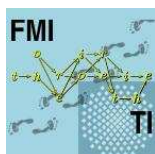


Institut für Formale Methoden
der Informatik
Abteilung Theoretische Informatik
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Laboratoire Spécification et Vérification
École Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France



Studienarbeit Nr. 2270

A Continuous Truth Domain for Runtime Verification

Normann Decker

Course of Study: Computer Science

Examiner: Prof. Volker Diekert

Supervisor: Dr. Benedikt Bollig

Commenced: 1 April 2010

Completed: 1 October 2010

CR-Classification: F.3.1, D.2.4

Abstract. Specification formalisms like LTL specify properties for infinite traces whereas monitors only observe finite prefixes. These can be conclusive, however, in general they are not. Restricting monitors to only two possible conclusions is therefore inappropriate in many cases and known approaches distinguish only between a small number of verdicts. We extend the setting to an infinite, continuous domain that enables us to further discriminate runs and allows for more detailed appraisals in presence of inconclusive information. The concept of continuous space induces new notions of monitorability that we study and relate to well known classes.

Keywords. Runtime Verification

Contents

1. Introduction	3
2. Runtime Verification	6
2.1. Observing systems at runtime	6
2.2. Specifying system properties	7
2.3. Monitoring	8
2.4. Applications	10
3. Monitoring finite state systems	12
3.1. Properties	13
3.2. Discrete monitoring	14
3.3. Continuous monitoring	15
3.3.1. Prefix monotonic monitors	16
3.3.2. Distinguishing inconclusive prefixes	19
3.3.3. Computing future and past aspects	24
3.3.4. Choosing a monitor function	29
4. Conclusions	34
A. Linear Temporal Logic	39

1. Introduction

Software is complex. Even small applications consist of thousands of lines of codes, hundreds of modules that are connected like the gears of a mechanical clockwork. A typo in the source code or a wrongly connected gate in hardware designs might effect an error, an interface, implemented inaccurately, might do so as well. Any abstract algorithm or protocol might take assumptions that do not hold for every use case of the problem they are supposed to solve. It is not always easy to find the reason for an observed misbehaving of a system and it is harder to assure their absence.

Architectures evolve, the number of components in today's soft- and hardware rise and distribution, ubiquity becomes inherent. That is where quality assurance gets difficult.

Information processing systems interlace every part of modern society. Transportation, communication, production or health care got heavily dependent on reliably working digital systems. Efficiency, safety and privacy are, among others, expected properties. That is why their assessment becomes important.

The most abstract and least pragmatical but yet most consequent and scientifically interesting approach to verification is the use of formal methods [Wol97].

Founded on the rich and consistent base of mathematics, these methods provide the most convincing notion of assurance. They work on abstract models of systems and specifications and the level of abstraction must be traded off between applicability of results and computational expense.

The major approaches to formal verification can be grouped roughly into the domains of *model checking*, *theorem proving*, *testing* and *runtime verification*.

Certain properties can be proven manually in an inductive, logic based manor comparable to classical mathematical proofs. For this method, referred to as *theorem proving*, software tools can be applied for many small, direct steps. However, a user is usually needed to complete proofs or take steps that cannot be done automatically. Thus a considerable technical expertise and understanding of the specification is required. Even though already verified code can be reused reducing e.g. the workload on consecutive versions of a program, a unique and new proof is needed for every piece of software in general, making theorem proving very expensive (see [KEH⁺09] for an extensive case study).

The - theoretically - most exhaustive automatic verification technique is the approach of *model checking*. The objective is to answer the question if a given structure models (i.e. is consistent with) a specification. Considering a formalized (and usually abstracted) model of the system under scrutiny and its specification, the problem can be seen e.g. as language inclusion problem. If every behaviour of the implemen-

tation model is also covered by the specification, the system is considered correct. The practical drawback of this approach is a generally exponential blow-up in the state space of the implementation model.

A classical approach, that is formally less exhaustive but more pragmatical, is *testing*. There is no essential need for modelling the implementation, but only comparing in-/output behaviour with respect to a set of test cases, called a test suite. Unless the whole input domain is covered (which is reasonably not the case) testing can never provide complete proof of correctness. However, considering some model of the system under test (i.e. white box testing or structural testing) allows creation of test suites that satisfy certain coverage criteria such as path or instruction coverage, strengthening at least confidence in the tests [Kor90].

In addition to these methods, that are only applied *before* the actual deployment, *runtime verification* is used to oversee the system during the run. The exponentially growing number of possible executions that need to be taken into account during model checking now breaks down to a single sequence of observed states. Even though violations to the specification might exist, the system has the chance to react on their occurrence, thus complementing (incomplete) pre-executional verification.

This report focuses on runtime verification. Faults may be observable before they effect failures¹ and we will study approaches to monitor and assess a system's run according to a formal specification. The earlier a violation is recognized or the more precisely it is estimated the more adequately can reactions be planned and preformed (e.g. starting by signalling a failure to the user and rescheduling actions up to shutting down processes). Recent approaches to monitoring properties of reactive systems are presented. Fulfilment of properties during execution can often neither be guaranteed nor excluded and a simple, two valued truth domain is not adequate in many applications. More recent research improved the situation by proposing up to four different verdicts [BLS06b, BLS07a] and our contribution is to extend to a an infinite domain. It enables us to further discriminate runs and allows for more detailed appraisals in presence of inconclusive information. The concept of continuous space induces, furthermore, new notions of monitorability that we study with regard to our monitoring approach.

Outline. In Chapter 2 runtime verification is discussed in general, presenting approaches to observe, specify and check system behaviour from the literature.

Chapter 3 concerns different approaches to assess partial runs regarding a specification. The first section formally introduces the popular language classes of safety, co-safety and monitorability (ugly-freeness). In the subsequent section we construct a three valued monitor for ω -regular languages from Büchi automata.

In Section 3.3 we introduce monitor functions that give verdicts from a continuous domain. Desired function characteristics lead to different notions of monitorability and classes of monitorable ω -languages. We discuss monitor functions that are

¹A failure is a deviation between the observed behaviour and the required behaviour of a system, a fault occurs during the execution and results in an incorrect state that may or may not lead to a failure. [DGR04]

monotonic with respect to the prefix relation in Section 3.3.1.

To distinguish prefixes regarding to their estimated compliance to a specification a future-past relation is proposed in Section 3.3.2. Functions that are monotonic regarding this relation lead to a new class of monitorable languages that is related to safety, co-safety and ugly-freeness.

Section 3.3.3 shows how to use finite automata to compute future and past aspects that characterize finite prefixes with regard to a given ω -language. The chapter ends with a procedure to choose a continuous monitor for a given language in Section 3.3.4. It follows our conclusion and perspectives.

2. Runtime Verification

This report focuses on monitoring and before we proceed to the technical part in Chapter 3 let us have a look at the context. Even if the actual monitoring process is the core part of runtime verification it is accompanied by other issues. To check and assess a running system it is essential to clarify the target of observation. A monitor itself will check against an abstract specification and it is necessary to define how symbolic properties are subsumed from real implementations.

2.1. Observing systems at runtime

A set of propositions, atomic properties that may or may not hold in a certain system state, must be defined by the user since they need to capture the specific target of the analysis. The properties can concern physical measures and hardware, e.g. "temperature $< 80^{\circ}\text{C}$ " or "hard drive 3 reports an error". Other properties can be taken from program variables or method invocations ($x < 1000$, "backup process started", "all active users are authorized").

The propositions and the methods to evaluate them obviously depend on the implementation and on the intention of the specification. One way to evaluate these propositions is to inject instructions into the source code or provide a (virtual) machine with according interfaces. The decision, how much influence the monitoring process should have into the system, is important. Code injections or synchronization points for observing devices might influence implementational or functional properties of the system. Needed resources such as computing time and memory can increase, monitoring code that is used for debugging might have to be kept even in the productive application to avoid behavioural changes. Off-line monitoring could make expensive monitoring feasible by allowing random access to the trace and not consuming any system resources. It can be suitable for properties that can be decided only after the system stops or properties that require a backward traversal of the trace. However, it does not allow for any real-time feedback or other interaction with the system.

Delgado et al. [DGR04] did an extensive study on the different approaches to oversee systems and compare implementations of monitoring frameworks.

2.2. Specifying system properties

To make use of observations, it is necessary to specify desired behaviour. Formal specification frameworks allow a flexible and automated analysis. Properties are defined by means of sequences of observations, that is, sequences of states for that atomic propositions are evaluated.

Pnueli recognized that the major notions of correctness in system verification concerns the concepts of invariance and eventuality, in particular in non-terminating systems. He introduced in [Pnu77] a formal approach to temporal reasoning about programs by time dependence of events that considerably influenced the field of formal verification.

The notions of eventuality, invariance and temporal implication are reflected by the *linear temporal logic*, LTL¹. In this logic, models are infinite sequences, ω -words, over a finite alphabet. Thus properties are, in that sense, sets of valid runs represented by infinite strings and we will therefore use the terms “language” and “property” interchangeably. For models that allow branching of traces, an extended version called computational tree logic, CTL, can be used. LTL is not as expressive as monadic second order (MSO) logic [Kam68] and finite automata, but yet powerful enough to serve as intuitive specification language.

Additionally, MITL, metric interval temporal logic [AFH96, MNP06], and TLTL, timed linear temporal logic [Ras99], provide extensions for specifying also timing constraints.

Given the formalisms to specify properties it is possible to differentiate between certain classes. Lamport first introduced the terms of *safety* and *liveness* to the verification setting in [Lam77]. Different techniques for proving those properties motivated this distinction.

Safety properties are commonly understood as to describe that “nothing bad will happen”. From a (runtime) verification point of view they can be seen as properties that give, eventually, witness for every violation and is therefore safe as long as the opposite is not evident. Checking a system (or a run of a system) against a safety property, we can notice its violation after a finite time. For their complements, *co-safety* properties, not violations but satisfaction is thus observable after a finite prefix of a run. Liveness properties specify notions of “something good”, that must happen eventually during an execution.

Alpern and Schneider clarified these terms in [AS85] providing a formal definition and characterizations. They describe liveness by the possibility to continue any finite prefix of a run to an infinite one satisfying the property.

The safety languages are the open sets in the so called natural topology over the universe of languages of infinite words. In that topology the co-safety languages are then the closed sets and languages representing a liveness property are exactly the dense sets. Intuitively, one can find an arbitrarily “close” member (in terms of common prefixes) to every excluded word. The topological characterization is used

¹See Appendix A for a formal definition.

Formula	Safety	So-safety
$\Box p$	✓	
$\Diamond q$		✓
Xq	✓	✓
$\Box \Diamond p$		
$Xp \vee \Box \Diamond p$		
$p U q$		✓

Table 2.1.: Examples for safety and co-safety properties expressed in LTL.

by Alpern and Schneider to proof that every property is the intersection of a safety and a liveness property. Their characterization of safety and liveness is independent of the used notion for expressing properties. In [AS87] they furthermore give a construction to decompose a property given by a Büchi automaton into a safety and liveness part whose intersection yields the property.

For runtime verification, Pnueli and Zaks introduced in [PZ06] the notion of *monitorability* for temporal properties. A property is monitorable after observing a finite trace, as long as a monitor can still yield a conclusive verdict, i.e. satisfaction or violation. Otherwise the observed finite word is called “ugly” for the specific property. We will define the class formally in Section 3.1.

It has recently been shown in [Bau10] that the problem to decide, whether there exists such ugly prefixes for a given ω -regular language, is complete for PSpace (in terms of the length of an LTL formula or the size of a Büchi automaton).

2.3. Monitoring

The motivating issue in runtime verification is, after all, to relate the observed information to the specified behaviour.

Once, the system information and system behaviour are clarified and formalized to meet the abstract notion of states, the actual verification process is performed. This particular performance of comparing the sequence of condensed observations (system states, abstract behaviour) and the specification is referred to as *monitoring*. The more or less abstract device of a monitor is constructed from a specification and transforms the observation to some sort of statement about the current run. Monitoring hence involves the questions what type of verdict a monitor should yield and how it can be obtained and implemented.

Monitor verdicts

In first place, monitoring is basically solving the word problem for a language. That is, given a system run, verifying that this run is valid (included) in the specification. It is thus initially a matter of *true* (\top) and *false* (\perp). A finite automaton accepts a given word or it does not and the classical semantics of LTL maps formulas to the

boolean truth domain $\mathbb{B} = \{\top, \perp\}$ [Pnu77]. However, the setting of runtime verification obliges to combine the binary problem with the fact of (in general) insufficient information to solve it.

Observing a system can only provide finite prefixes of the whole infinite run. Sometimes, a precise decision is possible, for example because a required event already happened or a proposition that is supposed to always hold was violated at some point. If this is not the case, one way to cope with finite prefixes can be found in [GH01]. Events that have to happen eventually or never can be checked on a finite trace. In the former case this is a pessimistic approach since it might happen later, in the latter case it is optimistic.

In [PM95], Manna and Pnueli suggest with FLTL to use explicitly a strong (existential) and weak (universal) next operator. They differ in their interpretation of non-observed next-states in the sense, that it has in particular to exist for the strong version to evaluate to \top while the weak version only evaluates to \perp under a contradicting witness.

A strong and weak semantics, LTL^+ and LTL^- , are combined in [EFH⁺03] to their “truncated” semantics where the weak view is consistent with a preference for false positives, the strong view with a preference for false negatives.

Multivalued monitor domains

Since checking properties leads in general not to final conclusive results, Bauer et al. found the boolean truth domain inappropriate and introduced in [BLS06b] a three valued monitor domain $\mathbb{B}_3 = \{\top, \perp, ?\}$. Using \mathbb{B}_3 , they defined a semantics on LTL formulas that assigns the value ? to inconclusive words. Bauer et al. also give a monitor construction that we will outline in Section 3.2.

In [BLS07a], The authors branch the inconclusive verdict and propose a four valued semantics for LTL formulas. The authors describe four maxims that they consider important for LTL semantics aimed at runtime verification.

- (1) *Existential next* requires the inclusion of a strong next operator,
- (2) *Complementation by negation* requires that a negated formula evaluates to the complemented and different truth value.

These maxims are motivated for semantics on finite words in general. In runtime verification, however, any finite word is to be interpreted as *finite prefix* of an infinite word. There is, indeed, a next state, only the evaluation is not yet possible. To reflect this particularity of runtime verification, two more maxims were proposed.

- (3) *Impartiality* requires that a finite trace is not evaluated to \top (\perp) if there still exists an infinite continuation leading to another verdict,
- (4) *Anticipation* requires that once every infinite continuation of a finite trace leads to the same verdict, this finite trace evaluates to the very same one.

The authors compare different concepts for semantics on finite traces regarding to these maxims. The two-valued approach in [PM95] only meets Maxim (1) and (2).

For the combination of LTL^+ and LTL^- [EFH⁺03], Bauer et al. argue that Maxim (1) is met by the strong version, LTL^+ but not by the weak one. The approach does not respect any of the other maxims and so they propose their own solution, called RV-LTL. It combines FLTL and LTL_3 in the way that FLTL is used to distinguish inconclusive results in LTL_3 . That leads to a four valued truth domain $\mathbb{B}_4 = \{\top, \perp, \top^P, \perp^P\}$, consisting of *true*, *false*, *probably true* and *probably false* and is justify by respecting all four of their maxims.

A continuous monitor domain

To further distinguish between inconclusive words the idea of an even larger, continuous domain of monitor verdicts arises naturally. Then a monitor does not only choose a qualitative result from a finite set of cases but makes a quantitative statement about a prefix with respect to a given property. These statements can, for example, be interpreted as “believe” or amount of “evidence” for fulfilment or violation of requirements. Imagine, for example, the safety property $\Box a$ and the two observations

$$\begin{aligned} u_1 &= aaa \quad \text{and} \\ u_2 &= aaaaaaaaaa. \end{aligned}$$

Both, u_1 and u_2 , are formally inconclusive or, considering the semantics discussed for \mathbb{B}_4 , at most “probably true”. Intuitively, however, the confidence in u_2 is higher than in u_1 , simply because we observed the system longer (confirming non-violation so far). Similarly, for $\Diamond b$, u_2 is less promising since we waited longer for the required event that still has not happened.

The aspects of quality for prefixes, regarding a property, may differ depending on the system, or the global setting. In Chapter 3 we study different approaches to assess and, in particular, compare prefixes. In combination with general characteristics monitors should comport with (e.g. convergence to concise truth values) we formulate different desired features of monitor functions. We investigate classes of languages (i.e. properties) that correspond to these features since not every property can be monitored by functions providing them. Additionally we provide a procedure to obtain monitor functions for a given property and construct automata to help compute their values.

2.4. Applications

Before going deeper into some details of monitoring in the next chapter, we outline some ideas on how runtime verification can be used in application. In [BLS06a] a “runtime reflection framework” for distributed systems is presented by Bauer et al.

The framework uses monitoring techniques to observe failures and is intended to provide methods for subsequently identify failures and recover. For identification, a diagnostic engine takes the current state of the system and monitor verdicts and uses a SAT-solver to reason about system components that cannot work correctly.

The idea of a system observing itself, that is able to recognize its own (mis)behaviour, evolves to whole self-managing computer-systems. They have the potential to be safer, more reliable and cost-effective [HS06]. Runtime verification coincides with self-monitoring that plays an important role in such systems since they need to keep track of their actions and internal states in order to act and adapt.

Another interesting application concerns electronic contracts. These contracts formalize the idea of paper contracts to obtain a representation that allows to answer questions about the expected behaviour from a partner in the future, to foresee contract violations and to identify the responsible party in case of violations [XJ10]. A formalism for contract specifications has been proposed by Prisacariu and Schneider [PS07] and led to a monitor construction in [KPS08]. A contract specifies obligations, rights and prohibitions of parties and, furthermore, it defines penalties for violating parties which can be seen in analogy to the mitigation process in runtime reflection [LS09].

3. Monitoring finite state systems

After describing the general setting, we go into more detail in the following section. In order to give characterizations and proof properties of functions and languages we will first clarify some notation and define important terms.

We follow the convention that $\mathbb{N} = \{1, 2, 3, \dots\}$ and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. We also use the extensions $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ and $\mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty, \infty\}$ along with the following sense of ordering and behaviour under addition and multiplication.

- ∞ is the supremum and $-\infty$ is the infimum: $\forall z \in \mathbb{Z} : -\infty < z < \infty$.
- ∞ and $-\infty$ are additionally inverse to each other: $\infty - \infty = 0$.
- $\pm\infty$ dominates integers: $\forall z \in \mathbb{Z}, n \in \mathbb{N}_\infty :$
 $z \pm \infty = \pm\infty, \pm\infty \cdot n = \pm\infty, \pm\infty \cdot (-n) = \mp\infty$.

Note that $\pm\infty \cdot 0$ is undefined, however, we will make sure not to run into this case.

For this chapter, let Σ always be a fixed, finite alphabet. A word w over Σ is a sequence $w_1w_2w_3\dots$, with each $w_i \in \Sigma$. We consider finite words, as elements of the free monoid Σ^* with concatenation and the empty word λ as neutral element. Let Σ^ω denote the set of infinite sequences, called ω -words, over Σ . The union of finite and infinite words is denoted $\Sigma^\infty := \Sigma^* \cup \Sigma^\omega$. The complement of languages is always taken according to the considered superset, i.e for languages $L \subset \Sigma^\omega$ and $L' \subset \Sigma^*$ the complements are $\overline{L} := \Sigma^\omega - L$ and $\overline{L'} := \Sigma^* - L'$, we sometimes write $\beta \notin L$ meaning $\beta \in \overline{L}$.

Furthermore, for a word w and $n \in \mathbb{N}$ we denote

- the suffix of w , beginning at position n as $w^{(n)} = w_nw_{n+1}w_{n+2}\dots$,
- the prefix of length n of w as $w_{(n)} = w_1w_2\dots w_n$ and $w_{(0)} = \lambda$,
- the set of all prefixes of w as $\overleftarrow{w} = \{u \in \Sigma^* \mid \exists v \in \Sigma^\infty : uv = w\}$,
- for finite words w the set of all infinite extensions $\overrightarrow{w} = \{wv \in \Sigma^\omega \mid v \in \Sigma^\omega\}$.

In most cases, we will use $\alpha, \beta, \gamma, \dots$ to denote infinite words and u, v, w, \dots for finite words.

3.1. Properties

Properties, i.e. ω -languages, can be categorized by different means. We will now systematically define some of the most important classes, starting with a formal definition of the previously used terms good, bad (cf. [KV01]) and ugly prefix (cf. [BLS07b]).

Definition 3.1 (Good, bad and ugly prefixes). *Let $L \subseteq \Sigma^\omega$ be an ω -language and $u \in \Sigma^*$ a finite word. We call u*

- a good prefix for L , iff $\vec{u} \subseteq L$,
- a bad prefix for L , iff $\vec{u} \cap L = \emptyset$
- an ugly prefix for L , iff $\forall v \in \Sigma^* : uv$ is neither good nor bad.

Emerson formulated in [Eme83] the limit closure restriction to classify sets of computation paths (i.e. ω -languages). This class coincides naturally with the topological notion of closed sets that are connected to safety properties by Alpern and Schneider in [AS85]. We follow the similar, and as we think intuitive, formulation from [KV01] that is based on the notion of good prefixes.

Definition 3.2 (Safety/Co-safety languages). *A language $L \subseteq \Sigma^\omega$ is called*

- a safety language, or safe, if for all $\alpha \in \bar{L}$, there is a prefix $u \in \overleftarrow{\alpha}$ which is bad for L .
- a co-safety language, or co-safe, if for all $\alpha \in L$, there is a prefix $u \in \overleftarrow{\alpha}$ which is good for L .

The class of ω -languages, that are safe (co-safe) is called SAFETY (CO-SAFETY).

The notion of *monitorability* was proposed by Pnueli and Zaks in [PZ06] and reflects the occurrence of ugly prefixes in the sense, that once an ugly prefix is observed no conclusive result can be obtained anymore and thus monitoring can be stopped. Recall Table 2.1 on page 8 for some examples.

Definition 3.3 (Non-monitorability/ugly-freeness). *Let $L \subseteq \Sigma^\omega$ be an ω -language. We call L non-monitorable after a word $u \in \Sigma^*$, if u is an ugly prefix for L . If there is no ugly prefix for L , the language is called ugly-free (or monitorable). $\overline{\text{UGLY}}$ denotes the class of all ugly-free ω -languages.*

The term “monitorable”, can be misleading, since monitoring even non-monitorable properties can still be useful. Even if an ugly prefix exists, a monitor can still report conclusively as long as it does not become manifest in the current run (which might never happen as the possible behaviour of the system can be a proper subset of the specified property excluding the “ugly runs”). For example, the language $\mathcal{L}((a \wedge \Box \Diamond a) \vee \Box b)$ becomes only non-monitorable if the observation starts with a . Since we discuss different types of monitors and therefore different notions of “monitorability” we will rather use the term ugly-free for the sense above.

Corollary 3.1 (Safety/co-safety languages are ugly-free). *Safety and co-safety languages $L \in \text{SAFETY} \cup \text{CO-SAFETY}$ are ugly-free.*

For a safety or a co-safety language L , there are no ugly prefixes. However, this property also holds for (some) non safety/co-safety properties:

Remark (Ugly-freeness is more than safety and co-safety). *The class of ugly-free ω -languages is strictly larger than the union of safety and co-safety languages. Consider, for example, the language defined by the LTL formula $\varphi = ((p \vee q)Ur) \vee \Box p$.*

3.2. Discrete monitoring

In order to handle inconclusiveness, Bauer et al. introduced in [BLS06b] the three valued truth domain $\mathbb{B}_3 = \{\top, \perp, ?\}$ for semantics of finite prefixes (i.e. words in Σ^*) with respect to languages of infinite words. They also gave a construction for a finite state machine (FSM) implementing a function $m_L : \Sigma^* \rightarrow \mathbb{B}_3$, where L is an ω -regular language, expressed by an LTL formula and

$$m_L(u) = \begin{cases} \top & \text{if } u \text{ is a good prefix for } L \\ \perp & \text{if } u \text{ is a bad prefix for } L \\ ? & \text{otherwise.} \end{cases}$$

We will refer to these functions, mapping finite words to some truth domain, (or FSMs implementing them) as monitors. The above specified mapping is discrete in the sense, that the truth domain is finite. Furthermore, to distinguish the monitor described here from the ones in the following chapter, we want to call it the \mathbb{B}_3 -monitor as it uses the tree valued truth domain.

The monitor function, as defined, respects the maxim of *impartiality*, meaning that \top or \perp are not taken unless the given word is indeed a good or bad prefix, and *anticipation* in the sense that the monitor reports a conclusive verdict as soon as it is possible. We summarize this behaviour under the term *accuracy* (of final conclusion).

We outline a monitor construction based on the procedure given in [BLS06b], however, instead of starting on an LTL formula φ and constructing two Büchi automata accepting $\mathcal{L}(\varphi)$ and its complement we start direct on a given automaton for an ω -language L .

Definition 3.4 (Büchi automaton [Bü62]). *A Büchi automaton over a finite alphabet Σ is a tuple $\mathcal{A} = (Q, q_0, \delta, F)$, where*

- Q is a finite non-empty set of states,
- $q_0 \in Q$ is the initial state,
- $\delta \subseteq Q \times \Sigma \times Q$ the transition relation and
- $F \subseteq Q$ is a set of accepting states.

A run $\rho = \rho_0, \rho_1, \dots$ ($\rho_i \in Q$) of a Büchi automaton on a word $\alpha = \alpha_1\alpha_2\cdots \in \Sigma^\omega$ is an infinite sequence of states such that $\rho_0 = q_0$ and $\forall_{i \in \mathbb{N}} : (\rho_{i-1}, \alpha_i, \rho_i) \in \delta$. We say ρ is accepting if and only if there is an accepting state $q \in F$ such that q occurs infinitely often in ρ (i.e. $|\rho|_q = \infty$).

The accepted language of a Büchi automaton \mathcal{A} is

$$\mathcal{L}(\mathcal{A}) = \{\alpha \in \Sigma^\omega \mid \exists \text{ an accepting run of } \mathcal{A} \text{ on } \alpha\}.$$

For $q \in Q$, $\mathcal{A}_q = (Q, q, \delta, F)$ denotes the automaton that is similar to \mathcal{A} but starts in state q .

Let from now on $\mathcal{A} = (Q, q_0, \delta, F)$ be fixed, accepting an ω -language L . We define the set

$$E_{\mathcal{A}} := \{q \in Q \mid \mathcal{L}(\mathcal{A}_q) \neq \emptyset\}$$

of states that result in a non-empty language accepted by \mathcal{A}_q .

Büchi automata are constructively closed under complementation [Bü62], furthermore Safra gave in [Saf88] a construction that is optimal in size ($2^{\mathcal{O}(n \log(n))}$). Thus, we can obtain an automaton $\bar{\mathcal{A}}$, such that $\mathcal{L}(\bar{\mathcal{A}}) = \bar{L}$.

The NFAs $\mathcal{B}_{\mathcal{A}} = (Q, q_0, \delta, E_{\mathcal{A}})$ and $\mathcal{B}_{\bar{\mathcal{A}}} = (Q, q_0, \delta, E_{\bar{\mathcal{A}}})$ accept all words $w \in \Sigma^*$, that are not bad prefixes, respectively not good prefixes, for L .

To obtain deterministic versions $\det(\mathcal{B}_{\mathcal{A}})$ and $\det(\mathcal{B}_{\bar{\mathcal{A}}})$ we use the classical powerset construction. The state space of them will then be 2^Q .

If a word $w \in \Sigma^*$ leads $\det(\mathcal{B}_{\bar{\mathcal{A}}})$ to a state $P \in 2^Q$ where every $q \in P$ is not in $E_{\bar{\mathcal{A}}}$, then w is a good prefix since no run passing any $q \in P$ is accepted by $\bar{\mathcal{A}}$. To classify w as bad prefix, we can proceed similarly using $E_{\mathcal{A}}$. This motivates the following definition for a function $\lambda : 2^Q \times 2^Q \rightarrow \mathbb{B}_3$.

$$\lambda(P, P') := \begin{cases} \top & \text{if } P \cap E_{\bar{\mathcal{A}}} = \emptyset \\ \perp & \text{if } P' \cap E_{\mathcal{A}} = \emptyset \\ ? & \text{otherwise} \end{cases}$$

It follows that, by using λ to label the states $Q' \subseteq 2^Q \times 2^Q$ of the product automaton

$$\det(\mathcal{B}_{\bar{\mathcal{A}}}) \times \det(\mathcal{B}_{\mathcal{A}}),$$

we obtain an FSM $\mathcal{M} = (Q', q'_0, \delta', \lambda)$ that yields \top , \perp or $?$ for a word that is a good, bad or inconclusive prefix, respectively.

The minimized version of \mathcal{M} is unique (modulo isomorphic changes) for an ω -language L , given by any Büchi automaton and therefore what we want to consider the \mathbb{B}_3 -monitor for L from now on.

3.3. Continuous monitoring

In order to study continuous monitorability we start by defining properties of functions that are intuitively desirable in the context of runtime verification. From these

properties we derive classes of languages that can be monitored by a function satisfying some of these properties and compare them to well-known classes such as SAFETY and CO-SAFETY.

For a continuation of boolean truth domains it appears natural to generalize the domain in a way that allows for continuous values in between the extremes \perp and \top . We therefore will consider mappings from finite words over a given finite alphabet Σ to the closed interval $[0, 1] \subset \mathbb{R}$.

We then leave the so far described finite domain setting of monitors and use a notion of convergence to argue about behaviour of these functions on infinite sequences of arguments and in continuous space.

We say a function $m : \Sigma^* \rightarrow [0, 1]$ *converges* against a value $x \in \mathbb{R}$ with respect to a given sequence of finite strings $(u_n)_{n \in \mathbb{N}}$, if and only if each ϵ -neighbourhood of x contains almost every member of the sequence $(m(u_n))_{n \in \mathbb{N}}$. We then also write

$$\lim_{n \rightarrow \infty} m(u_n) = x.$$

For an ω -word $\alpha = \alpha_1 \alpha_2 \dots$, let the *sequence of prefixes* of α be

$$\left(\alpha_{(n)} \right)_{n \in \mathbb{N}_0} = \lambda, \alpha_1, \alpha_1 \alpha_2, \alpha_1 \alpha_2 \alpha_3, \dots .$$

The characteristic function for any language L is defined as usual, i.e.

$$\mathbb{1}_L(\alpha) = \begin{cases} 1, & \text{if } \alpha \in L \\ 0, & \text{if } \alpha \notin L . \end{cases}$$

Definition 3.5 (Monitor). *Given an ω -language $L \subseteq \Sigma^\omega$, let the class of continuous monitors for L be*

$$\mathfrak{M}(L) = \left\{ m : \Sigma^* \rightarrow [0, 1] \mid \forall \alpha \in \Sigma^\omega : \lim_{n \rightarrow \infty} m(\alpha_{(n)}) = \mathbb{1}_L(\alpha) \right\}.$$

That is, following the prefixes of a word α , the values of any monitor for L will finally stay arbitrarily close to 1 if L contains α and to 0 otherwise.

We now examine several desirable properties of monitor functions, such as accuracy and different notions of monotonicity.

3.3.1. Prefix monotonic monitors

The first class we study contains languages for that monitors exist which are monotonic with respect to prefixes.

Definition 3.6 (Prefix monotonic monitors). *A function $m : \Sigma^* \rightarrow [0, 1]$ is monotonic with respect to the proper prefix relation $< \subseteq \Sigma^* \times \Sigma^*$ (prefix monotonic) if*

$$\begin{aligned} & (\forall u_1, u_2 \in \Sigma^* : u_1 < u_2 \Rightarrow m(u_1) \leq m(u_2)) \\ & \vee (\forall u_1, u_2 \in \Sigma^* : u_1 < u_2 \Rightarrow m(u_1) \geq m(u_2)). \end{aligned}$$

The class of languages, that can be monitored by prefix monotonic functions is then

$$\mathcal{K}_{pm} := \{L \subseteq \Sigma^\omega \mid \exists m \in \mathfrak{M}(L) : m \text{ is prefix monotonic}\}.$$

Intuitively, for a word α , these functions can only once choose a direction and then have to follow it, as growing prefixes can only all get „better“ or all get „worse“.

Theorem 3.1. *The class \mathcal{K}_{pm} of languages monitorable by prefix monotonic functions is exactly the union of SAFETY and CO-SAFETY.*

Proof.

1. SAFETY \cup CO-SAFETY $\subseteq \mathcal{K}_{pm}$. Let $L \in \text{SAFETY} \cup \text{CO-SAFETY}$. If L is safe, then a prefix monotonic monitor is

$$m_L : \Sigma^* \rightarrow [0, 1], u \mapsto \begin{cases} 0 & \text{iff. } u \text{ is bad prefix} \\ 1 & \text{otherwise.} \end{cases}$$

Note, since L is safe, any ω -word not in L has a (finite) bad prefix. Therefore m converges against the right value as it stays at 1 or switches to 0 if the exclusion is evident. m also switches at most once and is thus monotonic.

If L is not safe then L is necessarily a co-safety language. To obtain a monitor in that case we only change the a priori value to 0 and switch to 1 when a good prefix is observed.

$$m_L : \Sigma^* \rightarrow [0, 1], u \mapsto \begin{cases} 1 & \text{iff. } u \text{ is good prefix} \\ 0 & \text{otherwise} \end{cases}$$

2. SAFETY \cup CO-SAFETY $\supseteq \mathcal{K}_{pm}$. To verify that also each language in \mathcal{K}_{pm} is safe or co-safe let m be a prefix monotonic monitor for $L \in \mathcal{K}_{pm}$. We derive from the monotonicity of m that at least each word in L has a good or each word not in L a bad prefix. We split the cases for the monitor value of the empty word $m(\lambda)$.

If $m(\lambda)$ is neither 0 nor 1, the monitor has to “chose a direction” at some (finitely reachable) point. Then the prefix monotonicity prevents m from going back again and all infinite extensions have to share the same membership property (see Figure 3.1 for an example). Thus, every word then has ether a good or a bad prefix and L is both, safe and co-safe.

If $m(\lambda)$ takes 0, then, for any $\alpha \in L$, m_L has eventually to increase its value for a finite prefix v . After increasing, the monotonicity prohibits to ever reach 0 again for any extension of v . Thus every member of L must has such a good prefix and L is thus co-safe. On the contrary, if $m(\lambda)$ takes 1, any $\beta \notin L$ has to have a bad prefix and L must at least be safe. \square

Interestingly, if we restrict the class \mathcal{K}_{pm} even further by not allowing values 0 or 1 for the empty word, we obtain exactly the intersection of SAFETY and CO-SAFETY. Moreover, considering “accuracy” in terms of good and bad prefixes still yields the same class.

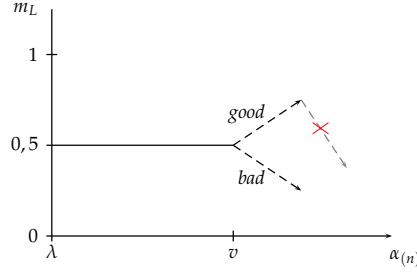


Figure 3.1.: Assume, for example, $\alpha \in L$. m converges to 1 and has therefore to increase its initial value after some finite prefix v . v then is already a good (respectively bad) prefix since every extension has v as prefix and cannot reach the value 0 anymore.

Definition 3.7 (Accuracy). Let $L \subseteq \Sigma^\omega$ an ω -language. Then the class of accurate monitors for L is

$$\mathfrak{M}_{\text{acc}}(L) := \{m \in \mathfrak{M}(L) \mid \forall u \in \Sigma^* : m(u) = 1 \text{ iff. } \vec{u} \subseteq L \\ \wedge m(u) = 0 \text{ iff. } \vec{u} \cap L = \emptyset\}.$$

Theorem 3.2 (Characterization of SAFETY \cap CO-SAFETY). For an ω -language $L \subseteq \Sigma^\omega$, the following characterizations are equivalent

1. $L \in \text{SAFETY} \cap \text{CO-SAFETY}$
2. $\exists m \in \mathfrak{M}(L) : m$ is prefix monotonic and $0 < m(\lambda) < 1$
3. $\exists m \in \mathfrak{M}_{\text{acc}}(L) : m$ is prefix monotonic

Proof.

1. \Leftrightarrow 2. From the previous proof of Theorem 3.1, it is evident, that each language L is safe and co-safe if there is a prefix monotonic monitor and we exclude the second case. Furthermore, for all languages $L \in \text{SAFETY} \cap \text{CO-SAFETY}$ that are safe and co-safe

$$m_L : \Sigma^* \rightarrow [0,1], u \mapsto \begin{cases} 1 & \text{iff. } u \text{ is good prefix and } |u| > 0 \\ 0 & \text{iff. } u \text{ is bad prefix and } |u| > 0 \\ 0.5 & \text{otherwise} \end{cases}$$

is clearly a monitor since a good or bad prefix must eventually occur. Note, that m is not accurate in general. If λ is already good or bad, m would only evaluate to 1 or 0, respectively, for words of a minimal length of 1. This, however, does not effect a violation of the convergence property.

1. \Leftrightarrow 3. If m is accurate for a language L , then $m(\lambda) = 1$ or $m(\lambda) = 0$ implies $L = \Sigma^\omega$ or $L = \emptyset$ and thus $L \in \text{SAFETY} \cap \text{CO-SAFETY}$. In any other case, as

before, the prefix monotonicity yields membership in SAFETY \cap CO-SAFETY. For the only-if-part, we consider the accurate and prefix monotonic monitor

$$m_L : \Sigma^* \rightarrow [0, 1], u \mapsto \begin{cases} 1 & \text{iff. } u \text{ is good prefix} \\ 0 & \text{iff. } u \text{ is bad prefix} \\ 0.5 & \text{otherwise.} \end{cases}$$

□

3.3.2. Distinguishing inconclusive prefixes

As we saw, prefix monotonic monitors can not make use of the extended truth domain but only of three different values. The following approach to determine a more detailed evaluation of an actual inconclusive verdict is to propose an ordering relation on finite words that can be seen to distinguish between more and less “promising” prefixes with respect to fulfilling a property. It helps us, to quantify the relation between prefixes and languages.

Future aspect

For assessing prefixes with respect to a given language, we want to take into account, what we already can assure about all possible extensions (i.e. the “future”). The most obvious criterion in that sense is whether the membership of all extensions is already approved or disapproved, i.e. if we face a good or a bad prefix.

Furthermore, we want to measure the smallest number of letters, that need to be appended to a word in order to reach a good or bad prefix reachable. A word is considered the better the closer, in that sense, the next good prefix can be found (the “good future”) and we loose confidence the closer a bad prefix appears (the “bad future”). We use these two measures to compare finite prefixes in terms of a given ω -language.

Definition 3.8 (Future aspect). *Let $L \subseteq \Sigma^\omega$ be an ω -language and $u \in \Sigma^*$ a finite word. The distance to the shortest good prefix for L that is an extension of u is*

$$\text{dist}_g^L(u) := \min \{ |v| \mid v \in \Sigma^*, \overrightarrow{uv} \subseteq L \} \cup \{ \infty \}$$

and distance to the shortest bad prefix for L that is an extension of u is

$$\text{dist}_b^L(u) := \min \{ |v| \mid v \in \Sigma^*, \overrightarrow{uv} \cap L = \emptyset \} \cup \{ \infty \}.$$

The future aspect of a prefix u with regard to L is the difference

$$f(u) := \text{dist}_b^L(u) - \text{dist}_g^L(u).$$

Note that, if there is no finite extension of a word u to a good or bad prefix, the respective distance is infinite. For easier reading we omit the L if the language is fixed in the context.

Comparing, for a language L , the future aspects of two prefixes we can define a relation $<_f \subseteq \Sigma^* \times \Sigma^*$ such that

$$\forall u, v \in \Sigma^* : u <_f v \text{ iff } f(u) < f(v) .$$

Past aspect

When extensions do not yield any good reason for distinguishing between two words we want to assess the past behaviour. Although it is hard to formalize a notion of “attitude”, it appears natural to examine the past behaviour of a system and reason in an inductive manner.¹ We want therefore also take past aspects into account.

Let, for $L \subseteq \Sigma^\omega$, $n \in \mathbb{N}_0$,

$$\text{Good}^L(n), \text{Bad}^L(n) \subseteq \Sigma^n$$

be the sets of good and bad prefixes, respectively, for L with length n . For a word $u \in \Sigma^*$ that is not yet conclusive we can interpret

- $\text{Good}^L(|u|)$ as “missed chances” and
- $\text{Bad}^L(|u|)$ as “avoided failures”

that occurred while observing u . Analogously to the future aspect we define the past.

Definition 3.9 (Past aspect). *Let $L \subseteq \Sigma^\omega$, $u \in \Sigma^*$ and $\text{Good}^L(n)$, $\text{Bad}^L(n)$ defined as above. The past aspect of u regarding L is*

$$p(u) := |\text{Bad}^L(|u|)| - |\text{Good}^L(|u|)|.$$

Again, we omit to write the L explicitly if the context is clear and define a relation $\leq_p \subseteq \Sigma^* \times \Sigma^*$ such that

$$\forall u, v \in \Sigma^* : u \leq_p v \text{ iff } p(u) \leq p(v) .$$

¹This is a common technique, for example, in machine learning. Also, learning of formal languages and their respective automata is an acknowledged field of current research, cf. for example [ELS10], [BKKL10].

Comparing prefixes by future and past

Now that we developed a notion to compare prefixes according to how promising they appear, we combine the aspects to form a single relation between finite words, depending on a given ω -language. This is used to formulate a monotonicity restriction for monitor functions as was done in Section 3.3.1 based on the prefix relation.

Definition 3.10 (Future-past relation). *For an ω -language $L \subseteq \Sigma^\omega$, the future-past relation with respect to L , $\leq_L \subseteq \Sigma^* \times \Sigma^*$, is a relation on finite words, such that, for $u, v \in \Sigma^*$, $u \leq_L v$ if and only if*

$$u <_f v \text{ or } u =_f v \wedge u \leq_p v.$$

Given L , a monitor function that is monotonic with respect to \leq_L (*fp-monotonic*) will yield a higher value to more promising words. Desirable is furthermore that it is accurate and meets our general limit condition for monitors.

The question arises, which languages can be monitored by functions underlying these restrictions, that is, when does such a monitor exist and how can it be constructed. We will define and characterize this class in the following.

Definition 3.11 (Fp-monitorable languages). *An ω -language L is called fp-monitorable if there is a monitor for L that is accurate and monotonic with respect to future and past (fp-monotonic). The class of fp-monitorable languages is denoted by*

$$\mathcal{K}_{\text{fp}} := \{L \subseteq \Sigma^\omega \mid \exists_{m \in \mathcal{M}_{\text{acc}}(L)} \forall_{u, v \in \Sigma^*} : u \leq_L v \Rightarrow m(u) \leq m(v)\}.$$

Theorem 3.3. *The class \mathcal{K}_{fp} is strictly included in the class $\overline{\text{UGLY}}$ of ugly-free languages.*

Proof. Let $L \in \mathcal{K}_{\text{fp}}$ be an fp-monitorable language and m an according monitor. We proof by contradiction that L cannot have an ugly prefix (cf. Definition 3.3).

Assume there were an ugly prefix u for L . Let $\alpha, \beta \in \overrightarrow{u}$ be infinite extensions of u such that $\alpha \in L$ and $\beta \notin L$. Since m is a monitor for L , $\lim_{n \rightarrow \infty} m(\alpha_{(n)}) = 1$ and $\lim_{n \rightarrow \infty} m(\beta_{(n)}) = 0$. That is, for any ε -neighbourhood of 1 or 0, respectively, there are only finitely many members of the prefix-value sequence that are excluded. Thus, for α

$$\forall_{\varepsilon > 0} \exists_{n \in \mathbb{N}} \forall_{i > n} : 1 - m(\alpha_{(i)}) < \varepsilon.$$

Consequently, there must be an infinite subsequence

$$v_1, v_2, v_3, \dots$$

of $\alpha_{(1)}, \alpha_{(2)}, \alpha_{(3)}, \dots$, such that $m(v_1) < m(v_2) < m(v_3) < \dots$. For $v \leq_L v' \Rightarrow m(v) \leq m(v')$ is equivalent to $m(v') < m(v) \Rightarrow v' <_L v$, the sequence is also strictly increasing in terms of the relation $<_L$:

$$v_1 <_L v_2 <_L v_3 <_L \dots$$

Since u is ugly, neither a good nor a bad prefix can be reached thereafter.

$$\forall w \in u\Sigma^* : \text{dist}_b(w) = \text{dist}_g(w) = \infty$$

Particularly, the future aspect of all prefixes of α and β behave the same (f always evaluates to $\infty - \infty = 0$ after u). Furthermore the past aspect does not depend on the subtree under u and is therefore also equal for α and β . Hence,

$$\forall i \in \mathbb{N} : m(\alpha_{(i)}) = m(\beta_{(i)})$$

which violates the convergence property of m and therefore contradicts the assumption that an ugly prefix for L exists. \square

The future and past aspect operate on \mathbb{Z}_∞ and therefore we will make use of a projection to our intended monitor domain $[0,1]$. For the following, let

$$\pi : \mathbb{N}_\infty \rightarrow [0,1]$$

be a strictly monotonic injection such that $\pi(0) = 0$ and $\pi(\infty) = 1$. As an example, we can imagine the mapping

$$x \mapsto 1 - \frac{1}{1+x} \quad (x \in \mathbb{N}_0)$$

and to cover ∞ , we can use the limit, i.e.

$$\infty \mapsto \lim_{x \rightarrow \infty} 1 - \frac{1}{1+x} = 1.$$

Theorem 3.4. *The intersection of SAFETY and CO-SAFETY is (strictly) included in \mathcal{K}_{fp} .*

Proof. Let $L \in \text{SAFETY} \cap \text{CO-SAFETY}$. We can define a monitor $m_{\text{sc}} : \Sigma^* \rightarrow [0,1]$ for L by

$$m_{\text{sc}}(u) := \pi(\exp(f(u)) + \pi(\exp(p(u))) \cdot \exp(f(u))) \quad (3.1)$$

Note that f and p can take values $\pm\infty$. In that case we want to consider the limit, for example $\exp(x)$ shall yield 0 for $x = \infty$. Then $\pi(\exp(x) + c \exp(y))$ yields 1 for $x = \infty$ or $y = \infty$ (or both) and 0 for $x = y = -\infty$.

We first verify that m_{sc} is indeed a monitor for L . Let $\alpha \in L$. Since L is *co-safe*, α has a *good prefix* of finite length n . The future of any good prefix u evaluates to $f(u) = \text{dist}_b(u) - \text{dist}_g(u) = \infty - 0 = \infty$, hence the right hand side of Equation (3.1) evaluates to 1. Therefore $\forall k > n : m_{\text{sc}}(\alpha_{(k)}) = 1$. Considering any *bad prefix* v , $f(v)$ evaluates to $f(v) = \text{dist}_b(v) - \text{dist}_g(v) = 0 - \infty = -\infty$ and therefore (3.1) to 0. For any ω -word $\beta \notin L$, there is such a bad prefix of finite length m since L is *safe* and, by concluding that $\forall k > m : m_{\text{sc}}(\beta_{(k)}) = 0$, we see that m_{sc} satisfies our Definition 3.5 of a monitor with respect to L .

m_{sc} is also accurate since, for $u \in \Sigma^*$,

$$\begin{aligned} m_{\text{sc}}(u) = 1 &\Leftrightarrow f(u) = \infty \\ &\Leftrightarrow \text{dist}_{\text{b}}(u) = \infty && \text{(a)} \\ &\Leftrightarrow u \text{ is a good prefix for } L. && \text{(b)} \end{aligned}$$

Setting $f(u)$ to ∞ is the *only* way to evaluate the monitor value to 1, and that can only be achieved by rising the distance from u to a bad prefix to infinity. Note, that no bad prefix is reachable after u , if and only if all infinite extensions of u are included in L , i.e. u is a good prefix itself, since L is safe (b). L is also co-safe, hence all of these extensions have good prefixes and $\text{dist}_{\text{g}}(u)$ cannot be infinite, which yields the if-part of (a). An analogous argument applies for a monitor value of 0.

To see that m_{sc} is monotonic in terms of the definition of \mathcal{K}_{fp} (Definition 3.11) let us consider two finite words $u, v \in \Sigma^*$ such that $u \leq_L v$ and show that $m_{\text{sc}}(u) \leq m_{\text{sc}}(v)$ by splitting the following two cases.

1. For $f(u) < f(v)$, we approximate the following subterm of m_{sc} .

$$\begin{aligned} &\exp(f(u)) + \pi(\exp(p(u))) \cdot \exp(f(u)) \\ &\leq \exp(f(u)) + 1 \cdot \exp(f(u)) && \text{The factor } \pi \text{ can at most be } \\ &&& \text{1, i.e. when } p(u) = \infty. \\ &< \exp(f(u) + 1) && \text{Note that } f(u) \text{ cannot be } \\ &&& \infty. \\ &\leq \exp(f(v)) + \pi(\exp(p(v))) \cdot \exp(f(v)) && \text{Equality for } f(u) = f(v) - \\ &&& \text{1 and } p(v) = -\infty. \end{aligned}$$

Thus, $m_{\text{sc}}(u)$ is also (strictly) smaller than $m_{\text{sc}}(v)$.

2. Let $f(u) = f(v)$. If their value is finite $p(u) \leq p(v) \Rightarrow m_{\text{sc}}(u) \leq m_{\text{sc}}(v)$ since m_{sc} is strictly monotonic in p . Infinite values $\pm\infty$ for f evaluate m_{sc} equally to 1 or 0, respectively, regardless of p . Therefore $u \leq_L v \Rightarrow m_{\text{sc}}(u) \leq m_{\text{sc}}(v)$ still holds and m_{sc} is monotonic. \square

Remark. For $\Sigma = \{a, b\}$ we observe witnesses for the following non-empty classes

$$a\Sigma^\omega \in \text{SAFETY} \cap \text{CO-SAFETY}$$

$$\overline{L_1} \in \text{SAFETY} - \text{CO-SAFETY} - \mathcal{K}_{\text{fp}}$$

$$L_1 \in \text{CO-SAFETY} - \text{SAFETY} - \mathcal{K}_{\text{fp}}$$

$$L_3 \in \mathcal{K}_{\text{fp}} - \text{CO-SAFETY} - \text{SAFETY}$$

$$a^\omega \in (\mathcal{K}_{\text{fp}} \cap \text{SAFETY}) - \text{CO-SAFETY}$$

$$\overline{a^\omega} \in (\mathcal{K}_{\text{fp}} \cap \text{CO-SAFETY}) - \text{SAFETY}$$

$$L_2 \in \overline{\text{UGLY}} - \text{SAFETY} - \text{CO-SAFETY} - \mathcal{K}_{\text{fp}}$$

L_1 denotes the language $\mathcal{L}(aa^*ba\Sigma^\omega)$ (see Figure 3.2). The word a^ω is the only word, that has neither a good nor a bad prefix. Since it is included in L_1 , the language is co-safe but not safe and its complement $\overline{L_1}$ is safe but not co-safe. a^ω also causes neither of them to be included in \mathcal{K}_{fp} . Following the prefixes of a^ω , the future aspect remains constant whereas the past increases for L_1 , preventing any assumed (monotonic) monitor from ever decreasing, as it should. For $\overline{L_1}$ the past decreases and the prefix sequence of a^ω gets worse even though the word is included.

The language L_3 (see Figure 3.3) strictly separates \mathcal{K}_{fp} from SAFETY and CO-SAFETY. It is obviously neither safe nor co-safe. Considering the monitor m_{sc} , used to proof Theorem 3.4 we confirm that L_3 is indeed included in \mathcal{K}_{fp} by investigating the only two words that do not have a bad or good prefix: $a^\omega \in L_3$ and $b^\omega \notin L_3$. For the the prefix sequence of a^ω , the future aspect increases infinitely, causing m to converge to 1. Along the the prefix sequence of $b^\omega \notin L_3$, the future aspects decrease infinitely, causing m to converge to 0. Note, that L_3 is not ω -regular as can be seen by a pumping argument.

Switching the membership of a^ω and b^ω we obtain $L_2 = (L_3 - \{a^\omega\}) \cup \{b^\omega\}$. Using a similar argument to the one for L_1 , it is evident, that L_2 does not belong to \mathcal{K}_{fp} and is still not safe or co-safe. It is, however, ugly-free. For this class we can also give ω -regular examples, e.g. $L_2' = \mathcal{L}(\Box a \vee \Diamond(b \wedge Xb))$.

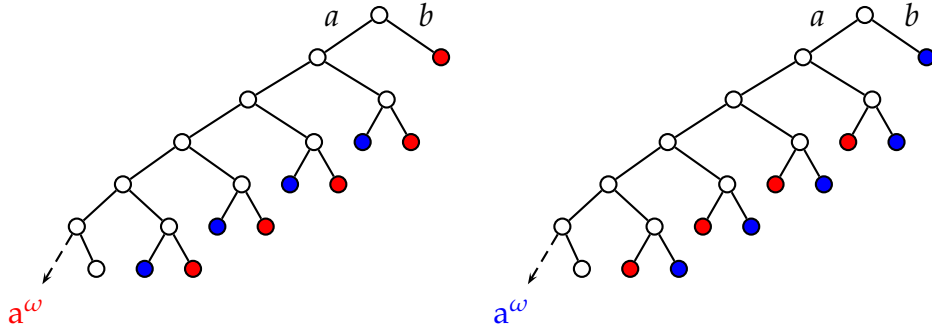


Figure 3.2.: Sketch of the language $L_1 = \mathcal{L}(aa^*ba\Sigma^\omega)$ (l) and its complement (r). Nodes represent finite words, starting at the root with the empty word λ . Appending letter a branches to the left, letter b to the right. Red nodes indicate bad, blue nodes good prefixes. Note that a^ω is not included in L_1 .

3.3.3. Computing future and past aspects

The aspect functions f and p are essential for the definitions and characterization we made. Therefore, we give an approach to compute them, using a given language specification. We restrict here to languages, that can be specified by Büchi automata,

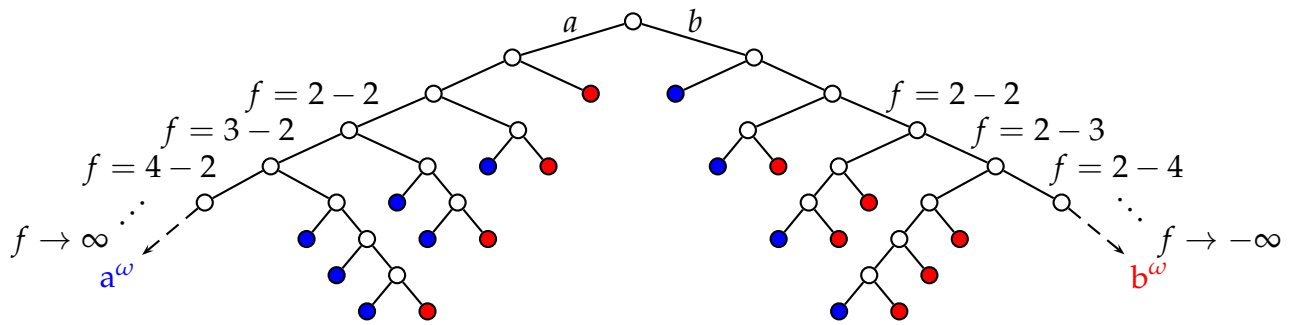


Figure 3.3.: Sketch of the language $L_3 = (a\Sigma^\omega \setminus \bigcup_{n \in \mathbb{N}} a^n b^n \Sigma^\omega) \cup (\bigcup_{n \in \mathbb{N}} b^n a^n \Sigma^\omega)$. Note, that b^ω is not included. Blue and red nodes indicate good and bad prefixes, respectively. For some prefixes of a^ω and b^ω the future aspect f is outlined.

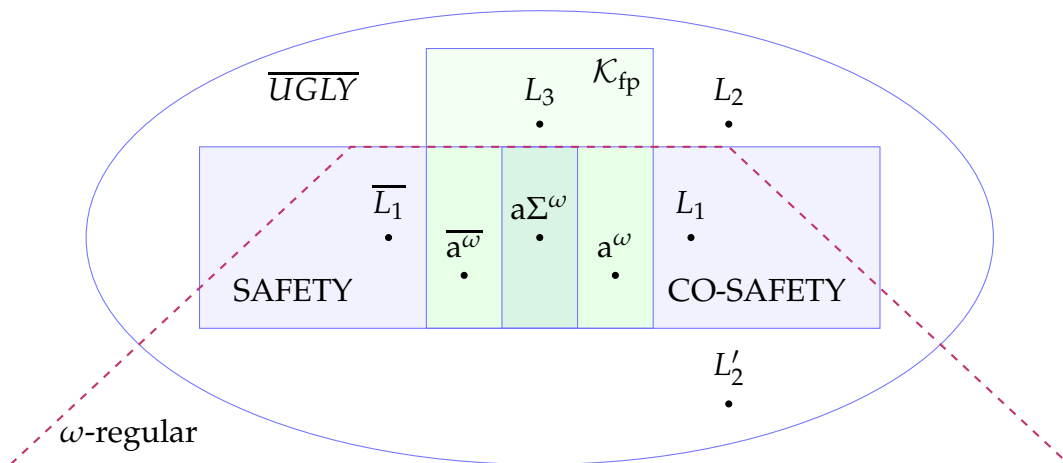


Figure 3.4.: Overview of the relation between \mathcal{K}_{fp} and other classes of ω -languages.

i.e. that are ω -regular. Recall, however, that this covers not the whole class (cf. Figure 3.4).

We keep the framework of automata and construct a Moore machine that outputs the future aspect for a given finite input word and a weighted automaton for computing the past.

In the following, we assume to have the \mathbb{B}_3 -monitor $\mathcal{M} = (Q, q_0, \delta, \lambda)$ for a property $L \in \Sigma^\omega$ under observation. As we saw in Section 3.2, it can be obtained e.g. from Büchi automata.

Computing $f(u)$

For a given ω -language L , f_g and f_b measure the distance of a given finite word to the closest good and bad continuation, respectively. The \mathbb{B}_3 -monitor \mathcal{M} gives a verdict for each of its states that is either \perp , \top or $?$. Since we can assume the verdict accurate (maxims of impartiality and anticipation) and each edge is labeled with one letter, the distance to the closest state yielding \perp or \top is also the exact minimal distance in terms of words to a bad respectively good prefix.

For every state $q \in Q$ we therefore can do an a priori breadth first search on the automaton graph of \mathcal{M} to obtain the distances $d_{\text{bad}}, d_{\text{good}}$ to the closest bad state q_\perp (i.e. $\lambda(q_\perp) = \perp$) and the closest good state q_\top (i.e. $\lambda(q_\top) = \top$). The distances are used to define a new labeling function $\lambda_{\mathcal{F}} : Q \rightarrow \mathbb{Z}_\infty$,

$$\lambda_{\mathcal{F}}(q) := d_{\text{bad}}(q) - d_{\text{good}}(q),$$

where d_{bad} and d_{good} evaluate to ∞ if no such state is reachable.

This gives us a Moore machine $\mathcal{F} = (Q, q_0, \delta, \lambda_{\mathcal{F}})$ to keep track of the distances while new information (letters) arrives.

Computing $p(u)$

Next we construct a weighted automaton for computing the past aspect. A weighted automaton is, similar to classical FSMs, a tuple representing a graph augmented with weight functions over a semiring and a semantics on finite words over a fixed alphabet.

Definition 3.12 (Weighted automaton [DVK09, Section 2.1]). *A weighted automaton over a semiring $\mathbb{S} = (S, \oplus, \odot, \mathbf{0}, \mathbf{1})$ and a finite alphabet Δ is a structure $\mathcal{A} = (Q, \eta, \mu, \gamma)$ where*

- Q is the nonempty finite set of states,
- $\eta : Q \rightarrow S$ is the initial-weight function,
- $\mu : Q \times \Sigma \times Q \rightarrow S$ is the transition-weight function and
- $\gamma : Q \rightarrow S$ is the final-weight function.

The semantics of \mathcal{A} is a function $\llbracket \cdot \rrbracket : \Sigma^* \rightarrow \mathbb{S}$ (a so called formal power series). Given a word $w = w_1 w_2 \dots w_n \in \Sigma^*$, all possible paths that follow the transition-weight function μ along w are weighted and summed up, i.e

$$\llbracket \mathcal{A} \rrbracket(w) := \bigoplus_{(q_0, \dots, q_n) \in Q^{n+1}} \eta(q_0) \odot \mu(q_0, w_1, q_1) \odot \dots \odot \mu(q_{n-1}, w_n, q_n) \odot \gamma(q_n).$$

The values of p are integers, therefore we use the semiring $(\mathbb{Z}, +, \cdot, 0, 1)$ for our purpose.

Now, given the \mathbb{B}_3 -monitor \mathcal{M} we construct a weighted automaton $\mathcal{P} = (Q, \eta, \mu, \gamma)$, keeping the original set of states Q . Since only the length of a word matters here, we use an alphabet consisting of only one letter and thus omit to specify it explicitly.

We use the initial-weight function η to restrict the considered paths to those beginning with the initial state.

$$\eta(q) := \begin{cases} 1 & \text{if } q = q_0 \\ 0 & \text{otherwise} \end{cases}$$

For the past aspects we need to count the exact number of paths of a given length, regardless of their label. Therefore, the transition-weight function μ is set to the number of edges in δ between two states. Note that invalid paths in \mathcal{M} will then contain at least one step weighted 0, hence annihilating the whole path value.

$$\mu(q, q') := |\{a \in \Sigma \mid (q, a, q') \in \delta\}|$$

As $p(u) = |\text{Bad}(|u|)| - |\text{Good}(|u|)|$, we can obtain p by summing up each valid path, weighted with 1, if it ends in a bad state and -1 if it ends in a good state. We do not count inconclusive states.

$$\gamma(q) = \begin{cases} 1 & \text{if } \lambda(q) = \perp \\ -1 & \text{if } \lambda(q) = \top \\ 0 & \text{if } \lambda(q) = ? \end{cases}$$

From this construction we see, that the semantics of the automaton \mathcal{P} is exactly the difference between the number of good and bad prefixes of the length of the input word.

Weighted automaton lack, in general, the advantage of iterative processing of an input word as done by, for example, Moore machines. That is, given a word of w and the state of the FSM after reading w , it just requires a quick computation (e.g. a look up in the transition table) to obtain the state after reading the by one letter extended word wa .

In our case, however, we fortunately can still compute the semantics of the automaton \mathcal{P} inductively, keeping track of some constant additional information.

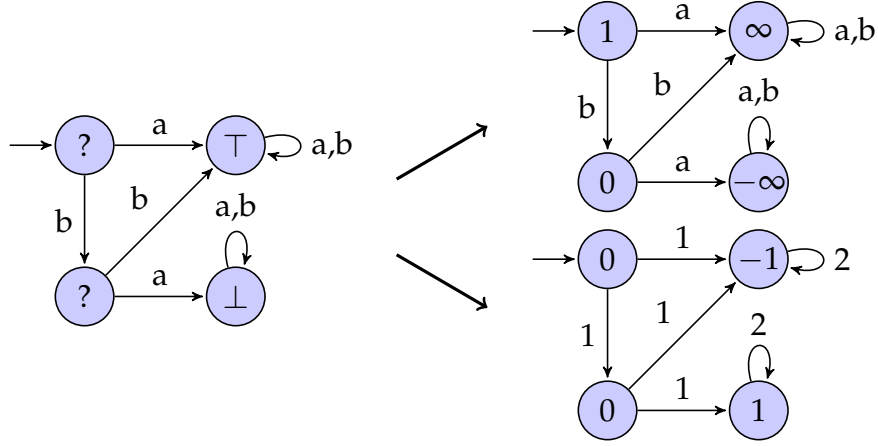


Figure 3.5.: Example of a \mathbb{B}_3 -monitor \mathcal{M} (left) and the resulting automata \mathcal{F} (top) and \mathcal{P} (bottom). States q are labeled with their output $\lambda(q)$ or their final-weight $\gamma(q)$, respectively. For the weighted automaton \mathcal{P} , edges with weight more then 0 are shown and accordingly labelled with μ .

Let k be the number of states in $Q = \{q_0, \dots, q_{k-1}\}$. The semantics of a weighted automaton for a word $w = w_1 w_2 \dots w_n$ can be computed using a matrix representation

$$\llbracket \mathcal{A} \rrbracket(w) = \eta \underline{\odot} M(w_1) \underline{\odot} \dots \underline{\odot} M(w_n) \underline{\odot} \gamma \quad (3.2)$$

where η and γ ,

$$\begin{aligned} \eta &= (\eta(q_0) \quad \eta(q_1) \quad \dots \quad \eta(q_{k-1})) \\ \gamma &= (\gamma(q_0) \quad \gamma(q_1) \quad \dots \quad \gamma(q_{k-1}))^T, \end{aligned}$$

are, respectively, the row vector of all inital-weights and the column vector consisting of all final-weights. $M(w_i)$ is the $(Q \times Q)$ -matrix given by $M(a)_{q,q'} = \mu(q, a, q')$. Here, $\underline{\odot}$ is the matrix operation according to \odot .

Since we use only one letter as alphabet, $M(w_i)$ is constant, thus

$$M(w_1) \underline{\odot} \dots \underline{\odot} M(w_n) = M^{|w|}.$$

We extend γ to the inital-weight *matrix* Γ of the form

$$\left(\begin{array}{c|cccc} \gamma_0 & 0 & 0 & \dots & 0 \\ \gamma_1 & 1 & 0 & \dots & 0 \\ \vdots & 0 & 1 & & \vdots \\ & \vdots & & \ddots & 0 \\ \gamma_{k-1} & 0 & \dots & 0 & 1 \end{array} \right)$$

where the first line on the right-hand side, consisting of only zeros, is swapped with the first row i , where $\gamma_i \neq 0$. That way all columns are linearly independent and Γ is thus invertable in our setting since $(\mathbb{Z}, +, \cdot, 0, 1)$ is a commutative ring.

Note that this row must exist. Otherwise the \mathbb{B}_3 -monitor \mathcal{M} would have only inconclusive states and L would thus not be monitorable.

While Equation 3.2 results in a scalar, we obtain a vector by extending γ to the matrix Γ . This can be seen as a constant information overhead that we have to keep for the iterative process as we have to keep track of the current state for FSMs.

$$\llbracket \mathcal{A} \rrbracket_{\text{it}}(w) := \eta \odot M^{|w|} \odot \Gamma.$$

However, the current (scalar) semantics value of the automaton can be identified easily as it is always the first entry, i.e.

$$\llbracket \mathcal{A} \rrbracket(w) = \pi_1^k(\llbracket \mathcal{A} \rrbracket_{\text{it}}(w)).$$

We achieve an iterative procedure with a single matrix multiplication per step for computing $\llbracket \mathcal{A} \rrbracket_{\text{it}}(wa)$. Let $M_\Gamma := \Gamma^{-1}M\Gamma$ be the transition-weight matrix conjugated with the final-weight matrix. Then

$$\begin{aligned} \llbracket \mathcal{A} \rrbracket_{\text{it}}(wa) &= \eta \odot M^{|w|+1} \odot \Gamma \\ &= \eta \odot M^{|w|} \odot \Gamma \odot \Gamma^{-1}M\Gamma \\ &= \llbracket \mathcal{A} \rrbracket_{\text{it}}(w) \odot M_\Gamma. \end{aligned}$$

3.3.4. Choosing a monitor function

After characterizing the class \mathcal{K}_{fp} of languages for which an fp-monotonic monitor function *exists* it is of interest how such a monitor can be obtained explicitly. We also saw that automata can be used to compute the aspect functions f and p and will investigate now how to assemble them to obtain a quantitative monitor for ω -regular languages.

Lemma 3.1. *Let $\text{REG}_{\text{fp}} = \mathcal{K}_{\text{fp}} \cap \text{REG}_\omega$ be the intersection of fp-monotonically monitorable and ω -regular languages. Then REG_{fp} is included in the union of safety and co-safety languages.*

$$\text{REG}_{\text{fp}} \subseteq \text{SAFETY} \cup \text{CO-SAFETY}$$

Proof. Let $L \in \text{REG}_{\text{fp}}$ be an ω -regular language and m an fp-monotonic monitor for L . We consider the case, where L is not safe and derive that L is necessarily co-safe. The argumentation applies analogously for the other case, where L is not co-safe.

For $L \notin \text{SAFETY}$ there is an ω -word $\beta \notin L$ without bad prefix. m has to converge to 0 for the sequence of prefixes of β and thus, as argued earlier, there must be a subsequence $v_1 <_L v_2 <_L v_3 <_L \dots$.

The regularity of L yields the existence of a Büchi automaton accepting L and, using the construction outlined in Section 3.2, the \mathbb{B}_3 -monitor \mathcal{M} for L . We observe that the future aspect f cannot decrease infinitely since the maximal length of any path in the automaton graph of \mathcal{M} is bounded by the (finite) number of states. The

function f has therefore to become constant after some length l and thus the value for the past aspect p has to strictly decrease thereafter.

Since the past aspect depends only of the length of the observation, p behaves the same for all words. If there were a word $\alpha \in L$ without good prefix, again, the future aspect would have to become constant at some point whereas always eventually the past aspect decreases strictly, preventing the monitor m from approaching to 1. This contradicts the assumption that such an α exists and L is thus co-safe. \square

To choose an appropriate monitor we classify a given language $L \in \mathcal{K}_{\text{fp}}$ into one of the cases where L is

1. safe and co-safe,
2. either safe or co-safe and
 - a) only one type of prefixes (good or bad) exists,
 - b) both, good and bad prefixes, exist.

Lemma 3.1 verifies that these cases are exhaustive.

For the first case it is easy to check if a language is both, safety and co-safety. The corresponding \mathbb{B}_3 -monitor must be a directed, acyclic graph connected to two capture states, the one yielding \top and the other yielding \perp . Recall that the proof of Theorem 3.4 already used an appropriate monitor

$$m_{\text{sc}} := \pi(\exp(f(u)) + \pi(\exp(p(u))) \cdot \exp(f(u)))$$

for all languages of that class.

This monitor is, however, in general not suitable for non-safe or non-co-safe languages since it evaluates to 1 as soon as no bad prefix is reachable any more. Consider, for example, the language $\{a^\omega\}$ where $f(a) = \infty$ and hence $m(a) = 1$ even though a is not a good prefix.

It remains to distinguish between the other two types. If a language provides no good prefixes or no bad prefixes at all (see Figure 3.6 for an example) it can be monitored by only considering the past aspects since the future is constantly infinite ($f = \pm\infty$). Recall that no language in \mathcal{K}_{fp} has an ugly prefix, thus dist_b and dist_g cannot both be infinite at the same time. A quantitative monitor then is

$$m_p(u) = \begin{cases} 0 & \text{if } \text{dist}_g(u) = 0 \\ 1 & \text{if } \text{dist}_b(u) = 0 \\ \pi(\exp(p(u))) & \text{otherwise.} \end{cases}$$

We see that m_p respects good/bad prefixes and assures compatibility with the future-past relation $<_{\text{fp}}$ as long, as only one type of conclusive prefix is reachable. Assuming the language is indeed monitorable in terms of \mathcal{K}_{fp} , it suffices to check whether the according \mathbb{B}_3 -monitor only has one conclusive state (i.e yielding either \top or \perp) in order to see if this monitor is applicable.

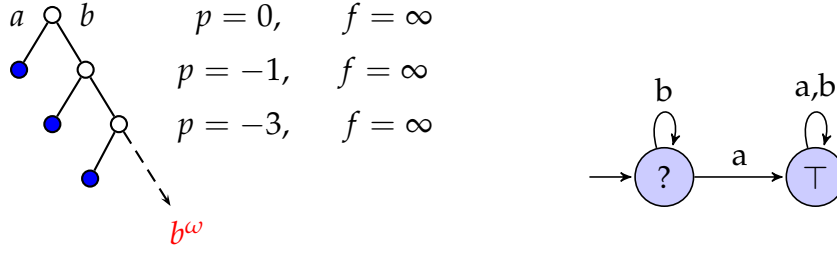


Figure 3.6.: The language $\mathcal{L}(\diamond a)$ has only good prefixes and the future aspect stays thus constantly infinite. The according \mathbb{B}_3 -monitor thus has just one conclusive state, which yields \top .

To cover the last type of languages (case 2b) we need to examine their structure to a further extend. Lemma 3.2 provides a structural property for these languages, a threshold k for the future aspect f .

Lemma 3.2. *Let $L \in \text{REG}_{\text{fp}} = \mathcal{K}_{\text{fp}} \cap \text{REG}$ be an ω -regular language such that*

- a) $L \notin \text{CO-SAFETY}$ or
- b) $L \notin \text{SAFETY}$,
- $\exists u, v \in \Sigma^* : u$ is a good, v is a bad prefix for L ,
- there is an fp -monotonic monitor m for L .

There is a number $k \in \mathbb{Z}$ with the following properties:

a) If $L \notin \text{CO-SAFETY}$,

- for every infinite word $\alpha \in L$, that has no good prefix, f eventually stays constantly at value k on the according prefix series, i.e. $\exists i \in \mathbb{N} \forall j \in \mathbb{N}, j > i : f(\alpha_{(i)}) = k$ and
- k is the maximal value for f : $\max_{i \in \mathbb{N}} (f(\alpha_{(i)})) = k$.

b) If $L \notin \text{SAFETY}$,

- for every infinite word $\beta \notin L$, that has no bad prefix, f eventually stays constantly at value k on the according prefix series, i.e. $\exists i \in \mathbb{N} \forall j \in \mathbb{N}, j > i : f(\beta_{(i)}) = k$ and
- k is the minimal value for f : $\min_{i \in \mathbb{N}} (f(\beta_{(i)})) = k$.

Before giving the proof we clarify how this can be used. The number k represents the maximum or minimum of the future aspect f . Hence, knowing its value allows us to decide whether we still have to limit the influence of f to assure fp -monotonicity or if we can give enough weight to the past p in order to converge on infinitely inconclusive paths. For safety languages, where k is the maximum of f ,

$$m_{\text{safe}}(u) := \begin{cases} \pi(\exp(f(u)) + \pi(\exp(p(u))) \cdot \exp(f(u))) & \text{if } f(u) < k \\ \pi(2 \cdot \exp(f(u)) + \exp(p(u))) & \text{otherwise} \end{cases}$$

is fn-monotonic since f completely dominates p in the first case. In the second case, by the previous lemma, $f = k$ or $f = \infty$ and stays at either of these values. The monitor is determined by p without ever falling beyond the values in the first case (where f is always smaller). For $f = \pm\infty$ (only), m_{safe} evaluates to 1 and 0 respectively. Once reached $f = k$, $p \rightarrow \infty$ lets the monitor converge to 1. Dually

$$m_{\text{co-safe}}(u) := \begin{cases} \pi(\exp(f(u)) + \pi(\exp(p(u))) \cdot \exp(f(u))) & \text{if } f(u) > k \\ \pi(\exp(p(u)) \cdot \exp(f(u))) & \text{otherwise} \end{cases}$$

suites the co-safe case. The number k then is the minimum of f and can be estimated by $-\infty$ as soon as it drops down to k , thus allowing $m_{\text{co-safe}}$ to converge to 0 for $p \rightarrow -\infty$. On the other hand, the second term's upper limited is the lower limit of the first. Therefore the function does not violate the monotonicity in $<_{\text{fp}}$.

Proof of Lemma 3.2. Let $\mathcal{F} = (\sigma, Q \cup \{q_{\top}, q_{\perp}\}, q_0, \delta, \lambda)$ be the FSM proposed in Section 3.3.3, that outputs the value of the future aspect function f . Let q_{\top} and q_{\perp} be the two conclusive states, i.e. $\lambda(q_{\top}) = \top$ and $\lambda(q_{\perp}) = \perp$.

First we will show that no state can have a larger (if L is a safety language) respectively smaller (if L is a co-safety language) label than a state on any circle in the automaton graph. Thus all states on all circles are labelled maximal or minimal, respectively. All infinite words without conclusive prefix have to (eventually only) traverse the states of these circles and hence \mathcal{F} outputs eventually only this value (i.e. k). Secondly, we reason that the label (and thus k) is indeed an integer, i.e. not $\pm\infty$.

1. To see that states on circles have maximal respectively minimal future values f assume, for $L \in \text{SAFETY}$, there are states $q_1, q_2 \in Q$ such that q_1 is part of a circle and $k := \lambda(q_1) < \lambda(q_2)$. Consider any finite word u ending in q_2 (i.e. $\delta^*(q_0, u) = q_2$) and an infinite word $\alpha \in \Sigma^{\omega}$, running through q_1 infinitely often. Since L is safe, $\alpha \in L$ and m must converge to 1 on the according prefix series $\alpha_{(n)}$. But since $f(u) = \lambda(q_2) > f(\alpha_{(i)}) = k$ for infinitely many i such a state q_2 cannot exist. An analogous argument applies, if $L \in \text{CO-SAFETY}$, for a state q'_2 with $\lambda(q'_2) < \lambda(q_1)$. Note, as all states of any circle are maximal/minimal, they are in particular equal.

2. Let $L \in \text{SAFETY}$. For a state $q \in Q$, an infinite label $\lambda(q) = \infty$ means that q_{\perp} or (i.e. a bad prefix) is not reachable from q any more. Note that q_{\top} and q_{\perp} are the only states that are labelled with \top and \perp , respectively, thus $\lambda(q) = ?$. Assume q_{\perp} is not reachable, i.e. every word ending in q has no extension to a bad prefix. Since L is safe, q would already be conclusive and $q = q_{\top}$. Now assume q_{\top} is not reachable. If the automaton, being in state q , cannot proceed to any circle, each word that led to q is already a bad prefix and $q = q_{\perp}$. Otherwise if there is reachable circle, q_{\top} is also not reachable from any state q' on that circle and, as argued earlier, its label must be maximal, thus $\lambda(q') = \max_{q \in Q}(\lambda(q)) = -\infty$. Considering that q_{\top} and q_{\perp} are both reachable at least from q_0 we have $\lambda(q_0) > -\infty$, contradicting the maximality of circle nodes. Again, the prove is analogous for $L \in \text{CO-SAFETY}$, exchanging $-\infty/\infty$ and \top/\perp . \square

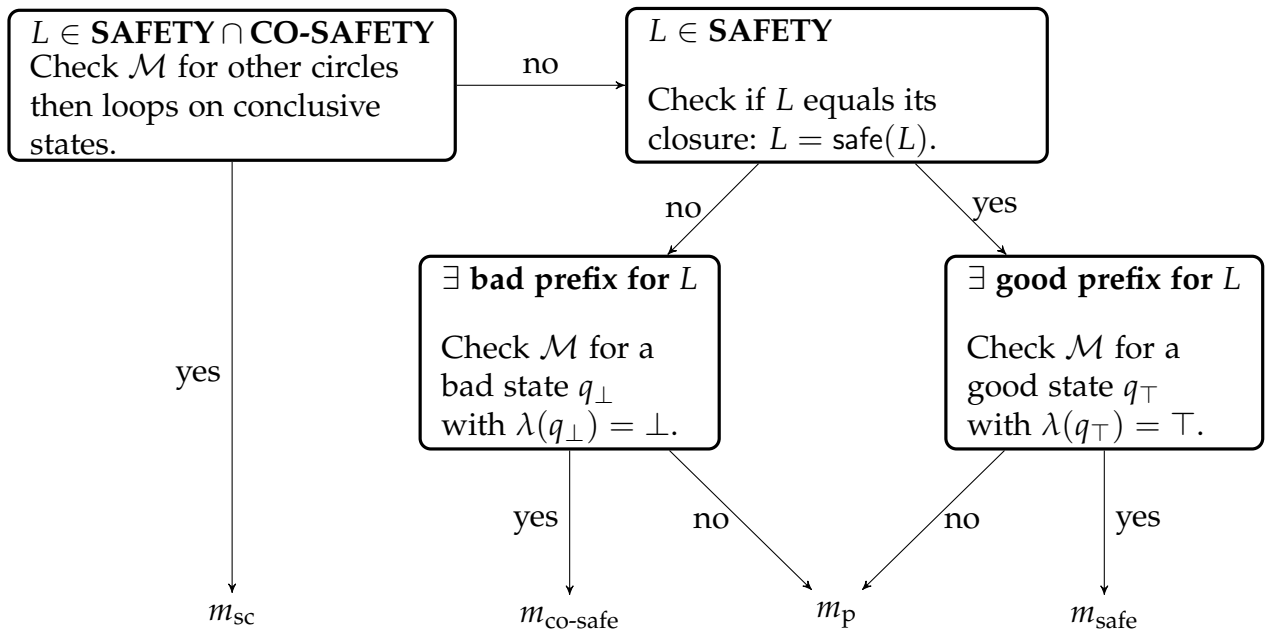


Figure 3.7.: Procedure to choosing a monitor function. To decide if L is safe, it is possible to compare L to its closure $\text{safe}(L)$ [AS87].

4. Conclusions

We discussed runtime verification with approaches from the literature to observe systems and check the current run for conformance to given properties. Popular classes of properties were presented. The discrepancy between specification in terms of infinite words and the fact that only finite prefixes are observable led to different semantics of specification languages like LTL and the necessity for extended domains of monitor verdicts. Monitors are devices that assess the run of a system regarding a certain requirement. For a three valued truth domain, consisting of true, false and inconclusive, a finite state machine can be constructed to function as monitor. To distinguish inconclusive prefixes to a further extend we proposed to use the continuous interval $[0, 1]$ as quantitative verdict.

Continuous monitors must meet a limit requirement, that is, for any word fulfilling or violating a property they must converge to 1 or 0, respectively, on the according prefix series.

Further restrictions result in different classes of monitorable languages. Prefix monotonic monitors are applicable for the union of SAFETY and CO-SAFETY, additionally requiring accuracy yields the intersection. However, we saw, that these monitors do not make use of the extended domain but will only use one constant value for all inconclusive prefixes.

Future and past aspects were introduced to rate prefixes with regard to a language. The future considers the distance to the closest good and bad prefix, the past counts missed chances to satisfy and avoided failures to violate a property.

A language is fp-monitorable if it has a continuous monitor that is also monotonic in terms of the future-past relation. In other words the monitor always yields a higher value for prefixes that are “more promising” in terms of their future and past aspects. Every fp-monitorable language is ugly-free. Every language that is safe or co-safe and ω -regular is fp-monitorable.

We saw that we can construct a Moore machine from a property specification to compute the future aspect and a weighted automaton to compute the past. Also, a procedure to choose a monitor for a given specification was presented.

* * *

The idea of quantitative judgement over prefixes of infinite words appears natural and was the initial motivation. The difficulty lies in the comparison of prefixes. We studied one particular approach. It remains to investigate other criteria than the here proposed future and past aspects to relate prefixes. Additionally, we required global monotonicity for our functions, i.e. a monitor verdict is absolute and comparable to every other verdict of any other prefix. It might be possible to relax this requirement

in the way, that monotonicity must only be accomplished locally on a prefix series but not between prefixes of different runs.

The relation of function characteristics in general and known classes of languages can be of further interest. One might think of an exact characterization of known classes in terms of monitor restrictions.

Finally, the question for applications cannot be avoided. Qualitative assessment of runs could find application in the mentioned self-managing systems, e.g. to decide to be more careful with following actions or to abort a risky operation. Here, again, it remains a challenge to find a reasonable way to assess and relate incomplete traces.

Acknowledgements

I want to thank Benedikt Bollig for advising me during the work on this report. I appreciated our discussions and the so friendly affiliation in Cachan. I learned a lot.

Also, I thank Prof. Martin Leucker for initially proposing the topic and all his explanations.

My stay at Cachan and the experience I made at the ENS would not have been possible without the recommendation by Prof. Volker Diekert and the financial support of the Laboratoire Spécification et Vérification, ENS de Cachan.

References

- [AFH96] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.
- [AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
- [AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
- [Bau10] Andreas Bauer. Monitorability of ω -regular languages. 1006.3638, June 2010.
- [BKKL10] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning communicating automata from MSCs. *IEEE Transactions on Software Engineering*, 36(3):390–408, 2010.
- [BLS06a] A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *Software Engineering Conference, 2006. Australian*, page 10 pp., 2006.
- [BLS06b] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of Real-Time properties. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. 2006.
- [BLS07a] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification*, pages 126–138. 2007.
- [BLS07b] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. techreport, 2007.
- [Bü62] Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic Methodology and Philosophy of Science (Proc. 1960 Internat. Congr .)*, pages 1–11. Stanford Univ. Press, Stanford, Calif., 1962.
- [DGR04] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.
- [DVK09] Manfred Droste, Heiko Vogler, and Werner Kuich. *Handbook of Weighted Automata*. Springer, Berlin, 1 edition, October 2009.

- [EFH⁺03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Computer Aided Verification*, pages 27–39. 2003.
- [ELS10] Javier Esparza, Martin Leucker, and Maximilian Schlund. Learning workflow petri nets. In *Applications and Theory of Petri Nets*, pages 206–225. 2010.
- [Eme83] E. Allen Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26(1-2):121–130, September 1983.
- [GH01] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 412–416, 2001.
- [HS06] M.G. Hinchey and R. Sterritt. Self-managing software. *Computer*, 39(2):107–109, 2006.
- [Kam68] J. A. W. Kamp. *Tense Logic and the Theory of Linear Order*. Ph.D. thesis, University of California at Los Angeles (UCLA), 1968.
- [KEH⁺09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, Big Sky, Montana, USA, 2009. ACM.
- [Kor90] B. Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, 1990.
- [KPS08] Marcel Kyas, Cristian Prisacariu, and Gerardo Schneider. Run-Time monitoring of electronic contracts. In Sungdeok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *Automated Technology for Verification and Analysis*, volume 5311 of *Lecture Notes in Computer Science*, pages 397–407. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-88387-6_34.
- [KV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, November 2001.
- [Lam77] L. Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, SE-3(2):125–143, 1977.
- [LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

- [MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 274–289. 2006.
- [PM95] Amir Pnueli and Zohar Manna. *Temporal Verification of Reactive Systems: Safety*. Springer, Berlin, 1 edition, August 1995.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 1977.
- [PS07] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In Marcello Bonsangue and Einar Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-72952-5_11.
- [PZ06] A. Pnueli and A. Zaks. PSL model checking and Run-Time verification via testers. In *FM 2006: Formal Methods*, pages 573–586. 2006.
- [Ras99] Jean-Francois Raskin. *Logics, Automata and Classical Theories for Deciding Real Time*. Ph.D. thesis, University of Namur, Belgium, 1999.
- [Saf88] S. Safra. On the complexity of ω -automata. In *Foundations of Computer Science, Annual IEEE Symposium on*, volume 0, pages 319–327, Los Alamitos, CA, USA, 1988. IEEE Computer Society.
- [Wol97] Pierre Wolper. The meaning of “formal”: from weak to strong formal methods. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):6–8, 1997.
- [XJ10] Lai Xu and Manfred Jeusfeld. Pro-active monitoring of electronic contracts. In Johann Eder and Michele Missikoff, editors, *Advanced Information Systems Engineering*, volume 2681 of *Lecture Notes in Computer Science*, pages 1028–1028. Springer Berlin / Heidelberg, 2010. 10.1007/3-540-45017-3_39.

A. Linear Temporal Logic

Definition A.1 (LTL formulas). *The set of linear temporal logic formulas over a finite alphabet Σ is denoted by LTL and given by the following grammar:*

$$\varphi ::= \text{True} \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid \varphi \text{ U } \varphi \mid a \quad (a \in \Sigma)$$

Definition A.2 (Boolean semantics of LTL formulas). *Let the Boolean semantics of LTL formulas over a finite alphabet Σ be the mapping*

$$\llbracket \cdot \rrbracket : \text{LTL} \rightarrow (\Sigma^\omega \rightarrow \{\perp, \top\}).$$

For a word $w \in \Sigma^\omega$, a letter $a \in \Sigma$ and formulas $\varphi, \psi \in \text{LTL}$, the semantics is defined inductively by

$$\begin{aligned} \llbracket \text{True} \rrbracket(w) &:= \top \\ \llbracket a \rrbracket(w) &:= \top \text{ iff } a = w_1 \\ \llbracket \varphi \rrbracket(w) &:= \perp \text{ iff } \llbracket \varphi \rrbracket(w) \neq \top \\ \llbracket \neg\varphi \rrbracket(w) &:= \top \text{ iff } \llbracket \varphi \rrbracket(w) = \perp \\ \llbracket X\varphi \rrbracket(w) &:= \top \text{ iff } \llbracket \varphi \rrbracket(w^{(2)}) = \top \\ \llbracket \varphi \wedge \psi \rrbracket(w) &:= \top \text{ iff. } \llbracket \varphi \rrbracket(w) = \top \text{ and } \llbracket \psi \rrbracket(w) = \top \\ \llbracket \varphi \text{ U } \psi \rrbracket(w) &:= \top \text{ iff. } \exists j \in \mathbb{N} : \llbracket \psi \rrbracket(w^{(j)}) = \top \text{ and} \\ &\quad \forall i \in \mathbb{N}, i < j : \llbracket \varphi \rrbracket(w^{(i)}) = \top. \end{aligned}$$

For formulas $\varphi, \psi \in \text{LTL}$, the following abbreviations may also be used.

$$\begin{aligned} \text{False} &:= \neg\text{True} \\ \varphi \vee \psi &:= \neg(\neg\varphi \wedge \neg\psi) \\ \diamond\varphi &:= \text{True U } \varphi && \text{(i.e. "eventually")} \\ \square\varphi &:= \neg\diamond\neg\varphi && \text{(i.e. "globally")} \end{aligned}$$

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Normann Decker)