is different: the watched guards must satisfy their invariant, mainly, that on root branches that are not satisfied, *Rear* and *Front* watch the highest and the second highest unassigned node and every other node above them is valued 0. Backtracking must thus return the positions of the guards to the previous state.

This is achieved by using *guard snapshots* that are created when decisions happen. A snapshot at the decision level L consists of two vectors: 1) one capturing the state of *new guards*, those that exist after the unit propagation triggered by the decision terminates but did not exist before the decision, 2) and one capturing the state of *old guards*, those that existed before but not after L. When the decision L is undone by backtracking, the new guards get removed and the old guards get reestablished.

As depicted in Figure 6, both lists of guards are empty when a new decision level is enetered and get updated when guards move downwards within the decision level. Namely, the functions `move-front-down` and `move-rear-down` would additionally call a procedure that updates the snapshot. The first move of a guard within the decision level adds its position to the list of old guards, and every further move replaces the guard in the list of new guards by its descendants received by the children of the current node.

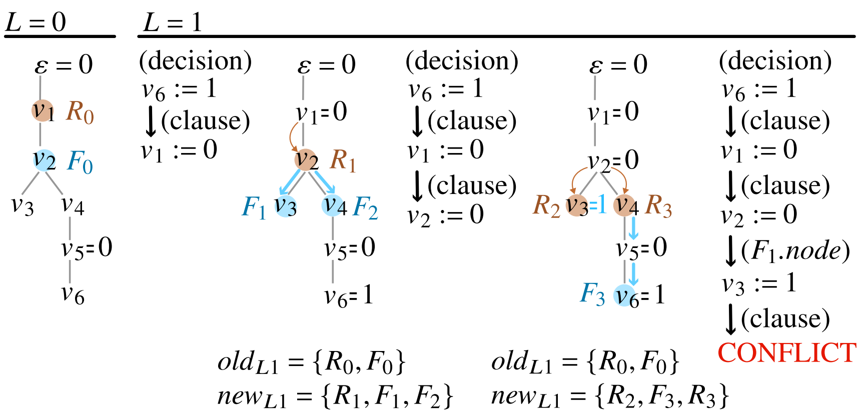


Figure 6 Backtracking. On the left, the position of *Rear* R_0 and *Front* F_0 at the end of level 0 (*L0*) is shown. Then, at the start of L1, the assignment $v_6 := 1$ is decided, from which an external clause unit propagates $v_1 := 0$. This triggers the movement of R_0, which further triggers the movement of F_0. As R_0 and F_0 were not created in the current level L1, they are added to the list of old guards. The list of new guards is updated with R_1, F_1, F_2. The unit propagation continues and another external clause unit propagates $v_2 := 0$, which triggers further guard movement. The moved guards have been created in L1, therefore, old_{L1} is untouched. The moved guards are replaced in new_{L1} with their descendants. As F_1 was moved down from the leaf node, it caused a unit propagation $v_3 := 1$ at its rear R_2. The consequent unit propagation raises a conflict. If we were in the context of CDCL, a conflict analysis follows, during which the node of the former F_1 is queried for the reason of the propagation. The reason contains negations of 0-valued literals of the root branch that ends in the queried leaf node, namely $\neg v_1 \wedge \neg v_2$. Backtracking from L1 recreates the guards from old_{L1} and deletes the guards listed in new_{L1}, which reverts the trie to the state at the end of L0.

This mechanism can be implemented fairly efficiently by equipping every guard with a pointer to its place in the snapshot. The pointer allows a constant-time quick removal/replacement of the guard from the snapshot and an addition of a guard to the snapshot is constant time.

6.2 Constructing a Trie

A SAT solver can use the trie in two ways. A generic mode that can be used in general SAT solving can be optimized and fine-tuned to fit the generation of newpre(c, *Visited*) in the AFA emptiness check. We will first comment on the generic mode and then contrast the AFA specialised mode against it.

6.2.1 Constructing a Trie For General SAT Solving

The set of clauses in the input of a SAT solver is added to the trie in a standard way. For the input clauses to be added to the trie, we first need to pick an ordering of literals and sort the clauses (a good option is for instance ordering by the frequency of literals in the input clauses).

A clause is then added by traversing the trie from the root (ε) down, simultaneously with the added clause. The child into which the traversal descends is chosen as the one with the ith literal of the clause. If the child with the ith literal is not present, the descend ends and the remaining literals of the clause are added as a new branch of the current node.

▶ **Example 2.** Let us add the clause $v_1 \vee v_3 \vee \neg v_5$ into the trie from Figure 3 b). The first literal v_1 of the clause is found among the children of the root. The second one is not found among the children of the node v_1, so the new branch $v_3, \neg v_5$ is added under the node v_1.

In case that all literals of the added clause are exhausted (i is larger than the length of the clause), the descend also ends and the entire subtree beyond the $(i-1)$th visited node is removed (rather all the sub-trees rooted by the children of the $(i-1)$th node). Indeed, the removed root branches represented clauses strictly larger than the added clause, hence they were redundant (by boolean absorption).

▶ **Example 3.** Adding the clause v_3 into the trie from Figure 3b would remove the nodes $\neg v_5, v_6, v_7$. Note that not all boolean absorption is detected this way – it is highly dependent on the order of clauses. E.g., the clause v_4 would be created as a new branch of ε and the absorbed branches v_1, v_4 and v_2, v_4 would remain in the trie.

6.2.2 Constructing a Trie for AFA

In the AFA emptiness check, we optimize handling of the trie for the particular case of generating $\mathsf{newpre}(c, \mathit{Visited})$. We use the mixed-order trie, with root branches representing clauses ordered arbitrarily. The flexible ordering of root branches allows to maximize sharing of prefixes and hence to reduce the size of the trie. Normally, the cost of allowing the mixed order would be heavy. Whenever adding a clause, we would have to perform an expensive depth-first search for a branch with a longest prefix included in the clause. However, as we will explain below, due to the particular way we are using the trie in generating $\mathsf{newpre}(c, \mathit{Visited})$ and the particular way of how guards are implemented (Section 6.1.1), an addition of a clause is cheap and requires no search at all.

Let us now describe the mechanism in detail. Recall that root branches are clauses of the formula $\varphi_{\mathit{Visited}}$. The search for a new predecessor starts in a state in which the previous SAT call finished with a model m_k that encodes the newly found predecessor k such that for each $q \in Q$, $(m_k \models \Delta(q)) \leftrightarrow q \in k$.

Adding a predecessor to the antichain. To find the next predecessor, we need to add k to *Visited* and restart the SAT solver, preferably in a way that reuses as much of the previous computation as possible. Recall that adding k to *Visited* means adding to the trie the clause

$$\psi_k = \bigvee_{q \in Q \setminus k} \Delta(q) = \bigvee_{q \in Q \,\wedge\, m_k \models \neg\Delta(q)} \Delta(q)$$

As discussed in Section 6.1.1, the SAT solver finished in a state where on every branch π, the *Rear* guard is on the first non-$\mathtt{0}$-valued literal, and everything in the prefix $\bar{\pi}$ above it is valued $\mathtt{0}$.

▶ **Lemma 4.** *On every branch π, the prefix $\bar{\pi}$ ending at the parent of the rear guard is the largest prefix valued only by zeroes.*

Hence, on every branch π, the path $\bar{\pi}$ from the root (excluded) to the parent of the rear guard is the longest prefix included in ψ_k We can therefore add ψ_k efficiently to the trie as follows. We chose a root branch π with a deepest rear guard. Thus, π is a root branch with a largest prefix, $\bar{\pi}$, included in ψ_k. We append elements of $\psi_k \setminus \bar{\pi}$ as a new branch after $\bar{\pi}$. We call it *newly added branch.*
(We abuse the notation and implicitly convert between sequences of nodes and sets of literals. For example, $\psi_k \setminus \bar{\pi}$ above stands for the branch that arises from the branch ψ_k by removing the nodes with literals of the path $\bar{\pi}$.)

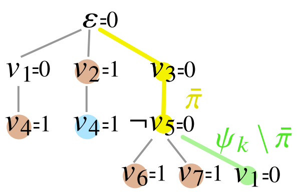


Figure 7 Adding a new clause $\psi_k = v_1 \vee v_3 \vee \neg v_5$ to a trie. *Rear*/*Front* guards are tan/blue, the largest prefix included in ψ_k is yellow, the added branch is green.

▶ **Example 5.** In Figure 7, we are adding a clause $\psi_k = v_1 \vee v_3 \vee \neg v_5$ to the trie. The deepest rear guards are v_6, v_7, we select one of them arbitrarily, v_6. The path from ε to the parent of v_6 is $\bar{\pi} = v_3, \neg v_5$. It is a longest prefix of the root branches that is included in the set ψ_k. The branch $\psi_k \setminus \bar{\pi} = v_1$ is added to the end node of $\bar{\pi}$. Note that 1) the new root branch $v_3, \neg v_5, v_1$ is fully valuated with 0, hence conflicting; and 2) if we used the fixed order $v_1 < \cdots < v_7$ instead, we would add a longer branch $v_3, \neg v_5$ under the node v_1.

Restarting the SAT solver. With a clause k added to the antichain, the next call of the SAT solver that finds the next maximal predecessor can reuse much of the previous state. First, note that after finding m_k, addition of the branch ψ_k will cause a conflict exactly in ψ_k (since all its literals are 0 in m_k). We may reuse much of the solver's state by just starting backtracking from the conflict in ψ_k. We need to ensure that after the backtracking, the guard invariant from Section 6.1.1 will be satisfied. These invariants would hold in common backtracking during SAT solving, but as we have added a new branch $\psi_k \setminus \bar{\pi}$, we have to enrich the snapshots of the guard history with guards in the newly added branch, as if it always existed. The snapshots of the guards are reconstructed using a simulation of the decisions that led to constructing m_k.

The simulation uses decision levels of literals in the new root branch to determine positions of *Rear* and *Front* in each decision level. It is based on the property that whenever a guard movement is triggered in a level L, the guard moves down through all nodes valuated with a lesser or equal level to L, because they already have the value 0, and it stops at the first node with a higher level than L, which is still unassigned, or raises a conflict if such node does not exist. The guard may also temporarily stop at nodes with the same decision level L, as they might not be valuated yet, but it will continue from them later in L (we are interested only in the position of the guard at the end of the decision level L).

For efficient backtracking in the newly added branch, we sort it by the decision levels of the literals. This way, during the backtracking, the guards in the branch will simply climb up.

▶ **Example 6.** Before describing the general algorithm, this example shows an intuitive reconstruction of the guard history in a newly added branch. Figure 8 shows a new root branch ψ_k, with the newly added branch $\psi_k \setminus \bar{\pi}$ starting at its fifth node v_5. Literals at nodes are not used in the reconstruction (we only know that they all have values 0, and, consequently, the deduction always ends with a conflict). The reconstruction is based solely on decision levels when literals were valuated in the previous SAT solving. The decision levels are shown on the left side of the nodes. At the right side, we can see the results of the reconstruction – deduced positions of *Rear* and *Front* guards at the end of levels when they moved.

We start the example at the end of level 1, when *Rear* is at v_1 and *Front* is at v_3 (we skip explanation of why the guards are at those positions, anyway, before level 3, the guard positions are irrelevant for enriching snapshots, as no guards are present in the newly added branch).

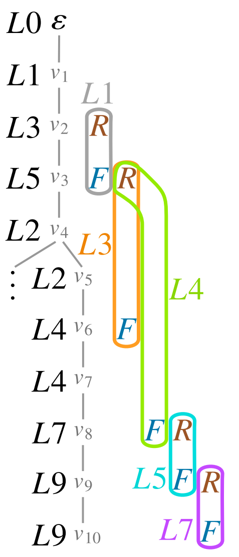


Figure 8 Reconstructing the history of guards in a new branch from decision levels of variable assignments.

During level 3, node v_2 is assigned `0` and the *Rear* triggers the downward move of the *Front*, which stops as far as at node v_6, as it is the first node that is assigned after level 3. The *Front* is then moved two steps down during level 4, because of `0`-assignment to the literals of the two nodes. In level 5, `0` is assigned to the literal under *Rear*, and *Rear* therefore moves down to *Front*, triggering the move of *Front* to the next node. In level 7, another `0` assignment moves *Rear* one step down, which again moves *Front* one step down. In level 9, both guards move from their positions but raise a conflict.

Let us generalize the deduction of guard positions from the previous example in a *deduction algorithm.*
The input to the deduction is the new root branch ψ_k, partitioned to the shared prefix $\bar{\pi}$ and the newly added branch $\psi_k \setminus \bar{\pi}$. The root branch ψ_k is ordered as it is present in the trie – the new suffix $\psi_k \setminus \bar{\pi}$ is ordered by the decision levels of its literals, but it may not be the case for the shared prefix $\bar{\pi}$.
The output of the deduction are two *movement sequences*, one for *Rear* the other for *Front*. A movement sequence is either an empty sequence (if the guard does not visit the newly added branch before the conflict level) or it is an alternation $L_1, N_1, L_2, N_2, \ldots, N_{n-1}, L_n$ of levels L_i and such nodes N_i, onto which the guard moved in L_i. The level L_1 is when the guard enters the newly added branch and L_n is the level of the conflict.

1. First, we find the level L_{R0}, when the *Rear* enters the newly added branch. It is the greatest level of the shared prefix $\bar{\pi}$, or 0 if $\bar{\pi}$ is empty.
2. Then, we find the node N_{R0}, where *Rear* resides at the time when *Front* enters the newly added branch (*Rear* stays at that node as long as until L_{R0}). It is the topmost node with L_{R0}, possibly ε.
3. Next, we find the level L_{F0} when *Front* enters the newly added branch. It is the highest level of $\bar{\pi} \setminus \{N_{R0}\}$, or 0 if $|\bar{\pi}| \leq 1$.
   This can be explained as follows. *Rear* visits N_{R0} in the highest level above N_{R0} (or in level 0 if N_{R0} is the first in π), let us call that level L_{push}. It immediately triggers the

downward move of *Front*. The *Front* then continues its movement in the suffix of $\bar{\pi}$ below N_{R0} without being pushed down by *Rear* and exits $\bar{\pi}$ in the highest level of the suffix (or immediately if the suffix is empty or L_{push} is higher).

4. Next, we record *Front* movements in the levels L such that $L_{F0} \leq L < L_{R0}$ using the following loop. The loop starts with $L \leftarrow L_{F0}$ and ends when $L_{R0} \leq L$ (which may hold initially). The following is the loop body. To the movement sequence of *Front*, we append a movement L, N where N is the topmost node of $\psi_k \setminus \bar{\pi}$ that has a greater level than L. If there is no such node, conflict is reached and the whole deduction is finished. Otherwise, we set L to the level of that node and repeat.
5. In the levels L such that $L_{R0} \leq L$, both *Rear* and *Front* move in the newly added branch. *Rear* moves in the same way as *Front* moved in the previous point. *Front* is always just one node below *Rear*, which follows from the ascending order of decision levels in the newly added branch. If *Rear* gets to the leaf, *Front* cannot be lower, conflict is reached and the deduction is done.

After the movements of the two guards in the relevant levels are deduced, snapshots are enriched in an obvious way. Namely, for all levels where a guard moves down (or causes a conflict), 1) the guard is added to the list of old guards if the move started in the newly added branch, 2) the moved copy of the guard is added to the list of new guards if it lands in the newly added branch.

▶ **Example 7.** In Figure 8, the *Front* movement sequence is $L3, v_6, L4, v_8, L5, v_9, L7, v_{10}, L9$ and the *Rear* movement sequence is $L5, v_8, L7, v_9, L9$. In level 3, *Front* has moved into the newly added branch from outside and landed at node v_6. Therefore, only the list of new guards of L3 is enriched with *Front* at that node. In level 4, the old (v_6) and new (v_8) position of *Front* is added to the old and new list respectively. In level 5, *Rear* is coming from outside, so its position at v_8 is added only to the new guards, while both old (v_8) and new (v_9) positions of *Front* are added to the respective lists. In level 7, both lists are enriched with old/new positions of both guards. In level 9, both *Rear* at v_9 and *Front* at v_{10} are added only to the list of old guards, as they started to move but a conflict was raised.

Initialization of the trie. The trie is initialized with the starting configuration $Q \setminus F$ of the backward search represented as the clause $\bigvee_{q \in Q \setminus F} \Delta(q)$ with an arbitrary ordering of the literals, and both guards are placed at the root.

7 Experiments

Implementation. We have implemented the clausal and the trie antichain in the tool AntiSAT, in C++, as a modification of MiniSAT [46], which we have chosen as a competent and modern solver, yet still simple and well documented enough to be modified with a reasonable effort. To implement the trie, we needed the version 1.12b, which is the last version that supports other constraints than clauses. We have however ported back major enhancements of the newer versions (except improvements of preprocessing), namely `StrengthenCC` [45]. We have implemented the trie data structure with memory locality and cost of allocations in mind: siblings are stored in a contiguous vector; added branch is allocated in a single array; guard data are present just next to the data of the guarded node. For SAT problems, the trie does not change during the solving, so we allocate a single memory region where the whole trie is moved after the construction.

Evaluation of AFA emptiness checking. The main focus of our experiment is to evaluate our AFA emptiness checker against other checkers, and evaluate the impact of using trie on the performance. We do not present a detailed evaluation of predecessor maximisation. Without it, our solver was not competitive at all, hence we include it to the baseline.

We have used the benchmark set from [24] with the following modifications. We have regenerated AFA from LTL because there was an unnecessary removal of final states applied previously. Additionally, to mitigate the effect of obvious redundancies in automata structure, we have applied simple preprocessing consisting of basic structural subformula sharing detection, merging of states with the same subformula and the same finalness, removing structurally unreachable states and their transitions, removing variables that occur only positively or only negatively, and applying few boolean laws: double negation, idempotence, annulment, absorption and complement law.[3]

We have compared the two variants of AntiSAT (with and without a trie) with the two most successful tools from [24]: ABC [10] (using IC3/PDR) and Mata [13] (Mata does not check AFA but boolean combinations of NFA, so it is not applicable to all benchmarks). All solvers and preprocessing tools are run with a time limit of 60 seconds and a memory limit of 10GiB. The 60s seem sufficient and extending the time limit does not seem to influence the overall comparison of tools significantly. Memory errors (which are very rare) are displayed together with timeouts. The experiments were run on a machine with 12th Gen Intel(R) Core(TM) i7-1260P CPU, running Linux. The results can be seen in Table 1 and Figure 9. The following can be concluded:

1. The trie in AntiSAT improves solving times (by a factor of 2 to 11 on average), except on "automata inclusion", where it is slightly slower (but Mata is here much faster anyway).
2. ABC and Mata are both complementary to AntiSAT. AntiSAT shines in "bool comb" benchmarks, Mata is very good at solving "automata inclusion", while ABC wins at "stranger afa", "ltl afa", and "noodler" benchmarks. There are several exceptions though.
3. The "email filter" benchmarks are easy for all solvers and most of the "noodler" benchmarks too.

The main positive takeout is that AntiSAT solves the entire "bool comb" and many examples from "ltl afa" and "noodler" much faster than the others, and it is strongly orthogonal to the other tools.

Regarding the impact of the trie data structure, Table 2 compares numbers of literals in the clausal and the trie representation of *Visited* (i.e. the number of trie nodes for AntiSAT[trie], or the total number of literals in all clauses of φ_{mut}) at the end of solving an AFA instance. In fact, for the "bool comb" benchmark, where the trie had the greatest effect in terms of solving time, less prefix sharing is shown than for other benchmarks (but still saving 500 thousand of literals on average).

Figure 10 shows the solving of three chosen nontrivial benchmark instances: the time that was spent on each predecessor query, the overall number of predecessor queries, and the results of the queries. Trie and clausal AntiSAT are compared. In the "bool comb" benchmark, clausal SAT spends a lot of time on finding predecessors (the SAT queries that find a model), while final proofs that there are no more predecessors (when the query results in UNSAT) are very fast. Trie is fast in both cases. The other two benchmarks are different: SAT queries with the UNSAT result are usually slower than queries that found predecessors. In the last benchmark, from "automata inclusion", the trie is slower than clausal AntiSAT. The performance of the trie does not seem to correlate with the number of states or symbol-variables.

[3] The instances are available at `https://github.com/p4l1ly/antisat-afa-benchmarks` at tag `sat2024`.

Table 1 Statistics from the solver run-times. $|B|$ is the number of instances in the benchmark, S_1 is the number of benchmarks solved only by the first solver of the two in the column, S_2 only by the second one, S_{12} by both. T_i is the average time the first/second solver has taken to solve interesting instances. An instance is *interesting* if both solvers solved it within 60 seconds and at least one solver took more than 1 second (to have a comparison of run-times not cluttered by trivial instances).

		AntiSAT[clause] x AntiSAT[trie]					ABC x AntiSAT[trie]					Mata x AntiSAT[trie]				
benchmark	$\|B\|$	S_1	S_2	S_{12}	T_1	T_2	S_1	S_2	S_{12}	T_1	T_2	S_1	S_2	S_{12}	T_1	T_2
automata inclusion	136	0	0	123	6.3	9.0	4	9	114	10.3	7.4	12	0	123	1.0	8.7
bool comb	653	0	6	540	11.6	1.0	0	78	468	14.3	1.6	6	97	449	4.1	2.2
email filter	500	0	0	500	NaN	NaN	0	0	500	7.3	0.2	0	0	500	1.2	0.1
ltl afa	7087	0	19	5724	14.9	7.9	1161	58	5685	0.3	14.2	Mata not applicable				
noodler	13840	0	6	13790	9.0	5.6	42	14	13782	0.4	9.9	Mata not applicable				
stranger afa	4058	0	291	3131	31.9	18.1	597	0	3422	0.7	34.0	Mata not applicable				

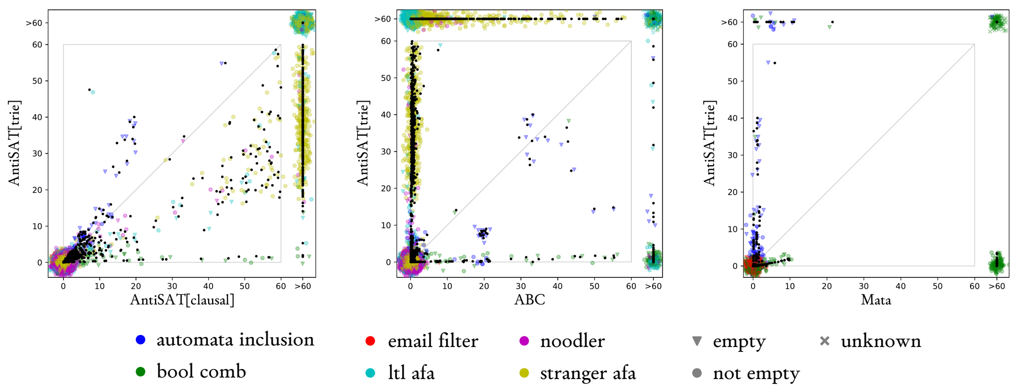


Figure 9 Detailed comparison of pairs of AFA emptiness checkers. Exact times are marked with black dots but to give an idea about the number of overlapping points, there are semitransparent associated points with small random shifts. The colour of the associated points indicates the benchmark set, while the shape indicates the answer (empty or not).

Table 2 Statistics of sizes of the clausal and trie representation of *Visited* taken from the interesting instances of the comparison AntiSAT[clause] x AntiSAT[trie]. $|B_I|$ is the number of the interesting instances in the benchmark. The abbreviations SzC/SzT mean size (i.e., the total number of literals) of clausal/trie representation of *Visited*.

		SzC / 1000			SzT / 1000		
benchmark	$\|B_I\|$	min	max	avg	min	max	avg
automata inclusion	86	108	5792	2509	51	1019	456
bool comb	60	182	19360	1510	65	18727	1065
ltl afa	62	244	60569	4145	8	15910	540
noodler	62	88	4605	1169	19	1470	245
stranger afa	87	270	32892	10110	5	622	312
total	357	88	60569	4245	5	18727	501

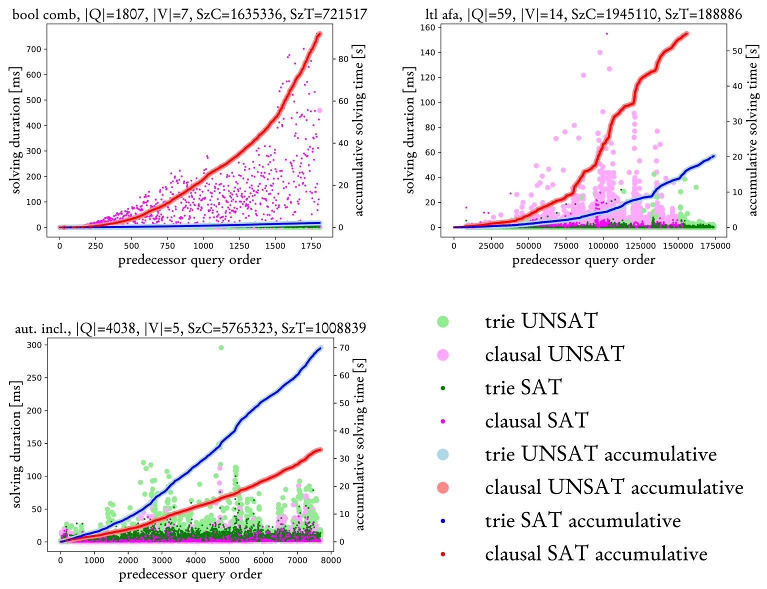


Figure 10 A detailed insight into solving three emptiness tests. Each graph shows SAT solving times throughout the solving of AntiSAT[clausal] and AntiSAT[trie] – the individual solving times and their accumulation.

8 Conclusion and Future Directions

We have proposed an AFA emptiness check that uses a SAT solver to generate a maximal predecessor not subsumed by the set of visited configurations. Our techniques performed significantly better than the state-of-the-art on a large portion of benchmarks. The same techniques can most probably be used also in the IC3-based AFA emptiness check (which can be seen as an augmentation of the backward antichain with a form of an abstraction refinement). Application of our techniques in IC3 may potentially be even more interesting than the presented adaptation to the backward antichain. A question is whether tries could be implemented in some top-performing SAT solver as CaDiCaL and have an impact in general SAT solving or whether a faster base solver would improve the performance of the AFA emptiness check.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Wei-Cheng Wu. Efficient handling of string-number conversion. In *Proc. of PLDI'20*, pages 943–957. ACM, 2020.

2 Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In *Proc. of CONCUR'02*, volume 2421 of *LNCS*, pages 116–130. Springer, 2002.

3 Saeed Amizadeh, Sergiy Matyunin, Helmut Weimer, and Marco Maratea. Learning combinatorial optimization algorithms over graphs. In *Proc. of NeurIPS'18*, volume 31, 2018.

4 Gilles Audemard and Laurent Simon. Glucose: A solver that predicts learnt clauses quality. In *SAT Competition*, 2009.

5 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In *SAT Competition*, 2020.

6 Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: an overview. In *Proc. of ICLP'02*, volume 2401 of *LNCS*, pages 1–19. Springer, 2002.

7 Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *Proc. of CAV'20*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.

8 Aaron R Bradley. SAT-based model checking without unrolling. In *Proc. of VMCAI'11*, pages 70–87. Springer, 2011.

9 Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Proc. of FMCAD'07*, pages 173–180. IEEE Computer Society, 2007.

10 Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *Proc. of CAV'10*, pages 24–40. Springer, 2010.

11 Thierry Castell, Claudette Cayrol, Michel Cayrol, and Daniel Le Berre. Using the Davis and Putnam procedure for an efficient computation of preferred models. In *Proc. of ECAI'96*, volume 96, pages 350–354. Citeseer, 1996.

12 Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

13 David Chocholatý, Tomáš Fiedor, Vojtěch Havlena, Lukáš Holík, Martin Hruška, Ondřej Lengál, and Juraj Síč. Mata: A fast and simple finite automata library. In *Proc. of TACAS'24*, pages 130–151. Springer, 2024.

14 Arlen Cox. Model Checking Regular Expressions. URL: `https://mosca19.github.io/slides/cox.pdf`, 2019. Presented at MOSCA'19.

15 Arlen Cox and Jason Leasure. Model checking regular language constraints. *CoRR*, abs/1708.09073, 2017. `arXiv:1708.09073`.

16 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.

17 Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. *ACM SIGPLAN Notices – POPL'14*, 49(1):541–553, 2014.

18 Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

19 M. De Wulf, L. Doyen, N. Maquet, and J. F. Raskin. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *Proc. of TACAS'08*, pages 63–77. Springer, 2008.

20 Giorgio Delzanno and Jean-François Raskin. Symbolic representation of upward-closed sets. In *Proc. of TACAS'00*, pages 426–441. Springer, 2000.

21 Emanuele Di Rosa, Enrico Giunchiglia, and Marco Maratea. Solving satisfiability problems with preferences. *Constraints*, 15:485–515, 2010.

22 Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In *Proc. of TACAS'10*, LNCS. Springer, 2010. `doi:10.1007/978-3-642-12002-2_2`.

23 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proc. of SAT'03*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

24 Tomáš Fiedor, Lukáš Holík, Martin Hruška, Adam Rogalewicz, Juraj Síč, and Pavol Vargovčík. Reasoning about regular properties: a comparative study. In *Proc. of CADE'23*, pages 286–306. Springer, 2023.

25 Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. Lazy automata techniques for WS1S. In *Proc. of TACAS'17*, volume 10205 of *LNCS*, pages 407–425. Springer, 2017.

26 Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2):101–127, 2004.

27 Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

28 Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. Unbounded model-checking with interpolation for regular language constraints. In *Proc. of TACAS'13*, LNCS. Springer, 2013. `doi:10.1007/978-3-642-36742-7_20`.

29 Pierre Ganty, Nicolas Maquet, and Jean-François Raskin. Fixed point guided abstraction refinement for alternating automata. *Theory of Computer Science*, 411(38-39):3444–3459, 2010.

30 Gilles Geeraerts, Gabriel Kalyon, Tristan Le Gall, Nicolas Maquet, and Jean-Francois Raskin. Lattice-valued binary decision diagrams. In *Proc. of ATVA'10*, pages 158–172. Springer, 2010.

31 Hyojung Han and Fabio Somenzi. Alembic: an efficient algorithm for CNF preprocessing. In *Proc. of DAC'07*, DAC '07, pages 582–587. ACM, 2007.

32 Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Proc. of TACAS '95*, volume 1019 of *LNCS*. Springer, 1995.

33 Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Proc. of SAT'12*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012.

34 Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL), 2018.

35 Peter Kelb, Tiziana Margaria, Michael Mendler, and Claudia Gsottberger. MOSEL: A sound and efficient tool for m2l(str). In *Proc. of CAV '97*, volume 1254 of *LNCS*, pages 448–451. Springer, 1997. doi:10.1007/3-540-63166-6_45.

36 Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Logic*, 2(3):408–429, 2001.

37 Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A library for efficient manipulation of non-deterministic tree automata. In *Proc. of TACAS'12*, volume 7214 of *LNCS*. Springer, 2012. doi:10.1007/978-3-642-28756-5_7.

38 João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, May 1999.

39 Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

40 Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Proc. of DAC '06*, pages 532–535. ACM, 2006.

41 Anders Møller et al. Brics automata library. URL: https://www.brics.dk/automaton/.

42 M. W. Moskewicz, L. Zhang, C. F. Madigan, S. Malik, and Y. Zhao. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, pages 530–535. IEEE Computer Society, 2001.

43 Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *Proc. of ICLR'19*, 2019.

44 Mate Soos. Enhancing CDCL SAT solvers with extended resolution. In *Proc. of Pragmatics of SAT*. EasyChair, 2010.

45 Niklas Sörensson and Niklas Eén. Minisat – A SAT solver with conflict-clause minimization. *Proc. of SAT'05*, pages 1–2, 2005.

46 Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *Proc. of SAT'09*, page 31, 2009.

47 Dmitriy Traytel. A coalgebraic decision procedure for WS1S. In *Proc. of CSL'15*, volume 41 of *LIPIcs*, pages 487–503. Schloss Dagstuhl, 2015.

48 Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of LICS'96*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

49 Pavol Vargovčík and Lukáš Holík. Simplifying alternating automata for emptiness testing. In *Proc. of APLAS'21*, pages 243–264. Springer, 2021.

50 Pavol Vargovčík. AntiSAT. Software, version 0.1.0., Czech Ministry of Education, Youth and Sports ERC.CZ LL1908 FIT BUT FIT-S-23-8151, swhId: swh:1:rev:5901055e699e7b95ba3dec63d11445fcd791f5de (visited on 2024-07-31). URL: https://github.com/p4l1ly/antisat/tree/5901055e699e7b95ba3dec63d11445fcd791f5de.

51 Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. String analysis via automata manipulation with logic circuit representation. In *Proc. of CAV'16*, volume 9779 of *LNCS*. Springer, 2016.

52 Pierre Wolper. On the use of automata for deciding linear arithmetic. In *Proc. of TABLEAUX'09*, volume 5607 of *LNCS*, page 16. Springer, 2009.

53 Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. of CAV'98*, pages 88–97. Springer, 1998.

54 Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.

55 Hantao Zhang and Mark Stickel. Implementing the Davis–Putnam method. *Journal of Automated Reasoning*, 24(1):277–296, 2000.