Centre Fédéré en Vérification

Technical Report number 2009.109

Fixpoint Guided Abstraction Refinement for Alternating Automata

Pierre Ganty, Nicolas Maquet, Jean-François Raskin

This work was partially supported by a FRFC grant: 2.4530.02
and by the MoVES project. MoVES (P6/39) is part of the IAP-Phase VI Interuniversity
Attraction Poles Programme funded by the Belgian State, Belgian Science Policy

http://www.ulb.ac.be/di/ssd/cfv

Fixpoint Guided Abstraction Refinement for Alternating Automata

Pierre Ganty[1], Nicolas Maquet[2*] and Jean-François Raskin[2]

[1] University of California, Los Angeles, USA
[2] Université Libre de Bruxelles (ULB), Belgium

Abstract. In this paper, we develop and evaluate two new algorithms for checking emptiness of alternating automata. Those algorithms build on previous works. First, they rely on antichains to efficiently manipulate the state-spaces underlying the analysis of alternating automata. Second, they are abstract algorithms with built-in refinement operators based on techniques that exploit information computed by abstract fixed points (and not counter-examples as it is usually the case). The efficiency of our new algorithms is illustrated by experimental results.

1 Introduction

Alternating automata are a generalization of both nondeterministic and universal automata. In an alternating automaton, the transition relation is defined using positive Boolean formulas: disjunctions allow for the expression of nondeterministic transitions and conjunctions allow for the expression of universal transitions. The emptiness problem for alternating automata being PSPACE-COMPLETE [3], several computationally-hard automata-theoretic and model-checking problems can be reduced in polynomial time to the emptiness problem for those automata. Here are some illustrative examples. The emptiness problem for a product of n nondeterministic automata, the language inclusion between two nondeterministic automata, or the LTL model-checking problem can be reduced in linear time to the emptiness problem for alternating automata. It is thus very desirable to design efficient algorithms for checking emptiness of those automata. In this paper, we propose new algorithms for efficiently checking the emptiness problem for alternating automata over finite words. Those new algorithms combine two recent lines of research.

First, we use efficient techniques based on *antichains*, initially introduced in [6], to symbolically manipulate the state-spaces underlying the analysis of alternating automata. Antichain-based techniques have been applied to several problems in automata theory [6, 8, 9, 1] and for solving games of imperfect information [13]. For example, in [9], we show how to solve the language inclusion problem between nondeterministic Büchi automata efficiently by exploiting the structures of the automata-based constructions underlying this problem. Automata that were out of reach of existing algorithms can be treated with these new antichain algorithms, see also [10] for new developments

[*] This author is supported by an FNRS-FRIA grant.

on that problem. Those techniques have also been applied with success to the satisfiability and model-checking of LTL specifications [8]. Our team has implemented these algorithms in a tool called ALASKA [7], which is available for download[1].

Second, to apply this antichain technique to even larger instances of alternating automata, we instantiate a generic abstract-refinement method that we have proposed in [5] and further developed in [11, 12]. This abstract-refinement method does not use counter-examples to refine unconclusive abstractions contrary to most of the methods presented and implemented in the literature, see for example [4]. Instead, our algorithm uses the entire information computed by the abstract analysis and combines it with information obtained by one application of a concrete predicate transformer. The algorithm presented in [5] is a generic solution that does not lead directly to efficient implementations. In particular, as shown in [11], in order to obtain an efficient implementation of this algorithm, we need to define a family of abstract domains on which abstract analysis can be effectively computed, as well as practical operators to refine the elements of this family of abstract domains. In this paper, we use the set of *partitions* of the locations of an alternating automaton to define the family of abstract domains. Those abstract domains and their refinement operators can be used both in *forward* and *backward* algorithms for checking emptiness of alternating automata.

To show the practical interest of these new algorithms, we have implemented them into the ALASKA tool. We illustrate the efficiency of our new algorithms on examples of alternating automata constructed from LTL specifications interpreted over finite words. With the help of those examples, we show that our algorithms are able to concentrate the analysis on important parts of the state-space and abstract away the less interesting parts *automatically*. This allows us to treat much larger instances than with the concrete forward or backward algorithms. We are confident that those new algorithms will allow us to solve problems of practical relevance that are currently out of reach of automatic methods.

Structure of the paper. In Sect. 2, we recall some important notions about alternating automata and about the lattice of partitions. In Sect. 3, we recall the basis for antichain algorithms and their application to the emptiness of alternating automata. In Sect. 4, we develop an adequate family of abstract domain based on the lattice of partitions along with the tools to refine elements of this family. In Sect. 5, we present our abstract forward and backward algorithms with refinement. In Sect. 6, we report on experiments that illustrate the efficiency of our algorithms. Finally, we draw some conclusions and evaluate future directions in Sect. 7.

2 Preliminaries

Alternating Automata. Let S be a set. We note $\mathcal{B}^+(S)$ the set of *positive Boolean formulas* over S. Formally, $\mathcal{B}^+(S) ::= s \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2$, where $s \in S$. A valuation for a set of proposition S is encoded as a subset of S. For each formula $\phi \in \mathcal{B}^+(S)$ we write $[\![\phi]\!] \subseteq 2^S$ the set of valuations that satisfy ϕ; as usual, $s \in [\![\phi]\!]$ is interpreted as the valuation that assigns "true" only to the variables in s. Let Σ be a finite alphabet.

[1] See http://www.antichains.be

A finite word w is a finite sequence $w = \sigma_0\sigma_1 \ldots \sigma_{n-1}$ of letters from Σ. We write Σ^* the set of finite words over Σ. We now recall the definition of *alternating automata over finite words* (AFA for short).

Definition 1. *An* alternating finite automaton *is a tuple* $\langle \mathsf{Loc}, \Sigma, q_0, \delta, F\rangle$ *where :* $\mathsf{Loc} = \{l_1, \ldots, l_n\}$ *is the set of locations;* $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ *is the set of alphabet symbols;* $q_0 \in \mathsf{Loc}$ *is the initial location;* $\delta \colon \mathsf{Loc} \times \Sigma \to B^+(\mathsf{Loc})$ *is the transition function; and* $F \subseteq \mathsf{Loc}$ *is the set of accepting locations.*

As we will often manipulate sets of sets of locations in the sequel, we will refer to the inner sets as *cells*. Let $\mathsf{Cells}(S) = 2^S$. A *cell* of an AFA with locations Loc is an element of $\mathsf{Cells}(\mathsf{Loc})$. Instead of defining the traditional notion of runs for AFA, we define their semantics as a *directed graph*, the nodes of which are cells. Each edge in the cell graph is labeled by an alphabet symbol.

Definition 2. *Let* $A = \langle \mathsf{Loc}, \Sigma, q_0, \delta, F\rangle$, $[\![A]\!] = \langle V, E\rangle$ *where:* $V = \mathsf{Cells}(\mathsf{Loc})$ *and* $\langle c, \sigma, c'\rangle \in E$ *iff* $c' \in [\![\bigwedge_{l \in c} \delta(l, \sigma)]\!]$. *A word* $w = \sigma_1, \ldots, \sigma_p$ *is* accepted *by the automaton* A *iff there exists a path* $c_0, c_1, \ldots, c_p$ *of cells of* V *such that* $q_0 \in c_0$, $c_p \in \mathsf{Cells}(F)$ *(the set of* accepting cells*), and* $\forall i \in [1, \ldots, p] : \langle c_{i-1}, \sigma_i, c_i\rangle \in E$.

In the sequel, we will consider $[\![A]\!]$ simply as the set of edges E of the cell graph and leave the set of vertices V implicit.

Predicate Transformers. We have defined the semantics of alternating automata as a directed graph of cells. To explore this graph, we use *predicate transformers* of type $2^{\mathsf{Cells}(\mathsf{Loc})} \to 2^{\mathsf{Cells}(\mathsf{Loc})}$.

Definition 3. *We consider the following* predicate transformers *(A is an* AFA*) :*

$$\begin{array}{rr}
post_\sigma[A](X) = \{c_2 \mid \exists \langle c_1, \sigma, c_2\rangle \in [\![A]\!] \colon c_1 \in X\} & post[A](X) = \bigcup_{\sigma \in \Sigma} post_\sigma[A](X) \\
\widetilde{post}_\sigma[A](X) = \{c_2 \mid \forall \langle c_1, \sigma, c_2\rangle \in [\![A]\!] \colon c_1 \in X\} & \widetilde{post}[A](X) = \bigcap_{\sigma \in \Sigma} \widetilde{post}_\sigma[A](X) \\
pre_\sigma[A](X) = \{c_1 \mid \exists \langle c_1, \sigma, c_2\rangle \in [\![A]\!] \colon c_2 \in X\} & pre[A](X) = \bigcup_{\sigma \in \Sigma} pre_\sigma[A](X) \\
\widetilde{pre}_\sigma[A](X) = \{c_1 \mid \forall \langle c_1, \sigma, c_2\rangle \in [\![A]\!] \colon c_2 \in X\} & \widetilde{pre}[A](X) = \bigcap_{\sigma \in \Sigma} \widetilde{pre}_\sigma[A](X)
\end{array}$$

These predicate transformers are actually two pairs which are *dual of each other*, as expressed in the following lemma.

Lemma 1. *For any* AFA A *with locations* Loc*, for any* $X \subseteq \mathsf{Cells}(\mathsf{Loc})$*, we have that* $\widetilde{post}[A](X) = \overline{post[A](\overline{X})}$ *and* $\widetilde{pre}[A](X) = \overline{pre[A](\overline{X})}$*, where* $\overline{X} \equiv \mathsf{Cells}(\mathsf{Loc}) \setminus X$.

The lattice of partitions. The heart of our abstraction scheme is to *partition* the set of locations Loc of an AFA, in order to build a smaller (hopefully more manageable) automaton. We recall the notion of partitions and some of their properties.

Let $\mathcal{P}$ be a partition of the set $S = \{l_1, \ldots, l_n\}$ into k *classes* (called *blocks* in the sequel) $\mathcal{P} = \{b_1, \ldots, b_k\}$. Partitions are classically ordered as follows: $\mathcal{P}_1 \preceq \mathcal{P}_2$ iff $\forall\, b_1 \in \mathcal{P}_1, \exists\, b_2 \in \mathcal{P}_2 : b_1 \subseteq b_2$. It is well known, see [2], that the set of partitions together with $\preceq$ form a complete lattice where $\{\{l_1\}, \ldots, \{l_n\}\}$ is the $\preceq$-minimal element, $\{\{l_1, \ldots, l_n\}\}$ is the $\preceq$-maximal element and the greatest lower bound of two partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, noted $\mathcal{P}_1 \curlywedge \mathcal{P}_2$, is the partition given by $\{b \neq \emptyset \mid \exists\, b_1 \in \mathcal{P}_1, \exists\, b_2 \in$

$\mathcal{P}_2 \colon b = b_1 \cap b_2\}$. The least upper bound of two partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, noted $\mathcal{P}_1 \curlyvee \mathcal{P}_2$, is the finest partition such that given $b \in \mathcal{P}_1 \cup \mathcal{P}_2$, for all $l_i \neq l_j : l_i \in b$ and $l_j \in b$ we have : $\exists\, b' \in \mathcal{P}_1 \curlyvee \mathcal{P}_2 \colon l_i \in b'$ and $l_j \in b'$. Also, we shall use $\mathcal{P}$ as a function such that $\mathcal{P}(l)$ simply returns the block b to which l belongs in $\mathcal{P}$.

Example 1. Given the set $S = \{a, b, c\}$ and two partitions $A_1 = \{\{a, b\}, \{c\}\}$ and $A_2 = \{\{a, c\}, \{b\}\}$. We have that $A_1 \curlywedge A_2 = \{\{a\}, \{b\}, \{c\}\}$, $A_1 \curlyvee A_2 = \{a, b, c\}$, and $A_2(a) = \{a, c\}$.

3 Deciding AFA Emptiness Using Antichains

A fundamental problem regarding AFA is the *emptiness problem*; i.e., to decide if there exists at least one word accepted by an AFA. Since nondeterministic automata (NFA, for short) emptiness can be solved in linear-time, a natural solution is to first perform an $\mathsf{AFA} \to \mathsf{NFA}$ translation and then check for emptiness. The translation is simple (albeit computationally difficult), as it amounts to a subset construction, similar to that of NFA determinization. Notice that the cell-graph semantics of AFA defined in the previous section is essentially an NFA obtained by subset construction. The following theorem exhibits two different methods of checking for emptiness, each evaluating a fixpoint-expression on the cell-graph.

Theorem 1. *Let $A = \langle \mathsf{Loc}, \Sigma, q_0, \delta, F \rangle$ be an AFA. The language of A is empty if and only if (the two expressions are equivalent, $\overline{X} \equiv \mathsf{Cells}(\mathsf{Loc}) \setminus X$) :*

$$(\mu\, x \cdot \mathit{post}[A](x) \cup [\![q_0]\!]) \subseteq \overline{\mathsf{Cells}(F)} \quad \textit{or} \quad (\mu\, x \cdot \mathit{pre}[A](x) \cup \mathsf{Cells}(F)) \subseteq \overline{[\![q_0]\!]}$$

Order relation on $\mathsf{Cells}(\cdot)$ *and antichains.* In earlier works [6, 8, 9], we have designed new efficient algorithms for several automata-theoretic problems. Those algorithms are based on efficient manipulations of sets of cells using *antichains*. The crucial property of antichains is that they are canonical representations of closed sets (for the set inclusion order) of cells. We summarize here some useful results on antichains for representing and manipulating closed sets. More details can be found in [6].

Let D be some finite domain. We define the *upward-closure* of $X \subseteq \mathsf{Cells}(D)$ as $\uparrow\! X = \{c \in \mathsf{Cells}(D) \mid \exists\, c' \in X \colon c \supseteq c'\}$. A set $X \subseteq \mathsf{Cells}(D)$ is *upward-closed* iff $X = \uparrow\! X$. The *downward-closure* of X is $\downarrow\! X = \{c \in \mathsf{Cells}(D) \mid \exists\, c' \in X \colon c \subseteq c'\}$. The set X is *downward-closed* iff $X = \downarrow\! X$. For any upward-closed set X, there exists a unique set of *minimal elements* $\lfloor X \rfloor = \{c \in X \mid \nexists\, c' \in X \colon c' \subset c\}$. Likewise, for any downward-closed set X, there exists a unique set of *maximal elements* $\lceil X \rceil = \{c \in X \mid \nexists\, c' \in X \colon c' \supset c\}$. Both sets $\lfloor X \rfloor$ and $\lceil X \rceil$ *antichains* and they *canonically represent* their upward- and downward-closure, respectively. In fact, if $X = \uparrow\! X$ then $X = \uparrow\! \lfloor X \rfloor$ and if $X = \downarrow\! X$ then $X = \downarrow\! \lceil X \rceil$.

Antichain manipulation and predidate transformers. Antichains of cells have additional useful properties. First, positive Boolean operation (union and intersection) on closed sets preserves closedness and can be carried out efficiently on antichains. Also set inclusions between closed sets, and set membership can be efficiently decided on the antichains representation. Second, as it will be established below, each of the four predicate transformers ($post, \widetilde{post}, pre, \widetilde{pre}$) can be evaluated *directly over antichains*, without the need to consider any non-minimal or non-maximal cell. Furthermore, each of these predicate transformers evaluates to sets which are *closed for subset inclusion*. In other words, the result of their computation can be canonically represented using antichains. In this work, we do not provide implementation-level details on how the predicate transformers are computed or how to compute the antichain representing a closed set of cells. Such information can be found in [8] and [7]. The two following lemmas respectively ensure that all four predicate transformers evaluate to sets of cells which can be represented with antichains; and that they can be transparently applied on antichains.

Lemma 2. *For any* AFA A *with locations* Loc, *for any* $X \subseteq \mathsf{Cells}(\mathsf{Loc})$ *we have that* $post[A](X)$ *is upward-closed,* $\widetilde{post}[A](X)$ *is downward-closed,* $pre[A](X)$ *is downward-closed, and* $\widetilde{pre}[A](X)$ *is upward-closed.*

Lemma 3. *Let* A *be an* AFA *with locations* Loc. *For any set* $X \subseteq \mathsf{Cells}(\mathsf{Loc})$ *we have that* $post[A](X) = post[A](\uparrow X)$, $\widetilde{post}[A](X) = \widetilde{post}[A](\downarrow X)$, $pre[A](X) = pre[A](\downarrow X)$, *and* $\widetilde{pre}[A](X) = \widetilde{pre}[A](\uparrow X)$.

Efficient computation on antichains representation. The union and intersection operators on upward- or downward-closed sets of cells can be efficiently computed directly over antichains in polynomial time. Let X and Y be two antichains. In the sequel, we note by $X \sqcup Y$ and $X \sqcap Y$ the unique antichain which respectively represent the union and the intersection of the sets represented by X and Y. Subset inclusion can also be decided in polynomial time on antichain representations, which we note $X \sqsubseteq Y$.

Finally, we show how to use antichains to evaluate more efficiently the fixpoint-expressions of Theorem 1. Notice that $[\![q_0]\!]$ and $\mathsf{Cells}(F)$ are respectively upward- and downward-closed sets of cells. Also, $\lceil \mathsf{Cells}(F) \rceil = \{\{F\}\}$, $\lfloor \overline{\mathsf{Cells}(F)} \rfloor = \{\{l\} \mid l \notin F\}$, $\lfloor [\![q_0]\!] \rfloor = \{\{q_0\}\}$, and $\lceil \overline{[\![q_0]\!]} \rceil = \{\mathsf{Loc} \setminus \{q_0\}\}$, all of which are antichains of linear size w.r.t. to the AFA. We can now rewrite the fixpoint expressions of Theorem 1 to exploit the properties of antichains.

Theorem 2. *Let* $A = \langle \mathsf{Loc}, \Sigma, q_0, \delta, F \rangle$ *be an* AFA. *The language of* A *is empty iff (the two expressions are equivalent and* $\overline{X} \equiv \mathsf{Cells}(\mathsf{Loc}) \setminus X$*) :*
$(\mu\, x \cdot \lfloor post[A](x) \rfloor \sqcup \lfloor [\![q_0]\!] \rfloor) \sqsubseteq \lfloor \overline{\mathsf{Cells}(F)} \rfloor$ *or* $(\mu\, x \cdot \lceil pre[A](x) \rceil \sqcup \lceil \mathsf{Cells}(F) \rceil) \sqsubseteq \lceil \overline{[\![q_0]\!]} \rceil$

This theorem provides the basis of efficient antichain-based algorithms to decide AFA emptiness. In the sequel, we will refer to them respectively as the *concrete forward* and *concrete backward* algorithms, as they directly on the semantics of AFA.

4 Abstraction of Alternating Automata

4.1 Abstract domain

In this section, we present an original algorithmic framework for the analysis of AFA, using antichains along with abstract interpretation. Given an AFA with locations Loc, our algorithm will use a family of abstract domains defined by the set of partitions $\mathcal{P}$

of Loc. The concrete domain is the complete lattice $2^{\mathsf{Cells}(\mathsf{Loc})}$, and each partition $\mathcal{P}$ defines the abstract domain as $2^{\mathsf{Cells}(\mathcal{P})}$. We refer to elements of $\mathsf{Cells}(\mathsf{Loc})$ as *concrete cells* and elements of $\mathsf{Cells}(\mathcal{P})$ as *abstract cells*. An abstract cell is thus a set of blocks of the partition $\mathcal{P}$ and it represents all the concrete cells which can be constructed by choosing at least one location from each block. To capture this representation role of abstract cells, we define the following predicate.

Definition 4. *The predicate* $\mathsf{Covers} : \mathsf{Cells}(\mathcal{P}) \times \mathsf{Cells}(\mathsf{Loc}) \rightarrow \{\top, \bot\}$ *is defined as follows :* $\mathsf{Covers}(c^\alpha, c)$ *iff* $c^\alpha = \{\mathcal{P}(l) \mid l \in c\}$.

Note that concrete cells are covered by a unique abstract cell while abstract cells usually cover many concrete cells.

Example 2. Let $\mathsf{Loc} = \{1, 2, 3, 4, 5\}$, $\mathcal{P} = \{b_1 = \{1\}, b_{2,3} = \{2, 3\}, b_{4,5} = \{4, 5\}\}$. We have that $\mathsf{Covers}(\{b_1, b_{4,5}\}, \{1, 3\})$ is false, $\mathsf{Covers}(\{b_1, b_{4,5}\}, \{1, 4\})$ is true, and $\mathsf{Covers}(\{b_1, b_{4,5}\}, \{1\})$ is false.

To make proper use of the theory of abstract interpretation, we define an *abstraction* and a *concretization* functions, and show that they form a *Galois connection* between the concrete domain and each of our abstract domains.

Definition 5. *Let* $\mathcal{P}$ *be a partition of the set* Loc, *we define the functions*
$\alpha_{\mathcal{P}} : 2^{\mathsf{Cells}(\mathsf{Loc})} \rightarrow 2^{\mathsf{Cells}(\mathcal{P})}$ *and* $\gamma_{\mathcal{P}} : 2^{\mathsf{Cells}(\mathcal{P})} \rightarrow 2^{\mathsf{Cells}(\mathsf{Loc})}$ *as follows :*
$\alpha_{\mathcal{P}}(X) = \{c^\alpha \mid \exists\, c \in X \colon \mathsf{Covers}(c^\alpha, c)\}$, $\gamma_{\mathcal{P}}(X) = \{c \mid \exists\, c^\alpha \in X \colon \mathsf{Covers}(c^\alpha, c)\}$.

In the sequel, we will omit the $\mathcal{P}$ subscript of α and γ when the partition is clear from the context. Additionaly, we define $\mu_{\mathcal{P}} = \gamma_{\mathcal{P}} \circ \alpha_{\mathcal{P}}$.

Lemma 4. *For any partition* $\mathcal{P}$ *of* Loc *:* $(2^{\mathsf{Cells}(\mathsf{Loc})}, \subseteq) \mathrel{\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}} (2^{\mathsf{Cells}(\mathcal{P})}, \subseteq)$.

Note that α and γ form a *Galois insertion* as it is easy to see that for all $\mathcal{P}$, $\alpha_{\mathcal{P}} \circ \gamma_{\mathcal{P}}$ is the identity function.

4.2 Efficient abstract analysis

In the sequel, we will need to evaluate fixpoint-expressions over the abstract domain. In theory, we could simply surround every predicate transformer occuring in the fixpoint-expressions by $\alpha \circ \cdot \circ \gamma$ to obtain an abstract fixpoint. However, for obvious performance concerns, we want to avoid as many concretization and abstraction steps as possible, and ideally make all the computations *directly over the abstract domain*. Furthermore, we would like that these *abstract predicate transformers* enjoy the same useful properties w.r.t. antichains so that we can reuse the results of the previous section. To achieve this goal, we proceed as follows. Given a partition $\mathcal{P}$ of the set of locations of an alternating automaton, we use a *syntactic transformation* θ that builds an *abstract* AFA which over-approximates the behavior of the original automaton. Later in this section we will show that the pre and $post$ predicate transformers can be directly evaluated on this abstract automaton to obtain the same result (but much faster) than the $\alpha \circ \cdot \circ \gamma$ computation on the original automaton. To express this syntactic transformation, we define *syntactic variants* of the abstraction and concretization functions.

Definition 6. *Let $\mathcal{P}$ be a partition of the set* Loc*. We define the following* syntactic *abstraction and concretization functions over positive Boolean formulas.*

$$\begin{aligned}
\hat{\alpha} : \mathcal{B}^+(\mathsf{Loc}) &\to \mathcal{B}^+(\mathcal{P}) & \hat{\gamma} : \mathcal{B}^+(\mathcal{P}) &\to \mathcal{B}^+(\mathsf{Loc}) \\
\hat{\alpha}(l) &= \mathcal{P}(l) & \hat{\gamma}(b) &= \bigvee_{l \in b} l \\
\hat{\alpha}(\phi_1 \vee \phi_2) &= \hat{\alpha}(\phi_1) \vee \hat{\alpha}(\phi_2) & \hat{\gamma}(\phi_1 \vee \phi_2) &= \hat{\gamma}(\phi_1) \vee \hat{\gamma}(\phi_2) \\
\hat{\alpha}(\phi_1 \wedge \phi_2) &= \hat{\alpha}(\phi_1) \wedge \hat{\alpha}(\phi_2) & \hat{\gamma}(\phi_1 \wedge \phi_2) &= \hat{\gamma}(\phi_1) \wedge \hat{\gamma}(\phi_2)
\end{aligned}$$

We formalize the link between the two variants of α and γ as follows.

Lemma 5. *For every $\phi \in \mathcal{B}^+(\mathsf{Loc})$ we have that $[\![\hat{\alpha}(\phi)]\!] = \alpha([\![\phi]\!])$, and for every $\phi \in \mathcal{B}^+(\mathcal{P})$ we have that $[\![\hat{\gamma}(\phi)]\!] = \gamma([\![\phi]\!])$.*

We can now define the θ transformation.

Definition 7. *Let $A = \langle \mathsf{Loc}, \Sigma, q_0, \delta, F \rangle$ and $\mathcal{P}$ a partition of* Loc*. $\theta(A, \mathcal{P}) = \langle \mathsf{Loc}^\alpha, \Sigma, b_0, \delta^\alpha, F^\alpha \rangle$ where: $\mathsf{Loc}^\alpha = \mathcal{P}$, $b_0 = \mathcal{P}(q_0)$, $\delta^\alpha(b, \sigma) = \hat{\alpha}(\bigvee_{l \in b} \delta(l, \sigma))$, and $F^\alpha = \{b \in \mathcal{P} \mid b \cap F \neq \emptyset\}$.*

Theorem 3. *Let A be an* AFA*, $\mathcal{P}$ a partition of its locations and $A^\alpha = \theta(A, \mathcal{P})$, $\alpha \circ post[A] \circ \gamma = post[A^\alpha]$ and $\alpha \circ pre[A] \circ \gamma = pre[A^\alpha]$.*

This theorem is crucial for the practical efficiency of our algorithms. In our framework, the evaluation of an abstract fixpoint on a large automaton amounts to compute a concrete fixpoint on a smaller automaton that is easy to obtain (the θ transformation can be done in linear time). This latter fixpoint computation can be performed with antichains, using all the results of Section 3.

4.3 Precision of the abstract domain

We now present some results about precision and representability in our family of abstract domains. In particular, for the automatic refinement of abstract domains, we will need an effective way of computing the *coarsest partition* which can represent an upward- or downward closed set of cells without loss of precision.

Definition 8. *A set of cells $X \subseteq \mathsf{Cells}(\mathsf{Loc})$ is representable in the abstract domain $2^{\mathsf{Cells}(\mathcal{P})}$ iff $\mu_{\mathcal{P}}(X) = X$ (recall that $\mu_{\mathcal{P}} = \gamma_{\mathcal{P}} \circ \alpha_{\mathcal{P}}$).*

Lemma 6. *Let $X \subseteq \mathsf{Cells}(\mathsf{Loc})$, let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two partitions of* Loc*. If X is representable with $\mathcal{P}_1$ and representable with $\mathcal{P}_2$, then X is representable with $\mathcal{P}_1 \curlyvee \mathcal{P}_2$.*

As the lattice of partition is a complete lattice, we have the following corollary.

Corollary 1. *For all $X \subseteq \mathsf{Cells}(\mathsf{Loc})$, there exists a coarsest partition $\mathcal{P} = \curlyvee \{\mathcal{P}' \mid \mu_{\mathcal{P}'}(X) = X\}$ such that $\mu_{\mathcal{P}}(X) = X$.*

For upward- and downward-closed sets, we have an efficient way to compute this coarsest partition. We start with upward-closed sets. To obtain an algorithm, we use the notion of *neighbour list*. The neighbour list of a location l with respect to an upward-closed set X, which we write $\mathcal{N}_X(l)$ is the set of subsets of Loc along which l appears in $\lfloor X \rfloor$.

Definition 9. *Let $X \subseteq \mathsf{Cells}(\mathsf{Loc})$ be an upward-closed set. The* neighbour list *of a location $l \in \mathsf{Loc}$ w.r.t. X is the set $\mathcal{N}_X(l) = \{c \setminus \{l\} \mid c \in \lfloor X \rfloor, l \in c\}$.*

The following lemma states that if two locations share the same neighbour lists w.r.t. an upward-closed set X, then they can be put in the same partition block and preserve the representability of X. Conversely, X cannot be exactly represented by any partition which puts into the same block two locations that have different neighbour lists.

Lemma 7. *For any partition $\mathcal{P}$ of Loc, for any upward-closed set X, the set X is representable in $2^{\mathsf{Cells}(\mathcal{P})}$ iff $\forall\, l, l' \in \mathsf{Loc} \cdot \mathcal{P}(l) = \mathcal{P}(l') \rightarrow \mathcal{N}_X(l) = \mathcal{N}_X(l')$.*

In other words, computing the neighbour list w.r.t. X for each element of Loc suffices to compute the coarsest partition which can represent X.

Corollary 2. *For all upward-closed set $X \subseteq \mathsf{Cells}(\mathsf{Loc})$, the partition $\mathcal{P}$ induced by the equivalence relation $l \sim l'$ iff $\mathcal{N}_X(l) = \mathcal{N}_X(l')$ is the coarsest partition that is able to represent X. Assuming that $\lfloor X \rfloor$ has been computed, this partition is computable in $O(n \log n)$ set comparisons, where n is the size of $\lfloor X \rfloor$.*

The representability of downward-closed sets is immediate with the following lemma. In practice, we simply compute the coarsest partition for the complementary upward-closed set.

Lemma 8. *Let $X \subseteq \mathsf{Cells}(\mathsf{Loc})$, $\mathcal{P}$ a partition of Loc. $\mu_P(X) = X$ iff $\mu_P(\overline{X}) = \overline{X}$.*

5 Abstraction Refinement Algorithm

This section presents two fixpoint-guided abstraction refinement algorithms for AFA. These algorithms share several ideas with the generic algorithm presented in [5] but they are formally different, so we provide arguments showing their correctness. To make the algorithms more readable, we have chosen not to include the antichain-specific notations in the pseudo-code. From the results of Sect. 3, is easy to see that the forward abstract algorithm only manipulates upward-closed sets while the backward abstract algorithm only manipulates downward-closed sets, so all these sets can be represented using antichains, which is what we implemented. We concentrate here on explanations related to the abstract forward algorithm. The abstract backward algorithm is the dual of this algorithm and its correctness can be established in a very similar way. We first give an informal presentation of the ideas underlying the algorithm and then we expose formal arguments for its soundness and completeness.

```
Input: $A = \langle \mathsf{Loc}, \Sigma, q_0, \delta, F \rangle$
Output: True iff $L(A) = \emptyset$
1  $\mathcal{P}_0 \leftarrow \{F, \mathsf{Loc} \setminus F\}$
2  $Z_0 \leftarrow \overline{\mathsf{Cells}(F)}$
3  for $i$ in $0, 1, 2, \dots$ do
4      $A_i^\alpha \leftarrow \theta(A, \mathcal{P}_i)$
5      $A_i^\alpha = \langle \mathsf{Loc}^\alpha, \Sigma, b_0, \delta^\alpha, F^\alpha \rangle$
6      $I_i \leftarrow [\![b_0]\!]$
7      $R_i \leftarrow \mu x \cdot (I_i \cup post[A_i^\alpha](x)) \cap \alpha_{\mathcal{P}_i}(Z_i)$
8      if $post[A_i^\alpha](R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$ then
9          return True
10     if $[\![q_0]\!] \not\subseteq Z_i$ then
11         return False
12     $Z_{i+1} \leftarrow \gamma_{\mathcal{P}_i}(R_i) \cap \widetilde{pre}[A](\gamma_{\mathcal{P}_i}(R_i))$
13     $\mathcal{P}_{i+1} \leftarrow \curlyvee\{\mathcal{P} \mid \mu_{\mathcal{P}}(Z_{i+1}) = Z_{i+1}\}$
```

```
Input: $A = \langle \mathsf{Loc}, \Sigma, q_0, \delta, F \rangle$
Output: True iff $L(A) = \emptyset$
1  $\mathcal{P}_0 \leftarrow \{\{q_0\}, \mathsf{Loc} \setminus \{q_0\}\}$
2  $Z_0 \leftarrow \overline{[\![q_0]\!]}$
3  for $i$ in $0, 1, 2, \dots$ do
4      $A_i^\alpha \leftarrow \theta(A, \mathcal{P}_i)$
5      $A_i^\alpha = \langle \mathsf{Loc}^\alpha, \Sigma, b_0, \delta^\alpha, F^\alpha \rangle$
6      $B_i \leftarrow \mathsf{Cells}(F^\alpha)$
7      $R_i \leftarrow \mu x \cdot (B_i \cup pre[A_i^\alpha](x)) \cap \alpha_{\mathcal{P}_i}(Z_i)$
8      if $pre[A_i^\alpha](R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$ then
9          return True
10     if $\mathsf{Cells}(F) \not\subseteq Z_i$ then
11         return False
12     $Z_{i+1} \leftarrow \gamma_{\mathcal{P}_i}(R_i) \cap \widetilde{post}[A](\gamma_{\mathcal{P}_i}(R_i))$
13     $\mathcal{P}_{i+1} \leftarrow \curlyvee\{\mathcal{P} \mid \mu_{\mathcal{P}}(Z_{i+1}) = Z_{i+1}\}$
```

Fig. 1. The *abstract-forward* (left) and *abstract-backward* (right) FGAR algorithms.

Description of the forward abstract algorithm. The most important information computed in the algorithm is Z_i, which is an over-approximation of the set of reachable cells which cannot reach an accepting cell in i steps or less. In other words, all the cells outside Z_i are either unreachable, or can lead to an accepting cell in i steps or less (or both). Our algorithm always uses the coarsest partition $\mathcal{P}_i$ that allows Z_i to be represented in the corresponding abstract domain. The algorithm begins by initializing Z_0 with the set of accepting cells and by initializing $\mathcal{P}_0$ accordingly (lines 1 and 2). The main loop proceeds as follows. First, we compute the abstract reachable cells R_i which are within Z_i, which is done by applying the θ transformation using $\mathcal{P}_i$ (line 4), and by computing a forward abstract fixpoint (line 7). If R_i does not contain a cell which can leave Z_i, we know (as we will formally prove later in this section) that the automaton is empty (line 8). If on the other hand, an initial cell (i.e., a cell containing q_0) is no longer in Z_i then we know that it can lead to an accepting cell in i steps or less (as it is obviously reachable) and we conclude that the automaton is non-empty (line 11). In the case where both tests failed, we *refine* the information contained in Z_i by removing all the cells which can leave R_i in one step, as we know that these cells are either surely unreachable or can lead to an accepting cell in $i + 1$ steps or less. Finally, the current abstract domain is changed to be able to represent the new Z_i (line 13), using the neighbour list algorithm of Corollary 2. It is important to note that this refinement operation is not the traditional refinement used in counter-example guided abstraction refinement. Note also that our algorithm does not necessarily choose a new abstract domain that is strictly more precise than the previous one as in [5]. Instead, the algorithm uses the most abstract domain possible at all times. As we cannot rely on the termination proof from [5], we provide a new one at the end of this section.

Completness and correctness of the forward abstract algorithm. Correctness and completness relies on the properties formalized in the following lemma.

Lemma 9. *Let* $\mathsf{Reach} = \mu x \cdot [\![q_0]\!] \cup post[A](x)$ *be the reachable cells of* A*, let* $\mathsf{Bad}^k = \cup_{j=0}^{j=k} pre^j[A](\mathsf{Cells}(F))$ *be the cells that can reach an accepting cell in* k *steps or less, and let us note* $\mathsf{Safe}^k = \mathsf{Cells}(\mathsf{Loc}) \setminus \mathsf{Bad}^k$*, i.e. the set of cells that cannot reach an accepting cell in* k *steps or less. The following four properties hold:*

1. $\forall i \geq 0\colon \mu_{\mathcal{P}_i}(Z_i) = Z_i$*, i.e.* Z_i *is representable in the successive abstract domains;*
2. $\forall i \geq 0\colon Z_{i+1} \subseteq Z_i$*, i.e. the sets* Z_i *are decreasing;*
3. $\forall i \geq 0\colon \mathsf{Reach} \cap \mathsf{Safe}^i \subseteq Z_i$*, i.e.* Z_i *over-approximates the reachable cells that cannot reach an accepting cell in* i *steps or less;*
4. *if* $Z_i = Z_{i+1}$ *then* $post[A^\alpha](R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$*.*

Proof. We prove each point in turn. Point 1 is straightforward as $\mathcal{P}_0$ is chosen in line 1 to be able to represent Z_0, and $\mathcal{P}_{i+1}$ is chosen in line 13 to be able to represent Z_{i+1}. Point 2 follows directly from the fact that $R_i \subseteq \alpha_i(Z_i)$, Z_i is representable in $\mathcal{P}_i$ by the previous point, and the definition of Z_{i+1} in line 12. Point 3 is established by induction. The property is clearly true for Z_0. Let us establish it for Z_{i+1} using the induction hypothesis that it is true for Z_i. By soundness of the theory of abstract interpretation, we know that in line 7 we compute a set R_i which over-approximates the set $\mathsf{Reach} \cap \mathsf{Safe}^i$. In line 12 we remove cells that can leave this set in one step, so Z_{i+1} is an over-approximation of the reachable cells that cannot reach an accepting cell in $i+1$ steps or less, i.e. $\mathsf{Reach} \cap \mathsf{Safe}^{i+1} \subseteq Z_{i+1}$, which concludes the proof. Point 4 is established as follows. If $Z_i = Z_{i+1}$, then clearly $post[A](\gamma_i(R_i)) \subseteq \gamma_i(R_i)$ as no cell can leave $\gamma_i(R_i)$ in one step (from line 12). Then $\gamma_i(R_i) \subseteq Z_i$ shows that $post[A](\gamma_i(R_i)) \subseteq Z_i$. Finally we conclude from monotonicity of α_i (itself a consequence of the Galois connection, see lemma 4) that $\alpha_i(post[A](\gamma_i(R_i))) \subseteq \alpha_i(Z_i)$ which is equivalent to $post[A^\alpha](R_i) \subseteq \alpha_i(Z_i)$ by theorem 3. ∎

We can now establish the soundness and completeness of our algorithm with the following theorem.

Theorem 4. *The forward abstract algorithm with refinement is sound and complete to decide the emptiness of* AFA*.*

Proof. Let A be the AFA on which the algorithm is executed. First, let us show that the algorithm is sound. Assume that the algorithm returns "True". In this case, the test of line 8 evaluates to true which implies that $post[A^\alpha](R_i) \subseteq R_i$ and so $post[A](\gamma_i(R_i)) \subseteq \gamma_i(R_i)$. Because $\gamma_i(R_i)$ is an over-approximation of the concrete reachable cells and as $\gamma_i(R_i) \subseteq Z_i$ we know that all the accepting cells are unreachable. Now, assume that the algorithm returns "False". Then $[\![q_0]\!] \not\subseteq Z_i$ which means that q_0 is able to reach an accepting cell in i steps or less. Since q_0 is obviously reachable, we can conclude that the language of A is non-empty. To prove the completeness of the algorithm, we only need to establish its termination. This is a direct consequence of point 2 and point 4 of the previous lemma. ∎

6 Experimental Evaluation

In this section, we evaluate the practical performance of our techniques with three series of benchmarks. Each benchmark is composed of a pair of LTL formulas $\langle \psi, \phi \rangle$ interpreted on finite words, and for which we want to know if ϕ is a logical consequence of ψ, i.e. if $\psi \models \phi$ holds. To solve this problem, we translate the formula $\psi \land \neg\phi$ into an AFA and check that the language of the AFA is empty. This translation is linear in the size of the formula and creates a location in the AFA for each subformula. As we will see, our ψ formulas are constructed as large conjunctions of constraints and model the behavior of finite-state systems, while the ϕ formulas model properties of those systems. We defined properties with varying degrees of *locality*. Intuitively, a property ϕ is local when only a small number of subformulas of ψ are needed to establish $\psi \models \phi$. This is not a formal notion but it will be clear from the examples. We will show in this section that our abstract algorithms are able to automatically identify subformulas which are not needed to establish the property. Due to lack of space, we only report results where $\psi \models \phi$ holds. Positive instances are clearly the most difficult, as must be prove that the corresponding AFA is empty, which requires to compute the entire fixed point (See Theorem 2). We now briefly recall the definitions of LTL interpreted over finite words and we follow by presenting each benchmark in turn.

Finite-Word LTL. Let Prop be a finite set of propositions. A LTL formula ϕ over Prop is of the form: $\phi ::= p \in \mathsf{Prop} \mid \neg\phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X\phi \mid \phi_1 U \phi_2$. Let $\Sigma = 2^{\mathsf{Prop}}$. The semantics of a finite-word LTL formula ϕ, which we note $[\![\phi]\!]$, is a subset of Σ^* as defined by the following semantic rules. Let $\omega \in \Sigma^*$. We use the following notations : ω_i is the letter in ω at the position i, starting at zero; $|\omega|$ is the length of ω; and $\omega_{i\rightarrow}$ is the suffix of ω starting at position i.

$\omega \in [\![p]\!]$ iff $p \in \omega_0$; $\omega \in [\![\neg\phi]\!]$ iff $\omega \notin [\![\phi]\!]$
$\omega \in [\![\phi_1 \land \phi_2]\!]$ iff $\omega \in [\![\phi_1]\!]$ and $\omega \in [\![\phi_2]\!]$
$\omega \in [\![\phi_1 \lor \phi_2]\!]$ iff $\omega \in [\![\phi_1]\!]$ or $\omega \in [\![\phi_2]\!]$
$\omega \notin [\![X\phi]\!]$ if $|\omega| < 2$, otherwise $\omega \in [\![X\phi]\!]$ iff $\omega_{1\rightarrow} \in [\![\phi]\!]$
$\omega \in [\![\phi_1 U \phi_2]\!]$ iff $\exists\, i, 0 \leq i < |\omega| : \omega_{i\rightarrow} \in [\![\phi_2]\!]$ and $\forall\, j, 0 \leq j \leq i : \omega_{j\rightarrow} \in [\![\phi_1]\!]$

We define the syntactic shortcuts *true* and *false* in the usual way, as well as $F\phi \equiv true U \phi$ and $G\phi \equiv \neg F \neg\phi$. Notice that no word of length 0 or 1 can satisfy $X\,true$, which is convenient to detect the end of the word. The formula $F\neg X\,true$ is thus valid in finite-word LTL, and $GX\,true$ is not satisfiable.

Benchmark 1. The first benchmark takes 2 parameters $n > 0$ and $0 < k \leq n$: $\mathsf{Bench1}(n,k) = \langle \bigwedge_{i=0}^{n-1} G(p_i \rightarrow (F(\neg p_i) \land F(p_{i+1}))), Fp_0 \rightarrow Fp_k \rangle$. Clearly we have that $\psi \models \phi$ holds for all values of k and also that the subformulas of ψ for $i > k$ are not needed to establish $\psi \models \phi$.

Benchmark 2. This second benchmark is used to demonstrate how our algorithms can automatically detect less obvious versions of locality than for $\mathsf{Bench1}$. It uses 2 parameters k and n with $0 < k \leq n$ and is built using the following recursive nesting definition: $\mathsf{Sub}(n,1) = Fp_n$; for odd values of $k > 1$ $\mathsf{Sub}(n,k) = F(p_n \land X(\mathsf{Sub}(n,k-1)))$; and for even values of $k > 1$ $\mathsf{Sub}(n,k) = F(\neg p_n \land X(\mathsf{Sub}(n,k-1)))$. Our second benchmark is : $\mathsf{Bench2}(n,k) = \langle \bigwedge_{i=0}^{n-1} G(p_i \rightarrow \mathsf{Sub}(i+1,k)), Fp_0 \rightarrow Fp_n \rangle$. It is relatively easy to see that $\psi \models \phi$ holds for any value of k, and that for odd values of k, the nested subformulas beyond the first level are not needed to establish the property.

Benchmark 3. This third and final benchmark aims to demonstrate the usefulness of our abstraction algorithms in a more realistic setting. We specified the behavior of a lift with n floors with a parametric LTL formula. An example of such formulas can be found in the appendix. For n floors, $\mathsf{Prop} = \{f_1, \ldots, f_n, b_1, \ldots, b_n, open\}$. The f_i propositions represent the current floor. Only one of the f_i's can be true at any time, which is initially f_1. The b_i propositions represent the state (lit or unlit) of the call-buttons of each floor and there is only one button per floor. The additional $open$ proposition is true when the doors of the lift are open. The constraints on the dynamics of this system are as follows : (i) initially the lift is at the first floor and the doors are open, (ii) the lift must close its doors when changing floors, (iii) the lift must go through floors in the correct order, (iv) when a button is lit, the lift eventually reaches the corresponding floor and opens its doors, and finally (v) when the lift reaches a floor, the corresponding button becomes unlit. Let n be the number of floors. We apply our algorithms to check two properties which depend on a parameter k with $1 < k \leq n$, namely $\mathsf{Spec1}(k) = G((f_1 \wedge b_k) \rightarrow (\neg f_k U f_{k-1}))$, and $\mathsf{Spec2}(k) = G((f_1 \wedge b_k \wedge b_{k-1}) \rightarrow (b_k U \neg b_{k-1}))$.

Experimental results. All the results of our experiments are found in Fig. 2, and were performed on a quad-core 3,2 Ghz Intel CPU with 12 Gb of memory. Due to lack of space, we only report results for the concrete forward and reverse backward algorithms which were the fastest (by a large factor) in all our experiments. The columns of the table are as follows. *ATC* is the size of the largest antichain encountered, *iters* is the number of iterations of the fixpoint, ATC^{α} and ATC^{γ} are respectively the sizes of the largest abstract and concrete antichains encountered, *steps* is the number of execution of the refinement steps and $|\mathcal{P}|$ is the maximum number of blocks in the partitions. *Benchmark 1.* The partition sizes of the first benchmark illustrate how our algorithm exploits the locality of the property to abstract away the irrelevant parts of the system. For local properties, i.e. for small values of k, $|\mathcal{P}|$ is small compared to $|\mathsf{Loc}|$ meaning that the algorithm automatically ignores many subformulas which are irrelevant to the property. For larger values of k, the abstraction overhead becomes larger, but that overhead becomes less important as the system grows. *Benchmark 2.* On the second benchmark, our abstract algorithm largely outperforms the concrete algorithm. Notice how for $k \geq 3$ the partition sizes do not continue to grow (it also holds for values of k beyond 5). This means that contrary to the concrete algorithm, FGAR does not get trapped in the intricate nesting of the F modalities (which are not necessary to prove the property) and abstracts it completely with a constant number of partition blocks. The speed improvement is considerable. *Benchmark 3.* On this final benchmark, the abstract algorithm outperforms the concrete algorithm when the locality of the property spans less than 5 floors. Beyond that value, the abstract algorithm starts to take longer than the concrete version. From the *ATC* column, one can see that the antichain sizes remain constant in the concrete algorithm, when the number of floors increases. This strongly indicates that the difficulty of this benchmark comes mainly from the exponential size of the alphabet rather than the state-space itself. Because our algorithms only abstracts the locations and not the alphabet, these results are not surprising. But again, for local properties, the gains are very significant.

					concrete forward			*abstract backward*					
	n	k	$\lvert\mathsf{Loc}\rvert$	$\lvert\mathsf{Prop}\rvert$	time	ATC	iters	time	ATC^{α}	ATC^{γ}	iters	steps	$\lvert\mathcal{P}\rvert$
Bench1	11	5	50	12	0,10	6	3	0,23	55	2	5	3	27
	15	5	66	16	1,60	6	3	0,56	55	2	5	3	31
	19	5	82	20	76,62	6	3	8,64	55	2	5	3	35
	11	7	50	12	0,13	8	3	0,87	201	2	5	3	31
	15	7	66	16	2,04	8	3	1,21	201	2	5	3	35
	19	7	82	20	95,79	8	3	9,99	201	2	5	3	39
	11	9	50	12	0,16	10	3	12,60	779	2	5	3	35
	15	9	66	16	2,69	10	3	13,42	779	2	5	3	39
	19	9	82	20	125,85	10	3	46,47	779	2	5	3	43
Bench2	7	1	19	8	0,06	8	2	0,10	11	2	4	3	14
	10	1	25	11	0,06	10	2	0,10	14	2	4	3	17
	13	1	31	14	0,08	14	2	0,12	17	2	4	3	20
	7	3	33	8	0,78	201	14	0,13	11	2	4	3	26
	10	3	45	11	802,17	4339	20	0,30	14	2	4	3	35
	13	3	57	14	> 1000	-	-	1,26	17	2	4	3	44
	7	5	47	8	88,15	2122	26	0,14	11	2	4	3	26
	10	5	65	11	> 1000	-	-	0,37	14	2	4	3	35
	13	5	83	14	> 1000	-	-	1,47	17	2	4	3	44
Lift : Spec1	8	3	84	17	0,30	10	17	0,51	23	40	7	4	21
	12	3	116	25	17,45	10	25	1,63	23	40	7	4	21
	16	3	148	33	498,65	10	33	26,65	23	40	7	4	21
	8	4	84	17	0,26	10	17	1,29	37	72	10	6	24
	12	4	116	25	17,81	10	25	5,02	37	72	10	6	24
	16	4	148	33	555,44	10	33	78,75	37	72	10	6	24
	8	5	84	17	0,32	10	17	3,70	42	141	12	8	27
	12	5	116	25	20,24	10	25	47,45	42	141	12	8	27
	16	5	148	33	543,27	10	33	> 1000	-	-	-	-	-
Lift : Spec2	8	3	84	17	0,46	10	17	1,18	58	72	8	4	22
	12	3	116	25	17,98	10	25	3,64	58	72	8	4	22
	16	3	148	33	557,75	10	33	48,90	58	72	8	4	22
	8	4	84	17	0,29	10	17	3,04	124	126	11	6	25
	12	4	116	25	19,29	10	25	10,63	124	126	11	6	25
	16	4	148	33	576,56	10	33	128,40	124	126	11	6	25
	8	5	84	17	0,31	10	17	15,88	131	266	14	8	28
	12	5	116	25	19,47	10	25	283,90	131	266	14	8	28
	16	5	148	33	568,83	10	33	> 1000	-	-	-	-	-

Fig. 2. Experimental results. Times are in seconds.

7 Discussion

We have proposed in this paper two new abstract algorithms with refinement for deciding language emptiness for AFA. Our algorithm is based on an abstraction-refinement scheme inspired from [5], which is different from the usual refinement techniques based on counter-example elimination [4]. Our algorithm also builds on the successful technique of antichains, that we have introduced in [6], to symbolically manipulate closed sets of cells (sets of sets of locations). We have demonstrated with a set of benchmarks that our algorithm is able to find coarse abstractions for complex automata constructed from large LTL formulas. For a large number of instances of those benchmarks, the abstract algorithms outperform by several order of magnitude the concrete algorithms. We believe that this clearly shows the interest of our new algorithms and their potential future developments. Several lines of future works can be envisioned. First, we should try to design a version of our algorithms where refinements are based on counter-examples and compare the relative performance of the two methods. Second, we have developed our technique for automata on finite words. We need to develop more theory to be able to apply our ideas to automata on infinite words. The fixed points involved in deciding emptiness for the infinite word case are more complicated (usually nested fixed points) and our theory must be extended to handle this case. Finally, it would be interesting to enrich our abstraction framework to deal with very large alphabets, possibly by partitioning the set of alphabet symbols.

References

1. A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA*, pages 57–67, 2008.
2. S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Springer, 1981.
3. A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
5. P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS '07*, volume 4634 of *LNCS*, pages 333–348. Springer, 2007.
6. M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV 2006*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.
7. M. De Wulf, L. Doyen, N. Maquet, and J.-F. Raskin. Alaska. In *ATVA*, pages 240–245, 2008.
8. M. De Wulf, L. Doyen, N. Maquet, and J.-F. Raskin. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *TACAS*, volume 4963 of *LNCS*, 2008.
9. L. Doyen and J.-F. Raskin. Improved algorithms for the automata-based approach to model-checking. In *TACAS*, volume 4424 of *LNCS*, pages 451–465. Springer, 2007.
10. S. Fogarty and M. Vardi. Buechi complementation and size-change termination, 2009. to appear in TACAS.
11. P. Ganty. *The Fixpoint Checking Problem: An Abstraction Refinement Perspective*. PhD thesis, Université Libre de Bruxelles, 2007.
12. P. Ganty, J.-F. Raskin, and L. Van Begin. From many places to few: automatic abstraction refinement for petri nets. *Fundamenta Informaticae*, 88(3):275–305, 2008.
13. J.-F. Raskin, K. Chatterjee, L. Doyen, and T. A. Henzinger. Algorithms for omega-regular games with imperfect information. *Logical Methods in Computer Science*, 3(3), 2007.

8 Appendix

This is the formula for the lift system with 2 floors :

$$f1 \wedge \neg f2 \wedge open \wedge G(((b1 \rightarrow (b1\, U\, (f1 \wedge open))) \wedge (b2 \rightarrow (b2\, U\, (f2 \wedge open)))))\wedge$$
$$G((open \rightarrow (f1 \vee f2))) \wedge G(((f1 \rightarrow (\neg f2)) \wedge (f2 \rightarrow (\neg f1))))\wedge$$
$$G((f1 \rightarrow \neg X f2) \wedge (f2 \rightarrow \neg(X f1)))\wedge$$
$$G((((f1 \wedge X^2(true)) \rightarrow X^2((f1 \vee f2))) \wedge ((f2 \wedge X^2(true)) \rightarrow X^2((f1 \vee f2)))))\wedge$$
$$G(((f1 \rightarrow \neg b1) \wedge (f2 \rightarrow \neg b2)))$$