A Fragment of Linear Temporal Logic for Universal Very Weak Automata*

Keerthi Adabala and Rüdiger Ehlers

University of Bremen, Bremen, Germany

Abstract. Many temporal specifications used in practical model checking can be represented as universal very weak automata (UVW). They are structurally simple and their states can be labeled by simple temporal logic formulas that they represent. For complex temporal properties, it can be hard to understand why a trace violates a property, so when employing UVWs in model checking, this information helps with interpreting the trace. At the same time, the simple structure of UVWs helps the model checker with finding short traces.
While a translation from computation tree logic (CTL) with only universal path quantifiers to UVWs has been described in earlier work, complex temporal properties that define sequences of allowed events along computations of a system are easier to describe in linear temporal logic (LTL). However, no direct translation from LTL to UVWs with little blow-up is known.
In this paper, we define a fragment of LTL that gives rise to a simple and efficient translation from it to UVW. The logic contains the most common shapes of safety and liveness properties, including all nestings of "Until"-subformulas. We give a translation from this fragment to UVWs that only has an exponential blow-up in the worst case, which we show to be unavoidable. We demonstrate that the simple shape of UVWs helps with understanding counter-examples in a case study.

1 Introduction

Complex reactive systems often have complex specifications. To obtain a sufficient degree of quality assurance, a model of the system can be verified against the specification. Automata-based model checking is a classical approach in this context, as it permits the specification to be written in a powerful logic such as linear temporal logic (LTL, [1]), which is then translated to an automaton for the verification process [2].

Whenever the system to be verified is found to violate the specification, a model checker can compute a (lasso-shaped) *counter-example trace* [3,2]. Such traces are often lengthy and the problem of explaining why the system behaves in the way observed in the trace has received some attention in the literature [4,5]. However, finding out why the behavior actually violates the property is

* This work was supported by DFG grant EH 481/1-1 and the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative.

also difficult [5]. While the trace includes a run of the automaton built from the specification, the various optimizations in the translation process from the specification to the automaton normally lead to a loss of structure. Hence, the run of the automaton does not give rise to an easy interpretation of the reason for the violation of the specification written by a system engineer. When not optimizing an automaton, it frequently becomes huge, which translates to a higher computational workload and can also lead to longer counter-example traces.

These observations give rise to the question if we can help a model checker with finding easy to interpret counter-example traces by employing very structured, but still small automata in the verification process. We present an approach for this purpose in this paper that is based on *universal very weak ω-automata*. Maidl [6] showed that this automaton class captures exactly the specifications that are representable both in linear temporal logic (LTL) and computation tree logic (CTL), where in the latter case only universal path quantifiers are used. Universal very weak automata (UVWs) expose the sequences of events that must not lead to errors, deadlocks or livelocks. They can be decomposed into a finite number of so-called *simple chains* that represent these sequences of events. There are multiple reasons for why this makes them interesting for counter-example trace generation:

1. Whenever a property is violated, we can search for counter-example traces for all simple chains. The information for which of these chains a violation can be found is helpful for pinpointing the error.
2. Counter-example traces for different simple chains can have different lengths, so the shortest one can be reported to the system engineer.
3. Along a trace, a UVW run can move to a different state only few times. These state changes represent points in time in which interesting events happen, so they can be highlighted to the engineer.
4. Every state in a UVW can be labeled by a relatively simple temporal logic formula that the state represents, and no two states in a minimized automaton are labeled in the same way, which eases the interpretation of a trace by the engineer.

So for those specification parts that can be represented as universal very weak automata, employing them for model checking the specification part simplifies debugging the model and hence speeds up the iterations of model and specification refinement that are characteristic for a model-based system development process.

Despite their nice properties, universal very weak automata are not well-studied. It is for example currently unknown how much blow-up is unavoidable when translating from LTL to UVW. Earlier work [7] contained a translation construction, but it requires the input to be represented as a deterministic Büchi automaton, which implies at least a doubly-exponential translation time and potentially large automata. Furthermore, the construction computes the UVW in an iterative way, which further increases the computation times.

To counter this problem, we provide a characterization of a subset of linear temporal logic (LTL) that permits an efficient translation to universal very weak automata in this paper. This characterization is given in the form of a

context-free grammar and captures, for example, all possible nestings of the *Until*-operator of LTL. We provide a translation procedure from formulas in the grammar to UVW. All states in the resulting UVWs represent languages of Boolean combinations of subformulas in the LTL specification. While we do employ simulation-based state minimization techniques, they are used in a way in which they do not invalidate the temporal logic state labeling in the UVW case. At the same time, no two states represent the same LTL (sub-)formulas, which can happen for minimally-sized classical Büchi automata, which are normally used in model checking. Hence, the state information in counter-example traces produced by a model checker is easy to interpret.

We demonstrate in a case study (using the model checker `spin` [8]) that the structure of the specification UVWs helps with finding the root cause of a specification violation. Since our LTL fragment covers the majority of specification shapes found in the literature, our construction is applicable in many application contexts.

1.1 Related Work

Translating properties from linear temporal logic (LTL) to automata is a classical topic in the formal methods literature as it is a required step for automata-based model checking (or reactive synthesis). When translating to non-deterministic Büchi automata, an exponential blow-up cannot be avoided [9], but by applying *simulation-based minimization* of the resulting automaton, automata sizes can be substantially reduced in practice [10,3]. Since model checking problems generally become easier when employing small automata, they are normally preferred. It has been noted, however, that the efficiency of model checking is also influenced by the *shape* of the specification automata. In particular, automata that delay the first visit to an accepting state have been found to lead to better model checking efficiency [3].

Another special automaton shape are *very weak* ω-automata. In such automata, all loops are self-loops, and universal very weak automata (UVW) have been identified as the automaton class that exactly characterizes the word languages that can be represented in LTL and for which the containment of all paths in a computation tree in the language can also be represented by a formula in computation tree logic (CTL) using only universal path quantifiers [6] (abbreviated as ACTL). This fragment is interesting as it unifies the two commonly used specification logics and because UVWs can be decomposed for distributed model checking, as we show in Section 2. While Maidl gave a construction to translate from ACTL to UVW whenever possible, the subset of LTL for which she gave a translation to UVW is highly restrictive and does not even allow to express $a \mathcal{U} b$ (a holds until b holds at least once). Effectively, her approach requires the specification engineer to encode the structure of a UVW into the logical specification. The grammar that we define in the next section does not have this restriction and allows arbitrary nestings of $\mathcal{U}$ operators. It also includes Maidl's LTL subset as a special case.

All of the automata translations discussed so far compute automata that can have a very complicated structure and that are hard to interpret. For example, one of the classical approaches to translating from LTL to Büchi automata involves *de-alternation* [11], which introduces *breakpoints* into the automaton structure. The main alternative translation appraoch involves *de-generalizing* generalized Büchi automata [12], which introduces a similar automaton structure. Subsequent automaton minimization steps [10] lead to additional incoherence between the automaton structure and the original specification. As a consequence, observing a run of an automaton does not help to explain *why* a trace satisfies a specification or not.

To solve this issue, Basin et al. [5] defined a calculus for *annotating* a counter-example obtained from a model checker (which has a lasso shape) with an explanation why it violates a given LTL property. Their approach is only applicable after a lasso has been computed, and there is no guarantee that the model checker picks a lassos that has an easy to explain reason for violating the specification. While short lassos make this more likely, their length is still influenced by the structure of the specification automaton. Asking for a counter-example trace of the form uv^ω with $|u| + |v|$ as short as possible would solve this problem, but approximating the minimal attainable length $|u| + |v|$ by any factor has been shown to be NP-hard [13], unlike finding shortest lassos.

In the approach that we present in this paper, we solve this problem by computing automata that have a simple structure and whose states are labeled by the LTL property that the state represents. The automata can be decomposed so that a model checker that searches for short lassos also searches for lassos that have an easy explanation.

2 Preliminaries

An ω-word automaton over some finite alphabet Σ (which we assume to be 2^{AP} for some set AP for the scope of this paper) is a tuple $\mathcal{A} = (Q, \delta, Q_0, \mathcal{F})$ with the finite set of states Q, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the set of initial states $Q_0 \subseteq Q$, and the acceptance condition $\mathcal{F} \subseteq Q$.

Given a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, we say that $\mathcal{A}$ induces an *infinite run* $\pi = \pi_0 \pi_1 \ldots \in Q^\omega$ if $\pi_0 \in Q_0$ and for all $i \in \mathbb{N}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta$. For the scope of this paper, we are only interested in infinite runs.

Word automata come in different types. In this paper, we will consider two types, namely *non-deterministic Büchi automata* (NBA) and *universal very weak automata* (UVW). For the former, we say that the automaton accepts a word w if there exists a run π induced by it and $\mathcal{A}$ along which states in $\mathcal{F}$ occur infinitely often. For a universal very weak automaton $\mathcal{A}$, we say that it accepts a word w if for *all* infinite runs π induced by $\mathcal{A}$ and w, we have that states in $\mathcal{F}$ appear only finitely often along π. For an automaton to be a UVW, its states furthermore need to be ranked, i.e., there exists a ranking function $r : Q \to \mathbb{N}$ such that for every $q \in Q$ and $q' \in Q$, if there exists a $x \in \Sigma$ with $(q, x, q') \in \delta$ then $r(q') < r(q)$ or $q = q'$. Intuitively, this means that all loops in the automaton are

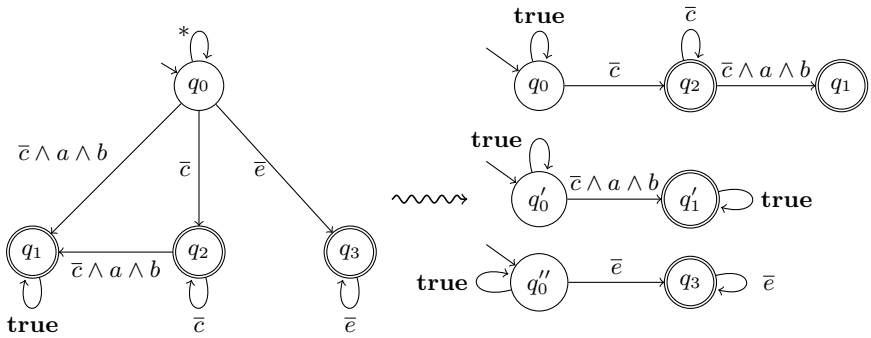

Fig. 1. A UVW and its decomposition into simple UVW chains for the property of $\mathsf{G}((a \rightarrow b)\,\mathcal{U}\,c) \wedge \mathsf{GF}(d\,\mathcal{U}\,e)$

self-loops (as visualized in Figure 1). Due to the existence of a ranking function, the acceptance of a word basically boils down to stating that no infinite run should eventually get stuck in a state $q \in \mathcal{F}$. We call $\mathcal{F}$ the set of *rejecting states* in case of UVWs, and the set of *accepting* states for NBAs. The language of an automaton $\mathcal{A}$, denoted as $\mathcal{L}(\mathcal{A})$, is defined to be the set of words accepted by $\mathcal{A}$.

If the set AP is suitable for modeling the current state of a system to be verified, ω-word automata over the alphabet $\Sigma = 2^{\mathsf{AP}}$ serve as an (internal) representation of a specification for model checking. They are however cumbersome to write, so a temporal logic such as linear temporal logic (LTL, [1]) typically serves as specification language used by system engineers, with the aim to automatically translate LTL properties to automata. LTL enriches Boolean logic by the addition of the *next* (X), *until* ($\mathcal{U}$), *weak until* ($\mathcal{W}$), *release* ($\mathcal{R}$), *globally* (G), and *finally* (F) operators, and a formal definition of the logic and its semantics can be found in [1]. We say that an automaton is equivalent to an LTL formula if the set of words over 2^{AP} that are models of the LTL formula is the same as the language of the automaton. A finite word over the character set 2^{AP} is a bad prefix for some LTL formula ψ if it cannot be extended to a word that satisfies the formula. A good prefix of some LTL formula ψ is a finite word all of whose infinite extensions satisfy the LTL formula. A specification for which all words that violate it have a bad prefix is called a *safety specification*. A specification without bad prefixes is called a *liveness specification*.

In the following, we use subsets of atomic propositions and their characteristic (Boolean) functions interchangeably. A transition (q_1, t, q_2) for some Boolean formula t represents transitions from a state q_1 to q_2 for all $x \in \Sigma$ that satisfy t. The $\bot$ symbol henceforth represents an invalid Boolean formula – applying any operation to it yields $\bot$ again. We also use these notations in figures depicting automata, where states are given as circles, states in $\mathcal{F}$ are doubly-circled, and transitions are depicted by arrows that are labeled by Boolean formulas t. When depicting UVWs, we furthermore draw them in a way that their ranking

functions become apparent, e.g., by letting all non-self-loop transitions lead to the right or down.

In the verification literature, non-deterministic Büchi automata are often used to represent a set of traces that a system to be verified should *not* permit and hence represent the complement of a specification. By switching from non-deterministic to universal branching (as common in the literature on ACTL $\cap$ LTL [6,14]), we avoid this complementation in reasoning, as UVWs accept all traces that *do* satisfy the specification. The complement of a specification representable as a UVW can be represented as a nondeterministic Büchi automaton (with exactly the same automaton tuple elements).

A UVW $\mathcal{A}$ can be decomposed into multiple sub-automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ (for some $n \in \mathbb{N}$), where each sub-automaton represents one path through $\mathcal{A}$, as shown in Figure 1. We call these paths *simple chains*, and formally, the intersection of their languages is the language of $\mathcal{A}$, i.e., we have $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2) \cap \ldots \cap \mathcal{L}(\mathcal{A}_n) = \mathcal{L}(\mathcal{A})$.

3 A Temporal Logic for Universal Very Weak Automata

In this section, we give a context-free grammar that captures a subclass of LTL formulas and a translation from this subclass to UVWs. Without loss of generality, we assume that occurrences of the negation operator in front of temporal operators have already been pushed inwards, just like in the *negation normal form* [2] of LTL. Negation operators located in front of pure Boolean sub-formulas do not have to be pushed inwards. The grammar for UVWs has the following components:

$$\chi ::= p \mid \neg\chi \mid \chi \wedge \chi \mid \chi \vee \chi \mid \mathbf{true} \mid \mathbf{false}$$
$$\psi ::= \chi \mid \psi \vee \psi \mid \mathsf{F}\psi \mid \phi\,\mathcal{U}\,\psi$$
$$\phi ::= \psi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{G}\phi \mid \mathsf{X}\phi \mid \psi\,\mathcal{R}\,\phi \mid (b \wedge \phi)\,\mathcal{U}\,(\neg b \wedge \phi) \mid (b \wedge \phi)\,\mathcal{W}\,(\neg b \wedge \phi)$$

In this grammar, p denotes an atomic proposition and b is a Boolean formula without temporal operators. Such formulas are accepted by the nonterminal χ. Note that in the last two rules for ϕ, we assume that the Boolean formula b is the same for both occurrences.

The acceptance of an LTL formula by the top-level nonterminal ϕ indicates that the LTL formula can be translated to a UVW, as we show below. The nonterminal ψ represents subformulas for which *quitting points* can be detected, which are defined as follows:

Definition 1. *Let f be an LTL formula with (strict) subformulas $\mathcal{S}$ for some set of propositions AP. A prefix word $w = w_0 \ldots w_n \in (2^{\mathsf{AP}})^*$ is called a* quitting point *if there exists a Boolean combination f' of subformulas from $\mathcal{S}$ such that for all words $u = w_0 \ldots w_{n-1} w_n u_{n+1} u_{n+2} \ldots \in \Sigma^\omega$, we have $u \models f$ if and only if $w_n u_{n+1} u_{n+2} \ldots \models f'$.*

Quitting points intuitively represent prefix words for some LTL formula for which the top-level formula does not need to be monitored in order to find out if a word

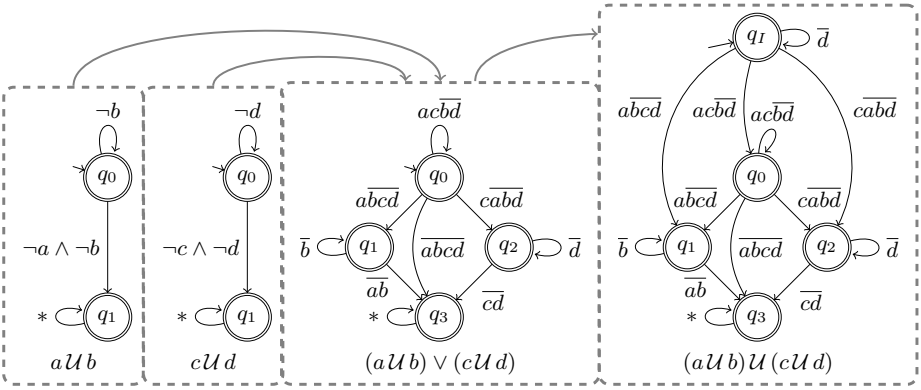

Fig. 2. Building a UVW for the LTL formula $f = (a\,\mathcal{U}\,b)\,\mathcal{U}\,(c\,\mathcal{U}\,d)$ in a step-by-step way from the UVW for the subformulas. Rejecting states are doubly-circled.

satisfies the formula after the point has been reached. For example, for the LTL formula $f = (a\,\mathcal{U}\,b)\,\mathcal{U}\,c$, any prefix word that ends with a character that includes c is such a quitting point, as after a c is seen along a trace, the outer-level obligation encoded in f is satisfied. However, when a quitting point has been reached, this does not necessarily mean that the satisfaction of the LTL formula is already established. For example, for $f = (a\,\mathcal{U}\,b)\,\mathcal{U}\,c$, the prefix word $\{a\}\{a, c\}$ is a quitting point, but the remainder of the word still has to satisfy $a\,\mathcal{U}\,b$ for f to be satisfied along the complete trace.

In the grammar given above, ψ has been carefully defined to only contain subformulas for which quitting points can be detected without recall for the history of the prefix word observed earlier. This enables us to construct UVWs for a specification with liveness objectives. Take for example the specification $f = (a\,\mathcal{U}\,b)\,\mathcal{U}\,(c\,\mathcal{U}\,d)$. The sub-formula $(a\,\mathcal{U}\,b)$ has all words ending with b as quitting points, whereas the second sub-formula has all words ending with d as quitting points. We can implement a translation to a UVW by adding one UVW state for each until-subformula $f^1\,\mathcal{U}\,f^2$ such that at least one run stays in this state until a quitting point has been seen, and until that is the case, the run branches to a state representing that $f^1 \vee f^2$ should hold. The overall translation is depicted in Figure 2. Note that a disjunction of two sub-formulas that enable history-free detection of quitting points has this property again, which we use in the translation.

The (recursive) Translate function that builds on this idea is given in Algorithm 1. The function takes an LTL formula and returns a pair consisting of a UVW for the LTL formula and a Boolean formula that encodes the set of characters with which quitting point prefixes for the LTL formulas end. For our implementation that we evaluate in Section 5, we cache the results of calls to Translate on an LTL subformula in case it occurs multiple times for the input formula. The number of generated UVW nodes is then at most exponential in the size of the formula (as every node generated by the algorithm can be

Algorithm 1 Translation procedure from an LTL subformula to a UVW and the characters that indicate that a quitting point has been just seen.

function TRANSLATE(f)
  if $f = t$ for some subformula t without temporal operators **then**
    $\mathcal{A} \leftarrow (\{q_0, q_1\}, \{(q_0, \neg t, q_1), (q_1, \mathbf{true}, q_1)\}, \{q_0\}, \{q_1\})$
    return $(\mathcal{A}, t)$
  if $f = f^1 \wedge f^2$ **then**
    $((Q^1, \delta^1, Q_0^1, \mathcal{F}^1), X^1) \leftarrow$ TRANSLATE(f_1)
    $((Q^2, \delta^2, Q_0^2, \mathcal{F}^2), X^2) \leftarrow$ TRANSLATE(f_2)
    $\mathcal{A} \leftarrow (Q^1 \uplus Q^2, \delta^1 \cup \delta^2, Q_0^1 \cup Q_0^2, \mathcal{F}^0 \cup \mathcal{F}^1)$
    return $(\mathcal{A}, \bot)$
  if $f = f^1 \vee f^2$ **then**
    $(\mathcal{A}^1, X^1) \leftarrow$ TRANSLATE(f_1), $(\mathcal{A}^2, X^2) \leftarrow$ TRANSLATE(f_2)
    return $(\mathcal{A}, X^1 \vee X^2)$, where $\mathcal{A}$ is the product of $\mathcal{A}^1$ and $\mathcal{A}^2$, where every state is rejecting for which both factor states are rejecting.
  if $f = f^1 \,\mathcal{U}\, f^2$ **then**
    $((Q^1, \delta^1, Q_0^1, \mathcal{F}^1), X^1) \leftarrow$ TRANSLATE($f_1 \vee f_2$)
    $((Q^2, \delta^2, Q_0^2, \mathcal{F}^2), X^2) \leftarrow$ TRANSLATE(f_2)
    $\mathcal{A} \leftarrow (Q^1 \uplus Q^2 \uplus \{q_0\}, \{(q_0, x, q_1) \mid \exists q_0^1 \in Q_0^1, (q_0^1, x, q_1) \in \delta^1, x \not\models X^2\} \cup \{(q_0, x, q_1) \mid \exists q_0^2 \in Q_0^2, (q_0^2, x, q_1) \in \delta^2, x \models X^2\} \cup \{(q_0, x, q_0) \mid x \not\models X^2\}, \{q_0\}, \mathcal{F}^0 \cup \mathcal{F}^1 \cup \{q_0\})$
    return $(\mathcal{A}, X^2)$
  if $f = f^2 \,\mathcal{R}\, f^1$ **then**
    $((Q^1, \delta^1, Q_0^1, \mathcal{F}^1), X^1) \leftarrow$ TRANSLATE($f_1 \vee f_2$)
    $((Q^2, \delta^2, Q_0^2, \mathcal{F}^2), X^2) \leftarrow$ TRANSLATE(f_2)
    $\mathcal{A} \leftarrow (Q^1 \uplus Q^2 \uplus \{q_0\}, \{(q_0, x, q_1) \mid \exists q_0^1 \in Q_0^1, (q_0^1, x, q_1) \in \delta^1\} \cup \{(q_0, x, q_1) \mid \exists q_0^2 \in Q_0^2, (q_0^2, x, q_1) \in \delta^2, x \models X^2\} \cup \{(q_0, x, q_0) \mid x \not\models X^2\}, \{q_0\}, \mathcal{F}^0 \cup \mathcal{F}^1 \cup \{q_0\})$
    return $(\mathcal{A}, \bot)$
  if $f = \mathsf{X} f^1$ **then**
    $((Q^1, \delta^1, Q_0^1, \mathcal{F}^1), X^1) \leftarrow$ TRANSLATE(f^1)
    $\mathcal{A} \leftarrow (Q^1 \cup \{q_0\}, \delta^1 \cup \{(q_0, \mathbf{true}, q_1) \mid q^1 \in Q_0^1\}, \{q_0\}, \mathcal{F}^1)$
    return $(\mathcal{A}, \bot)$
  if $f = (b \wedge f^1) \,\mathcal{U}\, (\neg b \wedge f^2)$ or $f = (b \wedge f^1) \,\mathcal{W}\, (\neg b \wedge f^2)$ **then**
    $((Q^1, \delta^1, Q_0^1, \mathcal{F}^1), X^1) \leftarrow$ TRANSLATE(f^1)
    $((Q^2, \delta^2, Q_0^2, \mathcal{F}^2), X^2) \leftarrow$ TRANSLATE(f^2)
    $\mathcal{A} \leftarrow (Q^1 \uplus Q^2 \uplus \{q_0\}, \{(q_0, x \wedge b, q_1) \mid \exists q_0^1 \in Q_0^1, (q_0^1, x, q_1) \in \delta^1\} \cup \{(q_0, x \wedge \neg b, q_1) \mid \exists q_0^2 \in Q_0^2, (q_0^2, x, q_1) \in \delta^2\} \cup \{(q_0, b, q_0)\}, \{q_0\}, \mathcal{F}^0 \cup \mathcal{F}^1 \cup K)$ for $K = \{q_0\}$ if $f = (b \wedge f^1) \,\mathcal{U}\, (\neg b \wedge f^2)$ and $K = \emptyset$ otherwise
    return $(\mathcal{A}, \bot)$

labeled by an LTL formula for its language, which is always a disjunction of subterms present in the original LTL formula). The algorithm does not show the implementations of the G and F operators, as they are special cases of the other operators (using the equivalences $\mathsf{G} f \equiv \mathbf{true}\,\mathcal{R}\, f$ and $\mathsf{F} f \equiv \mathbf{true}\,\mathcal{U}\, f$).

The construction is mostly straight-forward. For the disjunction case, we have to build a product automaton, which can lead to some blow-up.

Theorem 1. *Algorithm 1 computes a correct UVW for a given LTL formula under the assumption that the LTL formula is accepted by the grammar given in Section 3.*

Proof. Before we start with the main part of the proof, we need to show that for every subformula $f^1 \mathcal{U} f^2$ and $f^2 \mathcal{R} f^1$ in an overall LTL formula that is accepted by the nonterminal φ, we have that TRANSLATE(f^2) returns a UVW with exactly a single initial state. Since both of these temporal operators require that the operand f^2 is accepted by the ψ nonterminal, we only have to prove this for all subformulas accepted by this non-termininal. For all non-temporal subformulas, this sub-claim is true, as the UVW computed have exactly two states each, where only one is initial. This case forms our induction basis. For the disjunction case ($\psi ::= \psi \vee \psi$), the claim is also true as when taking the product of two UVW with one initial state each, the product also has only one initial state. Finally, the part of Algorithm 1 for the $\phi \mathcal{U} \psi$ and $\psi \mathcal{R} \phi$ cases all return UVWs with one initial state each.

Similarly, it can also be shown that for every subformula accepted by the ψ nonterminal, the second element of the tuple returned by TRANSLATE is never $\bot$, which we use for the proof.

Now to the main part of the proof. We prove the claim by induction on the structure of the LTL formula, where we use the induction hypothesis that for every subformula f, TRANSLATE(f) returns a pair $(\mathcal{A}, X)$ consisting of

1. a UVW $\mathcal{A}$ for f and
2. a subset $X \subseteq \Sigma$ such that
   (a) every prefix word ending with a letter from the subset is a quitting point,
   (b) X characterizes the one-letter prefix words that are good prefixes for f,
   (c) every word that is a model of f has to contain a character from X, and
   (d) for every prefix word $w_0 \dots w_n \in \Sigma^*$ that is a good prefix of f, we have that $w_n \in X$ and $w_1 \dots w_n$ is a good prefix for f as well.

   We also call X the set of *quitting characters* henceforth.

In this definition and henceforth, we treat character sets and LTL formulas that are free of temporal operators and that characterize such sets interchangeably. We still use $\bot$ to symbolize that no set/no Boolean function is provided.

Induction Basis: The only case in which TRANSLATE(f) does not recurse is when f is free of temporal operators. By the LTL semantics, the returned UVW should reject exactly the words not starting with a character that satisfies f. The UVW returned by the function has exactly two states. The non-initial one rejects all words. The initial one has a transition to the non-initial one that is taken whenever the first character of an input word does *not* satisfy f. Whenever this happens, the word is rejected as a run then visits the second non-initial state that is rejecting and self-loops on all characters. This implements exactly the semantics of an LTL formula that is free of temporal operators. The quitting characters returned along with the UVW are exactly the set of characters satisfying f (or, more precisely, for which exactly the words starting with one of them satisfy f), which is a valid set of quitting characters for f (by its definition).

Induction Step: We do a case split on the type of operator and assume that for the f^1 and f^2 sub-formulas, recursive calls to TRANSLATE yielded the UVWs $\mathcal{A}^1$ and $\mathcal{A}^2$ along with the quitting character sets X^1 and X^2, respectively.

- **Case** $f^1 \wedge f^2$: In this case, the resulting UVW should accept a word if and only if both of the UVWs for f^1 and f^2 accept a word. So all runs of both of them must accept a word. Under the inductive hypothesis that the UVWs returned by the calls to TRANSLATE(f^1) and TRANSLATE(f^2) are correct, this is achieved by merging the two UVWs into one and taking the initial states of both of them as new initial state set. The set of quitting characters is $\perp$, which means "does not apply" and is – by definition – a safe return value.
- **Case** $f^1 \vee f^2$: In this case, the resulting UVW should accept a word if and only if one of the UVWs for f^1 and f^2 accept the word. This case uses a product construction, where given the UVWs $\mathcal{A}^1 = (Q^1, \delta^1, Q_0^1, \mathcal{F}^1)$ and $\mathcal{A}^2 = (Q^2, \delta^2, Q_0^2, \mathcal{F}^2)$, the product UVW $\mathcal{A} = (Q, \delta, Q_0, \mathcal{F})$ with the following components is computed:
  - $Q = Q^1 \times Q^2$
  - $\delta = \{((q^1, q'^1), x, (q^2, q'^2)) \in Q \times \Sigma \times Q \mid (q^1, x, q^2) \in \delta^1, (q'^1, x, q'^2) \in \delta^2\}$
  - $Q_0 = Q_0^1 \times Q_0^2$
  - $\mathcal{F} = \mathcal{F}^1 \times \mathcal{F}^2$

  Let a word be given that is accepted by, w.l.o.g., $\mathcal{A}^1$. Then, every trace of $\mathcal{A}^1$ visits rejecting states only finitely often. All runs in $\mathcal{A}$ simulate runs of $\mathcal{A}^1$ and $\mathcal{A}^2$ in parallel. Since $\mathcal{F} = \mathcal{F}^1 \times \mathcal{F}^2$, we know that a run for $\mathcal{A}$ then also only visits rejecting states finitely often.
  On the other hand, let a word be rejected by both $\mathcal{A}^1$ and $\mathcal{A}^2$. Then there exist rejecting runs for both $\mathcal{A}^1$ and $\mathcal{A}^2$, and by the construction of $\mathcal{A}$, the product of these rejecting runs is a run of $\mathcal{A}$. Since both rejecting runs eventually get stuck in rejecting states, the product run in $\mathcal{A}$ also eventually gets stuck in a state in $\mathcal{F}^1 \times \mathcal{F}^2 = \mathcal{F}$, and hence is rejecting as well. Thus, the word is rejected by $\mathcal{A}$ as well.
  If furthermore a character set $X \subseteq \Sigma$ is returned by the TRANSLATE function for both f^1 and f^2 (i.e., not the $\perp$ element is returned), then the function definition declares its own returned character set to be the union of the character sets for f^1 and f^2. By the inductive hypothesis, any word starting with a character in the union of the characters satisfies one of f^1 and f^2. Likewise, every word without characters in this union is, by the inductive hypothesis, rejected by both $\mathcal{A}^1$ and $\mathcal{A}^2$. The same argument can be made for the conditions 2.(a) and 2.(d) of the inductive hypothesis given above.
- **Case** $f^1 \,\mathcal{U}\, f^2$**:** We assume that X^2 has the properties stated in the inductive hypothesis. By the definition, a word can only be a model of $f^1 \,\mathcal{U}\, f^2$ if eventually, a character from X^2 occurs in the word. The construction from Algorithm 1 for this case generates an initial state that is not left until such a character is read. Before the occurrence of this character, the outgoing transitions of the state are taken, which model the transitions leaving the initial states of a UVW for $f^1 \vee f^2$.
  So see why this construction is correct, let a word be given that satisfies $f^1 \,\mathcal{U}\, f^2$, where at positions 0 to j, f^1 is satisfied and at position $j+1$, f^2 is

satisfied. Let, without loss of generality, j be the least possible such index. A character from X^2 may first occur at a position $j' \geq j$ (it cannot occur earlier because otherwise j would not be the earliest possible such index). From positions 0 to j, the word surely satisfies $f^1 \vee f^2$ as it satisfies f^1. At position $j+1$ it satisfies f^2. In between positions j and j' in the word, we however now also know that f^2 is satisfied from there by the inductive hypothesis for X^2: by it, the word from position j onwards is a good prefix for f^2, and every suffix of this good prefix is a good prefix as well (except for the empty suffix). This includes the words from positions $j+1$, $j+2$, ..., until the character from X^2 occurs along the trace.
Note that the UVW generated for $f^1 \,\mathcal{U}\, f^2$ also does not accept too many words, as it enforces $f^1 \,\mathcal{U}\, f^2$ to hold until a letter has been seen that guarantees that f^2 is met. If $f^1 \vee f^2$ is always satisfied before this point, this implies that $f^1 \mathcal{U} f^2$ holds at the beginning of the word as well.
The algorithm returns X^2 as the set of quitting characters. This is correct as
1. no word not containing a character in X^2 can satisfy $f^1 \,\mathcal{U}\, f^2$
2. If a word satisfies $f^1 \mathcal{U} f^2$ from the first character, then it also satisfies $f^1 \mathcal{U} f^2$ from the second character onwards if f^2 is only satisfied later. If f^2 is satisfied from the first character onwards, then by the inductive hypothesis, the suffix of the word satisfies it as well (as otherwise X^2 would need to be $\bot$).

– **Case $f^1 \,\mathcal{R}\, f^2$:** This case is analogous to the $f^1 \,\mathcal{U}\, f^2$ case, except that $\bot$ is returned as quitting character set (which is always safe).
– **Case $\mathsf{X} f^1$:** In this case, a new UVW is generated that has one initial state from which all initial states of the UVW for f^1 are reached unconditionally. This implements exactly that the first character of a (suffix) trace is ignored. The algorithm returns $\bot$ as quitting character set, which is a safe choice.
– **Cases $(\alpha \wedge \phi) \,\mathcal{U}\, (\neg\alpha \wedge \phi)$ and $(\alpha \wedge \phi) \,\mathcal{W}\, (\neg\alpha \wedge \phi)$:** These special cases are similar to the $f^1 \,\mathcal{U}\, f^2$ and $f^1 \,\mathcal{R}\, f^2$ cases above, except that quitting character sets are not needed for determining whether at least one run should stay in the initial state added to $\mathcal{A}^1$ and $\mathcal{A}^2$ by the construction. Instead, the b condition is used to detect when every run should leave the added state. As quitting character set, the TRANSLATE function returns $\bot$ in this case, which is always safe.

The termination of the algorithm for every possible LTL formula follows from the fact that the algorithm only recurses on disjunctions of sub-formulas that are present in the original LTL specification and it always recurses into strict subformulas. Note that this observation also shows that the computed automata have a number of states that is at most exponential in the length of the LTL formula. Cichon et al. [9] showed that the smallest non-deterministic Büchi automata for LTL formulas of the shape $\bigwedge_{1 \leq i \leq n} \mathsf{F} p_1$ need a number of states that is exponential in n in general. Since the negation of these LTL formulas are accepted by the grammar given above, it follows that an exponential blow-up for translating LTL formulas in our grammar to UVWs is unavoidable (as every UVW for a specification is also a non-deterministic Büchi word automaton for the complement language).

After constructing a UVW with the procedure from Algorithm 1, it makes sense to minimize it. Unlike in the general Büchi automaton case [10], in UVWs it is always sound to merge states with the same language. The only case in which this would be unsound is if both states lie in the same strongly connecting component, which cannot happen in UVWs. When merging UVW states, we can simply reroute all transitions to a higher-ranked state to the lower-ranked states (for some arbitrary valid ranking function). In addition we merge states that are reachable using the same prefix words, and if for some pair of states q_1 and q_2, we have that q_1 has a language that is a subset of the language of q_2, but whenever q_2 is reached for some prefix trace, so is q_1, we remove q_2 (if q_1 and q_2 are not reachable from each other). For simplicity, we approximate language inclusion by fair simulation [10].

4 Discussion

Before looking into how UVWs can simplify the debugging process of models in the next section, we want to discuss the merits and drawbacks of the grammar and construction given in the preceding section.

The grammar that we defined in the preceding section does not support the use of the $\wedge$ operator for the nonterminal ψ. This is a necessity. For example, the property $\phi = a\,\mathcal{U}\,(b \wedge (c\,\mathcal{U}\,d))$ cannot be represented as a UVW. When building a state in which the UVW waits for $(b \wedge (c\,\mathcal{U}\,d))$ to hold and checks for a to hold along the way, we cannot predict when the state should be left. If the character $\{a, b, c\}$ occurs, then the next character could be $\{a\}$ (so that a UVW run has to stay in the state), but the next character could also be $\{c\}$ (and then we would have just observed a good prefix for the LTL formula). We verified that indeed no UVW for this LTL formula exists by using the tool `ltl2dstar` to translate it to a single-pair deterministic Rabin automaton, and then applying the test from [14] (implemented as part of the `bassist` reactive synthesis tool [7]).

The UVWs computed by the construction from the previous section can be labeled by temporal logic formulas that they represent. For example, Figure 1 shows a UVW for the LTL property $\psi = \mathsf{G}((a \rightarrow b)\mathcal{U}\,c) \wedge \mathsf{GF}(d\,\mathcal{U}\,e)$ that we computed with our approach. The states can be labeled by

- $q_0 \equiv \psi$,
- $q_1 \equiv \mathbf{true}$,
- $q_2 \equiv (a \rightarrow b)\,\mathcal{U}\,c$,
- $q_3 \equiv \mathsf{F}(d\,\mathcal{U}\,e)$,

which explains how the individual states contribute to the encoding of the LTL property. Our implementation of Algorithm 1 computes such a labelling automatically by keeping track of for which subformula a sub-UVW was computed. The later automaton minimization steps do not lead to a loss of this information, and since in UVWs, two states that represent the same language can always be merged, there is always only one state for each subformula, which makes them easy to understand. This is not the case for (non-deterministic) Büchi automata. Figure 3 shows an example nondeterministic Büchi automaton with two states

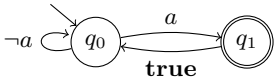

Fig. 3. A (minimally-sized) nondeterministic Büchi automaton for the language $\mathsf{GF}a$. All states represent the same language.

that represent the same language. In fact, all Büchi automata that encode the same LTL formula have this property.

5 Case Studies and Experiments

5.1 LTL to UVW Translation

We implemented the translation from LTL to UVWs in Python. All experiments reported in the following were conducted on a computer with an Intel Core i5-7200U CPU and 16 GB of memory while using `spin` version 6.4.7 and `spot` [15] version 2.4.1 under the Ubuntu 16.04 LTS operating system. From the formal verification framework `spot`, we only use the `ltl2tgba` [15] tool for translating LTL properties to (non-deterministic) Büchi automata. In many cases the automata computed by our construction and by `spot` (for the negation of the respective specification) are very similar, but our construction always guarantees that the output is a very weak automaton. For example, `spot` does not translate the negation of the LTL property $\psi = \mathsf{G}(a \vee \mathsf{X}b)$ to a very weak automaton, even though there exists an equivalent UVW for ψ.

As a first experiment, we tested how many of the properties that Blahoudek et al. [3] compiled for a study are accepted by the grammar that we define in this paper. Out of the 134 unique properties, 77 can be translated to UVWs, as we found out using the construction from [14]. Of these, 74 are accepted by our grammar, and their translation to UVWs took 257 milliseconds of computation time in total. Out of the remaining three properties, one is equivalent to **true** and the other two differ only in the names of the atomic propositions.

5.2 Case Study

The *General Inter-Orb Protocol (GIOP)* is a key component in the Common Object Request Broker Architecture (CORBA). Kamel and Leue [16] gave a model and specifications for this protocol. One of the specifications that they give for this model is quite convoluted, and we chose it as main benchmark, as it can be translated to a non-trivial UVW. The property is as follows:

$$\psi = \mathsf{G}(\mathsf{F}r \to (\mathsf{G}((s \wedge \mathsf{F}r) \to (\overline{r}\,\mathcal{U}\,p)) \\ \wedge \mathsf{G}((s \wedge \mathsf{F}r) \to ((\overline{p} \wedge \overline{r})\,\mathcal{U}\,(r \vee ((p \wedge \overline{r})\,\mathcal{U}\,(r \vee (\overline{p}\,\mathcal{U}\,r)))))))$$

It is neither a pure safety property, nor a pure liveness property. Proposition s represents that a user sends a request, p represents that a server processes a

request, and r represents that a user receives a reply. Intuitively, the formula states that if a user sends a request and eventually a reply message is received, that particular request was served exactly once in case of successful processing by the server or at most once in case of unsuccessful processing. So in any case, the same request should not be served and processed twice by the server.

The original model by Kamel and Leue is too large to model check it against the specification with `spin` and 16 GB of memory. To demonstrate how the simple shape of UVWs helps with understanding counter-example traces, we injected an error into the model, so that the model checker `spin` can compute a counter-example trace within the memory limit.

We use `spin`'s exhaustive verification algorithm. The `ltl2tgba` tool of `spot` translates the (negation of the) LTL specification above to a Büchi automaton comprising of 6 states (which happens to be very weak). When trying to verify the GIOP model with this automaton as specification, `spin` generates an error trace of length 526 in 3.4 seconds using 893 Mbytes of memory. The error trace is quite long and hence hard to inspect. While the trace involves only few state changes in the specification automaton, due to the absence of a labelling of the states with the LTL properties that they represent, interpreting the trace is difficult.

The same experiment when executed with a UVW constructed with the algorithm presented in this paper leads to an error trace of length 524 in 1.71 seconds using a total memory of 510 Mbytes. Figure 4 shows the full UVW computed for the LTL property given above. It can be decomposed into 6 simple chains, which are highlighted by different colors. The smallest chain comprises of just two states, whereas the longest one has five states. When running `spin` for all simple chains (and the model) separately, we first of all observe that spin finds counter-example traces for all chains except for the chains along $q_0 \to T$ and $q_0 \to q_2 \to T$, for which the verification process ran out of memory (in 59.9 and 77 seconds, respectively). Out of remaining four, for two chains a trace of length 526 was computed by `spin` in 3.69 and 4.0 seconds. For the other two, traces of length 455 were computed in 5.24 and 2.16 seconds, respectively.

We analyze one of the traces of length 455, as they are shorter and hence easier to understand. We show the values of the variables s, r, and p in the characters of the counter-example trace in Figure 5 along with the UVW chain. The UVW states are labeled by the following LTL formulas:

- $q_0 \equiv \psi$
- $q_4 \equiv \mathsf{G}\neg r \vee ((p \wedge \neg r)\mathcal{U}(r \vee (\neg p \,\mathcal{U}\, r)))$
- $q_5 \equiv \mathsf{G}\neg r \vee (\neg p \,\mathcal{U}\, r)$
- $q_2 \equiv \mathsf{G}\neg r$

Only those characters that lead to a state change in the UVW chain are shown. Restricting our attention to these characters gives us a summary of the error trace. The labelling shows that from state q_4, the trace character $s\overline{pr}$ leads to the second disjunct of q_4 to only be satisfiable if $\neg p \,\mathcal{U}\, r$ holds in the future. The following two highlighted characters then successively lead to the violation of every disjunct of the remaining obligation. We can also see that the s variable has a **true** value in all cases, which implies that user requests are sent more

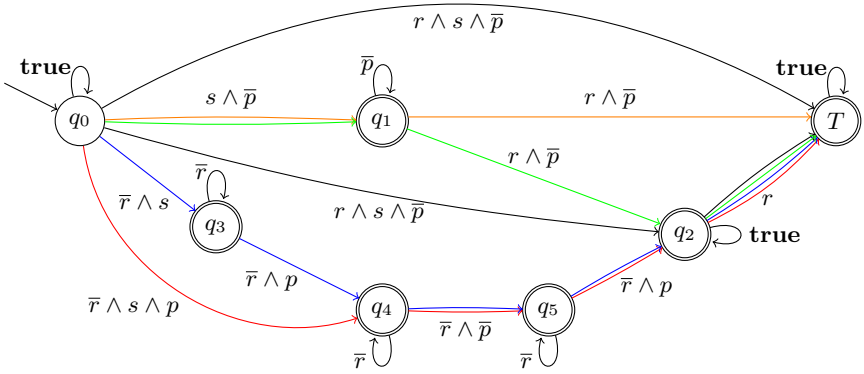

Fig. 4. UVW computed from our construction for the first case study. Each decomposed chain is highlighted with different color coding.

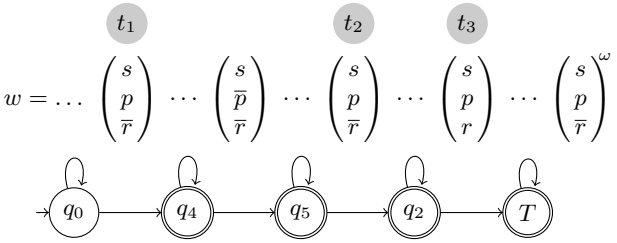

Fig. 5. Error trace analysis with a single chain of the UVW decomposition.

than once, or that the sending process does not leave the "just sent" state along the trace. After a request is sent in character t_1, it is processed twice in t_2 and t_3, which is the cause for the violation – an absorbing rejecting state is reached immediately afterwards. With this analysis of the cause of the error, we could now further inspect the trace to find the parts of the execution leading towards the double processing of the request.

6 Conclusion

We defined a context-free grammar for a subset of LTL and a translation from specifications accepted by this grammar to universal very weak automata. The key technical contribution was the definition of *quitting points* for LTL properties, which we exploited to give a grammar that covers the vast majority of the properties that are translatable to UVWs from an LTL property database compiled by Blahoudek et al. [3]. Furthermore, our grammar contains all possible nestings of the LTL *Until* operator. All states in the UVWs computed by our construction are automatically labeled by LTL formulas that they represent, and even when applying classical simulation-based state reduction techniques, this information is not lost. We demonstrated using a short case study how

the favourable properties of UVWs can be used to simplify a model debugging process. For space reasons, more thorough experiments are left for future work.

We believe that UVWs are also a useful automaton model for many other applications in the domain of formal methods. For instance, the translation presented in this paper is useful for *reactive synthesis*, where very large specifications need to be processed. Using UVWs to represent the specifications enables the use of *anti-chains* [17] as data structure for solving synthesis games without the introduction of counters that are normally used in *bounded synthesis* [18] for full LTL, which has the potential to substantially improve synthesis times.

References

1. Pnueli, A.: The temporal logic of programs. In: FOCS 1977, Proceedings. (1977) 46–57
2. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
3. Blahoudek, F., Duret-Lutz, A., Kretínský, M., Strejcek, J.: Is there a best Büchi automaton for explicit model checking? In: SPIN Symposium. (2014) 68–76
4. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.J.: Explaining counterexamples using causality. Formal Methods in System Design **40**(1) (2012) 20–40
5. Basin, D., Bhatt, B.N., Traytel, D.: Optimal proofs for linear temporal logic on lasso words. In: 16th International Symposium on Automated Technology for Verification and Analysis (ATVA 2018). (2018)
6. Maidl, M.: The common fragment of CTL and LTL. In: FOCS 2000, Proceedings. (2000) 643–652
7. Ehlers, R.: ACTL ∩ LTL synthesis. In: CAV 2012, Proceedings. (2012) 39–54
8. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
9. Cichon, J., Czubak, A., Jasinski, A.: Minimal Büchi automata for certain classes of LTL formulas. In: Fourth International Conference on Dependability of Computer Systems, (DepCos-RELCOMEX). (2009) 17–24
10. Gurumurthy, S., Bloem, R., Somenzi, F.: Fair simulation minimization. In: Computer Aided Verification, 14th International Conference, CAV. (2002) 610–624
11. Vardi, M.Y.: Nontraditional applications of automata theory. In: TACS, Proceedings. (1994) 575–597
12. Gerth, R., Peled, D.A., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification, Testing and Verification XV. (1995) 3–18
13. Ehlers, R.: Short witnesses and accepting lassos in ω-automata. In: LATA, Proceedings. (2010) 261–272
14. Bojańczyk, M.: The common fragment of ACTL and LTL. In: FOSSACS 2008, Proceedings. (2008) 172–185
15. Duret-Lutz, A.: LTL translation improvements in Spot 1.0. International Journal on Critical Computer-Based Systems **5**(1/2) (March 2014) 31–54
16. Kamel, M., Leue, S.: Validation of a remote object invocation and object migration in CORBA GIOP using Promela/Spin. In: International SPIN Workshop. (1998)
17. Filiot, E., Jin, N., Raskin, J.: Antichains and compositional algorithms for LTL synthesis. Formal Methods in System Design **39**(3) (2011) 261–296
18. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT **15**(5-6) (2013) 519–539