Impartiality and Anticipation for Monitoring of Visibly Context-free Properties

Normann Decker, Martin Leucker, and Daniel Thoma

Institute for Software Engineering and Programming Languages
Universität zu Lübeck, Germany
{decker, leucker, thoma}@isp.uni-luebeck.de

Abstract. We study monitoring of visibly context-free properties. These properties reflect the common concept of nesting which arises naturally in software systems. They can be expressed e.g. in the temporal logic CaRet which extends LTL by means of matching calls and returns. The future fragment of CaRet enables us to give a direct unfolding-based automaton construction, similar to LTL. We provide a four-valued, impartial semantics on finite words which is particularly suitable for monitoring. This allows us to synthesize monitors in terms of deterministic push-down Mealy machines. To go beyond impartiality, we develop a construction for anticipatory monitors from visibly push-down ω-automata by utilizing a decision procedure for emptiness.

1 Introduction

In Runtime Verification (RV) an actual execution of a system is checked with respect to a given correctness property [1]. Therefore, typically a so-called monitor is synthesized from the high-level specification of the correctness property, which yields an *assessment* or a *verdict*, denoting to which extent the property is satisfied by the current execution.

RV is a verification technique that is becoming more and more popular in recent years but is also a key ingredient in new programming paradigms such as *monitor-oriented programming* [2] or software architectures for reliable systems such as *runtime reflection* [1].

In runtime verification, one always faces a finite execution of a potentially infinite run of a system. Such an execution may be completed, and for example, completely stored in a log file and subsequently checked with respect to some property, or it may be checked on-line while it is continuously evolving. Depending on the application, different notions of correctness assessments are appropriate and monitors evaluating an execution and a property accordingly are needed.

As explained in [3] and [4], a two-valued assessment yielding *yes/true* or *no/false* seems appropriate when faced with completed executions as either a property is satisfied or not.

When checking an execution on-line, at least three different assessments (*true*/*false*/*inconclusive*) are needed to adhere to the maxim of *impartiality*.

This states that a property should only be evaluated to true or false, if any continuation of the execution will yield the same verdict. This ensures that runtime verification is not stopped prematurely with the (misleading) understanding that a property is violated or fulfilled although subsequent observations may yield a different verdict.

The inconclusive verdict can be refined further to a verdict of *presumably true* and *presumably false*. *Presumably true* expresses the fact that no violation has been seen but one might still occur in the future as the observation might be extended. *Presumably false* describes that some obligation is not satisfied but might still be fulfilled in the future. These verdicts are of particular interest when a system terminates as they still allow for some assessment where an inconclusive verdict would have provided no information at all.

The maxim of impartiality can trivially be fulfilled with a monitor always yielding the verdict *inconclusive*. The maxim of *anticipation* on the other hand states that a verdict of true or false should be evaluated as soon as this is possible, meaning for example for a violated safety property that the violation should be reported by a monitor for the shortest execution of a run (i.e. the shorted prefix of the run) violating the property.

The methods for checking properties of executions can broadly be divided into rewriting-based and automata-based approaches. As described in [4], the latter can sometimes be seen as pre-computations of rewriting-based approaches, highlighting that rewriting can be understood as on-the-fly automata constructions. Thus, typically, rewriting-based approaches are easier to implement, may have a better memory performance but may have a worse runtime performance. Moreover, anticipatory approaches to runtime verification need a complex check in each verification step which can be done more efficiently using pre-computations with automata.

A prominent specification formalism for denoting properties to check is Linear-time Temporal Logic (LTL) [5], which allows to specify star-free regular properties. A bunch of different approaches for checking LTL properties at runtime have been proposed. These can be categorized into two-valued rewriting-based approaches [6–8], impartial three and four-valued rewriting and automata-based approaches [9, 4], or impartial and anticipatory automata-based approaches [10, 11]. The latter approach was then generalized to arbitrary linear-time temporal logics which come with an automaton-based abstraction for a satisfiability check in [12].

For practical applications, plain LTL specifications are typically not enough. Besides enrichments like dealing with data or real-time aspects, one of the important goals is to specify context-free properties, as, in software systems, nesting structures arise naturally, in particular in the context of recursive programs with calls and returns. State-full protocols impose nested structures on message sequences. For example, a transaction protocol requires (recursively) any sub-transactions of some transaction to finish before its completion. Similar properties arise in nested document formats such as XML or serialization of nested data structures.

This common concept of nesting is reflected in the class of visibly context-free languages. Alur et al. proposed in [13] visibly pushdown automata as an automaton characterization of visibly context-free languages. The nature of this automaton model is that the stack action is determined by the input symbol. This is analogous to calls and returns in recursive programs. In contrast, a push-down property that is not visibly, is the language $a^n b a^n$ where a stack is needed rather for counting than for recognizing a nesting structure.

In the context of temporal specifications, the logic CaRet is a natural extension of LTL with the ability to express nesting [14, 15]. The concept of a direct temporal successor is extended to the concept of a so-called abstract successor. That is, the successor on the same level between a call and its matching return. CaRet, however does not cover the full class of visibly context-free properties. Logics with full expressiveness regarding visibly context-free languages are, for example VP-μTL [16] and MSO$_\mu$ [13].

Monitor synthesis for CaRet was first considered in [17]. More specifically, a monitoring approach for a version of CaRet is provided that allows for checking globally a past-time property, i.e. safety properties [18]. According to our taxonomy, the approach is rewriting-based. Due to the additional stack that has to be kept in this setting, a translation to an impartial automaton approach is not straightforward. Note that for CaRet the general scheme developed in [12] is not applicable.

In this paper, we study monitoring of visibly context-free properties. The future fragment of CaRet allows, similar to LTL a direct unfolding-based automaton construction. We provide a four-valued, impartial semantics on finite words in Section 3 which is particularly suitable for monitoring. It allows us to synthesize monitors in terms of deterministic push-down Mealy machines.

Additionally, we study an anticipatory approach to monitoring of visibly context-free properties in Section 4. We achieve to construct anticipatory monitors from visibly push-down ω-automata by utilizing a decision procedure for emptiness. Thus, this allows us to monitor properties expressed e.g. in full CaRet or VP-μTL. As such, we provide a complete picture of monitoring context-free properties in the taxonomy introduced in [4] and explained at the beginning of this paper.

2 Preliminaries

Alphabets and Words. Let AP be a finite set of atomic propositions and $\Sigma = 2^{\mathrm{AP}}$ a finite alphabet. We assume Σ to be the disjoint union of *call* symbols Σ_{c}, *return* symbols Σ_{r} and *internal* symbols Σ_{int}. Furthermore, call, int and ret denote propositional formulae characterizing exactly the call, internal and return symbols, respectively. A word over Σ is a possibly infinite sequence $w = w_0 w_1 w_2 \dots$ s.t. $w_i \in \Sigma$. We denote by $w^{(i)} = w_i w_{i+1} \dots$ the suffix starting at position i and, if $w \in \Sigma^n$, by $|w| = n$ its length. Let Σ^* and Σ^ω denote the sets finite and infinite words over Σ, respectively, and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$, $\Sigma^+ := \Sigma^* \setminus \{\epsilon\}$.

We denote $\mathbb{B}_4 = \{\top, \top^{\mathsf{p}}, \bot^{\mathsf{p}}, \bot\}$ the four-valued De-Morgan lattice with linear order, i.e. the lattice with $\top \sqsupseteq \top^{\mathsf{p}} \sqsupseteq \bot^{\mathsf{p}} \sqsupseteq \bot$, $\top = \neg\bot$ and $\top^{\mathsf{p}} = \neg\bot^{\mathsf{p}}$. Note, that we assume big operators to have lower precedence than small ones, thus $\bigsqcup a \sqcap b = \bigsqcup(a \sqcap b)$.

Visibly push-down automata. A (non-deterministic) *push-down automaton* is a tuple $\mathcal{F} = (Q, \Sigma, \Gamma, \delta, Q_0, F)$ where

- Σ, Γ are the finite *input* and *stack alphabet*, respectively, and $\Gamma_\# := \Gamma \dot\cup \{\#\}$ the stack alphabet enriched by a new bottom symbol $\# \notin \Gamma$,
- Q is a finite set of control *states*,
- $Q_0 \subseteq Q$ is the set of *initial states*,
- $F \subseteq Q$ is the set of *accepting states* and
- $\delta : Q \times \Gamma_\# \times \Sigma \to 2^{Q \times \Gamma_\#^{\leq 2}}$ is the non-deterministic, *transition function.*

A *configuration* of $\mathcal{F}$ is a tuple $(q, s) \in Q \times (\Gamma^*\{\#\})$ comprising the current control state and a stack assignment ending with #. A *run* of $\mathcal{F}$ on a *finite* input word $w = w_0 w_1 \dots w_n \in \Sigma^*$ is a sequence $c_0 c_1 \dots c_{n+1}$ of configurations $c_i \in Q \times (\Gamma^*\{\#\})$ s.t.

- $c_0 \in Q_0 \times \{\#\}$ and
- if $c_{i+1} = (q', \gamma'\gamma'' s)$ then $c_i = (q, \gamma s)$ with $(q', \gamma'\gamma'') \in \delta(q, \gamma, w_i)$,

where $q \in Q$,$\gamma \in \Gamma_\#$. A run $(q_0, s_0)(q_1, s_1)\dots(q_{n+1}, s_{n+1})$ is accepting if $q_{n+1} \in F$.

We call a push-down automaton $\mathcal{P}$ reading *infinite* words a *push-down ω-automaton.* A run of $\mathcal{P}$ on an infinite word $u = u_0 u_1 \dots \in \Sigma^\omega$ is an infinite sequence of configurations $c_0 c_1 \dots$ defined as above. A run $(q_0, s_0)(q_1, s_1)\dots$ is accepting if the sequence of states $q_0 q_1 \dots$ satisfies a Büchi condition, i.e. there is some $q \in F$ that occurs infinitely often in the sequence.

A push-down (ω-)automaton accepts a word $w \in \Sigma^\infty$ if there is an accepting run on w. By $\mathcal{L}(\mathcal{P}) \subseteq \Sigma^\omega$ and $\mathcal{L}(\mathcal{F}) \subseteq \Sigma^*$ we denote the set of words accepted by $\mathcal{P}$ and $\mathcal{F}$, respectively.

$\mathcal{F}$ and $\mathcal{P}$ are called a *visibly push-down automaton* (VPA) and a *visibly push-down ω-automaton* (ω-VPA), respectively, if the input alphabet Σ is the union of three disjoint alphabets Σ_{c}, Σ_{r}, Σ_{int} and for $(q', u) \in \delta(q, \gamma, a)$

- $u = \epsilon$ iff $a \in \Sigma_{\mathsf{r}}$ and $\gamma \neq \#$,
- $u = \#$ iff $a \in \Sigma_{\mathsf{r}}$ and $\gamma = \#$,
- $u = \gamma$ iff $a \in \Sigma_{\mathsf{int}}$ and
- $u \in (\Gamma\{\gamma\})$ iff $a \in \Sigma_{\mathsf{c}}$.

Emptiness of Push-down automata. Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta_{\mathcal{P}}, Q_0, F)$ be a push-down ω-automaton. Following [19], we can represent the set configurations of $\mathcal{P}$, from which all inputs are rejected (empty configurations), by means of a *multi automaton* $\mathcal{A} = (S \dot\cup Q, \Gamma, \delta, Q, A)$. $S \dot\cup Q$ is the *state space* where S is a finite set and disjoint from the states Q of $\mathcal{P}$, which are the *initial* states of $\mathcal{A}$. The stack alphabet Γ of $\mathcal{P}$ is the *input alphabet* of $\mathcal{A}$ and $A \subseteq S$ are the *accepting states.* $\delta : S \dot\cup Q \times \Gamma \to 2^{S \dot\cup Q}$ is the *transition function.* The multi-automaton $\mathcal{A}$ accepts configurations $(q, s\#)$ of $\mathcal{P}$ by behaving like a finite automaton with initial state q and reading the stack configuration $s \in \Gamma^*$ as input.

Note, that in [19] the state space of $\mathcal{A}$ and $\mathcal{P}$ are disjoint but each initial state s of $\mathcal{A}$ corresponds (bijectively) to some $q \in Q$. We therefore identify those. Further, we can assume $\mathcal{A}$ to have a deterministic transition function δ since $\mathcal{A}$ is basically a compact representation of a set of finite automata.

3 Four-valued Semantics of CaRet$^+$ on Finite Traces and Impartial Monitoring

In this section, we consider the logic CaRet as a specification formalism for nesting structures. For its future fragment CaRet$^+$, we provide a four-valued, impartial semantics on finite words. We show how to construct a push-down Mealy machine as monitor that incrementally reads input symbols and outputs the semantics of the observed trace. Our aim is to give an easily implementable monitor construction for properties expressing nesting structures.

3.1 Four-valued CaRet$^+$

The syntax of CaRet$^+$ formulae is defined by the following grammar.

$$\begin{aligned}\varphi ::= \; & p \mid \varphi \wedge \varphi \mid \mathrm{X}\,\varphi \mid \mathrm{X}^{\mathrm{a}}\,\varphi \mid \varphi\,\mathrm{U}\,\varphi \mid \varphi\,\mathrm{U}^{\mathrm{a}}\,\varphi \mid \\ & \neg p \mid \varphi \vee \varphi \mid \overline{\mathrm{X}}\,\varphi \mid \overline{\mathrm{X}^{\mathrm{a}}}\,\varphi \mid \varphi\,\mathrm{R}\,\varphi \mid \varphi\,\mathrm{R}^{\mathrm{a}}\,\varphi\end{aligned}$$

The idea of how CaRet extends the operators known form LTL is as follows: Consider for example a program procedure. While the direct successor of a line might be the first line in a called procedure, the abstract successor jumps directly to the next line in the current procedure and omits to enter any called procedures. Moreover, the last line in the procedure has a direct successor, namely the return position in its caller, but no abstract successor. CaRet uses the abstract next modality X^{a} to specify a property at the next abstract position. Further, in general, it contains the abstract past modality $\mathrm{X}^{\mathrm{a}-}$ for specifying a property at the call position of the current procedure. Intuitively, consecutive application of $\mathrm{X}^{\mathrm{a}-}$ walks up the call stack. Also, CaRet provides abstract versions of the common until and since operators. However, for sake of simplicity, we do not support the past operators in this section. It shall be noted, that in contrast to LTL, past modalities add expressiveness to the logic.

For a formula Φ, we denote $\mathsf{sub}(\Phi)$ the set of sub-formulae including unfoldings, e.g. $\psi \vee \varphi \wedge \mathrm{X}(\varphi\,\mathrm{U}\,\psi)$ for a sub-formula $\varphi\,\mathrm{U}\,\psi$ of Φ.

Semantics on finite traces. The semantics of CaRet is defined on infinite traces. Since monitoring inherently deals with finite traces we provide the impartial finitary semantics FCaRet_4. It is intended to intuitively resemble the infinite trace semantics, similar to finitary semantics for LTL formulae, e.g. FLTL and FLTL_4 [4].

As the latter, FCaRet_4 uses the four truth values *true* ($\top$), *false* ($\bot$), *presumably true* ($\top^{\mathsf{p}}$) and *presumably false* ($\bot^{\mathsf{p}}$), allowing for impartiality.

The distinguishing aspect between finitary and infinitary semantics is the need for handling the end of a trace which is reflected by discriminating weak and strong operators. A formula $\mathrm{X}\,\varphi$ describes what should happen at the next time step. If the trace ends here, it needs to be specified if that is desired or not. The temporal operators X, X^{a}, U and U^{a} are considered as strong operators having an existential character. They require the next position to exist and evaluate to $\bot^{\mathsf{p}}$ if not. Consequently, their duals $\overline{\mathrm{X}}$, $\overline{\mathrm{X}^{\mathrm{a}}}$, R and R^{a} have a weak, i.e. universal character; they impose restrictions only on actually existing positions. If there is no successive position, they evaluate to $\top^{\mathsf{p}}$. Note that the next operators and their (weak) duals therefore do not coincide on finite traces as they do on infinite ones.

Abstract successors. For the semantics of the abstract temporal modalities, we use the notion of abstract steps in terms of the abstract successor function $\mathsf{succ}^{\mathsf{a}}$ as in the infinitary CaRet semantics. We define the partial function $\mathsf{succ}^{\mathsf{a}} : \Sigma^{\infty} \rightharpoonup \mathbb{N}_0$ mapping a word to the *abstract successor* of its first position. For any $w \in \Sigma^{\infty}$, $\mathsf{succ}^{\mathsf{a}}(w) = 1$ if $|w| \geq 2$, $w_0 \notin \Sigma_{\mathsf{c}}$ is not a call and $w_1 \notin \Sigma_{\mathsf{r}}$ is not a return. If w starts with a call, then the abstract successor is its matching return, if it exists: $\mathsf{succ}^{\mathsf{a}}(w) = i$ if $w_0 \in \Sigma_{\mathsf{c}}$ and $i \in \mathbb{N}_0$ is the smallest number s.t. $w_i \in \Sigma_{\mathsf{r}}$ and in $w_1 \dots w_i$ the number of positions j with $w_j \in \Sigma_{\mathsf{r}}$ is greater than the number of positions j' with $w_{j'} \in \Sigma_{\mathsf{c}}$. $\mathsf{succ}^{\mathsf{a}}(w)$ is undefined in all other cases. Further, we let $\mathsf{succ}^{\mathsf{a}*}(w)$ denote the set of positions $\{i_1, i_2, \dots\}$ on a word $w \in \Sigma^{\infty}$ s.t. $i_1 = 0$ and $i_{j+1} = \mathsf{succ}^{\mathsf{a}}(w^{(i_j)})$. Additionally, we define a predicate $\mathsf{complete}(w)$ that is true if $\mathsf{succ}^{\mathsf{a}*}(w)$ has a maximal element $i < |w| - 1$ and $w_{i+1} \in \Sigma_{\mathsf{r}}$ is a return, otherwise $\mathsf{complete}(w)$ is false. That is, $\mathsf{complete}(w)$ is true if a return in w is not matched and thus the abstract sequence formed by the positions in $\mathsf{succ}^{\mathsf{a}*}(w)$ terminates because of the first unmatched return.

We define the semantics in conformance with FLTL_4 as defined in [4]. The semantics of a formula is given in terms of a function that maps words $w \in \Sigma = 2^{\mathrm{AP}}$ to the $\mathbb{B}_4$-lattice. For propositions and boolean connectives, the two-valued semantics can be directly lifted to the $\mathbb{B}_4$-lattice.

$$[\![p]\!]_4(w) = \begin{cases} \top & \text{if } p \in w_0 \\ \bot & \text{if } p \notin w_0 \end{cases} \qquad [\![\neg p]\!]_4(w) = \begin{cases} \top & \text{if } p \notin w_0 \\ \bot & \text{if } p \in w_0 \end{cases}$$

$$[\![\varphi \wedge \psi]\!]_4(w) = [\![\varphi]\!]_4(w) \sqcap [\![\psi]\!]_4(w) \qquad [\![\varphi \vee \psi]\!]_4(w) = [\![\varphi]\!]_4(w) \sqcup [\![\psi]\!]_4(w)$$

For the (direct) strong and weak next operators the semantics is defined as discussed above.

$$[\![\mathrm{X}\,\varphi]\!]_4(w) = \begin{cases} [\![\varphi]\!]_4(w^{(1)}) & \text{if } |w| > 1 \\ \bot^{\mathsf{p}} & \text{otherwise} \end{cases} \qquad [\![\overline{\mathrm{X}}\,\varphi]\!]_4(w) = \begin{cases} [\![\varphi]\!]_4(w^{(1)}) & \text{if } |w| > 1 \\ \top^{\mathsf{p}} & \text{otherwise} \end{cases}$$

The standard semantics of the U and R operator is lifted to $\mathbb{B}_4$. The right parts of the definitions deal with the end of words.

$$[\![\varphi \,\mathrm{U}\, \psi]\!]_4(w) = \left(\bigsqcup_{i<|w|} [\![\psi]\!]_4(w^{(i)}) \sqcap \bigsqcap_{j<i} [\![\varphi]\!]_4(w^{(j)}) \right) \sqcup \left(\bot^{\mathsf{p}} \sqcap \bigsqcap_{i<|w|} [\![\varphi]\!]_4(w^{(i)}) \right)$$

$$[\![\varphi \,\mathrm{R}\, \psi]\!]_4(w) = \left(\bigsqcup_{i<|w|} [\![\varphi]\!]_4(w^{(i)}) \sqcap \bigsqcap_{j\leq i} [\![\psi]\!]_4(w^{(j)}) \right) \sqcup \left(\top^{\mathsf{p}} \sqcap \bigsqcap_{i<|w|} [\![\psi]\!]_4(w^{(i)}) \right)$$

The abstract next operator can be defined in a similar manner as their direct counter parts using the abstract successor $\mathsf{succ}^{\mathsf{a}}$ instead of the direct successor.

While the the two-valued semantics FLTL considers observations as terminated, the four-valued semantics FLTL_4 reflects the intuition, that a finite observation might still be continued and therefore next operators X and $\overline{\mathrm{X}}$ evaluate to $\bot^{\mathsf{p}}$ and $\top^{\mathsf{p}}$, respectively, at the end of a word. For the end of an abstract sequence, i.e. when there is no abstract successor, both cases are possible. When observing an (unmatched) return symbol as the direct successor, the current "procedure" definitely returns and there is no continuation. The abstract next operators shall then give a definite verdict, i.e. $\top$ or $\bot$. On the other hand, if the abstract sequence ends because the whole observation ends before the next abstract successor, there might be a continuation and hence the evaluation is preliminary, i.e. $\top^{\mathsf{p}}$ or $\bot^{\mathsf{p}}$.

$$[\![\mathrm{X}^{\mathrm{a}}\, \varphi]\!]_4(w) = \begin{cases} [\![\varphi]\!]_4(w^{(n)}) & \text{if } \mathsf{succ}^{\mathsf{a}}(w) = n \in \mathbb{N} \\ \bot & \text{if } \mathsf{succ}^{\mathsf{a}}(w) \text{ is undef. } \wedge\, w_1 \in \Sigma_{\mathsf{r}} \\ \bot^{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$[\![\overline{\mathrm{X}^{\mathrm{a}}}\, \varphi]\!]_4(w) = \begin{cases} [\![\varphi]\!]_4(w^{(n)}) & \text{if } \mathsf{succ}^{\mathsf{a}}(w) = n \in \mathbb{N} \\ \top & \text{if } \mathsf{succ}^{\mathsf{a}}(w) \text{ is undef. } \wedge\, w_1 \in \Sigma_{\mathsf{r}} \\ \top^{\mathsf{p}} & \text{otherwise} \end{cases}$$

Note that the semantics of X^{a} is slightly different from the one in [14] to fit together with the U^{a} operator.

Based on the same idea, the abstract until and release operators are defined as follows.

$$[\![\varphi \,\mathrm{U}^{\mathrm{a}}\, \psi]\!]_4(w) = \begin{cases} \left(\bigsqcup_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\psi]\!]_4(w^{(i)}) \;\sqcap \prod_{\substack{j \in \mathsf{succ}^{\mathsf{a}*}(w) \\ j<i}} [\![\varphi]\!]_4(w^{(j)}) \right) & \text{if } \mathsf{complete}(w) \\ \left(\bigsqcup_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\psi]\!]_4(w^{(i)}) \;\sqcap \prod_{\substack{j \in \mathsf{succ}^{\mathsf{a}*}(w) \\ j<i}} [\![\varphi]\!]_4(w^{(j)}) \right) & \\ \qquad \sqcup \quad \left(\bot^{\mathsf{p}} \sqcap \prod_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\varphi]\!]_4(w^{(i)}) \right) & \text{otherwise} \end{cases}$$

$$[\![\varphi \,\mathrm{R}^{\mathrm{a}}\, \psi]\!]_4(w) = \begin{cases} \left(\bigsqcup_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\varphi]\!]_4(w^{(i)}) \;\sqcap \prod_{\substack{j \in \mathsf{succ}^{\mathsf{a}*}(w) \\ j\leq i}} [\![\psi]\!]_4(w^{(j)}) \right) & \text{if } \mathsf{complete}(w) \\ \left(\bigsqcup_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\varphi]\!]_4(w^{(i)}) \;\sqcap \prod_{\substack{j \in \mathsf{succ}^{\mathsf{a}*}(w) \\ j\leq i}} [\![\psi]\!]_4(w^{(j)}) \right) & \\ \qquad \sqcup \quad \left(\top^{\mathsf{p}} \sqcap \prod_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\psi]\!]_4(w^{(i)}) \right) & \text{otherwise} \end{cases}$$

In contrast to the until and release operators two cases have to be distinguished for their abstract counterparts. If the sequence of abstract successors for a word is complete, i.e. if the sequence terminates because of an umatched return, the operators cannot evaluate to $\bot^{\mathsf{p}}$ or $\top^{\mathsf{p}}$, respectively.

3.2 Visibly Push-down Mealy Machines (VPMM)

A typical approach to monitoring temporal properties is based on formula rewriting. When observing a symbol, the formula is evaluated and additionally rewritten to maintain the gained information. This requires equations for transforming any formula into a formula where each temporal operator is an X operator or is guarded by some X. Then, every sub-formula can explicitly be evaluated when reading only one new letter. For the U operator, the unfolding equation is standard and the abstract operator can be unfolded analogously:

$$\varphi \,\mathrm{U}\, \psi \equiv \psi \vee (\varphi \wedge \mathrm{X}(\varphi \,\mathrm{U}\, \psi))$$
$$\varphi \,\mathrm{U}^{\mathrm{a}}\, \psi \equiv \psi \vee (\varphi \wedge \mathrm{X}^{\mathrm{a}}(\varphi \,\mathrm{U}^{\mathrm{a}}\, \psi))$$

What remains is an unfolding of the abstract next operator X^{a}. According to the semantics, reading a return or an internal symbol it behaves like the

classical X operator, accept that there must not follow a return symbol in the next step. Evaluating a formula $\mathrm{X}^{\mathrm{a}}\,\varphi$ for a call symbol, the evaluation of φ needs to be postponed until the matching return symbol is read. If a return follows immediately it is matching. Otherwise the matching return can be reached by following the abstract sequence of positions until the first unmatched return.

$$\begin{aligned}\mathrm{X}^{\mathrm{a}}(\varphi) \equiv \quad & (\ \neg\mathsf{call} \wedge \mathrm{X}(\ \neg\mathsf{ret} \wedge \varphi)) \\ \vee(\quad & \mathsf{call} \wedge \mathrm{X}(\quad \mathsf{ret} \wedge \varphi)) \\ \vee(\quad & \mathsf{call} \wedge \mathrm{X}(\ \neg\mathsf{ret} \wedge \mathsf{true}\,\mathrm{U}^{\mathrm{a}}(\neg\mathsf{call} \wedge \mathrm{X}(\mathsf{ret} \wedge \varphi))))\end{aligned}$$

Using this equality for substituting X^{a} operators, we can equivalently transform any CaRet formula s.t. every temporal operator is guarded by X and hence do a step-wise evaluation. When only the classical operators are considered, the number of different formulae arising during evaluation is bounded (as long as the formulae are kept in e.g. disjunctive normal form). This is no longer the case for the abstract operators as during evaluation an unbounded number of inequivalent formulae may occur. This is expected, as the formula describes a pushdown-language and encodes a stack. In the following, we will use pushdown machines to handle the stack explicitly. This simplifies implementation as well as theoretical discussion.

The outline for the rest of this section is as follows. We introduce non-deterministic push-down Mealy machines (PMM) and show how they can be determinized. Next, based on the FCaRet_4 semantics defined above, we give a procedure to construct a PMM from a CaRet^+ formula, that reads symbols and outputs the FCaRet_4 semantics of the word read so far.

Mealy machines. A non-deterministic Mealy machine can, in general reach multiple configurations at a time. Each such current configuration yields an output. To consistently define the overall output of the automaton, we need to be able to summarize the single outputs in each step. The existential character of a non-deterministic model is lifted to a supremum (join) operation on all possible outputs in each step. A configuration may have multiple successors, which have no order. We therefore need commutativity, associativity and idem-potency of the join operation on the outputs, that is, we require the output domain to be a semi-lattice.

Definition 1 (Push-down Mealy Machine). *A (non-deterministic)* push-down Mealy machine *(*PMM*) is a tuple* $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, Q_0, \mathbb{L})$ *where*

- Σ, Γ *are the finite* input *and* stack alphabet*, respectively, and* $\Gamma_\# := \Gamma \cup \{\#\}$ *the stack alphabet enriched by a new bottom symbol* $\# \notin \Gamma$*,*
- $\mathbb{L}$ *is the* output alphabet *with* $(\mathbb{L}, \sqcup)$ *forming a semi-lattice,*
- Q *is a finite set of control* states*,*
- $Q_0 \subseteq Q$ *is the set of* initial states *and*
- $\delta : Q \times \Gamma_\# \times \Sigma \to 2^{Q \times \Gamma_\#^{\leq 2} \times \mathbb{L}}$ *is the non-deterministic, labeled* transition function*.*

A *configuration* of $\mathcal{M}$ is a tuple $(q, s) \in Q \times (\Gamma^*\{\#\})$ comprising the current control state and a stack assignment ending with #. The *run* of $\mathcal{M}$ on

a non-empty input word $w = w_0 w_1 \dots w_n \in \Sigma^+$ is the alternating sequence $C_0 \xrightarrow{\ell_1} C_1 \dots \xrightarrow{\ell_n} C_n$ of sets of configurations $C_i \subseteq Q \times (\Gamma^*\{\#\})$ and output symbols $\ell_i \in \mathbb{L}$ s.t.

- $C_0 = Q_0 \times \{\#\}$,
- $L_{i+1} \subseteq \mathbb{L}$ and C_{i+1} are the smallest sets such that, for $\gamma \in \Gamma$, $(q, \gamma s) \in C_i$ and $(q', \gamma'\gamma'', \ell) \in \delta(q, \gamma, a_{i+1})$ implies $(q', \gamma'\gamma'' s) \in C_{i+1}$ and $\ell \in L_{i+1}$, and
- $\ell_i = \bigsqcup L_{i+1}$.

The *output* of $\mathcal{M}$ on w is $\mathcal{M}(w) := \ell_n$.

$\mathcal{M}$ is called a *visibly* PMM (VPMM), if it satisfies the corresponding constraints defined above for VPA.

3.3 Determinizing VPMM

In order to be able to actually implement a VPMM as monitor to evaluate observations it must be deterministic. We can lift the determinization construction for VPA [13] to determinize a VPMM $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, Q_0, \mathbb{L})$ by adding treatment of output symols. We construct an equivalent deterministic VPMM $\mathcal{P}' = (Q', \Sigma, \Gamma', \delta', q_0', \mathbb{L})$ as follows.

In the finite control $Q' = 2^{Q \times Q} \times 2^Q$ we store, as in the standard subset construction for finite automata, a set of *current states* $R \subseteq Q$ and additionally an *effect relation* $S \subseteq Q \times Q$.

Inbetween a call action a_c and its corresponding return action a_r, S summarizes the transitions that were made on every state. That is, when $\mathcal{P}$ were in some state q just after reading a_c and from there possibly reached some state q' before reading a_r, S contains the tuple (q, q').

The stack of $\mathcal{P}'$, stores triples (S', R', a_c) from $\Gamma \subseteq Q' \times \Sigma_c$ where R', S' are the current states and the effect relation at the time the last open call a_c occurred. In the initial state $q_0' = \{(\mathsf{Id}_Q, Q_0)\}$, there is no recorded effect, i.e. each q points to itself, and the current states are the initial states of $\mathcal{P}$.

Internal. An internal action $a_{\mathsf{int}} \in \Sigma_{\mathsf{int}}$ simply updates the set of current states by applying δ element-wise. The effect relation is updated analogously. If $(q, q') \in S$ is a recorded effect on q and q' is mapped to q'' by δ on reading a_{int}, then in the next state of $\mathcal{P}'$ we record the tuple (q, q'') as effect on q.

We let $\delta'((S, R), a_{\mathsf{int}}, \gamma) = (S', R', \gamma, \ell)$ such that

$$
\begin{aligned}
S' &= \{(q, q') \mid \exists q'', \gamma', \ell' : (q, q'') \in S, (q', \gamma', \ell') \in \delta(q'', a_{\mathsf{int}}, \gamma')\} \\
R' &= \{q' \mid \exists q \in R, \gamma', \ell' : (q', \gamma', \ell') \in \delta(q, a_{\mathsf{int}}, \gamma')\} \\
\ell &= \bigsqcup \{\ell' \mid \exists q \in R, \gamma', q' : (q', \gamma', \ell') \in \delta(q, a_{\mathsf{int}}, \gamma')\}
\end{aligned}
$$

As opposed to the construction for VPA, we have also to compute the current output ℓ. It is obtained from all possible transitions from the current states $q \in R$ via reading a_{int}. Since δ is non-deterministic these are in general multiple values that are considered in disjunction. We therefore take the join, i.e. the supremum, of those.

Call. Upon reading a call symbol $a_{\mathsf{c}} \in \Sigma_{\mathsf{c}}$, the current states set R and the current effect S is stored by pushing them, together with a_{c}, onto the stack. While the set of current states is maintained by applying a_{c} via δ, the effect relation is reset s.t. every state q maps to itself. The output is obtained in the same way as by reading an internal action.

We let $\delta'((S,R),a_{\mathsf{c}},\gamma) = (\mathsf{Id}_Q, R', (S,R,a_{\mathsf{c}})\gamma, \ell)$ such that

$$\begin{aligned} R' &= \{q' \mid \exists q \in R, \gamma', \gamma'', \ell' : (q', \gamma'', \ell') \in \delta(q, a_{\mathsf{c}}, \gamma')\} \\ \ell &= \bigsqcup\{\ell' \mid \exists q \in R, \gamma', \gamma'', q' : (q', \gamma'', \ell') \in \delta(q, a_{\mathsf{c}}, \gamma')\} \end{aligned}$$

Return. Having all the information from the stack when reading a return symbol $a_{\mathsf{r}} \in \Sigma_{\mathsf{r}}$, $\mathcal{P}'$ can simulate the transition relation δ on all current states. This, however is not done directly on R but on the current states at call-time R' by consecutively applying a_{c}, S and then a_{r} to obtain the new set of current states R'' and the new effect relation S''. We obtain the output from all possible transitions via a_{r}, after a_{c} and S have been applied.

We let $\delta'((S,R),a_{\mathsf{r}},(S',R',a_{\mathsf{c}})) = (S'', R'', \epsilon, \ell)$ such that

$$\begin{aligned} U &= \{(q, q', \ell') \mid \exists q_1, q_2, \gamma', \gamma'', \ell'' : (q_1, \gamma''\gamma', \ell'') \in \delta(q, a_{\mathsf{c}}, \gamma'), \\ &\qquad\qquad (q_1, q_2) \in S, (q', \epsilon, \ell') \in \delta(q_2, a_{\mathsf{r}}, \gamma'')\} \\ S'' &= \{(q, q') \mid \exists q_3, \ell' : (q, q_3) \in S', (q_3, q', \ell') \in U\} \\ R'' &= \{q' \mid \exists q \in R, \ell' : (q, q', \ell') \in U\} \\ \ell &= \bigsqcup\{\ell' \mid \exists q \in R, q' : (q, q', \ell') \in U\} \end{aligned}$$

Note that the stack might have been necessary for computing the effect S but once it is known, the effect can be applied to a set of states without using the stack.

Finally, we need to specially treat the case of a return action $a_{\mathsf{r}} \in \Sigma_{\mathsf{r}}$ when reading the bottom symbol. Let $\delta'((S,R),a_{\mathsf{r}},\#) = (S', R', \#, \ell)$ such that

$$\begin{aligned} S' &= \{(q, q') \mid \exists q'', \ell' : (q, q'') \in S, (q', \#, \ell') \in \delta(q'', a_{\mathsf{r}}, \#)\} \\ R' &= \{q' \mid \exists q \in R, \ell' : (q', \#, \ell') \in \delta(q, a_{\mathsf{r}}, \#)\} \\ \ell &= \bigsqcup\{\ell' \mid \exists q \in R, q' : (q', \#, \ell') \in \delta(q, a_{\mathsf{r}}, \#)\} \end{aligned}$$

3.4 Constructing a VPMM for CaRet$^+$

The idea of the construction is that the Mealy machine maintains the formulae that need to be proved. When reading an input symbol it verifies the propositional part and postpones the resulting future obligations. Standard LTL formulae are encoded into the finite control and evaluated on the next input. Abstract until and release operators are reduced to checking their unfolding.

When observing a call action, the abstract next modalities push their argument on the stack as future obligation. On return, this obligation is removed

from the stack and evaluated together with the current obligation stored in the finite control.

The transition function maintains a disjunctive clause form of stored obligations and thereby removes alternation on the fly. The output in every step is the truth value from the evaluation of the current finite control state combined with the current evaluation of the stack. The stack evaluation describes the truth values of the suspended abstract operators from higher levels.

Given a CaRet$^+$ formula Φ we construct a VPMM $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, Q_0, \mathbb{B}_4)$ where the control states $Q = 2^{\mathsf{sub}(\Phi)}$ store a set of formulae to be evaluated upon the next input symbol and the stack alphabet $\Gamma = 2^{\mathsf{sub}(\Phi)} \times \mathbb{B}_4 \times \mathbb{B}_4$ stores the future obligations, the current and the previous stack evaluation.

For better readability, the transition function $\delta : Q \times \Gamma \times \Sigma \to 2^{Q \times \mathbb{B}_4 \times \Gamma^{\leq 2}}$, is specified in two parts, one handling the evaluation of the finite control separately and another part working on the stack. Let therefore be $\delta_{\mathsf{c}} : Q \times \Sigma \to 2^{Q \times \mathbb{B}_4}$ the control transition function.

Finite control. The finite control evaluates propositional formulae directly according to the input symbol, also, next formulae just consume the input and delegate their argument to the next step. The semantics of a formula $\mathrm{X}^{\mathrm{a}}\,\varphi$ evaluates to $\bot$ if the next position is a return and the current position is not a call. However, δ_c will never be evaluated on X^{a} when reading a call symbol, as can be seen from the definition of the transition function δ below. Therefore we do not make a distinction for that case here. Until and release formulae are simply handled using their unfolding.

$$\delta_{\mathsf{c}}(\{p\}, a) = \begin{cases} \{(\emptyset, \top)\} & \text{if } p \in a \\ \emptyset & \text{if } p \notin a \end{cases}$$

$$\delta_{\mathsf{c}}(\{\neg p\}, a) = \begin{cases} \emptyset & \text{if } p \in a \\ \{(\emptyset, \top)\} & \text{if } p \notin a \end{cases}$$

$$\begin{aligned}
\delta_{\mathsf{c}}(\{\varphi \wedge \psi\}, a) &= \delta_{\mathsf{c}}(\{\varphi, \psi\}, a) \\
\delta_{\mathsf{c}}(\{\varphi \vee \psi\}, a) &= \delta_{\mathsf{c}}(\{\varphi\}, a) \cup \delta_{\mathsf{c}}(\{\psi\}, a) \\
\delta_{\mathsf{c}}(\{\mathrm{X}\,\varphi\}, a) &= \{(\{\varphi\}, \bot^{\mathsf{p}})\} \\
\delta_{\mathsf{c}}(\{\overline{\mathrm{X}}\,\varphi\}, a) &= \{(\{\varphi\}, \top^{\mathsf{p}})\} \\
\delta_{\mathsf{c}}(\{\mathrm{X}^{\mathrm{a}}\,\varphi\}, a) &= \{(\{\varphi, \neg\mathsf{ret}\}, \bot^{\mathsf{p}})\} \\
\delta_{\mathsf{c}}(\{\overline{\mathrm{X}^{\mathrm{a}}}\,\varphi\}, a) &= \{(\{\varphi, \neg\mathsf{ret}\}, \top^{\mathsf{p}})\} \\
\delta_{\mathsf{c}}(\{\varphi\,\mathrm{U}\,\psi\}, a) &= \delta_{\mathsf{c}}(\{\psi \vee (\varphi \wedge \mathrm{X}(\varphi\,\mathrm{U}\,\psi))\}, a) \\
\delta_{\mathsf{c}}(\{\varphi\,\mathrm{R}\,\psi\}, a) &= \delta_{\mathsf{c}}(\{\psi \wedge (\varphi \vee \overline{\mathrm{X}}(\varphi\,\mathrm{R}\,\psi))\}, a) \\
\delta_{\mathsf{c}}(\{\varphi\,\mathrm{U}^{\mathrm{a}}\,\psi\}, a) &= \delta_{\mathsf{c}}(\{\psi \vee (\varphi \wedge \mathrm{X}^{\mathrm{a}}(\varphi\,\mathrm{U}^{\mathrm{a}}\,\psi))\}, a) \\
\delta_{\mathsf{c}}(\{\varphi\,\mathrm{R}^{\mathrm{a}}\,\psi\}, a) &= \delta_{\mathsf{c}}(\{\psi \wedge (\varphi \vee \overline{\mathrm{X}^{\mathrm{a}}}(\varphi\,\mathrm{R}^{\mathrm{a}}\,\psi))\}, a)
\end{aligned}$$

Sets of formulae (clauses) are interpreted as conjunctions. We can therefore remove alternation by directly evaluating the single formulae on the input symbol

and combining the respective results:

$$\delta_{\mathsf{c}}(\{\varphi_1,\ldots,\varphi_n\},a) = \delta_{\mathsf{c}}(\{\varphi_1\},a) \boxdot \ldots \boxdot \delta_{\mathsf{c}}(\{\varphi_n\},a)$$

The result of a single evaluation $\delta_{\mathsf{c}}(\varphi,a)$ are sets of tuples (K,b) of clauses and verdicts. In order to combine them the clauses need to be brought back to the disjunctive clause form which is realized by an operation $\boxdot$: Let $\mathcal{K}_i = (K_i,b_i), \mathcal{H}_j = (H_j,c_j) \in Q \times \mathbb{B}_4$ be tuples of states (i.e. conjunctive clauses) and verdicts. Sets of such tuples are combined by means of a the meet-like operation $\boxdot\colon 2^{Q\times\mathbb{B}_4} \times 2^{Q\times\mathbb{B}_4} \to 2^{Q\times\mathbb{B}_4}$ as follows.

$$\{\mathcal{K}_1,\ldots,\mathcal{K}_n\} \boxdot \{\mathcal{H}_1,\ldots,\mathcal{H}_m\} = \bigcup_{\substack{i\in[1,n]\\ j\in[1,m]}} \{\mathcal{K}_i\} \boxdot \{\mathcal{H}_j\} \tag{1}$$

$$\{(K,b)\} \boxdot \{(H,c)\} = \{(K \cup H,\ b \sqcap c)\} \tag{2}$$

The operation maintains the disjunctive form of the clause structure (1). Single clauses are conjunctive and thus their meet is simply the union of the clauses. The truth values are combined in terms of the meet on $\mathbb{B}_4$ (2).

Stack control. The actual transition function of $\mathcal{M}$ makes direct use of the finite control function δ_{c} on internal action $a_{\mathsf{int}} \in \Sigma_{\mathsf{int}}$ actions since they do not involve a stack operation. Only the stack evaluation is used in the output:

$$\delta(\{\varphi_1,\ldots,\varphi_n\},(K,b_1,b_2),a_{\mathsf{int}}) = (\delta_{\mathsf{c}}(\{\varphi_1,\ldots,\varphi_n\},a_{\mathsf{int}}) \boxdot \{(\emptyset,b_1)\}) \times \{(K,b_1,b_2)\}$$

On return operations $a_{\mathsf{r}} \in \Sigma_{\mathsf{r}}$, the top-most stack symbol is removed and combined to the current control state. That is, the obligation suspended to the stack earlier on the matching call is now evaluated. Note, that the preliminary verdict at call time (b_1) now is obsolete and the previous one (b_2) is evaluated.

$$\delta(\{\varphi_1,\ldots,\varphi_n\},(K,b_1,b_2),a_{\mathsf{r}}) = (\delta_{\mathsf{c}}(\{\varphi_1,\ldots,\varphi_n\},a_{\mathsf{r}}) \boxdot \{(K,b_2)\}) \times \{\epsilon\}$$

For a call $a \in \Sigma_{\mathsf{c}}$ we have

$$\begin{aligned}
\delta(\{\varphi_1,\ldots,\varphi_n\},\gamma,a) &= \delta(\{\varphi_1\},\gamma,a) \tilde{\sqcap} \ldots \tilde{\sqcap} \delta(\{\varphi_n\},\gamma,a)\\
\delta(\{\mathrm{X}^{\mathrm{a}}\,\varphi\},(K,b_1,b_2),a) &= \{(\emptyset,\ \bot^{\mathsf{p}} \sqcap b_1,\ (\{\varphi\},\bot^{\mathsf{p}} \sqcap b_1,b_1)(K,b_1,b_2))\}\\
\delta(\{\overline{\mathrm{X}^{\mathrm{a}}}\,\varphi\},(K,b_1,b_2),a) &= \{(\emptyset,\ \top^{\mathsf{p}} \sqcap b_1,\ (\{\varphi\},\top^{\mathsf{p}} \sqcap b_1,b_1)(K,b_1,b_2))\}\\
\delta(\{\varphi\,\mathrm{U}^{\mathrm{a}}\,\psi\},\gamma,a) &= \delta(\{\psi \vee (\psi \wedge \mathrm{X}^{\mathrm{a}}(\varphi\,\mathrm{U}^{\mathrm{a}}\,\psi))\},\gamma,a)\\
\delta(\{\varphi\,\mathrm{R}^{\mathrm{a}}\,\psi\},\gamma,a) &= \delta(\{\psi \wedge (\psi \vee \overline{\mathrm{X}^{\mathrm{a}}}(\varphi\,\mathrm{R}^{\mathrm{a}}\,\psi))\},\gamma,a)
\end{aligned}$$

and for $\varphi \neq \mathrm{X}^{\mathrm{a}}\,\varphi'$, $\varphi \neq \varphi'\,\mathrm{U}^{\mathrm{a}}\,\psi$, $\varphi \neq \overline{\mathrm{X}^{\mathrm{a}}}\,\varphi'$ and $\varphi \neq \varphi'\,\mathrm{R}^{\mathrm{a}}\,\psi$

$$\delta(\{\varphi\},(K,b_1,b_2),a) = (\delta_{\mathsf{c}}(\{\varphi\},a) \boxdot \{(\emptyset,b_1)\}) \times \{(\emptyset,b_1,b_1)(K,b_1,b_2)\}$$

Let $\mathcal{K}_i = (K_i,b_i,\alpha_i\gamma), \mathcal{H}_j = (H_j,c_j,\beta_j\gamma) \in Q \times \mathbb{B}_4 \times \Gamma^2$ be tuples of states (i.e. conjunctive clauses), verdicts and the top-most stack symbols. Note, that

when ever a call occurs, the topmost stack symbol is not touched but a new symbol is pushed onto the stack. Therefore, the pushed symbols $\alpha_i = (A_i, u_i)$ and $\beta_i = (B_i, v_i)$ may differ whilst the symbol underneath is the same $\gamma = (G, g)$ for all tuples $\mathcal{K}_i$ and $\mathcal{H}_i$.

Sets of such tuples are combined by means of a the meet-like operation

$$\tilde{\sqcap} : 2^{Q \times \mathbb{B}_4 \times \Gamma^2} \times 2^{Q \times \mathbb{B}_4 \times \Gamma^2} \to 2^{Q \times \mathbb{B}_4 \times \Gamma^2}$$

as follows:

$$\{K_1, \ldots, K_n\} \,\tilde{\sqcap}\, \{H_1, \ldots, H_m\} = \bigcup_{\substack{i \in [1,n] \\ j \in [1,m]}} \{K_i\} \,\tilde{\sqcap}\, \{H_j\}$$

$$\begin{aligned} \{(K,\ b,\ (A, u, g_1)(G, g_1, g_2))\} \,\tilde{\sqcap}\, \{(H,\ c,\ (B, v, g_1)(G, g_1, g_2))\} \\ = \{(K \cup H,\ b \sqcap c,\ (A \cup B, u \sqcap v \sqcap g_1, g_1)(G, g_1, g_2))\} \end{aligned}$$

Theorem 1. *Let Φ be a CaRet formula and $w \in \Sigma^*$. Then $\mathcal{M}_\Phi(w) = [\![\Phi]\!]_4(w)$.*

Corollary 1. *Given a CaRet formula φ, we can construct in 2-*ExpTime *a push-down Mealy machine $\mathcal{M}$ implementing the four-valued FCaRet$_4$ semantics of φ.*

4 Anticipatory Monitoring of Visibly Context-free Properties

In this section we describe an anticipatory monitor construction for visibly context-free ω-languages. By basing the construction on properties given by ω-Vpa we provide support for complete CaRet including past operators and more expressive logics like VP-μTL and MSO$_\mu$ which are complete for the visibly context-free ω-languages. Furthermore, integrating an emptiness check into the monitor construction allows for the synthesis of *anticipatory* monitors, i.e. monitors that yield a definite verdict as early as possible.

Given a property $L \subseteq \Sigma^\omega$ we define a three-valued, anticipatory *monitor* function $\mathfrak{M}_3$ thereby lifting the concept of [10] from LTL to arbitrary ω-languages. $\mathfrak{M}_3 : 2^{\Sigma^\omega} \to (\Sigma^* \to \mathbb{B}_3)$ is given as

$$\mathfrak{M}_3(L)(w) = \begin{cases} \top & \text{if } \forall_{u \in \Sigma^\omega} : wu \in L \\ \bot & \text{if } \forall_{u \in \Sigma^\omega} : wu \notin L \\ ? & \text{otherwise.} \end{cases}$$

The monitor function yields $\top$ for a good prefix w i.e. if any continuation of w is in L, it yields $\bot$ for a bad prefix w i.e. if any continuation of w is not in L and it yields ? otherwise.

4.1 Emptiness per configuration

Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, Q_0, F)$ be an ω-VPA. In the following, we show how to construct a deterministic VPA that accepts exactly the good and inconclusive prefixes of $\mathcal{L}(\mathcal{P})$, i.e. $\{w \in \Sigma^* \mid \exists_{u \in \Sigma^\omega} : wu \in \mathcal{L}(\mathcal{P})\}$.

As Bouajjani et al. describe in [19], we can, in polynomial time, construct a multi-automaton $\mathcal{A} = (S \cup Q, \Gamma, Q, \delta_\mathcal{A}, A)$ accepting exactly the set of configurations from which there is an accepting run of $\mathcal{P}$. That is, $\mathcal{P}$ can still accept at least one word in a configuration $(q, w\#)$ iff $w \in \Gamma^*$ accepted by $\mathcal{A}$ when starting in the state $q \in Q$.

We construct a VPA $\mathcal{F} = (Q, \Sigma, \Gamma \times S^Q, \delta_\mathcal{F}, Q_0, F_\mathcal{F})$ that behaves like $\mathcal{P}$ but simultaneously simulates $\mathcal{A}$. The initial configuration of $\mathcal{F}$ represents the initial configurations of $\mathcal{P}$ and $\mathcal{A}$. Reading inputs form Σ, $\mathcal{F}$ simulates the behavior of $\mathcal{P}$ and when $\mathcal{P}$ pushes a symbol γ onto the stack, $\mathcal{F}$ additionally simulates $\mathcal{A}$ reading γ and stores the new configuration of $\mathcal{A}$ on the stack. When $\mathcal{P}$ removes the top-most symbol γ from the stack, $\mathcal{F}$ also removes the top-most symbol including the current configuration of $\mathcal{A}$ and thereby restoring the configuration of $\mathcal{A}$ before having read γ.

A configuration of $\mathcal{A}$ is a state $s \in S \cup Q$ for each initial state $q \in Q$, meaning if $\mathcal{A}$ started in q it would currently be in state s. A configuration is therefore a mapping from Q to $S \cup Q$. Let $\hat{\delta}_\mathcal{A} : S^Q \times \Gamma \to S^Q$ be the transition function of $\mathcal{A}$ lifted to mappings $f \in S^Q$ s.t. $\hat{\delta}_\mathcal{A}(f, \gamma) : q \mapsto \delta_\mathcal{A}(f(q), \gamma)$. That is $\hat{\delta}_\mathcal{A}$ applies a γ transition "state-wise" to f. Following this idea, we define the transition function of $\mathcal{F}$ as follows. Note, ω-VPA can, in general, not be determinized and thus we construct a non-deterministic automaton $\mathcal{F}$. However since $\mathcal{F}$ is a VPA, it can be determinized afterwards [13].

$$
\begin{aligned}
(q', (\gamma, f)) \in \delta_\mathcal{F}(q, (\gamma, f), a) &\Leftrightarrow (q', \gamma) \in \delta_\mathcal{P}(q, \gamma, a) && (\text{for } a \in \Sigma_{int}) \\
(q', \#_\mathcal{F}) \in \delta_\mathcal{F}(q, \#_\mathcal{F}, a) &\Leftrightarrow (q', \#_\mathcal{P}) \in \delta_\mathcal{P}(q, \#_\mathcal{P}, a) && (\text{for } a \in \Sigma_r) \\
(q', \epsilon) \in \delta_\mathcal{F}(q, (\gamma, f), a) &\Leftrightarrow (q', \epsilon) \in \delta_\mathcal{P}(q, \gamma, a) && \begin{pmatrix} \text{for } a \in \Sigma_r \\ \text{and } \gamma \neq \#_\mathcal{P} \end{pmatrix} \\
(q', (\gamma', f')(\gamma, f)) \in \delta_\mathcal{F}(q, (\gamma, f), a) &\Leftrightarrow \begin{matrix} (q', \gamma'\gamma) \in \delta_\mathcal{P}(q, \gamma, a) \\ \text{and } f' = \hat{\delta}_\mathcal{A}(f, \gamma') \end{matrix} && (\text{for } a \in \Sigma_c)
\end{aligned}
$$

Here, $\#_\mathcal{F}$ and $\#_\mathcal{P}$ denote the bottom stack symbols of $\mathcal{F}$ and $\mathcal{P}$, respectively. To correctly treat the empty stack, we interpret the bottom symbol $\#_\mathcal{F}$ of $\mathcal{F}$ as $(\#_\mathcal{P}, \mathsf{id})$ since for each state q of $\mathcal{P}$, $\mathcal{A}$ is initially in the corresponding initial state, which is q itself.

In every state $q \in Q$, $\mathcal{P}$ is in a non-empty configuration, iff the multi-automaton $\mathcal{A}$ accepts the current stack for q. The current configuration f of $\mathcal{A}$ is stored in the top-most stack symbol of $\mathcal{F}$. So, when $f(q)$ is an accepting state of $\mathcal{A}$ and the current control state is q, $\mathcal{P}$ had a non-empty configuration and we hence let $\mathcal{F}$ accept exactly in the configurations $(q, (\gamma, f)s)$ s.t. $f(q) \in F_\mathcal{A}$. This condition can be realized technically by storing the top-most stack symbol in the finite control and define the set of accepting states of $\mathcal{F}$ accordingly.

From that construction we conclude that $\mathcal{F}$ accepts exactly the non-bad prefixes for the language accepted by $\mathcal{P}$.

Theorem 2. *For all $w \in \Sigma^*$, $w \in \mathcal{L}(\mathcal{F})$ if and only if $\mathfrak{M}_3(\mathcal{L}(\mathcal{P}))(w) \neq \bot$.*

4.2 Anticipatory Monitors for Visibly Context-free Properties

Using the construction above we can now construct a Moore machine that computes the three-valued monitoring semantics $\mathfrak{M}_3(P)$ for any visibly context-free property $P \subseteq \Sigma^\omega$, assuming that P is presented as ω-VPA.

Definition 2 (Push-down Moore Machine). *A (deterministic)* push-down Moore machine *is a tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \Lambda, \lambda)$ where*
- *Q is a finite set of* states *and $q_0 \in Q$ the* initial state,
- *Σ, Γ, Λ are the finite* input-, stack- *and* output alphabets, *respectively, and $\Gamma_\# := \Gamma \cup \#$ the stack alphabet enriched by a new* bottom symbol $\# \notin \Gamma$,
- *$\delta : Q \times \Gamma_\# \times \Sigma \to Q \times \Gamma_\#^{\leq 2}$ the deterministic* transition function *and*
- *$\lambda : Q \to \Lambda$ the* output function.

A *configuration* of $\mathcal{M}$ is a tuple $(q, s) \in Q \times (\Gamma^*\{\#\})$ comprising the current control state and a stack assignment ending with #. The *run* of $\mathcal{M}$ on a word $w = a_1 \dots a_n \in \Sigma^*$ is the sequence of configurations $c_0 c_1 \dots c_{n+1}$ s.t. $c_0 = (q_0, \#)$ and for $\gamma \in \Gamma$, $c_i = (q, \gamma s)$ and $\delta(q, \gamma, a_{i+1}) = (q', \gamma'\gamma'')$ we have $c_{i+1} = (q', \gamma'\gamma''s)$. The *output* of $\mathcal{M}$ on w is $\mathcal{M}(w) := \lambda(q_{\mathsf{last}})$ where $(q_{\mathsf{last}}, s_{\mathsf{last}}) = c_{n+1}$.

The Moore Machine for $\mathfrak{M}_3$. In the fashion of [10] we construct $\mathcal{F}_P$ and also $\mathcal{F}_{\neg P}$ accepting all non-bad prefixes for the complement of P and combine them to a Moore machine. We know that if some $w \in \Sigma^*$ is rejected by $\mathcal{F}_P$, then $\mathfrak{M}_3(P)(w) = \bot$ and consequently if w is rejected by $\mathcal{F}_{\neg P}$ then $\mathfrak{M}_3(P)(w) = \top$. These cases exclude each other and if both accept then $\mathfrak{M}_3(P)(w) = ?$.

Note, while it is always possible to complement an ω-VPA for some property P and construct $\mathcal{F}_{\neg P}$ from it, it might be preferable to negate the property earlier. In particular, when using a logic that allows direct negation, it is advised to negate before constructing an automaton. Recall, we can assume $\mathcal{F}_P$ and $\mathcal{F}_{\neg P}$ determinized. We combine both and obtain a deterministic visibly push-down Moore machine $\mathcal{M}$, that outputs $\top$ for every good, $\bot$ for every bad and ? for every inconclusive prefix for P.

For $\mathcal{F}_P = (Q_P, \Sigma, \Gamma_P, \delta_P, I_P, F_P)$ and $\mathcal{F}_{\neg P} = (Q_{\neg P}, \Sigma, \Gamma_{\neg P}, \delta_{\neg P}, I_{\neg P}, F_{\neg P})$ we let $\mathcal{M} = (Q_P \times Q_{\neg P}, \Sigma, \Gamma_P \times \Gamma_{\neg P}, \delta, I_P \times I_{\neg P}, \mathbb{B}_3, \lambda)$
with $\delta((q_1, q_2), (\gamma_1, \gamma_2), a) := ((q_1', q_2'), (\gamma_1', \gamma_2')(\gamma_1'', \gamma_2''))$
where $(q_1', \gamma_1'\gamma_1'') = \delta_\varphi(q_1, \gamma_1, a)$ and $(q_2', \gamma_2'\gamma_2'') = \delta_{\neg\varphi}(q_2, \gamma_2, a)$.

The output of $\mathcal{M}$ is defined as

$$\lambda(q_1, q_2) = \begin{cases} \top & \text{if } q_2 \notin F_{\neg\varphi} \\ \bot & \text{if } q_1 \notin F_\varphi \\ ? & \text{otherwise.} \end{cases}$$

Note, that λ is well defined since P and $\neg P$ exclude each other.

Theorem 3. *Given an ω-VPA $\mathcal{P}$, we can construct a deterministic push-down Moore Machine $\mathcal{M}$ implementing the three-valued monitoring function for $\mathcal{L}(\mathcal{P})$, i.e. for all $w \in \Sigma^*$, $\mathcal{M}(w) = \mathfrak{M}_3(\mathcal{L}(\mathcal{P}))(w)$.*

Corollary 2. *Given a CaRet formula φ, we can construct in 3-ExpTime a push-down Moore machine $\mathcal{M}$ implementing the three-valued semantics function for φ.*

5 Conclusion

In this paper, we investigated the problem of monitoring visibly context-free properties. In particular we proposed a four-valued semantics for the future fragment of the temporal logic CaRet on finite words, together with a monitor synthesis algorithm yielding deterministic push-down Mealy machines for properties with calls and returns.

For the full CaRet logic, or more generally, for any visibly context-free language, we provided a three-valued monitoring approach adhering both, to the maxims of impartiality and anticipation. It comprises a three-valued anticipatory semantics as well as corresponding synthesis algorithm yielding deterministic push-down Moore machine.

Together with [17] this gives a complete picture of two-valued, impartial and anticipatory semantics for runtime monitoring.

References

1. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5) (2009) 293–303
2. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S., eds.: OOPSLA, ACM (2007) 569–588
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. J. Log. Comput. **20**(3) (2010) 651–674
4. Leucker, M.: Teaching runtime verification. In Khurshid, S., Sen, K., eds.: RV. Volume 7186 of Lecture Notes in Computer Science., Springer (2011) 34–48
5. Pnueli, A.: The temporal logic of programs. In: FOCS, IEEE Computer Society (1977) 46–57
6. Geilen, M.: On the construction of monitors for temporal logic properties. Electr. Notes Theor. Comput. Sci. **55**(2) (2001) 181–199
7. Havelund, K., Rosu, G.: Monitoring programs using rewriting. In: ASE, IEEE Computer Society (2001) 135–143
8. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In Katoen, J.P., Stevens, P., eds.: TACAS. Volume 2280 of Lecture Notes in Computer Science., Springer (2002) 342–356
9. d'Amorim, M., Rosu, G.: Efficient monitoring of omega-languages. In Etessami, K., Rajamani, S.K., eds.: CAV. Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 364–378

10. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In Arun-Kumar, S., Garg, N., eds.: FSTTCS. Volume 4337 of Lecture Notes in Computer Science., Springer (2006) 260–272
11. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. ACM Trans. Softw. Eng. Methodol. **20**(4) (2011) 14
12. Dong, W., Leucker, M., Schallhart, C.: Impartial anticipation in runtime-verification. In Cha, S.D., Choi, J.Y., Kim, M., Lee, I., Viswanathan, M., eds.: ATVA. Volume 5311 of Lecture Notes in Computer Science., Springer (2008) 386–396
13. Alur, R., Madhusudan, P.: Visibly pushdown languages. In Babai, L., ed.: STOC, ACM (2004) 202–211
14. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In Jensen, K., Podelski, A., eds.: TACAS. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 467–481
15. Alur, R., Arenas, M., Barceló, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. Logical Methods in Computer Science **4**(4) (2008)
16. Bozzelli, L.: Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In Caires, L., Vasconcelos, V.T., eds.: CONCUR. Volume 4703 of Lecture Notes in Computer Science., Springer (2007) 476–491
17. Rosu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: This time with calls and returns. In Leucker, M., ed.: RV. Volume 5289 of Lecture Notes in Computer Science., Springer (2008) 51–68
18. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In Dwork, C., ed.: PODC, ACM (1990) 377–410
19. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In Mazurkiewicz, A.W., Winkowski, J., eds.: CONCUR. Volume 1243 of Lecture Notes in Computer Science., Springer (1997) 135–150