Resets vs. Aborts in Linear Temporal Logic[*]

Roy Armoni[1], Doron Bustan[2], Orna Kupferman[3], and Moshe Y. Vardi[2]

[1] Intel Israel Development Center
[2] Rice University
[3] Hebrew University

Abstract. There has been a major emphasis recently in the semiconductor industry on designing industrial-strength property specification languages. Two major languages are ForSpec and Sugar 2.0, which are both extensions of Pnueli's LTL. Both ForSpec and Sugar 2.0 directly support reset/abort signals, in which a check for a property ψ may be terminated and declared successful by a reset/abort signal, provided the check has not yet failed. ForSpec and Sugar 2.0, however, differ in their definition of failure. The definition of failure in ForSpec is syntactic, while the definition in Sugar 2.0 is semantic. In this work we examine the implications of this distinction between the two approaches, which we refer to as the *reset* approach (for ForSpec) and the *abort* approach (for Sugar 2.0). In order to focus on the reset/abort issue, we do not consider the full languages, which are quite rich, but rather the extensions of LTL with the reset/abort constructs.

We show that the distinction between syntactic and semantic failure has a dramatic impact on the complexity of using the language in a model-checking tool. We prove that Reset-LTL enjoys the "fast-compilation property": there is a linear translation of Reset-LTL formulas into alternating Büchi automata, which implies a linear translation of Reset-LTL formulas into a symbolic representation of nondeterministic Büchi automata. In contrast, the translation of Abort-LTL formulas into alternating Büchi automata is nonelementary (i.e., cannot be bounded by a stack of exponentials of a bounded height); each abort yields an exponential blow-up in the translation. This complexity bounds also apply to model checking; model checking Reset-LTL formulas is exponential in the size of the property, while model checking Abort-LTL formulas is nonelementary in the size of the property (the same bounds apply to satisfiability checking).

1 Introduction

A key issue in the design of a model-checking tool is the choice of the formal specification language used to specify properties, as this language is one of the *primary* interfaces to the tool [7]. (The other primary interface is the modelling language, which is typically the hardware description language used by the designers). In view of this, there has been a major emphasis recently in the semiconductor industry on designing industrial-strength property specification languages (PSLs), e.g., Cadence's FormalCheck Specification

[*] A full version of this paper is available at `http://www.cs.rice.edu/~vardi/`.

Language [8], Intel's ForSpec [1][1], IBM's Sugar 2.0 [2][2], and Verisity's Temporal ***e*** [11]. These languages are all *linear* temporal languages (Sugar 2.0 has also a branching-time extension), in which time is treated as if each moment in time has a unique possible future. Thus, linear temporal formulas are interpreted over linear sequences, and we regard them as describing the behavior of a single computation of a system. In particular, both ForSpec and Sugar 2.0 can be viewed as extensions of Pnueli's LTL [13], with regular connectives and hardware-oriented features.

The regular connectives are aimed at giving the language the full expressive power of Büchi automata (cf. [1]). In contrast, the hardware-oriented features, *clocks* and *resets/aborts*, are aimed at offering direct support to two specification modes often used by verification engineers in the semiconductor industry. Both clocks and reset/abort are features that are needed to address the fact that modern semiconductor designs consist of interacting parallel modules. Today's semiconductor design technology is still dominated by synchronous design methodology. In synchronous circuits, clock signals synchronize the sequential logic, providing the designer with a simple operational model. While the asynchronous approach holds the promise of greater speed ([5]), designing asynchronous circuits is significantly harder than designing synchronous circuits. Current design methodology attempt to strike a compromise between the two approaches by using multiple clocks. This methodology results in architectures that are globally asynchronous but locally synchronous. ForSpec, for example, supports local asynchrony via the concept of *local clocks*, which enables each subformula to sample the trace according to a different clock; Sugar 2.0 supports local clocks in a similar way.

Another aspect of the fact that modern designs consist of parallel modules interacting asynchronously is the fact that a process running on one module can be reset by a signal coming from another module. As noted in [15], reset control has long been a critical aspect of embedded control design. Both ForSpec and Sugar 2.0 directly support reset/abort signals. The ForSpec formula "accept a in ψ" asserts that the property ψ should be checked only until the arrival of the reset signal a, at which point the check is considered to have *succeeded*. Similarly, the Sugar 2.0 formula "ψ abort on a" asserts that property ψ should be checked only until the arrival of the abort signal a, at which point the check is considered to have succeeded. In both ForSpec and Sugar 2.0 the signal a has to arrive before the property ψ has "failed"; arrival after failure cannot "rescue" ψ. ForSpec and Sugar 2.0, however, differ in their definition of *failure*.

The definition of failure in Sugar 2.0 is semantic; a formula fails at a point in a trace if the prefix up to (and including) that point cannot be extended in a manner that satisfies the formula. For example, the formula " next **false**" fails semantically at time 0, because it is impossible to extend the point at time 0 to a trace that satisfies the formula. In contrast, the definition of failure in ForSpec is syntactic. Thus, " next **false**" fails syntactically at time 1, because it is only then that the failure is actually discovered. As another example, consider the formula "(globally $\neg p) \wedge ($ eventually $p)$". It fails semantically

[1] ForSpec 2.0 has been designed in a collaboration between Intel, Co-Design Automation, Synopsys, and Verisity, and has been incorporated into the hardware verification language Open Vera, see `http://www.open-vera.com`.

[2] See `http://www.haifa.il.ibm.com/projects/verification/sugar/` for description of Sugar 2.0. We refer here to Version 0.8 (Draft 1), Sept. 12, 2002.

at time 0, but it never fails syntactically, since it is always possible to wait longer for the satisfaction of the eventuality (Formally, the notion of syntactic failure correspond to the notion of *informative prefix* in [6].) Mathematically, the definition of semantic failure is significantly simpler than that of syntactic failure (see formal definitions in the sequel), since the latter requires an inductive definition with respect to all syntactical constructs in the language.

In this work we examine the implications of this distinction between the two approaches, which we refer as the *reset* approach (for ForSpec) and the *abort* approach (for Sugar 2.0). In order to focus on the reset/abort issue, we do not consider the full languages, which are quite rich, but rather the extensions of LTL with the reset/abort constructs. We show that while both extensions result in logics that are as expressive as LTL, the distinction between syntactic and semantic failure has a dramatic impact on the complexity of using the language in a model-checking tool. In linear-time model checking we are given a design M (expressed in an HDL) and a property ψ (expressed in a PSL). To check that M satisfies ψ we construct a state-transition system T_M that corresponds to M and a nondeterministic Büchi automaton $\mathcal{A}_{\neg\psi}$ that corresponds to the negation of ψ. We then check if the composition $T_M || \mathcal{A}_{\neg\psi}$ contains a reachable fair cycle, which represents a trace of M falsifying ψ [19]. In a symbolic model checker the construction of T_M is linear in the size of M [3]. For LTL, the construction of $\mathcal{A}_{\neg\psi}$ is also linear in the size of ψ [3,18]. Thus, the front end of a model checker is quite fast; it is the back end, which has to search for a reachable fair cycle in $T_M || \mathcal{A}_{\neg\psi}$, that suffers from the "state-explosion problem".

We show here that Reset-LTL enjoys that "fast-compilation property": there is a linear translation of Reset-LTL formulas into alternating Büchi automata, which are exponentially more succinct than nondeterministic Büchi automata [18]. This implies a linear translation of Reset-LTL formulas into a symbolic representation of nondeterministic Büchi automata. In contrast, the translation of Abort-LTL formulas into alternating Büchi automata is nonelementary (i.e., cannot be bounded by a stack of exponentials of a bounded height); each abort yields an exponential blow-up in the translation. These complexity bounds are also shown to apply to model checking; model checking Reset-LTL formulas is exponential in the size of the property, while model checking Abort-LTL formulas is nonelementary in the size of the property (the same bounds apply to satisfiability checking).

Our results provide a rationale for the syntactic flavor of defining failure in ForSpec; it is this syntactic flavor that enables alternating automata to check for failure. This approach has a more operational flavor, which could be argued to match closer the intuition of verification engineers. In contrast, alternating automata cannot check for semantic failures, since these requires coordination between independent branches of alternating runs. It is this coordination that yields an exponential blow-up per abort. Our lower bounds for model checking and satisfiability show that this blow-up is intrinsic and not a side-effect of the automata-theoretic approach.

2 Preliminaries

A *nondeterministic Büchi word automaton* (NBW) is $\mathcal{A} = \langle \Sigma, S, S_0, \delta, F \rangle$, where Σ is a finite set of alphabet letters, S is a set of states, $\delta : S \times \Sigma \to 2^S$ is a transition function, $S_0 \subseteq S$ is a set of initial states, and $F \subseteq S$ is a set of accepting states. Let $w = w_0, w_1, \ldots$ be an infinite word over Σ. For $i \in \mathbb{N}$, let $w^i = w_i, w_{i+1}, \ldots$ denote the suffix of w from its ith letter. A sequence $\rho = s_0, s_1, \ldots$ in S^ω is a *run* of $\mathcal{A}$ over an infinite word $w \in \Sigma^\omega$, if $s_0 \in S_0$ and for every $i > 0$, we have $s_{i+1} \in \delta(s_i, w_i)$. We use $inf(\rho)$ to denote the set of states that appear infinitely often in ρ. A run ρ of $\mathcal{A}$ is *accepting* if $inf(\rho) \cap F \neq \emptyset$. An NBW $\mathcal{A}$ accepts a word w if $\mathcal{A}$ has an accepting run over w. We use $L(\mathcal{A})$ to denote the set of words that are accepted by $\mathcal{A}$. For $s \in S$, we denote by $\mathcal{A}^s$ the automaton $\mathcal{A}$ with a single initial state s.

Before we define an alternating Büchi word automaton, we need the following definition. For a given set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas **true** and **false**. Let $Y \subseteq X$. We say that Y *satisfies* a formula $\theta \in \mathcal{B}^+(X)$ if the truth assignment that assigns *true* to the members of Y and assigns *false* to the members of $X \setminus Y$ satisfies θ. A tree is a set $X \subseteq \mathbb{N}^*$, such that for $x \in \mathbb{N}^*$ and $n \in \mathbb{N}$, if $xn \in X$ then $x \in X$. We denote the length of x by $|x|$.

An *alternating Büchi word automaton* (ABW) is $\mathcal{A} = \langle \Sigma, S, s^0, \delta, F \rangle$, where Σ, S, and F are as in NBW, $s^0 \in S$ is a single initial state, and $\delta : S \times \Sigma \to \mathcal{B}^+(S)$ is a transition function. A run of $\mathcal{A}$ on an infinite word $w = w_0, w_1, \ldots$ is a (possibly infinite) S-labelled tree τ such that $\tau(\varepsilon) = s^0$ and the following holds: if $|x| = i$, $\tau(x) = s$, and $\delta(s, w_i) = \theta$, then x has k children $x_1, \ldots, x_k$, for some $k \leq |S|$, and $\{\tau(x1), \ldots, \tau(xk)\}$ satisfies θ. The run τ is *accepting* if every infinite branch in τ includes infinitely many labels in F. Note that the run can also have finite branches; if $|x| = i$, $\tau(x) = s$, and $\delta(s, a_i) = \mathbf{true}$, then x need not have children.

An *alternating weak word automaton* (*AWW*) is an ABW such that for every strongly connected component C of the automaton, either $C \subseteq F$ or $C \cap F = \emptyset$. Given two AWW $\mathcal{A}_1$ and $\mathcal{A}_2$, we can construct AWW for $\Sigma^\omega \setminus L(\mathcal{A}_1)$, $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$, and $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$, which are linear in their size, relative to $\mathcal{A}_1$ and $\mathcal{A}_2$ [12].

Next, we define the temporal logic LTL over a set of atomic propositions AP. The syntax of LTL is as follows. An atom $p \in AP$ is a formula. If ψ_1 and ψ_2 are LTL formulas, then so are $\neg\psi_1$, $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\mathbf{X}\,\psi_1$, and $\psi_1\,\mathbf{U}\,\psi_2$. For the semantics of LTL see [13]. Each LTL formula ψ induces a language $L(\psi) \subseteq (2^{AP})^\omega$ of exactly all the infinite words that satisfy ψ.

Theorem 1. [18] *For every* LTL *formula ψ, there exists an AWW $\mathcal{A}_\psi$ with $O(|\psi|)$ states such that $L(\psi) = L(\mathcal{A}_\psi)$.*

Proof. For every subformula φ of ψ, we construct an AWW $\mathcal{A}_\varphi$ for φ. The construction proceeds inductively as follows.

- For $\varphi = p \in AP$, we define $\mathcal{A}_p = \langle 2^{AP}, \{s_p^0\}, s_p^0, \delta_p, \emptyset \rangle$, where $\delta_p(s_p^0, \sigma) = \mathbf{true}$ if p is true in σ and $\delta_p(s_p^0, \sigma) = \mathbf{false}$ otherwise.
- Let ψ_1 and ψ_2 be subformulas of ψ and let $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ the automata for these formulas. The automata for $\neg\psi_1$, $\psi_1 \wedge \psi_2$, and $\psi_1 \vee \psi_2$ are the automata for $\Sigma^\omega \setminus L(\mathcal{A}_1)$, $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$, and $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$, respectively.

- For $\varphi = \mathbf{X}\psi_1$, we define $\mathcal{A}_\varphi = \langle 2^{AP}, \{s^0_\varphi\} \cup S_{\psi_1}, s^0_\varphi, \delta_0 \cup \delta_{\psi_1}, F_{\psi_1}\rangle$ where $\delta_0(s^0_\varphi, \sigma) = s^0_{\psi_1}$.
- For $\varphi = \psi_1 \mathbf{U} \psi_2$, we define $\mathcal{A}_\varphi = \langle 2^{AP}, \{s^0_\varphi\} \cup S_{\psi_1} \cup S_{\psi_2}, s^0_\varphi, \delta_0 \cup \delta_{\psi_1} \cup \delta_{\psi_2}, F_{\psi_1} \cup F_{\psi_2}\rangle$ where $\delta_0(s^0_\varphi, \sigma) = \delta_{\psi_2}(s^0_{\psi_2}, \sigma) \vee (\delta_{\psi_1}(s^0_{\psi_1}, \sigma) \wedge s^0_\varphi)$.

An automata-theoretic approach for LTL satisfiability and model-checking is presented in [20,21]. The approach is based on a construction of NBW for LTL formulas. Given an LTL formula ψ, satisfiability of ψ can be checked by first constructing an NBW $\mathcal{A}_\psi$ for ψ and then checking if $L(\mathcal{A}_\psi)$ is empty. As for model checking, assume that we want to check whether a system that is modelled by an NBW $\mathcal{A}_M$ satisfies ψ. First construct an NBW $\mathcal{A}_{\neg\psi}$ for $\neg\psi$, then check whether $L(\mathcal{A}_M) \cap L(\mathcal{A}_{\neg\psi}) = \emptyset$. (The automaton $\mathcal{A}_{\neg\psi}$ can also be used as a run-time monitor to check that ψ does not fail during a simulation run [6].)

Following [18], given an LTL formula ψ, the construction of the NBW for ψ is done in two steps: (1) Construct an ABW $\mathcal{A}'_\psi$ that is linear in the size of ψ. (2) Translate $\mathcal{A}'_\psi$ to $\mathcal{A}_\psi$. The size of $\mathcal{A}_\psi$ is exponential in the size of $\mathcal{A}'_\psi$ [10], and hence also in the size of ψ. Since checking for emptiness for NBW can be done in linear time or in nondeterministic logarithmic space [21], both satisfiability and model checking can be solved in exponential time or in polynomial space. Since both problems are PSPACE-complete [14], the bound is tight.

3 Reset-LTL

In this section we define and analyze the logic Reset-LTL. We show that for every Reset-LTL formula ψ, we can efficiently construct an ABW $\mathcal{A}_\psi$ that accepts $L(\psi)$. This construction allows us to apply the automata-theoretic approach presented in Section 2 to Reset-LTL. The logic Reset-LTL is an extension of LTL, with the operators accept in and reject in . Let ψ be a Reset-LTL formula over 2^{AP} and let b be a Boolean formula over AP. Then, accept b in ψ and reject b in ψ are Reset-LTL formulas. The semantic of Reset-LTL is defined with respect to tuples $\langle w, a, r\rangle$, where w is an infinite word over 2^{AP}, and a and r are Boolean formulas over AP. We refer to a and r as the *context* of the formula. Intuitively, a describes an *accept* signal, while r describes a *reject* signal. Note that every letter σ in w is in 2^{AP}, thus a and r are either true or false in σ. The semantic is defined as follows:

- For $p \in AP$, we have that $\langle w, a, r\rangle \models p$ if $w_0 \models a \vee (p \wedge \neg r)$.
- $\langle w, a, r\rangle \models \neg\psi$ if $\langle w, r, a\rangle \not\models \psi$.
- $\langle w, a, r\rangle \models \psi_1 \wedge \psi_2$ if $\langle w, a, r\rangle \models \psi_1$ and $\langle w, a, r\rangle \models \psi_2$.
- $\langle w, a, r\rangle \models \psi_1 \vee \psi_2$ if $\langle w, a, r\rangle \models \psi_1$ or $\langle w, a, r\rangle \models \psi_2$.
- $\langle w, a, r\rangle \models \mathbf{X}\psi$ if $w_0 \models a$ or ($\langle w^1, a, r\rangle \models \psi$ and $w_0 \not\models r$).
- $\langle w, a, r\rangle \models \psi_1 \mathbf{U} \psi_2$ if there exists $k \geq 0$ such that $\langle w^k, a, r\rangle \models \psi_2$ and for every $0 \leq j < k$, we have $\langle w^j, a, r\rangle \models \psi_1$.
- $\langle w, a, r\rangle \models$ accept b in ψ if $\langle w, a \vee (b \wedge \neg r), r\rangle \models \psi$.
- $\langle w, a, r\rangle \models$ reject b in ψ if $\langle w, a, r \vee (b \wedge \neg a)\rangle \models \psi$.

An infinite word w satisfies a formula ψ if $\langle w, \mathbf{false}, \mathbf{false}\rangle \models \psi$. The definition ensures that a and r are always disjoint, i.e., there is no $\sigma \in 2^{AP}$ that satisfies both a and r. It can be shown that this semantics satisfies a natural duality property: $\neg\mathsf{accept}\ a\ \mathsf{in}\ \psi$ is logically equivalent to $\mathsf{reject}\ b\ \mathsf{in}\ \neg\psi$. For a discussion of this semantics, see [1]. Its key feature is that a formula holds if the accept signal is asserted before the formula "failed". The notion of failure is syntax driven. For example, $\mathbf{X}\,\mathbf{false}$ cannot fail before time 1, since checking $\mathbf{X}\,\mathbf{false}$ at time 0 requires checking $\mathbf{false}$ at time 1.

Before we analyze the complexity of Reset-LTL, we characterize its expressiveness.

Theorem 2. Reset-LTL *is as expressive as* LTL.

The proof of Theorem 2 relies on the fact that although the accept and reject conditions a and r of the subformulas are defined by the semantic of Reset-LTL, they can be determined syntactically. We can use this fact to rewrite Reset-LTL formulas into equivalent LTL formulas.

We now present a translation of Reset-LTL formulas into ABW. Note, that the context that is computed during the evaluation of Reset-LTL formulas depends on the part of the formula that "wraps" each subformula. Given a formula ψ, we define for each subformula φ of ψ two Boolean formulas $acc_\psi[\varphi]$ and $rej_\psi[\varphi]$ that represent the context of φ with respect to ψ.

Definition 1. *For a* Reset-LTL *formula ψ and a subformula φ of ψ, we define the* acceptance context of φ*, denoted $acc_\psi[\varphi]$, and the* rejection context of φ*, denoted $rej_\psi[\varphi]$. The definition is by induction over the structure of the formula in a top-down direction.*

- *If $\varphi = \psi$, then $acc_\psi[\varphi] =$ **false** and $rej_\psi[\varphi] =$ **false**.*
- *Otherwise, let ξ be the innermost subformula of ψ that has φ as a strict subformula.*
  - *If $\xi = \mathsf{accept}\ b\ \mathsf{in}\ \varphi$, then $acc_\psi[\varphi] = acc_\psi[\xi] \vee (b \wedge \neg rej_\psi[\xi])$ and $rej_\psi[\varphi] = rej_\psi[\xi]$.*
  - *If $\xi = \mathsf{reject}\ b\ \mathsf{in}\ \varphi$, then $acc_\psi[\varphi] = acc_\psi[\xi]$ and $rej_\psi[\varphi] = rej_\psi[\xi] \vee (b \wedge \neg acc_\psi[\xi])$.*
  - *If $\xi = \neg\varphi$, then $acc_\psi[\varphi] = rej_\psi[\xi]$ and $rej_\psi[\varphi] = acc_\psi[\xi]$.*
  - *In all other cases, $acc_\psi[\varphi] = acc_\psi[\xi]$ and $rej_\psi[\varphi] = rej_\psi[\xi]$.*

A naive tree representation of the Boolean formulas $acc_\psi[\varphi]$ and $rej_\psi[\varphi]$ can lead to an exponential blowup. This can be avoided by using DAG representation of the formulas. Note that two subformulas that are syntactically identical might have different contexts. E.g., for the formula $\psi = \mathsf{accept}\ p_0\ \mathsf{in}\ p_1 \vee \mathsf{accept}\ p_2\ \mathsf{in}\ p_1$, there are two subformulas of the form p_1 in ψ. For the left subformula we have $acc_\psi[p_1] = p_0$ and for the right subformula we have $acc_\psi[p_1] = p_2$.

Theorem 3. *For every* Reset-LTL *formula ψ, there exists an AWW $\mathcal{A}_\psi$ with $O(|\psi|)$ states such that $L(\psi) = L(\mathcal{A}_\psi)$.*

Proof. For every subformula φ of ψ, we construct an automaton $\mathcal{A}_{\psi,\varphi}$. The automaton $\mathcal{A}_{\psi,\varphi}$ accepts an infinite word w iff $\langle w, acc_\psi[\varphi], rej_\psi[\varphi]\rangle \models \varphi$. The automaton $\mathcal{A}_\psi$ is then $\mathcal{A}_{\psi,\psi}$. The construction of $\mathcal{A}_{\psi,\varphi}$ proceeds by induction on the structure of φ as follows.

- For $\varphi = p \in AP$, we define $\mathcal{A}_{\psi,p} = \langle 2^{AP}, \{s_p^0\}, s_p^0, \delta_p, \emptyset \rangle$, where $\delta_p(s_p^0, \sigma) = \textbf{true}$ if $acc_\psi[\varphi] \vee (p \wedge \neg rej_\psi[\varphi])$ is true in σ and $\delta_p(s_p^0, \sigma) = \textbf{false}$ otherwise.
- For Boolean connectives we apply the Boolean closure of AWW.
- For $\varphi = \mathbf{X}\,\psi_1$, we define $\mathcal{A}_{\psi,\varphi} = \langle 2^{AP}, \{s_\varphi^0\} \cup S_{\psi_1}, s_\varphi^0, \delta_0 \cup \delta_{\psi_1}, F_{\psi_1} \rangle$ where

$$\delta_0(s_\varphi^0, \sigma) = \begin{bmatrix} \textbf{true} & \text{if } \sigma \models acc_\psi[\varphi], \\ \textbf{false} & \text{if } \sigma \models rej_\psi[\varphi], \\ s_{\psi_1}^0 & \text{otherwise.} \end{bmatrix}$$

- For $\varphi = \psi_1 \,\mathbf{U}\, \psi_2$, we define $\mathcal{A}_{\psi,\varphi} = \langle 2^{AP}, \{s_\varphi^0\} \cup S_{\psi_1} \cup S_{\psi_2}, s_\varphi^0, \delta_0 \cup \delta_{\psi_1} \cup \delta_{\psi_2}, F_{\psi_1} \cup F_{\psi_2} \rangle$, where $\delta_0(s_\varphi^0, \sigma) = \delta_{\psi_2}(s_{\psi_2}^0, \sigma) \vee (\delta_{\psi_1}(s_{\psi_1}^0, \sigma) \wedge s_\varphi^0)$.
- For $\varphi = \mathsf{accept}\ b\ \mathsf{in}\ \psi_1$ we define $\mathcal{A}_{\psi,\varphi} = \mathcal{A}_{\psi,\psi_1}$
- For $\varphi = \mathsf{reject}\ b\ \mathsf{in}\ \psi_1$ we define $\mathcal{A}_{\psi,\varphi} = \mathcal{A}_{\psi,\psi_1}$

Note that $\mathcal{A}_{\psi,\varphi}$ depends not only on φ but also on $acc_\psi[\varphi]$ and $rej_\psi[\varphi]$, which depend on the part of ψ that "wraps" φ. Thus, for example, the automaton $\mathcal{A}_{\psi,\psi_1}$ we get for $\varphi = \mathsf{accept}\ b\ \mathsf{in}\ \psi_1$ is different from the automaton $\mathcal{A}_{\psi,\psi_1}$ we get for $\varphi = \mathsf{reject}\ b\ \mathsf{in}\ \psi_1$, and both automata depend on b.

The construction of ABW for Reset-LTL formulas allows us to use the automata-theoretic approach presented in Section 2. Accordingly, we have the following (the lower bounds follow from the known bounds for LTL).

Theorem 4. *The satisfiability and model-checking problems of* Reset-LTL *are PSPACE-complete.*

Theorems 3 and 4 imply that the standard automata-theoretic approach to satisfiability and model checking extends to Reset-LTL in a fairly straightforward fashion. In particular, translation to alternating automata underlies the standard approaches to compilation of LTL to automata. Current compilers of LTL to automata, either explicit [4] or symbolic [3], are syntax driven, recursively applying fairly simple rules to each formula in terms of it subformulas. For example, to compile the formula $X\varphi$ symbolically, the compiler generates symbolic variables z_φ and $z_{X\varphi}$, adds the symbolic invariance $z_{Xp} \leftrightarrow z'_\varphi$ (by convention primed variables refer to the next point in time), and proceeds with the processing of φ. As the proof of Theorem 3 shows, the same approach applies also to Reset-LTL.

Remark 1. Theorem 4 holds only for formulas that are represented as trees, where every subformula of ψ has a unique occurrence. It does not hold in DAG representation, where subformulas that are syntactically identical are unified. In this case one occurrence of a subformula could be related to many automata that differ in their context. Thus, the size of the automaton could be exponential in the length of the formula, and the automata-based algorithm runs in exponential space. An EXPSPACE lower bound for the satisfiability of Reset-LTL formulas that are represented as DAGs can be shown, so, the bounds are tight.

4 Abort-LTL

In this section we define and analyze the logic Abort-LTL. We first present a construction of AWW for Abort-LTL formulas with size nonelementary in the size of the formula. This implies nonelementary solutions for the satisfiability and model-checking problems, to which we later prove matching lower bounds.

The Abort-LTL logic extends LTL with an abort on operator. Formally, if ψ is an Abort-LTL formula over 2^{AP} and b is a Boolean formula over AP, then ψ abort on b is an Abort-LTL formula. The semantic of the abort operator is defined as follows:

– $w \models \psi$ abort on b iff $w \models \psi$ or there is a prefix w' of w and an infinite word w'' such that b is true in the last letter of w' and $w' \cdot w'' \models \psi$.

For example, the formula "$(\mathbf{G}\, p)$ abort on b" is equivalent to the formula $(p\,\mathbf{U}\,(p \wedge b)) \vee \mathbf{G}\, p$. Thus, in addition to words that satisfy $\mathbf{G}\, p$, the formula is satisfied by words with a prefix that ends in a letter that satisfies b and in which p holds in every state. Such a prefix can be extended to an infinite word where $\mathbf{G}\, p$ holds, and thus the word satisfies the formula.

Before we analyze the complexity of Abort-LTL, we characterized its expressiveness.

Theorem 5. Abort-LTL *is as expressive as* LTL.

The proof of Theorem 5 relies on the fact that for every LTL formula ψ there exists a counter-free deterministic Rabin word automaton (DRW) $\mathcal{A}_\psi$ such that $L(\psi) = L(\mathcal{A}_\psi)$, and vice versa [16]. Given an LTL formula ψ we use the counter-free DRW $\mathcal{A}_\psi$ to construct a counter-free DRW $\mathcal{A}'$ such that $L(\mathcal{A}') = L(\psi$ abort on $b)$. Thus, there exists an LTL formula ψ' that is equivalent to ψ abort on b.

We now describe a construction of AWW for Abort-LTL formulas. The construction involves a nonelementary blow-up. This implies nonelementary solutions for the satisfiability and model-checking problems, to which we later prove matching lower bounds. For two integers n and k, let $exp(1, n) = 2^n$ and $exp(k, n) = 2^{exp(k-1,n)}$. Thus, $exp(k, n)$ is a tower of k exponents, with n at the top.

Theorem 6. *For every* Abort-LTL *formula ψ of length n and* abort on *nesting depth k, there exists an AWW $\mathcal{A}_\psi$ with $exp(k, n)$ states such that $L(\psi) = L(\mathcal{A}_\psi)$.*

Proof. The construction of AWW for LTL presented in Theorem 1 is inductive. Thus, in order to extend it for Abort-LTL formulas, we need to construct, given b and an AWW $\mathcal{A}_\psi$ for ψ, an AWW $\mathcal{A}_\varphi$ for $\varphi = \psi$ abort on b. Once we construct $\mathcal{A}_\varphi$, the inductive construction is as described in Theorem 1. Given b and $\mathcal{A}_\psi$, we construct $\mathcal{A}_\varphi$ as follows.

– Let $\mathcal{A}_n = \langle 2^{AP}, S_n, s^{n0}, \delta_n, F_n \rangle$ be an NBW such that $L(\mathcal{A}_n) = L(\mathcal{A}_\psi)$. According to [10], $\mathcal{A}_n$ indeed has a single initial state and its size is exponential in $\mathcal{A}_\psi$.
– Let $\mathcal{A}'_n = \langle 2^{AP}, S'_n, s'^{n0}, \delta'_n, F'_n \rangle$ be the NBW obtained from $\mathcal{A}_n$ by removing all the states from which there are no accepting runs, i.e, all states s such that $L(\mathcal{A}^s_n) = \emptyset$.

- Let $\mathcal{A}_{fin} = \langle 2^{AP}, S'_n, s'^{n0}, \delta, \emptyset\rangle$, be an AWW where δ is defined, for all $s \in S$ and $\sigma \in \Sigma$ as follows.

$$\delta(s,\sigma) = \begin{bmatrix} \textbf{true} & \text{if } \sigma \models b \text{ and } \delta_n(s,\sigma) \neq \emptyset, \\ \bigvee_{t\in\delta_n(s,\sigma)} t & \text{otherwise.} \end{bmatrix}$$

  Thus, whenever $\mathcal{A}'_n$ reads a letter that satisfies b, the AWW accepts. Intuitively, $\mathcal{A}_{fin}$ accepts words that contain prefixes where b holds in the last letter and ψ has not yet "failed".
- We define $\mathcal{A}_\varphi$ to be the automaton for $L(\mathcal{A}_\psi) \cup L(\mathcal{A}_{fin})$. Note that since both $\mathcal{A}_\psi$ and $\mathcal{A}_{fin}$ are AWW, so is $\mathcal{A}_\varphi$. The automaton $\mathcal{A}_\varphi$ accepts a word w if either $\mathcal{A}_\psi$ has an accepting run over w, or if $\mathcal{A}'_n$ has a finite run over a prefix w' of w, which ends in a letter σ that satisfies b.

For LTL, every operator increases the number of states of the automaton by one, making the overall construction linear. In contrast, here every abort on operator involves an exponential blow up in the size of the automaton. In the worst case, the size of $\mathcal{A}_\psi$ is $exp(k,n)$ where k is the nesting depth of the abort on operator and n is the length of the formula.

The construction of ABW for Abort-LTL formulas allows us to use the automata-theoretic approach presented in Section 2, implying nonelementary solutions to the satisfiability and model-checking problems for Abort-LTL.

Theorem 7. *The satisfiability and model-checking problems of* Abort-LTL *are in SPACE($exp(k,n)$), where n is the length of the specification and k is the nesting depth of* abort on .

Note that the proof of Theorem 6, buttressed by lower bounds below, shows that to have a general compilation of Abort-LTL to automata one cannot proceed in a syntax-directed fashion; rather, to compile φ abort on b one has to construct in sequence $\mathcal{A}_\varphi$, $\mathcal{A}_n$, $\mathcal{A}'_n$, $\mathcal{A}_{fin}$, and finally $\mathcal{A}_{\varphi \text{ abort on } b}$ (of course, these steps can be combined).

We now prove matching lower bounds. We first prove that the nonelementary blow-up in the translation described in Theorem 6 cannot be avoided. This proves that the automata-theoretic approach to Abort-LTL has nonelementary cost. We construct infinitely many Abort-LTL formulas ψ_n^k such that every AWW that accept $L(\psi_n^k)$ is of size $exp(k,n)$. The formulas ψ_n^k, are constructed such that $L(\psi_n^k)$ is closely related to $\{ww\Sigma^\omega : |w| = exp(k,n)\}$. Intuitively, we use the abort on operator to require that every letter in the first word is identical to the letter at the same position in the next word. It is known that every AWW that accept this language has at least $exp(k,n)$ states. The proof that every AWW that accepts $L(\psi_n^k)$ has at least $exp(k,n)$ states is similar to the known proof for $\{ww\Sigma^\omega : |w| = exp(k,n)\}$ and is discussed later.

We then show that the nonelementary cost is intrinsic and is not a side-effect of the automata-theoretic approach by proving a nonelementary lower bounds for satisfiability and model checking of Abort-LTL.

We start by considering words of length 2^n; that is, when $k = 1$. Let $\Sigma = \{0,1\}$. For simplicity, we assume that 0 and 1 are disjoint atomic propositions. Each letter of w_1 and w_2 is represented by block of n "cells". The letter itself is stored in the first cell of the block. In addition to the letter, the block stores its position in the word. The position is a number between 0 and $2^n - 1$, referred to as the *value* of the block, and we use an atomic proposition c_1 to encode it as an n-bit vector. For simplicity, we denote $\neg c_1$ by c_0. The vector is stored in the cells of the block, with the least significant bit stored at the first cell. The position is increased by one from one block to the next. The formulas in Γ requires that the first cell of each block is marked with the atomic proposition $\#$, that the first cell in the first block of w_2 is marked with the atomic proposition @, and that the first cell after w_2 is marked by \$. An example of a legal prefix (structure wise) is shown in Figure 1.

																\$
								@								
#		#		#		#		#		#		#		#		#
0	?	0	?	1	?	1	?	1	?	0	?	1	?	1	?	?
c_0	c_0	c_0	c_1	c_1	c_0	c_1	c_1	c_0	c_0	c_0	c_1	c_1	c_0	c_1	c_1	?

Fig. 1. An example for $n = 2$ that represents the case where $w_1 = 0011$ and $w_2 = 1011$. Each row represents a unique atomic proposition, which should hold at exactly the cell in which it is marked. An exception are the propositions 0 and 1 whose values are checked only in the first cell in each block (other cells are marked ?)

Formally, Γ contains the following formulas.

- $\gamma_1 = \# \wedge (c_0 \wedge (Xc_0 \wedge \overset{n}{\cdots} \wedge Xc_0)$
  After every $\#$ before the first @ there are $n-1$ cells without $\#$ or @, and then another $\#$.
- $\gamma_2 = (\# \rightarrow \bigwedge_{1 \leq i < n} \mathbf{X}^i(\neg\# \wedge \neg@) \wedge \mathbf{X}^n\, \#)\, \mathbf{U}\, @$
  The first cell is marked by $\#$ and the first block counter value is $000\ldots0$.

The following four formulas make sure that the position (that is encoded by c_0, c_1) is increased by one every $\#$. We use an additional proposition z that represents the carry. Thus, we add 1 to the least significant bit and then propagate the carry to the other bits. Note that the requirement holds until the last $\#$ before @.

- $\gamma_3 = (((\# \vee z) \wedge c_0) \rightarrow (\mathbf{X}(\neg z) \wedge \mathbf{X}^n\, c_1))\, \mathbf{U}\, (\# \wedge \mathbf{X}((\neg\#)\, \mathbf{U}\, @))$
- $\gamma_4 = ((\neg(\# \vee z) \wedge c_0) \rightarrow (\mathbf{X}(\neg z) \wedge \mathbf{X}^n\, c_0))\, \mathbf{U}\, (\# \wedge \mathbf{X}((\neg\#)\, \mathbf{U}\, @))$
- $\gamma_5 = (((\# \vee z) \wedge c_1) \rightarrow (\mathbf{X}\, z \wedge \mathbf{X}^n\, c_0))\, \mathbf{U}\, (\# \wedge \mathbf{X}((\neg\#)\, \mathbf{U}\, @))$
- $\gamma_6 = ((\neg(\# \vee z) \wedge c_1) \rightarrow (\mathbf{X}(\neg z) \wedge \mathbf{X}^n\, c_1))\, \mathbf{U}\, (\# \wedge \mathbf{X}((\neg\#)\, \mathbf{U}\, @))$

The following formulas require that the first @ is true immediately after w_1.

- $\gamma_7 = ((\# \wedge \bigvee_{0 \leq i < n} X^i c_0) \rightarrow ((\neg@)\, \mathbf{U}\, \mathbf{X}(\# \wedge \neg@)))\, \mathbf{U}\, @$
  as long as the counter is not $111\ldots1$ there not going to be @.
- $\gamma_8 = ((\# \wedge \bigwedge_{0 \leq i < n} X^i c_1) \rightarrow \mathbf{X}^n\, @)\, \mathbf{U}\, @$
  When the counter is $111\ldots1$ the next value going to be @.

The formulas for w_2 are similar, except that they begin with a $\neg @ \,\mathbf{U}\, (@ \wedge \ldots)$, and \$ replaces @. We add the formula $(\neg \$) \,\mathbf{U}\, @$ to make sure that the first \$ is immediately after w_2.

Next, we describe the formula θ, which requires that for all positions $0 \leq j \leq 2^{n-1}$, the j-th letter in w_1 is equal to the j-th position in w_2. While such a universal quantification on j is impossible in LTL, it can be achieved using the abort on operator.

We start with some auxiliary formulas:

$$\theta_= = \# \wedge \bigwedge_{i=0}^{n-1} ((\mathbf{X}^i\, c_0 \wedge ((\neg \$) \,\mathbf{U}\, (\$ \wedge \mathbf{X}^{i+1}\, c_0))) \vee (\mathbf{X}^i\, c_1 \wedge ((\neg \$) \,\mathbf{U}\, (\$ \wedge \mathbf{X}^{i+1}\, c_1))))$$

The formula requires the current position value to agree with the position value right after \$. Then, the formula

$$\theta_{next0} = (\theta_= \wedge ((\neg @) \,\mathbf{U}\, (@ \wedge (((\# \wedge \theta_=) \rightarrow 0) \,\mathbf{U}\, \$)))) \text{ abort on } \$.$$

requires that we are in a beginning of a block in w_1, and every block between @ and \$ whose position is equal to the position of the current block (note that there is exactly one such block) is marked with 0. Intuitively, let

$$\theta'_{next0} = \theta_= \wedge ((\neg @) \,\mathbf{U}\, (@ \wedge (((\# \wedge \theta_=) \rightarrow 0) \,\mathbf{U}\, \$)))$$

Then, θ'_{next0} requires that we are in a beginning of a block in w_1, the block position is equal to the position of the block that starts after \$, and every block between @ and \$ whose position is also equal to the position of the block that starts after \$ is marked with 0. Thus, θ'_{next0} is equivalent to θ_{next0} except that it fails when the current block does not match the block after \$. This is where the abort operator enters the picture. For every position, if the corresponding block is marked 0, the prefix of the word that ends at \$ can be extend such that the current block position match the position of the block that starts after \$. This extension would satisfy θ'_{next0}, thus the word satisfies θ_{next0}. The formula θ_{next1} is defined similarly.

Now, the formula θ requires that $w_1 = w_2$.

$$\theta = (((\# \wedge 0) \rightarrow \theta_{next0}) \wedge ((\# \wedge 1) \rightarrow \theta_{next1})) \,\mathbf{U}\, @$$

Words of length $exp(k, n)$. So far we have shown how to construct ψ_n^1, which defines equality between words of length $exp(1, n)$. We would like to scale up the technique to construct formulas ψ_n^k that define equality between words of length $exp(k, n)$. (As before, we use @ to mark the end of the first word and we use \$ to mark the end of the second word.) To do that, we encode such words by sequences consisting of $exp(k, n)$ $(k-1)$-blocks, of length $exp(k-1, n)$ each. Each such $(k-1)$-block, whose beginning is marked by $\#_{k-1}$, represents one letter, encoding both the letter itself as well as its position in the word, which requires $exp(k-1, n)$ bits. We need to require that (1) $(k-1)$-blocks behave as an $exp(k-1, n)$-counter, i.e., the first $(k-1)$-block is identically 0, and subsequent $(k-1)$-blocks count modulo $exp(k, n)$, and (2) if there are two $(k-1)$-blocks, b_1 in the first word and b_2 in the second word that encode the same position, then they must encode the same letter. To express (1) and (2), we have to refer to bits inside the $(k-1)$-blocks, which we encode using $(k-2)$-blocks, of length $exp(k-2, n)$.

Thus, we need an inductive construction. We start with 0-blocks, of length n, and use formulas Γ^0 to require that the 0-blocks behave as an n-bit counter (using the formulas $\gamma_1, \dots, \gamma_8$ from earlier). Inductively, suppose we have already required the $(k-2)$-blocks to behave as an $exp(k-2, n)$ counter. We now want every sequence of $exp(k-1, n)$ $(k-2)$-blocks, initially marked with $\#_{k-1}$, to encode a $(k-1)$ block. We use the values of a proposition c^{k-1} at the start of each $(k-2)$-block to encode the bits of the $(k-1)$-block.

We now need to write formulas analogous to $\gamma_1, \dots, \gamma_8$ to require that $(k-1)$-block to behave as an $exp(k-1, n)$-bit counter. The difficulty is in referring to bits in the same position of successive $(k-1)$-blocks using formulas of size polynomial in n (for $k = 1$ we can use $\mathbf{X}^n$ to refer to corresponding bits in successive 0-blocks). To refer to corresponding bits in successive $(k-1)$-blocks, we use the fact that each such bit is encoded using $(k-2)$-blocks. Thus, referring to such bits require the comparison of $(k-2)$-blocks. Also, to say that the two words, each of length $exp(k, n)$ are equal we need to express the analog of θ, which requires the analogue of $\theta_=$. But the latter use a conjunction of size n to range over all the n-bits of a 0-block. Here we need to range over all $(k-2)$ blocks and compare pair of such blocks.

Thus, the key is to be able to compare i-blocks, for $i = 0, \dots, k-1$. Once we are able to compare i-blocks we can go ahead and construct and compare $(i+1)$-blocks. To compare i-blocks for $i \geq 1$ we use the marker $\$_i$. Instead of directly comparing two i-blocks, we compare them both to the i-block that come immediately after $\$_i$, just as in $\theta_=$ we compared two 0-blocks to the 0-block that comes immediately after the $\$$ marker. By "aborting on" $\$_i$ we make sure that we are comparing the two i-blocks to *some* i-block that could come after $\$_i$; this way we are not bound to some specific i-block that actually comes after $\$_i$.

ψ_n^k is a conjunction of a sequence of sets of formulas. The construction of the sets of formulas is inductive, for every level i ($0 \leq i \leq k$), we define Γ^i and Θ^i that require level i to be "legal" and make some "tools" for level $i+1$. The set Γ^i requires the followings:

- γ_1^i requires that the counter value of the first i-block is $000\dots0$.
- γ_2^i requires that after every $\#_i$ before the next $\#_i$ there are $exp(i, n)$ many $(i-1)$-blocks without $\#_i$. This formula is only needed in level 0, after that it is taken care of by γ_7^{i-1} and γ_8^{i-1}.
- The following four formulas (γ_3^i, γ_4^i, γ_5^i, and γ_6^i) make sure that the counter (that is encoded by c_0^i, c_1^i) value is increased by one every $\#_i$. We use an additional proposition z^i that represents the carry.
- In the following two formulas (γ_7^i and γ_8^i), the first $k-1$ levels are a bit different from the kth level. The first $k-1$ levels require that at the $\#_{i+1}$ proposition will be true only at the beginning of every $(i+1)$-block. The formulas of the kth level require that the @ will be true exactly at the beginning of w_2.

A similar set of formulas is used for w_2. In addition for $i > 0$, we require that the first $\$_i$ marker appears after w_1 and w_2, and that the first $\$_i$ is proceeded by a legal i-block, and that i-block is proceeded by the first $\$_{i-1}$. These requirements can be formulated easily using formulas similar of formulas of Γ^i.

The Θ^i set requires two basic conditions:

1. A formula $\theta^i_{\#next0}$ that requires that the i-block between the next $\#_{(i+1)}$ and the one after, which has the same position value as the current i-block, represents the letter c_0^{i+1}. (A similar formula is needed for c_1^{i+1}).
2. A formula $\theta^i_{\$next0}$ that requires that the i-block in the $(i+1)$-block that starts after the first $\$_{(i+1)}$, and has the same position value as the current i-block, represents the letter c_0^{i+1}. (A similar formula is needed for c_1^{i+1}).

both formulas uses the auxiliary formula $\theta^i_=$ that requires the current i-block to be equivalent to the i-block that starts right after the first $\$_i$.

We present some examples that demonstrate the inductive construction. The base of the induction is the construction of Γ^0 and Θ^0. The formulas of Γ_0 are similar to the formulas that are presented in the former section, the main difference is that the 0-block with $11\ldots1$ value does not imply the end of the first word, but that the next 0-block should be marked with $\#_1$. Thus $\gamma_8^0 = ((\#_0 \wedge \bigwedge_{0\leq i<n} X^i c_1^0) \rightarrow \mathbf{X}^n\, \#_1)\,\mathbf{U}\,@$
As for the formulas of Θ^0, they are similar to the formulas that presented in the former section, only here we also need formulas that determine the value of the matching address in the next 1-block. For example, $\theta^0_{\#next0}$ requires the matching 0-block in the next 1-block to represents c_0^1. Thus,
$\theta^0_{\#next0} = (\theta^0_= \wedge \mathbf{X}((\neg\#_1)\,\mathbf{U}\,(\#_1 \wedge (((\#_0 \wedge \theta^0_=) \rightarrow c_0^1)\,\mathbf{U}\,\mathbf{X}\,\#_1))))$ abort on $\$_0$

Assume that for some $1 < i \leq k$, we already constructed Γ^j and Θ^j for every $j < i$. The structure of Γ^{i-1} is the base for Γ^i. For example, $\gamma_1^i = (\#_{i-1} \rightarrow c_0^i)\,\mathbf{U}\,\mathbf{X}\,\#_i$.

In the formulas that require the positions of the i-blocks to increased by one form one block to the next, we use the $\theta^{i-1}_{\#next0}$ and $\theta^{i-1}_{\#next1}$ formulas instead of the $\mathbf{X}^n$ operator. For example, $\gamma_3^i =$
$((\#_{i-1} \wedge (\#_i \vee z^i) \wedge c_0^i) \rightarrow (\mathbf{X}((\neg\#_{i-1})\,\mathbf{U}\,(\#_{i-1} \wedge (\neg z^i))) \wedge \theta^{i-1}_{\#next1}))\,\mathbf{U}\,(\#_i \wedge \mathbf{X}(\neg\#_i)\,\mathbf{U}\,@)$

Next, we describe Θ^i, the main change is in $\theta^i_=$, which requires that the current i-block position value is equal to the i-block that starts after $\$_i$. Thus,
$\theta^i_= = (((\#_{i-1} \wedge c_0^i) \rightarrow \theta^{i-1}_{\$next0}) \wedge ((\#_{i-1} \wedge c_1^i) \rightarrow (\theta^{i-1}_{\$next1})))\,\mathbf{U}\,\mathbf{X}\,\#_i$

In the rest of the formulas we use similar techniques. For example,
$\theta^i_{\#next0} = (\theta^i_= \wedge ((\neg\#_{i+1})\,\mathbf{U}\,(\#_{i+1} \wedge (((\#_i \wedge \theta^i_=) \rightarrow c_0^i)\,\mathbf{U}\,\mathbf{X}\,\#_{i+1}))))$ abort on $\$_i$, which requires that the matching i-block in the next $(i+1)$-block represents c_0^{i+1}.

The last formula that we define requires w_1 and w_2 to be equivalent. First we define $\theta^i_{@next0} = (\theta^{k-1}_= \wedge (\neg @)\,\mathbf{U}\,@ \wedge \mathbf{X}(((\#_{k-1} \wedge \theta^k_=) \rightarrow 0)\,\mathbf{U}\,\mathbf{X}\,@))$ abort on $\$_{k-1}$, which requires that the matching $(k-1)$-block in w_2 represents 0. Next, we define $\theta^i_{@next1}$ in a similar way. Then, we define
$\theta_{=w} = (((\#_{k-1} \wedge 0) \rightarrow \theta^{k-1}_{@next0}) \wedge ((\#_{k-1} \wedge 1) \rightarrow (\theta^{k-1}_{@next1})))\,\mathbf{U}\,\mathbf{X}\,@$,
which requires that $w_1 = w_2$

We now discuss the length of the formulas in the above construction. For every $0 \leq i \leq k$, we have a constant number of formulas in Γ^i and Θ^i, thus the number of formulas is $O(k)$. The problem is in formulas that recursively use other formulas. Since formulas like $\theta^i_=$ contains four sub-formulas $\theta^{i-1}_=$, the length of $\theta^k_=$ is $O(4^k)$. Thus the total length of the formulas is $O(4^k + n)$.

Lemma 1. *Every ABW that accepts ψ_n^k has at least $exp(k, n)$ states.*

Lemma 1 shows that the the automata-theoretic approach to Abort-LTL has a nonelementary cost. We now show that this cost is intrinsic to Abort-LTL and is not an artifact of the automata-theoretic approach.

Satisfiability and model-checking for Abort-LTL. We now prove that satisfiability checking for Abort-LTL is SPACE($exp(k, n)$)-hard. We show a reduction from a hyper-exponent version of the *tiling problem* [22,9,17]. The problem is defined as follows relative to a parameter $k > 0$. We are given a finite set T, two relations $V \subseteq T \times T$ and $H \subseteq T \times T$, an initial tile t_0, a final tile t_a,and a bound $n > 0$. We have to decide whether there is some $m > 0$ and an a tiling of an $exp(k, n) \times m$-grid: such that: (1) t_0 is in the bottom left corner and t_a is in the top left corner, (2) Every pair of horizontal neighbors is in H, and (3) Every pair of vertical neighbors is in V. Formally: Is there a function $f : (exp(k, n) \times m) \rightarrow T$ such that (1) $f(0, 0) = t_0$ and $t(0, m - 1) = t_a$, (2) for every $0 \leq i < exp(k, n)$, and $0 \leq j < m$, we have that $(f(i, j), f(i + 1, j)) \in H$, and (3) for every $0 \leq i < exp(k, n)$, and $0 \leq j < m - 1$, we have that $(f(j, i), f(j, i + 1)) \in V$. This problem is known to be SPACE($exp(k, n)$)-complete [9,17].

We reduce this problem to the satisfiability problem for $\mathsf{Abort} - \mathsf{LTL}$. Given a tiling problem $\tau = \langle T, H, V.t_0, t_f, n\rangle$, we construct a formula ψ_τ such that τ admits tiling iff ψ_τ is satisfiable. The idea is to encode a tiling as a word over T, consisting of a sequence of blocks of length $l = exp(k, n)$, each encoding one row of the tiling. Such a word represents a proper tiling if it starts with t_0, ends with a block that starts with t_a, every pair of adjacent tiles in a row are in H, and every pair of tiles that are $exp(k, n)$ tiles apart are in V. The difficulty is in relating tiles that are far apart. To do that we represent every tile by a $(k - 1)$-block, of length $exp(k - 1, n)$, which represent the tiles position in the row. As we had earlier, to require that the $(k - 1)$-blocks behave as a $exp(k - 1, n)$-bit counter and to compare $(k - 1)$-blocks, we need to construct them from $(k - 2)$-blocks, which needs to be constructed from $(k - 3)$-blocks, and so on. Thus, as we had earlier, we need an inductive construction of i-blocks, for $i = 1, \ldots, k - 1$, and we need to adapt the machinery of the previous nonelementary lower-bound proof.

It can be shown that there exists an exponential reduction from the nonelementary domino problem to the satisfiability of Abort-LTL formulas. Thus the satisfiability problem of the Abort-LTL is non-elementary hard.

Theorem 8. *The satisfiability and model-checking problems for* Abort-LTL *formulas nesting depth k of* abort on *are SPACE($exp(k, n)$)-complete.*

5 Concluding Remarks

We showed in this paper that the distinction between reset semantics and abort semantics has a dramatic impact on the complexity of using the language in a model-checking tool. While Reset-LTL enjoys the "fast-compilation property"–there is a linear translation of Reset-LTL formulas into alternating Büchi automata, the translation of Abort-LTL formulas into alternating Büchi automata is nonelementary, as is the complexity of satisfiability and model checking for Abort-LTL. This raises a concern on the feasibility of implementing a model checker for logics based on Abort-LTL(such as Sugar 2.0). While the nonelementary blow-up is a worst-case prediction, one can conclude from our results that while Reset-LTL can be efficiently compiled using a rather modest extension to existing LTL compilers (e.g., [4,3]), a much more sophisticated automata-theoretic machinery is needed to implement an compiler for Abort-LTL.

It is important to understand that the issue here is not simply the complexity blow-up for some convoluted formulas of Abort-LTL. As noted earlier, the proof of Theorem 3 shows that the standard syntax-driven approach to compiling LTL to automata applies also to Reset-LTL; in fact, the ForSpec compiler applies syntax-driven processing to all ForSpec's constructs [1]. In contrast, the proof of Theorem 6, buttressed by Lemma 1, shows that to have a general compilation of Abort-LTL to automata one cannot proceed in a similar syntax-directed fashion. Thus, the sketchy description of a syntax-directed compilation scheme provided in the documentation of Sugar 2.0 is not only incomplete but also seriously underestimates the effort required to implement a compiler for full Sugar 2.0.

Acknowledgements. Work of Bustan and Vardi supported in part by NSF grants CCR-9988322, CCR-0124077, IIS-9908435, IIS-9978135, and EIA-0086264, by BSF grant 9800096, and by a grant from the Intel Corporation. Kupferman is visiting UC Berkeley and is supported by NSF grant CCR-9988172.

References

1. R. Armoni, L. Fix, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, A. Tiemeyer, E. Singerman, M.Y. Vardi, and Y. Zbar. The ForSpec temporal language: A new temporal property-specification language. In *Proc. 8th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, Lecture Notes in Computer Science 2280, pages 296–311. Springer-Verlag, 2002.
2. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *Proc. Conf. on Computer-Aided Verification*, LNCS 2102, pages 363–367, 2001.
3. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
4. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, August 1995.
5. S.M. Nowick K. van Berkel, M.B. Josephs. Applications of asynchronous circuits. *Proceedings of the IEEE*, 1999. special issue on asynchronous circuits & systems.
6. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal methods in System Design*, 19(3):291–314, November 2001.
7. R.P. Kurshan. Formal verification in a commercial setting. In *Proc. Conf. on Design Automation (DAC'97)*, volume 34, pages 258–262, 1997.
8. R.P. Kurshan. *FormalCheck User's Manual.* Cadence Design, Inc., 1998.
9. H.R. Lewis. Complexity of solvable cases of the decision problem for the predicate calculus. In *Foundations of Computer Science*, volume 19, pages 35–47, 1978.
10. S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

11. M.J. Morley. Semantics of temporal ***e***. In T. F. Melham and F.G. Moller, editors, Banff'99 *Higher Order Workshop (Formal Methods in Computation)*. University of Glasgow, Department of Computing Science Technic al Report, 1999.
12. D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *Proc. 13th Int. Colloquium on Automata, Languages and Programming*, LNCS 226, 1986.
13. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.
14. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal ACM*, 32:733–749, 1985.
15. A comparison of reset control methods: Application note 11. `http://www.summitmicro.com/tech_support/notes/note11.htm`, Summit Microelectronics, Inc., 1999.
16. W. Thomas. A combinatorial approach to the theory of ω-automata. *Information and Computation*, 48:261–283, 1981.
17. P. van Emde Boas. The convenience of tilings. In *Complexity, Logic and Recursion Theory*, volume 187 of *Lecture Notes in Pure and Applied Mathetaics*, pages 331–363, 1997.
18. M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, LNCS 1043, pages 238–266, 1996.
19. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, 1986.
20. M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, April 1986.
21. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.
22. H. Wang. Dominoes and the aea case of the decision problem. In *Symposium on the Mathematical Theory of Automata*, pages 23–55, 1962.