T U M

I N S T I T U T F Ü R I N F O R M A T I K

Runtime Verification for LTL and TLTL

Andreas Bauer, Martin Leucker, Christian Schallhart

TUM-I0724
Dezember 07

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-12-I0724-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Runtime Verification for LTL and TLTL

Andreas Bauer, Martin Leucker, Christian Schallhart

***Abstract*— This paper studies runtime verification of properties expressed either in lineartime temporal logic (LTL) or timed lineartime temporal logic (TLTL). It classifies runtime verification in identifying its distinguishing features to model checking and testing, respectively. It introduces a three-valued semantics (with truth values *true, false, inconclusive*) as an adequate interpretation as to whether a partial observation of a running system meets an LTL or TLTL property.**

For LTL, a conceptually simple monitor generation procedure is given, which is *optimal* in two respects: First, the size of the generated deterministic monitor is *minimal*, and, second, the monitor identifies a continuously monitored trace as either satisfying or falsifying a property *as early as possible*. The presented approach is furthermore related to the properties monitorable in general and is compared to existing concepts in the literature. It is shown that the set of *monitorable properties* does not only encompass the *safety* and *co-safety* properties but is strictly larger.

For TLTL, the same road map is followed by first defining a three-valued semantics. The corresponding construction of a timed monitor is more involved, yet, as is shown, possible.

***Index Terms*— D.2.4.a - Assertion checkers · D.2.5.g - Monitors · F.3.1.a - Assertions**

I. Introduction

Verification comprises all techniques suitable for showing that a system satisfies its specification. *Runtime verification* deals with those verification techniques that allow checking whether an execution of a system under scrutiny satisfies or violates a given correctness property. It aims to be a *lightweight* verification technique complementing other verification techniques such as *model checking* [2] and testing [3].

In runtime verification, a correctness property φ is typically automatically translated into a *monitor*. Such a monitor is then used to check the *current* execution of a system or a (finite set of) *recorded* execution(s) with respect to the property φ. In the former case, we speak of *online monitoring* while in the latter case we speak of *offline monitoring*.

Formally, when $\mathcal{L}(\varphi)$ denotes the set of valid executions given by property φ, runtime verification boils down to checking whether the execution w is an element of $\mathcal{L}(\varphi)$. Thus, in its mathematical essence, runtime verification reduces to the *word problem*, i. e., the problem whether a given word is included in some language.

Correctness properties in runtime verification specify all admissible individual executions of a system and are usually formulated in some variant of linear temporal logic, such as LTL [4], as seen for example in [5], [6], [7], [8], [9], [10]. But also linear μ-calculus variants are used, for example in [11].

A preliminary version of this paper appeared at FSTTCS 2006 [1].
The authors are with the Technische Universität München, Germany.

Runtime verification deals (only) with the *detection* of violations (or satisfactions) of correctness properties—but it is not concerned with any consequential measures and thus it does not influence the program's functional behaviour. However, runtime verification is at the core of those approaches which react on faults at runtime: *Monitor-oriented programming* [12], for example, aims at a programming methodology that allows for the execution of code whenever monitors observe a violation of a given correctness property. *Runtime reflection* [13], to name a further example, is an architecture pattern that is applicable for systems in which monitors are enriched with a diagnosis and reconfiguration layer.

A. Runtime Verification versus Model Checking

While runtime verification shares also many similarities with *model checking*, there are important differences:

- In model checking, *all executions* of a given system are examined to answer whether these satisfy a given correctness property φ—which corresponds to the language inclusion problem. In contrast, runtime verification deals with the word problem. For most logical frameworks, the word problem is of far lower complexity than the inclusion problem, e. g. in case of LTL see [14] and [15].
- While model checking, especially when considering LTL, considers *infinite* traces, runtime verification deals with *finite* traces—as non-idealised executions are necessarily finite.
- While in model checking a complete model is given allowing to consider arbitrary positions of a trace, runtime verification, especially when dealing with online monitoring, considers finite executions of increasing size. For this, a monitor should be designed to consider executions in an *incremental fashion*.

These differences make it necessary to adapt the concepts developed in model checking to be applicable in runtime verification. For example, the second item asks for coming up with a semantics for LTL on finite traces that mimics LTL's semantics on infinite traces—which we do in the the first part of the paper. Note that LTL is originally defined on finite traces as well [16]. However, as we argue, this semantics is not suitable for runtime verification.

From an application point of view, there are also important differences between model checking and runtime verification: Runtime verification deals only with observed executions. Thus it is applicable to *black box systems* for which no system model is at hand. In model checking, however, a precise description of the system to check is mandatory as, before actually running the system, all possible executions must be

checked.[1]

Furthermore, model checking suffers from the so-called *state explosion problem*, which terms the fact that analysing all executions of a system is typically been carried out by generating the whole state space of the underlying system, which becomes often infeasibly huge. Considering a single run, on the other hand, does usually not yield any memory problems, provided that when monitoring online only a finite *history* of the current execution has to be stored.

In online monitoring, the complexity for *generating* the monitor procedure is often negligible, as the monitor is typically only generated once. However, the *complexity of the monitor*, i. e., its memory and computation time requirements for checking an execution are of important interest, as the monitor is part of the running system and should influence the system as little as possible.

B. Runtime Verification versus Testing

As runtime verification does not consider each possible execution of a system, but just a single or a finite subset, it shares similarities with *testing*: both are usually incomplete.

Typically, in testing one considers a finite set of finite input-output sequences forming a *test suite* [19]. Test-case execution is then checking whether the output of a system agrees with the predicted one, when giving the input sequence to the system under test.

A different form of testing, however, is closer to runtime verification, namely *oracle-based testing* [3]. Here, a test-suite is only formed by input-sequences. To make sure that the output of the system is as anticipated, a so-called *test oracle* has to be designed and "attached" to the system under test. This oracle then observes the system under test and checks a number of properties, i. e. in terms of runtime verification the oracle acts as a monitor. Thus, in essence, runtime verification can be understood as this form of testing. There are, however, differences in the foci of runtime verification and oracle-based testing: In testing, an oracle is typically defined directly, rather than generated from some high-level specification. On the other hand, in the domain of runtime verification, we do not consider the provision of a suitable set of input sequences to "exhaustively" test a system.

C. Monitoring of Discrete-Time Properties

In this paper, we introduce LTL_3 as a lineartime temporal logic designed for runtime verification. As Pnueli's LTL [4] is a well-accepted lineartime temporal logic used for specifying properties of infinite traces one usually wants to check properties specified in LTL in runtime verification as well. However, one has to interpret their semantics with respect to finite prefixes as they arise in observing actual systems. This approach to runtime verification is summarised in the following rationale:

> *Pnueli's LTL [4] is a well-accepted lineartime temporal logic used for specifying properties of infinite traces. In runtime verification, our goal is to check LTL properties given* finite prefixes of infinite traces.

Therefore, LTL_3's syntax coincides with LTL, while its semantics is given for finite traces. To implement the idea that, for a given LTL_3 formula, its meaning for a prefix of an infinite trace should correspond to its meaning considered as an LTL formula for the full infinite trace, we use *three* truth values: $true$, $false$, and *inconclusive*, denoted respectively by $\top$, $\bot$, and ?. More precisely, given a finite word u and an LTL_3 formula φ, the semantics is defined as follows:

- if there is no continuation of u satisfying φ (considered as an LTL formula), the value of φ is $false$;
- if every continuation of u satisfies φ (considered as an LTL formula), it is $true$;
- otherwise, the value is *inconclusive* since the observations so far are inconclusive, and neither $true$ or $false$ can be determined.

While there are actually semantics for LTL on finite traces [20], [21], these use (only) two truth values. We strongly believe that only two truth values lead to misleading results in runtime verification: Consider the formula $\neg p\, U\, init$ (read: *not* p *until init*) stating that nothing bad (p) should happen before the init function is called. If within an execution p becomes true before $init$, the formula is violated and thus $false$ (for any continuation of the current execution). If, on the other hand, the init function has been called and no p has been observed before, the formula is $true$, regardless of what will happen in the future. Besides observing failures, for testing and verification, it is equally important to know whether some property is indeed $true$ or whether the current observation is just inconclusive and a violation of the property to check may still occur.

Originally, we proposed this three-valued semantics and its use for runtime verification in [1]. However, some essential concepts were defined by Kupferman and Vardi: In [22] a *bad prefix* (of a Büchi automaton) is defined as a finite prefix which cannot be the prefix of any accepting trace. Dually, a *good prefix* is a finite prefix such that any infinite continuation of the trace will be accepted. It is exactly this classification that forms the basis of our 3-valued semantics: "bad prefixes" (of formulae) are mapped to $false$, "good prefixes" evaluate to $true$, while the remaining prefixes yield *inconclusive*.

For a given LTL_3 formula, we describe how to construct a (deterministic) finite state machine (FSM) with three output symbols. This automaton reads finite traces and yields their three-valued semantics. Thus, monitors for three-valued formulae classify prefixes as one of $good = \top$, $bad = \bot$, or $?$ (neither $good$ nor bad). Standard minimisation techniques for FSMs can be applied to obtain a unique FSM that is *optimal* with respect to its number of states. In other words, any smaller FSM must be non-deterministic or check a different property. As an FSM can straightforwardly be deployed, we obtain a practical framework for runtime verification.

The proposed semantics of LTL_3 has a valuable implication for a corresponding monitor. It requires the monitor to report a violation of a given property *as early as possible*: Since any continuation of a bad (good) prefix is bad (respectively

[1]Note, that it is possible to automatically *learn* [17] and verify a system model, thereby applying model checking techniques to an a priori unknown system [18].

good), there exists a *minimal* bad (good) prefix for every bad (good) prefix. In runtime verification, we are interested in getting feedback from the monitor as early as possible, i. e., for minimal prefixes, let them be either good or bad. Since all bad prefixes for a formula φ yield false and good prefixes yield true, also minimal ones do so. Thus, the correctness of our monitor procedure ensures that already for *minimal* good or bad prefixes either true or false is obtained.

In [23], a Büchi automaton was modified to serve as a monitor reporting false for minimal bad prefixes. However, no precise semantics in terms of LTL of the resulting monitor was given. As such, LTL_3 can be understood as a logic which complements the constructions carried out in [23] with a formal framework. Nevertheless, we feel that our constructions are more direct and therefore easier to understand.

In this paper, we further discuss, which LTL_3 properties are *monitorable* at all. We follow the definition given by Pnueli and Zaks in [24] essentially stating that a property is monitorable with respect to a trace whenever a corresponding monitor might still report a violation (or satisfaction). We point out the precise relation to Rosu's notion of *never violate states* [23] in monitors, which is similar yet not the same. Moreover, we recall the notion of safety and co-safety properties. We show that the popular belief that monitoring is only suitable for safety properties is misleading: The class of monitorable properties is richer than the union of safety and co-safety properties. Finally, we discuss runtime verification based on good/bad-prefixes compared to approaches based on Kupferman's and Vardi's notion of *informative prefixes*, as for example the approach shown in [25]. We argue that runtime verification should be based on good/bad prefixes rather than on informative prefixes, as it follows the *as early as possible* maxim.

Note that multi-valued versions of LTL have been considered, for example in [26]. There, the semantics is defined for *infinite* traces and the resulting logics and model checking approaches are completely different from LTL_3. Moreover, these logics are helpful in model checking abstractions of systems or of software product lines [27], and we do not see any benefit of the developed ideas in the setting of runtime verification.

D. Monitoring of Real-time Properties

In the second part of the paper, we address real-time systems. We base our ideas on the *timed lineartime temporal logic* (TLTL), a logic originally introduced by Raskin in [28]. TLTL, as argued by D'Souza, can be considered a natural counterpart of LTL in the timed setting: He showed in [29] that, over timed traces, TLTL is equally expressive as first-order logic, transferring Kamp's famous result that, over words, LTL and first-order logic coincide with respect to expressiveness [16] to the world of real-time systems.

We define a three-valued version of TLTL for finite *timed* traces resulting in the logic TLTL_3, following a similar approach as for LTL. Moreover, for a TLTL_3 formula we describe how to construct a monitor yielding the semantics for a finite *timed* trace, again, "as early as possible".

While the general scheme developed for LTL_3 proves to be applicable in the real-time setting as well, the monitor construction is technically much more involved. Automata for TLTL employ so-called *event recording* and *event predicting* clocks. Since in runtime verification, the future of a trace is not known, event predicting clocks are difficult to handle. We introduce *symbolic timed runs* and show their benefit for checking promises efficiently, avoiding a possible but generally expensive translation of event-clock automata to (predicting-free) timed automata [30].

So far, not many approaches for runtime verification of real-time properties have been given. [31] studies monitor generation based on LTL enriched with a freeze quantifier for time. In [32] and [33], *fault diagnosis* for timed systems is examined, a problem that shares some similarities with runtime verification yet is more complicated. However, in these approaches, only timed automata or event-recording automata are used and no prediction of events is supported. TLTL is event-based, meaning that the system emits events when the system's state has changed. In [34] monitoring of continuous signals is considered, which is intrinsically different to observing discrete signals in a continuous time domain.

E. Outline

In Section II, we develop our runtime verification approach for the discrete-time setting. After recalling standard LTL syntax and semantics, we introduce a three-valued semantics for LTL formulae on finite words, yielding the three-valued logic LTL_3. Then we develop and discuss a monitor construction technique to produce for an LTL-property φ a deterministic finite-state machine $\mathcal{M}^\varphi$ which evaluates φ on finite traces according to LTL_3. Finally, we demonstrate this approach with an example from concurrent C++-development practice.

Section III analyses the structure of the developed monitors, complements the notions of good and bad prefixes with *ugly* prefixes to characterise the instant when properties become *non-monitorable*. Moreover, we discuss monitoring in the light of safety and co-safety properties and compare our work with ideas based on *informative* prefixes.

In Section IV, we expand our runtime verification approach for the *real-time* setting. After recalling standard TLTL syntax and semantics, we introduce a three-valued semantics to evaluate standard TLTL formulae on finite *timed* words, yielding the three-valued logic TLTL_3. Then we develop and discuss a monitor construction technique to produce for an TLTL-property φ a deterministic monitor $\mathcal{M}^\varphi$ which evaluates φ on finite timed traces according to TLTL_3.

II. Three-valued LTL in the Discrete-Time Setting

In this section, we consider runtime verification for systems whose behaviour is characterised by a sequence of states which occur at discrete time steps. These states are then abstracted with a set of atomic properties AP which evaluate to either true or false in such a state. Thus, the behaviour of the system under scrutiny is described by an (in)finite word over the alphabet 2^{AP}. Linear temporal logic (LTL) is a well-accepted logic to

specify properties of infinite words [4], and, consequently, our developments in this section are for LTL specifications.

After recalling standard LTL syntax and semantics, we introduce in this section a three-valued semantics to evaluate standard LTL formulae on finite words yielding to the logic LTL_3 . Thereby, LTL_3 distinguishes three cases:

- Either the observed finite word u is sufficient to prove that the monitored property φ *holds independently of the yet unknown future* behaviour, or
- the observed finite word u already indicates that φ *cannot be satisfied in any possible future continuation*, or finally,
- *neither of both cases* occurred so far.

Having the semantics of LTL_3 at hand, we develop and discuss a monitor construction technique to produce for an LTL-property φ a deterministic finite-state machine $\mathcal{M}^\varphi$ which evaluates φ on finite traces according to LTL_3—thus enabling a most predictive evaluation of φ: Once it can be decided that φ will remain either satisfied or unsatisfied, the monitor will provide this information immediately. Finally, we demonstrate this approach with an example from concurrent C++-development practice.

A. Preliminaries

For the remainder of this section, let AP be a finite set of atomic propositions and $\Sigma = 2^{\text{AP}}$ a finite alphabet. We write a_i for any single element of Σ, i.e., a_i is a possibly empty set of propositions taken from AP. Finite traces over Σ are elements of Σ^*, and are usually denoted with $u, v, u', v', u_1, v_1, u_2, \ldots$, whereas infinite traces are elements of Σ^ω, usually denoted with $w, w', w_1, w_2, \ldots$ We also write e.g. $\{p,q\}\,\{p\}\ldots$ for a finite or infinite word $a_0a_1\ldots$ with $a_0 = \{p,q\}$ and $a_1 = \{p\}$. If clear from the context, we also drop the brackets around singletons, i.e., we write $\{p,q\}\,p\ldots$ for the same word $a_0a_1\ldots$ Finally, we call the concatenation uv of two finite words u and v *finite continuation* of u with v. Similarly, the concatenation uw of u with an infinite word w is called *infinite continuation* of u with w.

Then the syntax and semantics of LTL on infinite traces is defined as follows.

Definition 1 (LTL formulae) *The set of* LTL formulae *is inductively defined by the grammar*

$$\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\ U\ \varphi \mid X\varphi$$

with $p \in AP$.

In addition, we use three abbreviations, namely $\varphi \wedge \psi$ for $\neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi$ for $\neg\varphi \vee \psi$, $F\varphi$ for $\mathit{true}\ U\ \varphi$, and $G\varphi$ for $\neg(\mathit{true}\ U\ \neg\varphi)$.

Definition 2 (LTL semantics) *Let* $w = a_0a_1\ldots \in \Sigma^\omega$ *be a infinite word with* $i \in \mathbb{N}$ *being a position. Then we define the semantics of* LTL_3 *formulae inductively as follows*

$$
\begin{array}{lll}
w,i \models \mathit{true} & & \\
w,i \models \neg\varphi & \textit{iff} & w,i \not\models \varphi \\
w,i \models p & \textit{iff} & p \in a_i \\
w,i \models \varphi_1 \vee \varphi_2 & \textit{iff} & w,i \models \varphi_1 \textit{ or } w,i \models \varphi_2 \\
w,i \models \varphi_1 U \varphi_2 & \textit{iff} & \exists k \geq i \textit{ with } w,k \models \varphi_2 \\
 & \textit{and} & \forall i \leq l < k \textit{ with } w,l \models \varphi_1 \\
w,i \models X\varphi & \textit{iff} & w,i+1 \models \varphi
\end{array}
$$

Further, $w \models \varphi$ *holds iff* $w,0 \models \varphi$ *holds.*

We denote with $\mathcal{L}(\varphi) = \{w \in \Sigma^\omega \mid w \models \varphi\}$ the set of models of an LTL-formula φ. Two LTL-formulae φ and ψ are called *equivalent*, written as $\varphi \equiv \psi$, iff $\mathcal{L}(\varphi) = \mathcal{L}(\psi)$ holds. The language $\mathcal{L}(\varphi)$, generated by an LTL-formula φ, is a regular set of infinite traces and can be described by a corresponding Büchi automaton defined next.

Definition 3 ((Nondeterministic) Büchi automaton (NBA)) *A* (nondeterministic) Büchi automaton (NBA) *is a tuple* $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$*, where*

- Σ *is a finite alphabet,*
- Q *is a finite non-empty set of states,*
- $Q_0 \subseteq Q$ *is a set of initial states,*
- $\delta : Q \times \Sigma \rightarrow 2^Q$ *is a transition function, and*
- $F \subseteq Q$ *is a set of accepting states.*

We extend the transition function $\delta : Q \times \Sigma \rightarrow 2^Q$, as usual, to $\delta' : 2^Q \times \Sigma^* \rightarrow 2^Q$ by $\delta'(Q', \epsilon) = Q'$ and $\delta'(Q', ua) = \bigcup_{q' \in \delta'(Q',u)} \delta(q', a)$ for $Q' \subseteq Q$, $u \in \Sigma^*$, and $a \in \Sigma$. To simplify notation, we use δ for both δ and δ'.

A *run* of an automaton $\mathcal{A}$ on a word $w = a_0a_1\ldots \in \Sigma^\omega$ is a sequence of states and actions $\rho = q_0a_0q_1a_1q_2\ldots$, where q_0 is an initial state of $\mathcal{A}$ and where we have $q_{i+1} \in \delta(q_i, a_i)$ for all $i \in \mathbb{N}$. For a run ρ, let $\text{Inf}(\rho)$ denote the states visited infinitely often. A run ρ of an NBA $\mathcal{A}$ is called *accepting* iff $\text{Inf}(\rho) \cap F \neq \emptyset$.

A nondeterministic *finite automaton* (NFA) $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$ is an automaton where Σ, Q, Q_0, δ, and F are defined as for a Büchi automaton, but which operates on finite words. A *run* of $\mathcal{A}$ on a word $w = a_0\ldots a_n \in \Sigma^*$ is a sequence of states and actions $\rho = q_0a_0q_1a_1\ldots a_nq_{n+1}$, where q_0 is an initial state of $\mathcal{A}$ and for all $i \in \mathbb{N}$ we have $q_{i+1} \in \delta(q_i, a_i)$. The run is called accepting if $q_{n+1} \in F$. An NFA is called *deterministic* iff for all $q \in Q$, $a \in \Sigma$, $|\delta(q,a)| = 1$, and $|Q_0| = 1$. We use DFA to denote a deterministic finite automaton.

In case of Büchi automata, we did not introduce their deterministic variant since not every NBA can be converted into an equivalent deterministic one. Furthermore, our monitor construction allows to apply determinisation once we have converted all NBAs into NFAs—thereby yielding a deterministic finite-state machine. The resulting FSM can be minimised to obtain an FSM with a provable minimal number of states.[2]

Finally, let us recall the notion of a *finite-state machine* (FSM), which is a finite state automaton enriched with output,

[2]Note that in many practical cases, the monitor will be based directly on the underlying nondeterministic automata and will be determinised on-the-fly using the power-set construction (cf. the discussion at the end of Section II-C.)

formally denoted with a tuple $(\Sigma, Q, Q_0, \delta, \Delta, \lambda)$, where Σ, Q, Q_0, and δ are defined as before and where Δ is the output alphabet used in the output function $\lambda : Q \to \Delta$. The output of an FSM, defined by the function λ, is thus determined by the current state $q \in Q$ alone, rather than by input symbols. As before, δ extends to the domain of words as expected. For a deterministic FSM, we denote with λ also the function that yields for a given word u the output in the state reached by u rather than the sequence of outputs.

B. Syntax and Semantics of LTL$_3$

To overcome difficulties in defining an adequate Boolean semantics for LTL on finite traces, we propose a three-valued semantics. The intuition is as follows: in theory, we observe an infinite sequence w of some system. For a given formula φ, thus either $w \models \varphi$ or not. In practice, however, we can only observe a finite prefix u of w. Consequently, we let the semantics of φ with respect to u be true, if $uw' \models \varphi$ for every possible continuation w'. On the other hand, if uw' is not a model of φ for all possible infinite continuations w' of u, we define the semantics of φ with respect to u as false. In the remaining case, the truth value of uw' and φ depends on w'. Thus, we define the semantics of u with respect to φ to be *inconclusive*, denoted with ?, to signal that the so far observed prefix u itself is insufficient to determine how φ evaluates in any possible future continuation of u.

We define our three-valued semantics LTL$_3$ to interpret common LTL formulae, as defined in Definition 1 on finite prefixes to obtain a truth value from the set $\mathbb{B}_3 = \{\bot, ?, \top\}$ as follows:

Definition 4 (Three-valued semantics of LTL) *Let $u \in \Sigma^*$ denote a finite word. The* truth value *of a LTL$_3$ formula φ with respect to u, denoted with $[u \models \varphi]$, is an element of $\mathbb{B}_3$ defined as follows:*

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

Note that in the above definition, we use the semantic function $[u \models \varphi]$ as well as the standard notation $u\sigma \models \varphi$: Since we introduce a three-valued semantics on finite words, we have to use a semantic function $[u \models \varphi]$ to denote the truth value of φ with respect to a finite word u. On the other hand, for the standard two-valued semantics of LTL, we only write $u\sigma \models \varphi$ to assert that $u\sigma$ satisfies φ.

Note that already in [16], a coherent semantics for both, LTL on finite and infinite words is given. However, in runtime verification, we aim at checking LTL properties of infinite traces by considering their finite prefixes. This renders the standard two-valued LTL semantics on finite traces as inappropriate in our case.

C. Monitor Construction for LTL$_3$

Now we develop an automata-based monitor procedure for LTL$_3$. More specifically, for a given formula $\varphi \in$ LTL$_3$, we construct an FSM $\mathcal{M}^\varphi$ that reads finite words $u \in \Sigma^*$ and outputs $[u \models \varphi]$ which is a value in $\mathbb{B}_3$.

For an NBA $\mathcal{A}$, we denote by $\mathcal{A}(q)$ the NBA that coincides with $\mathcal{A}$ except for the set of initial states Q_0, which is redefined in $\mathcal{A}(q)$ as $Q_0 = \{q\}$. Let us fix $\varphi \in$ LTL for the rest of this section, and let $\mathcal{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ denote the NBA, which accepts all models of φ, and let $\mathcal{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, F^{\neg\varphi})$ denote the NBA, which accepts all words falsifying φ. The corresponding construction is standard [35] and explained, for example in [36]. Note that in order to obtain the complement of an NBA, we merely need to complement the formula, rather than the original Büchi automaton itself.

For the automaton $\mathcal{A}^\varphi$, we define a function $\mathcal{F}^\varphi : Q^\varphi \to \mathbb{B}$ (with $\mathbb{B} = \{\top, \bot\}$) where we set $\mathcal{F}^\varphi(q) = \top$ iff $\mathcal{L}(\mathcal{A}^\varphi(q)) \neq \emptyset$, i.e., we evaluate a state q to $\top$ iff the language of the automaton starting in state q is not empty. To determine $\mathcal{F}^\varphi(q)$, we identify in linear time the strongly connected components in $\mathcal{A}^\varphi$, which can be done using Tarjan's algorithm [37] or nested depth-first algorithms as examined in [38]. Using $\mathcal{F}^\varphi$, we define the NFA $\hat{\mathcal{A}}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, \hat{F}^\varphi)$ with $\hat{F}^\varphi = \{q \in Q^\varphi \mid \mathcal{F}^\varphi(q) = \top\}$. Analogously, we set $\hat{\mathcal{A}}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ with $\hat{F}^{\neg\varphi} = \{q \in Q^{\neg\varphi} \mid \mathcal{F}^{\neg\varphi}(q) = \top\}$.

Having $\hat{\mathcal{A}}^\varphi$ and $\hat{\mathcal{A}}^{\neg\varphi}$ at hand, we evaluate $[u \models \varphi]$ according to the following Lemma:

Lemma 5 (LTL$_3$ evaluation) *With the notation as before, we have*

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \notin \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \\ \bot & \text{if } u \notin \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \\ ? & \text{if } u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \cap \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \end{cases}$$

Proof: Let $\mathcal{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, F^{\neg\varphi})$ denote the NBA such that $\mathcal{L}(\mathcal{A}^{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. Feeding a finite word $u \in \Sigma^*$ to $\mathcal{A}^{\neg\varphi}$, we reach the set $\delta^{\neg\varphi}(Q_0^{\neg\varphi}, u) \subseteq Q^{\neg\varphi}$ of states. Thus, if there exists a state $q \in \delta^{\neg\varphi}(Q_0^{\neg\varphi}, u)$ such that $\mathcal{L}(\mathcal{A}^{\neg\varphi}(q)) \neq \emptyset$, then we can choose an infinite word $\sigma \in \mathcal{L}(\mathcal{A}^{\neg\varphi}(q))$ in order to expand u into $u\sigma \in \mathcal{L}(\mathcal{A}^{\neg\varphi})$. By definition of the NFA $\hat{\mathcal{A}}^{\neg\varphi}$, such a state $q \in \delta^{\neg\varphi}(Q_0^{\neg\varphi}, u)$ exists, iff $u \in \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$ holds.

Therefore, if $u \notin \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$ holds, then every possible continuation $u\sigma$ of u will be rejected by $\mathcal{A}^{\neg\varphi}$, i.e., every possible continuation $u\sigma$ will violate $\neg\varphi$ and satisfy φ and hence we have $u\sigma \models \varphi$ for all $\sigma \in \Sigma^\omega$. If this is the case, by Definition 4, $[u \models \varphi] = \top$.

By substituting φ for $\neg\varphi$, we obtain $[u \models \varphi] = \bot$ if $u \notin \mathcal{L}(\hat{\mathcal{A}}^{\varphi})$. Finally, if $u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \cap \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$, then there exist two continuations $\sigma \neq \sigma' \in \Sigma^\omega$ such that $u\sigma \models \varphi$ and $u\sigma' \not\models \varphi$ and therefore $[u \models \varphi] = ?$. ∎

The lemma yields the following procedure to evaluate the semantics of φ for a given finite trace u: we evaluate both $u \in \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$ and $u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi})$ and use Lemma 5 to determine $[u \models \varphi]$. As a final step, we now define a (deterministic) FSM $\mathcal{M}^\varphi$ that outputs for each finite word u its associated three-valued semantical evaluation with respect to some LTL-formula φ.

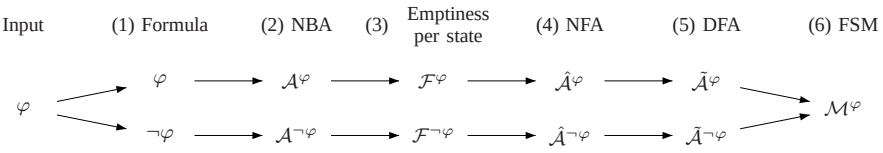


Fig. 1. The procedure for getting $[u \models \varphi]$ for a given φ

Let $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ be the deterministic versions of $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$, which can be computed in a standard manner using the power-set construction. Then, we define the FSM in question as a product of $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$.

Definition 6 (Monitor $\mathcal{M}^{\varphi}$ for an LTL$_3$ formula φ) *Let $\tilde{\mathcal{A}}^{\varphi} = (\Sigma, Q^{\varphi}, \{q_0^{\varphi}\}, \delta^{\varphi}, \tilde{F}^{\varphi})$ and $\tilde{\mathcal{A}}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, \{q_0^{\neg\varphi}\}, \delta^{\neg\varphi}, \tilde{F}^{\neg\varphi})$ be the DFAs with $\mathcal{L}(\tilde{\mathcal{A}}^{\varphi}) = \mathcal{L}(\hat{\mathcal{A}}^{\varphi})$ and $\mathcal{L}(\tilde{\mathcal{A}}^{\neg\varphi}) = \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$, where NFAs $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$ are as defined above. Then we define the* product automaton *$\bar{A}^{\varphi} = \tilde{\mathcal{A}}^{\varphi} \times \tilde{\mathcal{A}}^{\neg\varphi}$ as the FSM $(\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{\lambda})$, where $\bar{Q} = Q^{\varphi} \times Q^{\neg\varphi}$, $\bar{q}_0 = (q_0^{\varphi}, q_0^{\neg\varphi})$, $\bar{\delta}((q, q'), a) = (\delta^{\varphi}(q, a), \delta^{\neg\varphi}(q', a))$, and $\bar{\lambda} : \bar{Q} \to \mathbb{B}_3$ is defined by*

$$\bar{\lambda}((q, q')) = \begin{cases} \top & \text{if } q' \notin \tilde{F}^{\neg\varphi} \\ \bot & \text{if } q \notin \tilde{F}^{\varphi} \\ ? & \text{if } q \in \tilde{F}^{\varphi} \text{ and } q' \in \tilde{F}^{\neg\varphi}. \end{cases}$$

The monitor *$\mathcal{M}^{\varphi}$ of φ is obtained by minimising the product automaton $\bar{A}^{\varphi}$.*

We sum up our entire construction in Figure 1 and conclude by formulating the correctness theorem.

Theorem 7 (LTL monitor correctness) *Let $\varphi \in LTL_3$ and let $\mathcal{M}^{\varphi} = (\Sigma, Q, q_0, \delta, \lambda)$ be the corresponding monitor. Then, for all $u \in \Sigma^*$ the following holds:*

$$[u \models \varphi] = \lambda(\delta(q_0, u))$$

Proof: The theorem follows directly from the monitor construction given in Definition 6 and Lemma 5 on the evaluation of LTL$_3$. ∎

Complexity: Consider Figure 1: Given φ, step 1 requires replication and negation of φ, i.e., it is linear in the original size. Step 2, the construction of the NBAs, causes an exponential "blow-up" in the worst-case. Steps 3 and 4, leading to $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$, do not change the size of the original automata. Then, computing the deterministic automata in step 5, causes in general an exponential "blow-up" in size, for a second time. In total the FSM of step 6 will have double exponential size with respect to $|\varphi|$.

The size of the final FSM is in $O(2^{2^n})$ but can be minimised with standard algorithms for FSMs [39] to derive an *optimal* deterministic monitor with a minimal number of states. In the worst case, however, a lower bound of $O(2^{2^{\Omega(n)}})$ applies to the number of states, as proved in [40].

Thus, better complexity results in other approaches, like the one in [8], are due to one of the following reasons:

- First, one can use a fragment of LTL which is *strictly less expressive* than full LTL, i.e., one gives up the possibility to specify certain properties and thereby rules out some complicated cases exercising the worst case complexity. Note that our construction yields an optimal monitor regardless whatever fragment of LTL is considered.
- Second, it is possible to use a variant of LTL which is still capable to express all LTL-expressible properties but which requires *strictly longer formulae* for some of these properties.
- Third, one could abandon a single monolithic and deterministic automaton as monitor procedure, and use instead an alternative concept such as synchronising automata, hereby trading the size of automaton with an increased computational overhead at runtime [41].

Moreover, we have implemented the above construction of the finite-state automaton $\mathcal{M}^{\varphi}$ partly in an *on-the-fly* fashion. That is, for a given property φ, we construct the two NFAs, but we do not determinise them to obtain the two corresponding DFAs $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$. Consequently, we do not explicitly construct the final automaton $\mathcal{M}^{\varphi}$, but instead perform steps 4–6 on-the-fly to avoid the second exponential "blow-up".

To do so, our implementation employs the power-set construction, known from compiler construction [42], in an on-the-fly manner: Instead of only maintaining a single current state of a deterministic automaton, our monitor maintains the *set of reachable states* of the correspondingly underlying nondeterministic automaton. Then, the deterministic automaton would be in an accepting state, if and only if there exists at least one accepting state in the currently maintained set of states (of the nondeterministic automaton).

The reason for constructing the NFAs explicitly is that we have to check the emptiness per state for each state of the NBAs (recall Lemma 5 and its proof). Therefore, it is impractical to build these two NFAs on-the-fly as well.

D. Example

Now we discuss a simple but comprehensive real-world example in more detail, which also highlights most of the features described above.

In a C++-program, all static objects of an executable are initialised before the `main` method is entered, however, their order is undefined, and their initialisation is thus performed in a nondeterministic order (cf. [43]). In consequence, if threads get spawned before executing `main`, it is difficult to ensure that all resources necessary to synchronise those threads are already initialised, such as globally available and statically initialised mutex objects. This problem is generally known as the *static initialisation order fiasco* (cf. [44]). The "fiasco" is an especially complicated one when large applications are built

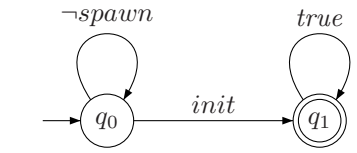
from a number of different frameworks which must remain independent from each other.

Using our monitor generator with a C++ logging layer such as the APACHE SOFTWARE FOUNDATION'S library, LOG4CXX[3], for gaining access to signals emitted by the application's threads, it is possible to construct a monitor over an alphabet $\Sigma = 2^{AP}$, where $\{spawn, init\} \subseteq AP$, for a property $\varphi \equiv \neg spawn\ U\ init$. In other words, the monitor reports a violation, once a thread is spawned before the application under scrutiny has properly finished its initialisation.

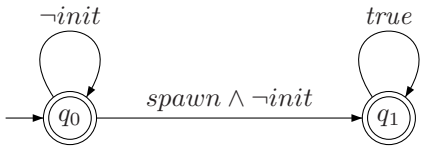
This example further illustrates the need for having three truth values, instead of two when monitoring a running system:

- Intuitively, a monitor for φ should raise an alarm only, if a thread was spawned before $init$ occurred.
- On the other hand, if $init$ occurs before any $spawn$ has occurred, the monitor should report that φ is satisfied irrespective of the future.
- Finally, until either happens, it should return ?, indicating the necessity for further observation.

Using the translation algorithm from formulae of LTL to Büchi automata as proposed by [45], one obtains for φ, respectively $\neg\varphi$, the Büchi automata depicted in Figure 2.



(a) Büchi automaton $\mathcal{A}^{\varphi}$.



(b) Büchi automaton $\mathcal{A}^{\neg\varphi}$.

Fig. 2. The Büchi automata for $\varphi \equiv \neg spawn\ U\ init$

Then the two nondeterministic finite automata $\hat{A}^{\varphi} = (\Sigma, Q^{\varphi}, Q_0^{\varphi}, \delta^{\varphi}, \hat{F}^{\varphi})$ and $\hat{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ are defined with the accepting states $\hat{F}^{\varphi}$ and $\hat{F}^{\neg\varphi}$, as described in the construction leading to Lemma 5.

In our particular case, all states in $\hat{A}^{\varphi}$ and $\hat{A}^{\neg\varphi}$ become accepting states, as only those states and transitions are shown which contribute to the accepted language. Also note that in this example, the two resulting finite automata are already deterministic.

Following Definition 6, we construct an FSM as monitor for the static initialisation order fiasco. For this purpose, we first build the product of $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$. Then we minimise this product automaton to obtain the FSM $\mathcal{M}^{\varphi}$ depicted in Figure 3. The figure shows the respective output symbols of the FSM below the corresponding state labels, e.g., for state q_1 we have $\bar{\lambda}(q_1) = \bot$. Note that the minimisation removed one of the originally four states of the product automaton. The FSM $\mathcal{M}^{\varphi}$ corresponds with the original intuition, and yields ? while neither event occurred, and either $\top$ or $\bot$, otherwise.

[3]See `http://www.apache.org/`.

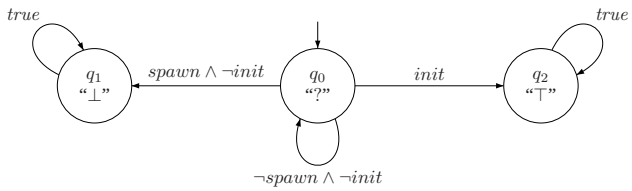


Fig. 3. The deterministic FSM $\mathcal{M}^{\varphi}$ for $\varphi \equiv \neg spawn\ U\ init$.

III. LTL$_3$ PUT IN PERSPECTIVE

Let us compare the approach carried out in the previous section with some accomplishments in the literature. More specifically, we compare LTL$_3$'s semantics when faced with so-called *good* and *bad* prefixes. Furthermore, we consider monitoring for the subclass of (co)-safety properties and we compare LTL$_3$'s semantics to approaches based on *informative* prefixes.

Let, as above, $\Sigma = 2^{\mathrm{AP}}$ be an alphabet for the remainder of this section.

A. Good/Bad Prefixes

Let us first recall the notion of *good* and *bad prefixes* as introduced in [22].

Definition 8 (Good/bad prefixes [22]) *Let $L \subseteq \Sigma^{\omega}$ be a language of infinite words over Σ. A finite word $u \in \Sigma^*$ is called*

- *a* bad prefix *for L, if for all $w \in \Sigma^{\omega}$, $uw \notin L$,*
- *a* good prefix *for L, if for all $w \in \Sigma^{\omega}$, $uw \in L$.*

Note that every continuation uv of a bad (good) prefix u for L by a finite word $v \in \Sigma^*$ is again a bad (good) prefix for L. A bad (good) prefix u is called *minimal*, if each strict prefix of u is not a bad (good) prefix anymore.

Using these terms, we can rephrase the semantics of LTL$_3$ as:

Remark 9 (LTL$_3$ identifies good and bad prefixes) *Given an LTL-formula φ and a finite word $u \in \Sigma^*$, then*

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \text{ is a good prefix for } \mathcal{L}(\varphi) \\ \bot & \text{if } u \text{ is a bad prefix for } \mathcal{L}(\varphi) \\ ? & \text{otherwise.} \end{cases}$$

Thus, the monitor procedure given in the previous section determines for a finite prefix of a potentially infinite word, whether it is good, bad, or neither good nor bad. More specifically, when considering the finite prefixes of an infinite word by increasing length, the monitor identifies its *minimal* good or bad prefix, if such a prefix exists.

Note that one of the contributions of [23] is to modify a given Büchi automaton, typically arising from a given LTL property, into a monitor which signals the occurrence of a minimal bad prefix. Thus, this construction yields a monitor

which distinguishes two cases, namely $[\ u \models \varphi] = \bot$ and $[\ u \models \varphi] \neq \bot$. At the same time, [23] does not discuss the semantics of the resulting monitor in terms a matching logical framework. Thus, LTL_3 can be understood as a logic which complements the constructions carried out in [23] with a formal framework. Nevertheless, we feel that our constructions are more direct and therefore easier to understand.

In practice, whenever a good or bad prefix is found, monitoring can be stopped, as every finite or infinite continuation of the prefix yields the same semantics with respect to LTL_3. For the (minimal) monitor $\mathcal{M}^\varphi$ (see Definition 6), a good or bad prefix leads to a state, which either outputs $\top$ or $\bot$, and which is only looping back to itself. We call such a state a *trap*.

Besides $\top$ and $\bot$, there can be a further trap in the monitor, as there can be a state, in which the output is ?, and from where no state with output $\top$ or $\bot$ is reachable anymore. Consider, for example, the language defined by GFp, stating that there are infinitely many states satisfying p. Any finite word can be extended to an infinite one satisfying the formula as well as to one falsifying the formula. Thus, given any finite word, no finite continuation yields $\top$ or $\bot$ with respect to LTL_3. For runtime verification, such a prefix is *ugly*, since after processing it, monitoring can be stopped yet with an inconclusive result.

Definition 10 (Ugly prefix) *Let $L \subseteq \Sigma^\omega$ be a language of infinite words over Σ. A finite word $u \in \Sigma^*$ is called an* ugly prefix *for L, if there is no $v \in \Sigma^*$ such that uv is either bad or good.*

We follow [24] in calling a formula φ *non-monitorable* with respect to a prefix u, if no $\bot$ or $\top$ verdict can be obtained. Using our terminology, we define:

Definition 11 ((Non)-monitorable) *Let φ be an LTL-formula and $u \in \Sigma^*$. We call φ* non-monitorable after u*, if u is an ugly prefix of $\mathcal{L}(\varphi)$. We call φ* monitorable*, if $\mathcal{L}(\varphi)$ has no ugly prefix.*

In other words, we call φ *monitorable*, if there is no $u \in \Sigma^*$ such that φ is non-monitorable after u.

The discussion above renders the structure of the deterministic monitor $\mathcal{M}^\varphi$ for an LTL_3 formula φ as depicted in Figure 4. In general, a monitor has three traps, corresponding to reading either a good, bad, or ugly prefix. As long as no trap is reached, the monitor outputs ?, while reaching a trap also implies that monitoring can be stopped (since the output will never change again).

In [23], the notion of a *never-violate state* was introduced for a state of a monitor, from which no bad state is reachable. Additionally, an algorithm was outlined for merging all never-violate states of a given Büchi automaton into a single never-violate state. In terms of Figure 4, both *ugly* and *good* are never-violate states, i.e., in [23], both are collapsed into a single never-violate state. Our monitor construction yields (at most) two such never-violate states, *good* and *ugly*. However, we think that it is essential for a prefix $u \in \Sigma^*$ to distinguish whether it is a good or an ugly prefix, as in the previous case the property to be monitored has been satisfied while in the latter case, no satisfaction or violation can be shown by considering continuations of u.

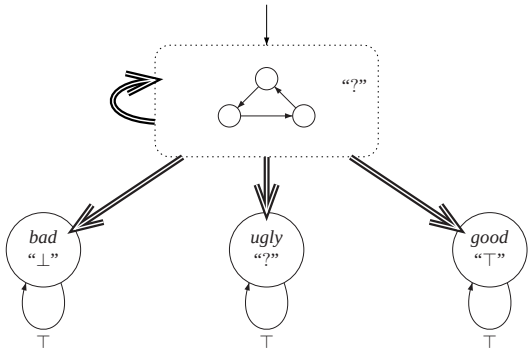


Fig. 4. The structure of the deterministic monitor

Note, that the notion of non-monitorable fits well to LTL_3. In [46], however, we suggest a more precise semantics of LTL-formulae with respect to finite words allowing to differentiate ugly prefixes. The idea is based on using a *strong* as well as a *weak* version of the next-state operator, essentially giving rise to a four-valued semantics. Then, monitoring of non-monitorable properties can still be considered meaningful, but this discussion is beyond the scope of this paper.

B. Safety and Co-safety Properties

We continue the comparison of LTL_3's semantics with existing concepts: The notion of bad and good prefixes was introduced in [22] in the context of safety and co-safety languages and formulae:

Definition 12 (Safety/Co-safety language [22]) *A language $L \subseteq \Sigma^\omega$ is called*

- *a* safety language*, if for all $w \notin L$, there is a prefix $u \in \Sigma^*$ of w which is a bad prefix for L.*
- *a* co-safety language*, if for all $w \in L$, there is a prefix $u \in \Sigma^*$ of w which is a good prefix for L.*

This notion is lifted to LTL formulae in the expected manner:

Definition 13 (Safety/Co-safety property) *A formula $\varphi \in$ LTL is called a* safety property *(*co-safety property*), if its set of models $\mathcal{L}(\varphi)$ is a safety language (co-safety language, respectively).*

Let us give some examples:

formula	safety	co-safety
Gp	•	
Fq		•
Xp	•	•
GFp		
$Xp \vee GFp$		
pUq		•

The definitions of safety and co-safety properties and languages immediately yield:

Remark 14 (Safety/Co-safety properties are monitorable) *Every LTL formula that is safety or co-safety is monitorable.*

In other words, for a safety or a co-safety language L, there are no ugly prefixes and for a safety or co-safety formula φ, the monitor $\mathcal{M}^\varphi$ has no ugly state. However, this property also holds for (some) non safety/co-safety properties:

Lemma 15 (Monitorable is more than safety & co-safety) *The class of monitorable LTL_3 properties is strictly larger than the union of safety and co-safety properties.*

Proof: Consider, for example, $\varphi = ((p \vee q)\,U r) \vee Gp$. Observe that the trace

- $ppp\ldots$ satisfies φ,
- $qqqq\ldots$ does not satisfy φ,
- $\ldots r$ is a good prefix for φ (provided that one of p or q holds in the positions denoted with $\ldots$, and
- $\ldots\{\neg p, \neg q, \neg r\}$ is a bad prefix for φ.

As $ppp\ldots$ satisfies φ but none of its finite prefixes is good, φ is not a co-safety property. As $qqq\ldots$ does not satisfy φ but none of its finite prefixes is bad, φ is neither a safety property. Nevertheless, any finite prefix that is neither good or bad can be extended to a good or a bad prefix: any letter containing r makes the prefix good, while a continuation by the letter $\{\neg p, \neg q, \neg r\}$ makes the prefix bad. Thus, the monitor $\mathcal{M}^\varphi$ for φ does not have any ugly state. ∎

The previous lemma contradicts the popular belief that monitoring is only suitable for safety properties. That said, there is something particular about safety properties: By definition, any infinite word w not satisfying a safety property φ, must have a bad prefix u. Hence, when we never reach the bad trap in an automaton for a safety property φ, then we know that the word w satisfies φ. Thus, assuming that one could predict the monitor output to be the infinite sequence $???\ldots$, one could classify the property as satisfied. This supports the intuition that, if nothing has gone wrong for an unbounded elapse of time, the property to be checked is indeed satisfied. The proof of the previous lemma shows that this property does not hold for all monitorable properties. For example, both $ppp\ldots$ and $qqq\ldots$ do not reach a trap when monitoring $\varphi = ((p \vee q)\,U r) \vee Gp$. Thus, the monitor $\mathcal{M}^\varphi$ will output the infinite sequence $???\ldots$ when monitoring either of these two words. However, $ppp\ldots$ satisfies φ while $qqq\ldots$ does not satisfy φ. Thus, even assuming that one could predict the monitor output to be $???\ldots$, one cannot classify the property as satisfied or violated, as both $ppp\ldots$ and $qqq\ldots$ yield the same output sequence $???\ldots$.

To summarise this discussion, we note that

- violations of safety properties are reported by monitoring procedures which check for finite prefixes of violating words, as well as
- successful validations of co-safety properties are reported by monitoring procedures which consider finite prefixes of satisfying words, but additionally,
- there are monitorable properties which are neither characterised by finite violating or finite satisfying prefixes.

C. Informative prefixes

In [22], the authors have introduced the notion of *informative prefixes*. The idea is to consider prefixes of infinite words that "*tell the whole story*" why a formula is (not) satisfied [22]. Consider, for example, the formula $X\,false$. While clearly unsatisfiable, one might argue that this becomes only "*obvious*" after considering a first letter of some word w: $X\varphi$ holds iff φ holds in the second position of w. For $X\,false$, this means that $false$ should hold in the second position of w, which is *obviously* not the case. Thus, while the empty prefix is not informative, every prefix of length one is.

Following the development of [22], we consider LTL formulae in negation normal form, i. e. the set of formulae defined by the following grammar:[4]

$$\varphi ::= true \mid false \mid p \mid \neg p \qquad (p \in \mathrm{AP})$$
$$\mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi U \varphi \mid \varphi R \varphi \mid X\varphi$$

The semantics is defined as expected, e. g. the semantics of the *release* quantifier is defined such that $\varphi_1 R \varphi_2$ is equivalent to $\neg(\neg\varphi_1\, U \neg\varphi_2)$. We use $\neg\varphi$ as a shorthand for its positive form, i. e. the formula obtained by negating φ and pushing all negations down reaching either a Boolean constant or an atomic proposition. The *closure* $cl(\varphi)$ of φ is defined as its set of subformulae.

Definition 16 (Informative prefix [22]) *For an LTL formula φ and a finite word $u = a_0 \ldots a_n$ with $a_i \in \Sigma$, we say that u is* informative *for φ, if there exists a mapping $\ell : \{0, \ldots n+1\} \to 2^{cl(\varphi)}$ such that the following holds:*

- $\neg\varphi \in \ell(0)$,
- $\ell(n+1) = \emptyset$, *and*
- *for all $0 \leq i \leq n$ and $\psi \in \ell(i)$, we have*
  - *if ψ is an atomic proposition, then a_i satisfies ψ,*
  - *if $\psi = \psi_1 \vee \psi_2$, then $\psi_1 \in \ell(i)$ or $\psi_2 \in \ell(i)$,*
  - *if $\psi = \psi_1 \wedge \psi_2$, then $\psi_1 \in \ell(i)$ and $\psi_2 \in \ell(i)$,*
  - *if $\psi = X\psi_1$, then $\psi_1 \in \ell(i+1)$,*
  - *if $\psi = \psi_1 U \psi_2$, then $\psi_2 \in \ell(i)$, or, $\psi_1 \in \ell(i)$ and $\psi_1 U \psi_2 \in \ell(i+1)$,*
  - *if $\psi = \psi_1 R \psi_2$, then $\psi_2 \in \ell(i)$ and, $\psi_1 \in \ell(i)$ or $\psi_1 R \psi_2 \in \ell(i+1)$.*

If u is informative for φ, the existing mapping ℓ is called a *witness* for $\neg\varphi$ in u. Note that the emptiness of $\ell(n+1)$ guarantees that all the requirements imposed by $\neg\varphi$ are fulfilled along u. The definition implies

Remark 17 (Informative implies bad) *Let φ be an LTL formula. Every informative prefix for φ is a bad prefix for φ.*

Note that the converse is not true, i.e., there are formulae which have bad prefixes but no informative ones, as shown in the examples below:

Example 18 (Informative prefixes)

[4]While the ideas presented below can also be developed in the version of LTL defined in Section II (as done for example in [21]), we follow [22] to simplify the presentation.

- *Consider Gp and $u = pq$. Note that u is a bad prefix for Gp and that $\neg Gp = F\neg p$.*[5] *Then, ℓ_1 defined by $\ell_1(0) = \{F\neg p\}$, $\ell_1(1) = \{F\neg p, \neg p\}$, $\ell_1(2) = \emptyset$ is a witness for $\neg\varphi$, showing that u is informative for φ.*
- *Consider $\varphi_2 = G(p \vee X false)$ and $u = pq$ as before. As φ_2 is equivalent to Gp, u is still a bad prefix for φ_2. Note, $\neg\varphi_2 = F(\neg p \wedge X true)$. Thus, some witness ℓ_2 should satisfy $F(\neg p \wedge X true) \in \ell_2(0)$. As p is satisfied in the first position of u, it has to hold that $\{F(\neg p \wedge X true), \neg p \wedge X true, X true\} \subseteq \ell_2(1)$. This implies that $true \in \ell_2(2) \neq \emptyset$. Thus, there is no witness for $\neg\varphi_2$ in u. However, adding an arbitrary letter to u turns it into an informative prefix and allows ℓ_2 to be extended to a witness for $\neg\varphi_2$.*
- *Consider $\varphi_3 = G(p \vee F false)$ and $u = pq$. As φ_3 is equivalent to Gp, u is still a bad prefix for φ_3. Note, $\neg\varphi_2 = F(\neg p \wedge G true)$. Now, having $G true$ as a subformula in some possible witness $\ell_3(i)$ requires $G true$ to be in any $\ell_3(j)$ for $j \geq i$, as $true$ cannot be falsified of any position of u. Thus, while u is a bad prefix showing that φ_3 does not hold for any continuation of u, there is no continuation of u that is informative.*

The example shows that the notion of *informativeness* for a property φ depends on the syntactical representation of φ. The example further highlights that checking for informative prefixes is closely related to the tableau-based [47] and alternating-automata-based approach to model checking LTL formulae [36]: The witness ℓ for some formula $\neg\varphi$ in u can be considered as a finite accepting tableaux for $\neg\varphi$, in the sense of [47].

The notion of informativeness is used to classify safety properties into three distinct safety levels:

Definition 19 (Safety levels [22]) *A safety property $\varphi \in LTL$ is called*

- intensionally safety, *if all its bad prefixes are informative,*
- accidentally safety, *if every word that violates φ has an informative prefix,*
- pathologically safety, *if there is a word that violates φ which has no informative prefix.*

We use the formulae previously studied to exemplify the notion of safety levels:

- Gp is intensionally safety.
- $G(p \vee X false)$ is accidentally safety.
- $G(p \vee F false)$ is pathologically safety.

Note, however, that all three formulae are equivalent, i.e., they accept the same set of models.

Note that, interestingly, [21] gives a semantical characterisation of informative prefixes in terms of a *weak* semantics of LTL on finite traces, though for the discussion to come, we stick to the syntactical presentation.

The monitor generation procedures given in [25] and [31] follow a tableau-style approach for checking violations of LTL (safety) properties. More specifically, these procedures will report informative bad prefixes (as stated explicitly in [25] and implicitly in [31]). In [22], the authors suggest that one could search for an informative prefix for φ as well as the negation of φ.

Because of Remark 17, such a search procedure would stop upon either some good or bad prefix—however, these prefixes have to be informative at the same time. Hence, in all the above mentioned approaches, it is possible that, for example, a bad prefix is examined, meaning the property to monitor is not satisfied, yet the word is not reported because it is not informative.

In the setting of safety properties, one might argue that the user of a monitor generation tool should only be allowed to generate monitors for intensionally safety properties and not also for accidentally or pathologically safety properties. Then, of course, monitors identifying only informative prefixes suffice to report all bad prefixes. However, while [22] provides a decision procedure for checking whether a formula is intensionally safety, no conversion algorithm from non-intensionally to intensionally safety formulae is given. For a user of a monitor generation tool, it might be interesting to learn that the property to monitor is not intensionally safety. However, it might be too hard and cumbersome for him or her to carry out a translation manually—and not necessary when following our construction.[6]

Though debatable, we consider monitors checking exclusively for informative bad prefixes, such as in [23], inferior to our monitors which check for bad and good prefixes, as the latter follow the maxim of reporting a violation (or satisfaction) as early as possible.

IV. Three-valued LTL in the Real-time Setting—TLTL

In this section, we consider runtime verification for real-time systems emitting *events* at dedicated *time points*. Thus, for monitoring, we may observe a sequence of events ranging over some alphabet Σ paired together with a *time stamp* (a real value), identifying when exactly the event happened. Thus, the behaviour of the system under scrutiny is described by an (in)finite *timed word* over the alphabet $\Sigma \times \mathbb{R}^{\geq 0}$.

Note, that in the discrete-time setting of LTL, we considered (sequences of) states of systems defined by Boolean combinations of atomic *propositions*, while here, we deal with systems emitting *events* at dedicated time points. We prefer this event-based approach in the real-time case, since otherwise one would have to deal with certain ambiguities: If one specifies that within 5 time units, both, propositions a and b must evaluate to true, the question arises, whether a and b are required to be true at the same point in time or not. If one is indeed interested in expressing that $a \wedge b$ must become true within 5 time units, then the semantics of the underlying logic must support the timed observation of such Boolean combinations of propositions—leading to a more complicated logic to start with. By following an event-based approach, we avoid these issues entirely. Moreover, one

[5] Recall that $\neg\varphi$ is a shorthand for the negation of φ in negation normal form.

[6] However, there might be a price to pay: When reporting a bad but not informative prefix to a user, it might be harder to understand for her or him *why* the prefix is indeed bad.

can still express a proposition-based property within the event-based framework by introducing an event for each change of the relevant Boolean formulae. Therefore, we do not consider the proposition-based approach in this section.

A logic suitable for expressing properties of such timed words is *timed lineartime temporal logic* (TLTL), which is a timed variant of LTL, originally introduced by Raskin in [28]. TLTL, as argued by D'Souza, can be considered a natural counterpart of LTL in the timed setting. Consequently, our developments in this section are for TLTL specifications.

TLTL is very well suited for expressing simple yet typical bounded response properties, such as requiring that an event a occurs in three time units.

In LTL, such a property is typically expressed as the formula $\varphi \equiv XXXa$. However, this formulation presumes a direct correspondence of discrete time delays with subsequent positions in the word. Actually, what should really be expressed is that the event a occurs after three time units *regardless how many other events between before*.

A main feature of TLTL is that it does not impose any mutual dependency between the frequency of the occurring events on the one hand side, and between the corresponding time stamps on the other hand. Henceforth, TLTL is especially suitable for specifying properties of *asynchronous* systems.

After recalling standard TLTL syntax and semantics, we introduce a three-valued semantics for evaluating standard TLTL formulae on finite timed words, following the same rationale as for LTL_3 (Sections IV-A and IV-B). In Section IV-C, we continue with a detailed overview on the now more involved monitor construction, which spans over Sections IV-D throughout IV-F. We conclude the real-time case with Section IV-G on issues arising in adapting our monitor construction to specific platforms.

A. Preliminaries

Let us fix an alphabet Σ of events for the rest of this section. In the timed setting, the occurrence of every event $a \in \Sigma$ is associated with a corresponding time stamp and therefore a timed word is a sequence $(a_0, t_0)(a_1, t_1)\dots$ of timed events $(\Sigma \times \mathbb{R}^{\geq 0})$:

Definition 20 (Timed Word [48]) *An (infinite)* timed word w *over the alphabet* Σ *is an (infinite) sequence* $(a_0, t_0)(a_1, t_1)\dots$ *of* timed events (a_i, t_i) *consisting of symbols* $a_i \in \Sigma$*, and non-negative numbers* $t_i \in \mathbb{R}^{\geq 0}$*, such that*

- *for each* $i \in \mathbb{N}$, $t_i < t_{i+1}$ *holds (*strict monotonicity*), and such that,*
- *in case of infinite words, for all* $t \in \mathbb{R}^{\geq 0}$ *there exists an* $i \in \mathbb{N}$ *with* $t_i > t$ *(*progress*).*

The *length* of a timed word is denoted with $|w|$ where we set $|w| = \infty$ for an infinite word and $|w| = n$ for a finite word $w = (a_0, t_0)\dots(a_{n-1}, t_{n-1})$.

To simplify notation, we abbreviate $(\Sigma \times \mathbb{R}^{\geq 0})$ by $T\Sigma$. Further, we define $T\Sigma^*$ and $T\Sigma^\omega$ as the set of finite and infinite timed words, respectively, i.e., every word in $T\Sigma^*$ satisfies strict monotonicity and every word in $T\Sigma^\omega$ satisfies both, strict monotonicity as well as progress.

We use *finite* and *infinite continuations* of finite timed words throughout the section. Thereby, the strict monotonicity of timed words is required to hold, i.e., for a finite timed word $u = (a_0, t_0)\dots(a_i, t_i) \in T\Sigma^*$, we consider only those timed words as continuations which start with a timed event (a_{i+1}, t_{i+1}) such that $t_{i+1} > t_i$ holds. For the sake of simplicity, when we refer to some continuation of u, we do not explicitly mention this condition $t_{i+1} > t_i$.

Furthermore, for w as above, we call its sequence of events (the projection to the first component) the *untimed word* of w, denoted with $\mathtt{ut}(w) = a_0 a_1 \dots$ and we write $\mathtt{ut}(\mathcal{L}) = \{\mathtt{ut}(w) \mid w \in \mathcal{L}\}$ for a finite or infinite timed language with $\mathcal{L} \subseteq T\Sigma^*$ or $\mathcal{L} \subseteq T\Sigma^\omega$, respectively.

Every event $a \in \Sigma$ is associated with an *event-recording clock*, x_a, and an *event-predicting clock*, y_a. Given an (infinite) timed word w, the value of the event-recording clock variable x_a at position i of w equals $t_i - t_j$, where j is the last position preceding i such that $a_j = a$. If no such position exists, then x_a is assigned the undefined value, denoted by $\perp$. The event-predicting clock variable y_a at position i equals $t_k - t_i$, where k is the next position after i such that $a_k = a$. If no such position exists, again, the variable is assigned $\perp$.

We compute the values of event-recording and event-predicting clocks with the following two functions which take a timed word $w = (a_0, t_0)(a_1, t_1)\dots \in T\Sigma^* \cup T\Sigma^\omega$, an event $a \in \Sigma$, and an index i as arguments:

$\mathsf{last}(w, a, i) = t_i - t_j$	iff	$a_j = a$ and	$0 \leq j < i$
	and	$a_k \neq a$ for all	$j < k < i$
$\mathsf{last}(w, a, i) = \perp$	iff	$a_j \neq a$ for all	$0 \leq j < i$
$\mathsf{next}(w, a, i) = t_j - t_i$	iff	$a_j = a$ and	$i < j < \lvert w \rvert$
	and	$a_k \neq a$ for all	$i < k < j$
$\mathsf{next}(w, a, i) = \perp$	iff	$a_j \neq a$ for all	$i < j < \lvert w \rvert$

For the set of all event-clocks $C_\Sigma = \{x_a, y_a \mid a \in \Sigma\}$, we summarise these evaluation rules with the next definition:

Definition 21 (Clock Valuation Function) *A* clock valuation function $\gamma_i : C_\Sigma \to T_\perp$ *with* $T_\perp = \mathbb{R}^{\geq 0} \cup \{\perp\}$ *over a timed word* $w = (a_0, t_0)(a_1, t_1)\dots \in T\Sigma^* \cup T\Sigma^\omega$ *assigns a positive real or the undefined value* $\perp$ *to each clock variable corresponding to position* i *such that the following holds:*

$$\gamma_i(x_a) = \mathsf{last}(w, a, i)$$
$$\gamma_i(y_a) = \mathsf{next}(w, a, i)$$

The set of clock valuation functions *over the clock* C_Σ *is denoted with* V_Σ.

Thus, γ_i describes the evaluation of the clocks in C_Σ at time t_i—but it *ignores* the event a_i: $\gamma_i(x_{a_i}) = \mathsf{last}(w, a_i, i)$ does *not* evaluate to 0 but refers to the penultimate occurrence of a_i (if no such occurrence exists, then $\gamma_i(x_{a_i}) = \perp$). Likewise $\gamma_i(y_{a_i}) = \mathsf{next}(w, a_i, i)$ does *not* evaluate to 0 but refers to the next future occurrence of a_i (analogously, if no such occurrence exists, then $\gamma_i(y_{a_i}) = \perp$). Therefore, at the time t_i when the event a_i occurs, we refer to the last past and the next future occurrence of a_i and ignore its current occurrence:

Remark 22 *Given a timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^* \cup T\Sigma^\omega$, and a corresponding sequence of clock valuation functions $\gamma_0, \gamma_1, \ldots$, note that each γ_i describes the time distances to the last preceding and next subsequent events relative to time instant t_i—but it is* independent from the current event a_i.

This definition leads to the following *initial* clock valuation γ_0 which holds before the first timed event (a_0, t_0) has been processed:

- $\gamma_0(x_a) = \bot$ for all x_a, and
- $\gamma_0(y_a) = \mathsf{next}(w, a, 0)$.

Thus, even the initial clock valuation function γ_0 (as well as every subsequent clock valuation function γ_i) depends on the entire word w because $\gamma_0(y_a) = \bot$ holds iff a does not occur in w at all. In the context of runtime verification, this imposes a problem, since a monitor observing a running system is unable to access future events—and consequently—it cannot evaluate the clock valuation function. To solve this problem, we introduce in Section IV-D *symbolic* timed runs and prove their appropriateness for our needs.

Clock valuation functions are defined with respect to a timed word and hence the strict monotonicity property and the potential infinite length of timed words imply the following two properties upon valid clock valuation functions:

Proposition 23 (Properties of Clock Valuation Functions) *Let $\gamma \in \mathrm{V}_\Sigma$ be a clock valuation function over the clocks C_Σ. Then the following two conditions hold:*

(a) Non-Coincidence. *For all events $a \neq a' \in \Sigma$, $\gamma(x_a) \neq \gamma(x_{a'})$ and $\gamma(y_a) \neq \gamma(y_{a'})$.*

(b) Continuity. *If γ refers to an infinite timed word, then there exists at least one clock $y_a \in C_\Sigma$ such that $\gamma(y_a) \neq \bot$ holds.*

Proof: Property (a), non-coincidence, holds because strict monotonicity disallows two events $(a_i, t_i)(a_{i+1}, t_{i+1})$ with $t_i = t_{i+1}$. Property (b), continuity, holds, since each infinite word has an infinite supply of events and hence there must occur some event in the future. ∎

We use the following notation to manipulate a clock valuation function $\gamma \in V_\Sigma$:

- For a clock $c \in C_\Sigma$ and a value $v \in T_\bot = \mathbb{R}^{\geq 0} \cup \{\bot\}$ we define $\gamma[c = v] \in V_\Sigma$ with

$$\begin{array}{ll} \gamma[c = v](c') = \gamma(c') & \text{iff } c' \neq c \\ \gamma[c = v](c') = v & \text{iff } c' = c \end{array}$$

- For $\delta \in \mathbb{R}^{\geq 0}$, we define

$$\begin{array}{ll} (\gamma \pm \delta)(x_a) = \gamma(x_a) \pm \delta & \text{iff } \gamma(x_a) \neq \bot \\ (\gamma \pm \delta)(x_a) = \bot & \text{iff } \gamma(x_a) = \bot \\ (\gamma \pm \delta)(y_a) = \gamma(y_a) \mp \delta & \text{iff } \gamma(y_a) \neq \bot \\ (\gamma \pm \delta)(y_a) = \bot & \text{iff } \gamma(y_a) = \bot \end{array}$$

  where we require $\gamma(y_a) \geq \delta$ for all event-predicting clocks y_a in case of $\gamma + \delta$ and $\gamma(x_a) \geq \delta$ for all event-recording clocks x_a in case of $\gamma - \delta$. Otherwise $\gamma \pm \delta$ is *invalid*.

To constrain the value of a clock at a certain point in time, i.e., to constrain the valuations of a γ_i, we need to formulate constraints over $T_\bot$. To do so, we define the set $\mathcal{I}$ to encompass the intervals over the positive reals $\mathbb{R}^{\geq 0}$ and the singleton $\{\bot\}$.

Definition 24 (Intervals) *The set $\mathcal{I}$ of* intervals *contains all intervals of the form $\llbracket l, r \rrbracket$ where $\llbracket$ and $\rrbracket$ either be (or [, respectively) or] and with $l, r \in \mathbb{N}$ and $l < r$ except for intervals of the form $[l, r]$ where we require $l \leq r$ instead. $\mathcal{I}$ also contains all intervals of the form $\llbracket l, \infty)$ for $l \in \mathbb{N}$. These intervals are interpreted as subsets from $\mathbb{R}^{\geq 0}$ in the usual way.*

Furthermore, $\mathcal{I}$ contains the interval $[\bot, \bot]$ with $[\bot, \bot] = \{\bot\}$.

For the sake of simplicity, we sometimes write the value t for an interval $[t, t]$, in particular in case of clock constraints, as defined next. The clock constraints over the clocks in C_Σ rely on the intervals $\mathcal{I}$ as their basis: Each such clock constraint requires a number of clocks in C_Σ to assume corresponding values in a respective interval $I \in \mathcal{I}$.

Definition 25 (Clock Constraint) *Let Σ be a finite alphabet of events with the associated set C_Σ of clocks. Then a* clock constraint *is a partial function $\psi : C_\Sigma \to \mathcal{I}$. If $\psi(c)$ is undefined, we write $\psi(c) = \mathsf{undef}$.*

A clock valuation function $\gamma \in V_\Sigma$ over the clocks C_Σ satisfies *a clock constraint ψ, iff $\gamma(c) \in \psi(c)$ holds for all $c \in C_\Sigma$ with $\psi(c) \neq \mathsf{undef}$. Then we write $\gamma \models \psi$.*

The set of constraints *on the clocks C_Σ is denoted with Ψ_Σ and contains all satisfiable constraints ψ.*

Thus, if $\psi(c) = \mathsf{undef}$ for a clock $c \in C_\Sigma$, then ψ does not constrain the value of c, i.e., the value $\gamma(c)$ for the clock c of a clock valuation γ with $\gamma \models \psi$ can be chosen arbitrarily.

We define the set of constraints Ψ_Σ to contain only the satisfiable constraints to meet Proposition 23: Every clock valuation function must satisfy non-coincidence and continuity—and hence each clock constraint $\psi \in \Psi_\Sigma$ must not enforce coincident or non-continuous clock valuation functions:

Remark 26 (Properties of Clock Constraints) *Each clock constraint $\psi \in \Psi_\Sigma$ has a non-coincident and continuous solution.*

The reason for using intervals in the definition of the clock constraints is twofold: First, we can use them for the definition of both, TLTL and the corresponding automaton model, i.e., event-clock automata. And second, we use the fact that Ψ_Σ is closed under conjunction for an efficient scheme to symbolically execute event-clock automata:

Remark 27 (Conjunction of Clock Constraints) *If the two clock constraints $\psi_0, \psi_1 \in \Psi_\Sigma$ are consistent, i.e., there exists a clock valuation function $\gamma \in V_\Sigma$ such that $\gamma \models \psi_i$ for $i = 0, 1$, then their* conjunction *$\psi = \psi_0 \wedge \psi_1$ is defined with*

$$\begin{array}{lll} \psi(c) = \psi_0(c) \cap \psi_1(c) & \textit{iff} & \psi_i(c) \neq \mathsf{undef} \textit{ for } i = 0, 1 \\ \psi(c) = \psi_i(c) & \textit{iff} & \psi_i(c) \neq \mathsf{undef} \\ & \textit{and} & \psi_{1-i}(c) = \mathsf{undef} \\ \psi(c) = \mathsf{undef} & \textit{iff} & \psi_i(c) = \mathsf{undef} \textit{ for } i = 0, 1 \end{array}$$

Above, we require $\gamma \models \psi_i$ for $i = 0, 1$ for $\psi_0 \wedge \psi_1$ to be defined and valid—since then $\gamma \models (\psi_0 \wedge \psi_1)$ holds and $\psi_0 \wedge \psi_1$ has indeed a non-coincident and continuous solution.

We also use the notation $\psi[c = I]$ for $\psi \in \Psi_\Sigma$, $c \in C_\Sigma$, and $I \in \mathcal{I}$, to denote the clock constraint which agrees with ψ for all clocks $c' \neq c$ and yields I for c. Hence we have $\psi[c = I](c') = \psi(c')$ for $c' \neq c$ and $\psi[c = I](c) = I$. Analogously, $\psi[c = \mathsf{undef}]$ is undefined for c and agrees with ψ for all other clocks $c' \neq c$. Finally, $\psi + \delta$ with $\delta \in \mathbb{R}^{\geq 0}$ is defined as

$$\begin{array}{ll}(\psi+\delta)(x_a) = [\![l+\delta, r+\delta]\!] & \text{iff } \psi(x_a) = [\![l, r]\!] \\ (\psi+\delta)(x_a) = \psi(x_a) & \text{iff } \psi(x_a) \in \{[\bot, \bot], \mathsf{undef}\} \\ (\psi+\delta)(y_a) = [\![l \dot{-} \delta, r-\delta]\!] & \text{iff } \psi(y_a) = [\![l, r]\!] \\ (\psi+\delta)(y_a) = \psi(y_a) & \text{iff } \psi(y_a) \in \{[\bot, \bot], \mathsf{undef}\}\end{array}$$

where we use $a \dot{-} b = \max\{0, a - b\}$ and where we require that no interval $(\psi + \delta)(y_a) = [\![l \dot{-} \delta, r - \delta]\!]$ becomes empty. Otherwise, if at least one interval becomes empty, $\psi + \delta$ is *invalid*.

In what follows, we use some basic relationships between clock valuation function and clock constraints:

Fact 28 (Clock Valuation Functions & Constraints) *Let $\gamma \in V_\Sigma$ be a clock valuation function and let $\psi \in \Psi_\Sigma$ be a clock constraint such that $\gamma \models \psi$ holds.*

(a) *If $\gamma + \delta$ is valid, then $\psi + \delta$ is valid as well and $\gamma + \delta \models \psi + \delta$ holds.*
(b) *For a clock $c \in C_\Sigma$ and a value $v \in T_\perp$ with $v \in I$ for some interval $I \in \mathcal{I}$, $\gamma[c = v] \models \psi[c = I]$ holds.*
(c) *For a clock $c \in C_\Sigma$ and an arbitrary value $v \in T_\perp$ $\gamma[c = v] \models \psi[c = \mathsf{undef}]$ holds.*

B. Syntax and Semantics of TLTL$_3$

For a finite set Σ of events, we introduce the formulae of TLTL by adding to LTL two new forms of atomic formulae: First, $\lhd_a \in I$ asserts that the time since $a \in \Sigma$ has occurred the last time lies within the interval $I \in \mathcal{I}$. And second, $\rhd_a \in I$ analogously asserts that the time until a occurs again lies within I. The semantics of $\lhd_a \in I$ is that $\gamma(x_a) \in I$ must hold at the point of evaluation, and analogously, in case of $\rhd_a \in I$, it is required that $\gamma(y_a) \in I$ holds. This timed variant of LTL is taken from [29] where is called LTL$_{ec}$.

Definition 29 (TLTL Formulae [29]) *The set of* formulae φ of TLTL *is defined by the grammar*

$$\varphi ::= \mathit{true} \mid a \mid \lhd_a \in I \mid \rhd_a \in I \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \; U \; \varphi \mid X\varphi,$$

for $a \in \Sigma$ and $I \in \mathcal{I}$.

Again, as in the discrete-time case, we use three abbreviations in our notation: $\varphi \wedge \psi$ for $\neg(\neg\varphi \vee \neg\psi)$, $F\varphi$ for $\mathit{true}\; U\; \varphi$, and $G\varphi$ for $\neg(\mathit{true}\; U\; \neg\varphi)$. Additionally, we write $\lhd_a \notin I$ for $\neg(\lhd_a \in I)$ and $\rhd_a \notin I$ for $\neg(\rhd_a \in I)$ respectively. The semantics of the untimed operators of TLTL formulae is defined as it is for (discrete time) LTL. By adding the semantics for $\lhd_a \in I$ and $\rhd_a \in I$, we obtain an inductive definition of the semantics of TLTL over infinite timed words:

Definition 30 (Semantics of TLTL [29]) *Let $w \in T\Sigma^\omega$ be an infinite timed word with $w = (a_0, t_0)(a_1, t_1)\ldots$, and let $i \in \mathbb{N}^{\geq 0}$. Then the following holds:*

$$\begin{array}{lll}w, i \models \mathit{true} & & \\ w, i \models \neg\varphi & \textit{iff} & w, i \not\models \varphi \\ w, i \models a & \textit{iff} & a_i = a \\ w, i \models \lhd_a \in I & \textit{iff} & \gamma_i(x_a) \in I \\ w, i \models \rhd_a \in I & \textit{iff} & \gamma_i(y_a) \in I \\ w, i \models \varphi_1 \vee \varphi_2 & \textit{iff} & w, i \models \varphi_1 \textit{ or } w, i \models \varphi_2 \\ w, i \models \varphi_1 U \varphi_2 & \textit{iff} & \exists k \geq i \textit{ with } w, k \models \varphi_2 \\ & \textit{and} & \forall l : (i \leq l < k \wedge w, l \models \varphi_1) \\ w, i \models X\varphi & \textit{iff} & w, i+1 \models \varphi\end{array}$$

Finally, we set $w \models \varphi$ iff $w, 0 \models \varphi$.

To illustrate the definition of the syntax and the semantics of TLTL, we give some example properties.

Example 31 (TLTL properties)

- $G(\neg alive \rightarrow \rhd_{alive} \in [0, 5])$ *means that whenever some event different from $alive$ occurs, then the event $alive$ must occur within 5 time units again. Note that this example does allow a sequence $\ldots, (alive, t_i)(alive, t_{i+1})\ldots$ with $t_{i+1} - t_i > 5$, i.e., two adjacent occurrences of $alive$ may be separated by an arbitrary period of time.*
- $G(\rhd_{alive} \in [0, 5])$ *requires that from every given point in time, $alive$ will occur within the next 5 time units. In this example, the subword $\ldots, (alive, t_i)(alive, t_{i+1})\ldots$ with $t_{i+1} - t_i > 5$ is ruled out, since $\gamma_i(y_{alive}) = t_{i+1} - t_i$ is required to be within $[0, 5]$.*
- $G(req \rightarrow \rhd_{ack} \in [0, 5])$ *means that if a request event req arrives, then it must be handled with an acknowledge event ack within 5 time units.*
- $\rhd_{alive} \in [0, 2] U done$ *states that the event $done$ has to occur eventually and that until then, the event $alive$ must occur every 2 time units.*
- $G(req \rightarrow \rhd_{req} \notin [0, 5])$ *requires that two subsequent request events req are separated by strictly more than 5 time units.*
- $G(actuator \rightarrow \lhd_{error} \in [\bot, \bot])$ *states that if an $actuator$ event occurs, then previously, no $error$ has occurred (equivalently, we could write $G(error \rightarrow G\neg actuator)$ in standard LTL).*

Analogously to the discrete-time case, we now define a 3-valued semantics for TLTL, yielding the logic TLTL$_3$.

Definition 32 (Semantics of TLTL$_3$) *Let $u \in T\Sigma^*$ denote a finite timed word. The* truth value *of a TLTL$_3$ formula φ with respect to u, denoted with $[u \models \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:*

$$[u \models \varphi] = \begin{cases} \top & \textit{if } \forall \sigma \textit{ such that } u\sigma \in T\Sigma^\omega \; u\sigma \models \varphi \\ \bot & \textit{if } \forall \sigma \textit{ such that } u\sigma \in T\Sigma^\omega \; u\sigma \not\models \varphi \\ ? & \textit{otherwise.} \end{cases}$$

In the above definition, the truth value of every possible infinite continuation σ of a given finite timed word u is evaluated according to TLTL-semantics. Since σ is a continuation

of $u = (a_0, t_0) \dots (a_i, t_i)$, we only have to consider those infinite words σ which start with a timed event (a_{i+1}, t_{i+1}) such that $t_{i+1} > t_i$ holds.

To illustrate the three-valued semantics, we discuss the evaluation of the first four example properties from above.

Example 33 (TLTL$_3$ Evaluation)

- $G(\neg alive \rightarrow \triangleright_{alive} \in [0,5])$ *evaluates always to* $\top$ *if* $\Sigma = \{alive\}$ *holds. If* Σ *contains any other element, then the TLTL*$_3$*-semantics yields either* $\bot$ *or* $?$*: If an event* $a \neq alive$ *occurred and* $alive$ *did not occur within 5 time units, then the semantics evaluates to* $\bot$*. Otherwise, the result is* $?$*.*
- $G(\triangleright_{alive} \in [0,5])$ *evaluates either to* $\bot$ *or* $?$ *again. If the evaluated finite prefix* u *contains a period of time which is longer than 5 time units and which does not contain a* $alive$ *action, then the result is* $\bot$*. Otherwise, it is* $?$*.*
- $G(req \rightarrow \triangleright_{ack} \in [0,5])$ *yields* $\bot$ *if there occurs a* req *event which is not followed by an* ack *event within 5 time units. Otherwise the result is* $?$*.*
- $\triangleright_{alive} \in [0,2] U done$ *evaluates to* $?$ *if* $done$ *has not occurred so far while two subsequent occurrences of* $alive$ *have never been separated by more than 2 time units. If* $done$ *occurred already and* $alive$ *has been signalled on time beforehand, then the formula evaluates to* $\top$*. Finally, if there are two subsequent occurrences of* $alive$ *which are separated by strictly more than 2 time units before* $done$ *occurred, then the formula evaluates to* $\bot$*.*

C. Overview on TLTL$_3$ Monitoring

In this section, we outline our monitor construction for TLTL$_3$ which we concretise in the subsequent sections. To build a monitor $\mathcal{M}^\varphi$ for a given TLTL$_3$-property φ, we follow roughly the approach taken in the discrete-time case. Thus, we look for a procedure to determine whether there exists an accepting and/or rejecting infinite continuation of a given finite prefix. To obtain such a procedure, we generate for a TLTL$_3$-property φ the two event-clock automata $\mathcal{A}^\varphi_{ec}$ and $\mathcal{A}^{\neg\varphi}_{ec}$ corresponding to φ and its negation $\neg\varphi$, which accept the timed words satisfying and respectively violating φ [49]. Then, following the concepts of the discrete-time case, we run both of them in parallel in order to check whether there exist infinite continuations which let $\mathcal{A}^\varphi_{ec}$ and/or $\mathcal{A}^{\neg\varphi}_{ec}$ accept.

However, in contrast to the discrete-time setting, this procedure faces *predicting clocks* and a more complex *emptiness check* as additional obstacles. Both issues are addressed separately in Sections IV-D and IV-E, respectively. Then in Section IV-F, having suitable techniques at hand, we build the final monitor, following closely the scheme used in the discrete-time setting.

Symbolic Runs of Event-Clock Automata (Section IV-D): As starting point for the monitoring procedure, we recall the Definition of event-clock automata (Definition 34) and their timed runs (Definition 35) over infinite words. In these timed runs, predicting clocks anticipate the time until some event occurs the next time in the future. Given a fixed infinite timed word, such an approach does not impose a problem, however, having only access to a finite prefix of a subsequently continued timed word, it is not possible to evaluate predicting clocks directly. Instead, our monitor executes the event-clock automaton symbolically by maintaining pairs of automaton states and symbolic clock valuations (Definition 36) which describe the viable values for each predicting clock as clock constraint.

After developing a procedure to implement a transition symbolically (Figure 6) and proving that this procedure is indeed abstracting all concrete transitions (Lemma 43), we define symbolic timed runs (Definition 41) of event-clock automata. These symbolic timed runs are not requiring any information beyond the currently known finite prefix of the observed timed word (Remark 42) and are therefore a suitable means for runtime verification. Then we prove that every timed run is abstracted by a corresponding symbolic timed run (Lemma 43).

It would remain to show the converse, i.e., that every symbolic timed run is concretised by a corresponding timed run. However, this is not the case as there are spurious symbolic timed runs which cannot be concretised (Proposition 44). But we can use a backward simulation argument to show that every *individual* symbolic transition has a corresponding concrete transition (Lemma 45). At this point, we can prove that, given an infinite timed word, every symbolic timed run over some *finite prefix* of the given word can be continued by an ordinary infinite timed run iff there exists a timed run over the entire infinite word (Theorem 46). This leads directly to a criterion for runtime verification (Corollary 47): Given a finite prefix of a timed word, this prefix can be continued into an accepted word iff there exists a symbolic timed run leading to a pair of a state and a symbolic clock valuation which gives rise to a non-empty language. Hence, we have to address the emptiness check for symbolic states, as discussed in the next section.

Emptiness Check for Symbolic States (Section IV-E): Thus, after reading a finite timed word, we have a pair with a state of the original event-clock automaton and a symbolic clock valuation describing the viable valuations of each predicting clock. Now we have to check whether there exists an infinite timed word which continues the given prefix and which leads the automaton to acceptance. Starting with general quotient automata (Definition 48) which work with any time-abstract bisimulation relation (Definition 49, [50]), and the emptiness check based upon such automata (Theorem 50, [48]), we obtain a look-up table which answers the question whether a pair consisting of a state and a bisimulation equivalence class has an empty language or not. To use this look-up table, we express symbolic clock valuations as the union of a set of equivalence classes of the underlying time-abstract bisimulation (following the condition given in Corollary 51). Finally, we recall the region equivalence for event-clock automata (Definition 52, [51], [48]) as one particular instance of a bisimulation relation and show how to compute a set of regions which covers a given symbolic clock valuation.

A Monitor Procedure for TLTL$_3$ (Section IV-F): As in the discrete-time setting, given a property φ we run two automata in parallel, namely one for φ and another one for $\neg\varphi$.

We symbolically execute the event-clock automaton $\mathcal{A}_{ec}^{\varphi}$ and check the emptiness for each reached pair consisting of a state and a symbolic clock valuation. In parallel, we do the same with the automaton $\mathcal{A}_{ec}^{\neg\varphi}$ corresponding to the negated property $\neg\varphi$. Then we combine the results of these two evaluations following directly the semantics of TLTL$_3$ to obtain the final verdict.

D. Symbolic Runs of Event-Clock Automata

We first recall event-clock automata: For a given finite alphabet Σ and a corresponding set C_Σ of clocks, an event-clock automaton is a finite state automaton whose edges are annotated both with input symbols from Σ and with clock constraints from Ψ_Σ. Intuitively, such an edge is enabled after reading some timed word, if the corresponding clock valuation function satisfies the clock constraint of the edge in concern.

Definition 34 (Event-Clock Automaton [30] as in [29]) *Let Σ be a finite alphabet and C_Σ the corresponding set of event-recording and event-predicting clocks. Then an* event-clock automaton *is defined as $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ with the following components:*

- *Q is a finite set of states,*
- *$Q_0 \subseteq Q$ is the set of initial states,*
- *$F \subseteq 2^Q$ is the set of accepting state sets following the generalised Büchi acceptance condition, as explained below, and*
- *$E \subseteq Q \times \Sigma \times \Psi_\Sigma \times Q$ as the finite set of transitions.*

We further define $K_{\mathcal{A}_{ec}}$ as the biggest constant appearing in the constraints of an event-clock automaton $\mathcal{A}_{ec}$.

For the sake of simplicity, we write K instead of $K_{\mathcal{A}_{ec}}$ when $\mathcal{A}_{ec}$ is clear from the context. An edge $e = (q, a, \psi, q')$ represents a transition from state q upon event a to state q', where the clock constraint ψ then specifies when e is enabled. A sequence of pairs consisting of states and clock valuation functions which corresponds to a sequence of respectively enabled transitions gives rise to a timed run.

Definition 35 (Timed Run) *A* timed run *θ of an automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ over a timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^\omega$ is an infinite sequence of state and clock valuation pairs $(q_0, \gamma_0)(q_1, \gamma_1) \ldots$ such that*

- *q_0 is an initial state, i.e., $q_0 \in Q_0$,*
- *each γ_i assumes values according to Definition 21 (thus γ_0 must be initial), and such that*
- *there exists a transition $(q_i, a_i, \psi, q_{i+1}) \in E$ with $\gamma_i \models \psi$ for all $i \geq 0$.*

A timed run θ of an automaton $\mathcal{A}_{ec}$ over a timed word $w \in T\Sigma^\omega$ is called *accepting*, iff for each $F_i \in F$, a state $q \in F_i$ exists such that q occurs infinitely often in θ. Finally, a timed word w is *accepted* by $\mathcal{A}_{ec}$, i.e., $w \in \mathcal{L}(\mathcal{A}_{ec})$, iff there exists an accepting run θ of $\mathcal{A}_{ec}$ over w. The use of extended B"uchi acceptance condition instead of the standard one is due to the construction given in [49].

For runtime verification, event-predicting clock variables pose a problem, since the time to the next future occurrence of an action a is predicted, while this information is not available yet—at least in the online monitoring approach. We solve this problem by representing the valuation of predicting clock variables *symbolically*.

When the automaton takes a transition $(q_i, a_i, \psi, q_{i+1})$, then the clock constraint $\psi \in \Psi_\Sigma$ either leaves a variable $c \in C_\Sigma$ unconstrained (i.e., $\psi(c) = \mathsf{undef}$) or associates a variable c with an interval $I \in \mathcal{I}$ (i.e., $\psi(c) = I$) to require $\gamma_i(c) \in I$. In the course of a symbolic run of an event-clock automaton, we do not know the actual value of any event-predicting clock and therefore we cannot evaluate any interval constraint $\gamma_i(y_a) \in I$ which involves an event-predicting clock y_a. However, we can assume that each such clock constraint will be satisfied in the future and add it to a list of constraints to be checked later on. But instead of maintaining each such constraint individually, we only maintain their conjunction—which is again a single clock constraint (see Remark 27).

Thus, when we symbolically execute an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$, we use pairs (q, Γ) with $\Gamma \in \Psi_\Sigma$ instead of pairs (q, γ) with $\gamma \in V_\Sigma$. During such a symbolic execution, we always know the values of event-recording clocks while we do not know the values of event-predicting clocks. Hence, the clock constraint Γ in such a pair (q, Γ) determines a single value for each event-recording clock using $\Gamma(x_a) = [l, l]$ with $l \in T_\perp$. In case of an event-predicting clock, Γ describes the valid range of values which are consistent with the constraints that occurred so far. Thus, $\Gamma(y_a)$ is either undefined or evaluates to an arbitrary interval from $\mathcal{I}$. For event-recording and event-predicting clocks, the interval $[\perp, \perp]$ is allowed in symbolic clock valuations: It means that the corresponding event either did not occur in the past or will not occur in the future.

This discussion yields the following definition:

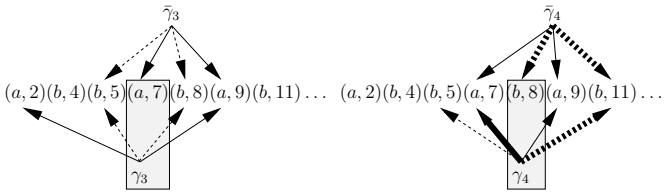
Definition 36 (Symbolic Clock Valuation) *A* symbolic clock valuation *is a clock constraint $\Gamma \in \Psi_\Sigma$ where $\Gamma(x_a) = [l, l]$ with $l \in T_\perp$ holds for all event-recoding clocks.*

Intuitively, a clock valuation function γ satisfies a symbolic clock valuation Γ, i.e., $\gamma \models \Gamma$, iff γ would have satisfied all guards subsumed by Γ during a symbolic run of the corresponding automaton.

When we symbolically run an event-clock automaton, our algorithm has to maintain a set of pairs (q, Γ). This set contains the pairs which are *reachable* from the initial set of pairs $\{(q_0, \Gamma_0) \mid q_0 \in Q_0\}$. Herein, Γ_0 is the initial symbolic clock valuation:

Definition 37 (Initial Symbolic Clock Valuation) *The* initial symbolic clock valuation *Γ_0 for a set C_Σ of clocks evaluates to $\Gamma_0(x_a) = [\perp, \perp]$ for all event-recoding clocks x_a and sets $\Gamma_0(y_a) = \mathsf{undef}$ for all event-predicting clocks y_a.*

But before describing symbolic runs, let us first consider how to compute the sequence of clock valuation functions $\gamma_0, \gamma_1, \ldots$ for a given timed word $w = (a_0, t_0)(a_1, t_1) \ldots$ This sequence can be computed directly, following their definition. However, in case of runtime verification, we are interested



i	γ_i				$\bar{\gamma}_i$			
	x_a	y_a	x_b	y_b	x_a	y_a	x_b	y_b
-1	—-				$\perp$	2	$\perp$	4
0	$\perp$	5	$\perp$	2	**0**	**5**	$\perp$	2
1	**2**	3	$\perp$	**1**	2	3	**0**	**1**
2	3	2	**1**	**3**	3	2	**0**	**3**
3	5	**2**	**2**	1	**0**	**2**	2	1
4	**1**	1	3	**3**	1	1	**0**	**3**

Fig. 5. Incremental and Ordinary Clock Valuations

in computing these sequences *incrementally* to derive an incremental scheme for the computation of symbolic timed runs.

For this, let us understand the sequences of clock valuation functions in full detail. In Figure 5, we show a prefix of a timed word over the alphabet $\Sigma = \{a, b\}$. Hence, every clock valuation γ_i refers two four timed events, namely to the last respective occurrence of a and b, as expressed by the values of x_a and x_b, and to the next occurrence of these two events, described by y_a and y_b. Thereby, the arrows in Figure 5 denote the events referred to by γ_3 and γ_4, respectively (we will explain $\bar{\gamma}_3$ and $\bar{\gamma}_4$ in the very next paragraphs). More precisely, the solid arrows show the events referred by x_a and y_a while the dashed ones correspond to x_b and y_b. So for example, $\gamma_3(x_a) = 5$ since γ_3 refers to $(a, 2)$ while $t_3 = 7$. In case of γ_4, we draw an arrow with a thick pen if the referred event changed from γ_3 to γ_4, e.g., x_a refers in γ_4 to $(a, 7)$ while it did refer to $(a, 2)$ in γ_3. Furthermore, the left part of the table in Figure 5 shows the clock valuations γ_i for $i = 0, \ldots, 4$ where we also typeset those values in boldface which are based on a newly referred event.

In general, to transit from (q_i, γ_i) to (q_{i+1}, γ_{i+1}), the timed event (a_i, t_i) is processed in following some enabled transition $e = (q_i, a_i, \psi, q_{i+1})$. Thus, we attempt to compute γ_{i+1} from γ_i and (a_i, t_i). But Definition 21 of clock valuation functions leads to the equation

$$\gamma_{i+1} = (\gamma_i + \delta_{i+1})[x_{a_i} = \delta_{i+1}][y_{a_{i+1}} = \mathsf{next}(w, a_{i+1}, i+1)] \tag{1}$$

for $i \geq 0$ where we use $\delta_{i+1} = t_{i+1} - t_i$ as abbreviation. Therein, the incremental computation of γ_{i+1} involves not only the timed event (a_i, t_i) but also the next timed event (a_{i+1}, t_{i+1}):

- The reset $[x_{a_i} = \delta_{i+1}]$ uses the time stamp t_{i+1}.
- The reset $[y_{a_{i+1}} = \mathsf{next}(w, a_{i+1}, i+1)]$ refers to a_{i+1} and to t_{i+1}.

This fact is reflected in the table of Figure 5: The values of x_{a_i} and of $y_{a_{i+1}}$ with respect to γ_{i+1} are typeset in bold face, since they both refer to different events than they did with respect to γ_i.

Therefore, we cannot define a relation $(q, \gamma) \xrightarrow{(a,\delta)} (q', \gamma')$ describing the transition from a pair (q, γ) to a pair (q', γ') on the occurrence of an action a after a delay δ without a reference to the next subsequently occurring event.

Since this problem persists at the symbolic level as well, i.e., we cannot define a relation $(q, \Gamma) \xrightarrow{(a,\delta)} (q', \Gamma')$ without referring to the next subsequently occurring event, we have to get rid of this look-ahead. To do so, we use a *sequence of incremental clock valuation functions*, denoted with $\bar{\gamma}_i$, in the incremental and symbolic execution of event-clock automata. Since the original definition of clock valuation functions is necessary to define the semantics of TLTL and TLTL_3, as well as to define timed runs, we could not use incremental valuation right from the beginning. Instead, depending on the context, we have to switch between both definitions.

Definition 38 (Incremental Clock Valuation Function) *For a finite alphabet Σ and an associated set $C_\Sigma = \{x_a, y_a \mid a \in \Sigma\}$ of clocks, an* incremental clock valuation function $\bar{\gamma}_i : C_\Sigma \to T_\perp$ *over a timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^* \cup T\Sigma^\omega$ assigns a positive real or the undefined value $\perp$ to each clock variable corresponding to position i such that the following holds:*

$$\begin{array}{lcll} \bar{\gamma}_i(x_a) & = & \mathsf{last}(w, a, i) & \textit{iff } a_i \neq a \\ \bar{\gamma}_i(x_a) & = & 0 & \textit{iff } a_i = a \\ \bar{\gamma}_i(y_a) & = & \mathsf{next}(w, a, i) & \end{array}$$

Thus, $\bar{\gamma}_i$ describes the values of the clocks from C_Σ *directly after* the timed event (a_i, t_i) occurred, i.e., $\bar{\gamma}_i(x_{a_i}) = 0$. In contrast, γ_i *ignores* the timed event (a_i, t_i), i.e., $\gamma_i(x_{a_i}) = \mathsf{last}(w, a_i, i)$ which evaluates either to $t_i - t_j$ for the largest $j < i$ with $a_j = a_i$ or to $\perp$ if no such j exists.

Hence, in Figure 5, we show the clock valuation functions γ_3 and γ_4 as a *cursor* which is *placed upon* some timed event, whereas the incremental clock valuation functions $\bar{\gamma}_3$ and $\bar{\gamma}_4$ are shown as a *cursor* which is *placed between* two timed events.

Since $\bar{\gamma}_0$ already depends on the event (a_0, t_0), we need an initial valuation function preceding $\bar{\gamma}_0$. We introduce the *initial incremental clock valuation function* $\bar{\gamma}_{-1}$ and define it with respect to an infinite timed word $w \in T\Sigma^\omega$ as follows:

- $\bar{\gamma}_{-1}(x_a) = \perp$ for all x_a,
- $\bar{\gamma}_{-1}(y_a) = t_j$ for $j \geq 0$ and $a_j = a$ and where $a_k \neq a$ holds for all $0 \leq k < j$, and
- $\bar{\gamma}_{-1}(y_a) = \perp$ if a does not occur in w at all.

Figure 5 shows the valuations of the incremental clock valuation functions $\bar{\gamma}_i$ in comparison to the original and corresponding clock valuation functions γ_i. Note that in case of $\bar{\gamma}_i$, either both, x_a and y_a, or x_b together with y_b, are changing their referred events. This is always the case, since the incremental computation of $\bar{\gamma}_{i+1}$ only involves (a_{i+1}, t_{i+1})—and does not refer to (a_i, t_i) anymore.

Assume that an automaton at $(q_i, \bar{\gamma}_{i-1})$ is about to process the timed event (a_i, t_i) from the timed word w with transition $e = (q_i, a_i, \psi, q_{i+1})$. To do so, it must first compute γ_i to check whether e is enabled, i.e., $\gamma_i \models \psi$. By following the

definitions, we find that

$$\gamma_i = (\bar{\gamma}_{i-1} + \delta_i)[y_{a_i} = \mathsf{next}(w, a_i, i)] \tag{2}$$

with $\delta_i = t_i - t_{i-1}$ for $i > 0$ and $\delta_0 = t_0$. If $\gamma_i \models \psi$ holds, the transition is enabled and we compute the next pair $(q_{i+1}, \bar{\gamma}_i)$ with

$$\bar{\gamma}_i = \gamma_i[x_{a_i} = 0] \,. \tag{3}$$

Thus, we use

$$\bar{\gamma}_i = (\bar{\gamma}_{i-1} + \delta_i)[y_{a_i} = \mathsf{next}(w, a_i, i)][x_{a_i} = 0] \tag{4}$$

for the incremental computation of $\bar{\gamma}_i$. Note that Equation (4) refers to future events beyond (a_i, t_i) only in terms of $\mathsf{next}(w, a_i, i)$. Hence, using Equations (2) and (4), we write

$$(q, \bar{\gamma}) \xrightarrow{(a,\delta)} (q', \bar{\gamma}')$$

for an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ with $q, q' \in Q$ and $\bar{\gamma}, \bar{\gamma}' \in V_\Sigma$ iff there exits a transition $e = (q, a, \psi, q)$ and a $v \in T_\perp$ such that $\bar{\gamma} + \delta$ is defined (see the discussion after Remark 27), and such that $(\bar{\gamma} + \delta)[y_a = v] \models \psi$ as well as $\bar{\gamma}' = (\bar{\gamma} + \delta)[y_a = v][x_a = 0]$ holds.

Furthermore, we expand the transition relation to finite and infinite timed words: $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{u} (q_{i+k}, \bar{\gamma}_{i+k-1})$ holds for a finite timed word $u = (a_0, t_0) \dots (a_{k-1}, t_{k-1}) \in T\Sigma^k$ if there exists a sequence of pairs $(q_i, \bar{\gamma}_{i-1}) \dots (q_{i+k}, \bar{\gamma}_{i+k-1})$ with $(q_{i+j}, \bar{\gamma}_{i+j-1}) \xrightarrow{(a_j,\delta_j)} (q_{i+j+1}, \bar{\gamma}_{i+j})$ for $0 \leq j \leq k-1$ and $\delta_0 = t_0$ and $\delta_j = t_j - t_{j-1}$ for $j > 0$. Note that for $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{u} (q_{i+k}, \bar{\gamma}_{i+k-1})$ to hold, u must be compatible to $\bar{\gamma}_{i-1}$, i.e., the evaluations of the event-predicting clocks must match the occurring events in u.

In case of an infinite word $\sigma = (a_0, t_0) \dots \in T\Sigma^\omega$, we write $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{\sigma}$ if there exists a sequence of pairs $(q_i, \bar{\gamma}_{i-1}) \dots$ with $(q_{i+j}, \bar{\gamma}_{i+j-1}) \xrightarrow{(a_j,\delta_j)} (q_{i+j+1}, \bar{\gamma}_{i+j})$ for $0 \leq j$ and again with $\delta_0 = t_0$ and $\delta_j = t_j - t_{j-1}$ for $j > 0$.

If the sequence of states $q_i, q_{i+1} \dots$ is accepting, then we write $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{\sigma} \downarrow$. As in the finite case, σ must be compatible to $\bar{\gamma}_{i-1}$ for $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{\sigma}$ to hold.

Definition 39 (Continuation Language) *Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ be an event-clock automaton. Then we define for a pair $(q, \bar{\gamma})$ with $q \in Q$ and $\bar{\gamma} \in V_\Sigma$ the* continuation language $\mathcal{L}(\mathcal{A}_{ec}(q, \bar{\gamma}))$ *of $\mathcal{A}_{ec}$ with*

$$\mathcal{L}(\mathcal{A}_{ec}(q, \bar{\gamma})) = \left\{ \sigma \in T\Sigma^\omega \mid (q, \bar{\gamma}) \xrightarrow{\sigma} \downarrow \right\}$$

We now raise incremental clock valuation functions and their transitions to the symbolic level: In the definition of symbolic runs of event clock automata, we use symbolic clock valuations that are abstractions of incremental clock valuation functions. We denote these *incremental symbolic clock valuations* with $\bar{\Gamma}_{-1}, \bar{\Gamma}_0, \dots$

Given a pair $(q_i, \bar{\Gamma}_{i-1})$, a transition $e = (q_i, a_i, \psi, q_{i+1})$, and a single timed event (a_i, t_i), we have to check whether the transition is enabled, and if so, we have to compute the corresponding new pair $(q_{i+1}, \bar{\Gamma}_i)$. To check whether the transition is enabled and to compute the resulting symbolic state, we use the procedure $\mathsf{symb\_step}((q_i, \bar{\Gamma}_{i-1}), \delta, e)$, shown in Figure 6. It takes the original pair $(q_i, \bar{\Gamma}_{i-1})$, the elapsed time δ, with $\delta = t_0$ for $i = 0$ and $\delta = t_i - t_{i-1}$ for $i > 0$, and the transition e. The procedure symbolically computes the transition according to Equation (4) and either returns $(q_{i+1}, \bar{\Gamma}_i)$ if the transition e is enabled or reports a constraint violation otherwise.

```
procedure symb_step((q_i, Γ̄_{i-1}), δ, e)
    { with e = (q_i, a_i, ψ, q_{i+1}) }
begin
{ ------------------------------------ }
{ step 1: elapse time }
if Γ̄_{i-1} + δ is invalid then
    return constraint_violation ;
Γ̄'_{i-1} := Γ̄_{i-1} + δ ;

{ ------------------------------------ }
{ step 2: reset y_{a_i} }
if Γ̄'_{i-1}(y_{a_i}) ≠ undef and 0 ∉ Γ̄'_{i-1}(y_{a_i}) then
    return constraint_violation ;
Γ_i := Γ̄'_{i-1}[y_{a_i} = undef] ;

{ ------------------------------------ }
{ step 3: process guard }
if not (∃γ with γ ⊨ Γ_i and γ ⊨ ψ) then
    return constraint_violation ;
Γ' := Γ_i ∧ ψ ;

{ ------------------------------------ }
{ step 4: reset x_{a_i} }
Γ̄_i := Γ'[x_{a_i} = 0] ;
return (q_{i+1}, Γ̄_i) ;
end
```

Fig. 6. Procedure $\mathsf{symb\_step}((q_i, \bar{\Gamma}_{i-1}), \delta, e)$

Note that the condition that Γ_i and ψ are consistent in step 3) of symb_step requires that there exists a $\gamma \in V_\Sigma$ with $\gamma \models \Gamma_i \wedge \psi$. Since each $\gamma \in V_\Sigma$ satisfies non-coincidence and continuity, see Proposition 23, this condition ensures that $\Gamma_i \wedge \psi$ satisfies these two properties as well.

In the following lemma, we show that each concrete transition from $(q_i, \bar{\gamma}_{i-1})$ to $(q_{i+1}, \bar{\gamma}_i)$ has a corresponding symbolic transition from $(q_i, \bar{\Gamma}_{i-1})$ to $(q_{i+1}, \bar{\Gamma}_i)$ as computed by symb_step. Thus, we write

$$(q_i, \bar{\Gamma}_{i-1}) \xrightarrow{(a_i,\delta)} (q_{i+1}, \bar{\Gamma}_i)$$

iff $(q_{i+1}, \bar{\Gamma}_i) = \mathsf{symb\_step}((q_i, \bar{\Gamma}_{i-1}), \delta, e)$ holds for some $e = (q_i, a_i, \psi, q_{i+1})$.

Lemma 40 (Abstracting a Transition) *Let $w = (a_0, t_0)(a_1, t_1) \dots \in T\Sigma^\omega$ be a timed word with the corresponding sequences of clock valuation functions $\gamma_0, \gamma_1, \dots$ and incremental clock valuation functions $\bar{\gamma}_{-1}, \bar{\gamma}_0, \dots$ Fix some $i \geq 0$ and set $\delta = t_i - t_{i-1}$ for $i > 0$ and $\delta = t_0$ for $i = 0$. Let $e = (q_i, a_i, \psi, q_{i+1})$ be an enabled transition, i.e., $\gamma_i \models \psi$.*

Then for a pair $(q_i, \bar{\Gamma}_{i-1})$ with $\bar{\gamma}_{i-1} \models \bar{\Gamma}_{i-1}$, $\mathsf{symb\_step}((q_i, \bar{\Gamma}_{i-1}), \delta, e)$ yields $(q_{i+1}, \bar{\Gamma}_i)$ with $\bar{\gamma}_i \models \bar{\Gamma}_i$.

Proof: We follow the procedure symb_step in a stepwise manner, where we first show that step 1) leads to $(\bar{\gamma}_{i-1} + \delta) \models$

$\bar{\Gamma}'_{i-1}$. Then step 2) yields a Γ_i which is an abstraction of γ_i, i.e., $\gamma_i \models \Gamma_i$, and consequently step 3) does not find any inconsistency. Finally, we prove that step 4) must produce a $\bar{\Gamma}_i$ such that $\bar{\gamma}_i \models \bar{\Gamma}_i$.

Below, we use the fact that either $\bar{\gamma}_{i-1}(y_a) = \bot$ or $\bar{\gamma}_{i-1}(y_a) \geq \delta$ must hold for each event-predicting clock y_a as at least δ time units pass by until the next event occurs.

1) **Elapse Time:** We have $\bar{\gamma}_{i-1} \models \bar{\Gamma}_{i-1}$, and hence by Fact 28, if $\bar{\gamma}_{i-1} + \delta$ is indeed valid, then $\bar{\gamma}_{i-1} + \delta \models \bar{\Gamma}_{i-1} + \delta = \bar{\Gamma}'_{i-1}$ must hold as well.
   But $\bar{\gamma}_{i-1} + \delta$ is valid since we have either $\bar{\gamma}_{i-1}(y_a) = \bot$ or $\bar{\gamma}_{i-1}(y_a) \geq \delta$ for all y_a—and henceforth $\bar{\gamma}_{i-1} + \delta \models \bar{\Gamma}'_{i-1}$ holds.
2) **Reset y_{a_i}:** From the preceding step, we know that $\bar{\gamma}_{i-1} + \delta \models \bar{\Gamma}'_{i-1}$ holds.
   $(\bar{\gamma}_{i-1} + \delta)(y_{a_i}) = 0$ must hold since a_i is the event being currently processed. Thus symb_step does not report a constraint violation.
   Then, by Fact 28, we obtain $(\bar{\gamma}_{i-1} + \delta)[y_{a_i} = \mathsf{next}(w, a_i, i)] \models \bar{\Gamma}'_{i-1}[y_{a_i} = \mathsf{undef}]$, i.e., $\gamma_i \models \Gamma_i$.
3) **Process the Guard:** Since the transition $e = (q_i, a_i, \psi, q_{i+1})$ is enabled, we know $\gamma_i \models \psi$. From the preceding step, we also have $\gamma_i \models \Gamma_i$, and therefore, ψ and Γ_i must be consistent with $\gamma_i \models \Gamma_i \wedge \psi = \Gamma'$.
4) **Reset x_{a_i}:** $\bar{\gamma}_i$ and γ_i differ only in the value for x_{a_i} which is reset to 0 in $\bar{\gamma}_i$ (see Equation (3)). From the preceding step, we have $\gamma_i \models \Gamma'$ and thus we obtain, by Fact 28, $\bar{\gamma}_i = \gamma_i[x_{a_i} = 0] \models \Gamma'[x_{a_i} = 0] = \bar{\Gamma}_i$.

This concludes the proof, as symb_step returns $(q_{i+1}, \bar{\Gamma}_i)$ with $\bar{\gamma}_i \models \bar{\Gamma}_i$. ∎

Based upon symb_step, and analogous to timed runs of an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$, we now define symbolic timed runs over a timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^\omega$ as an infinite sequence of pairs $\Theta = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_0) \ldots$ For these symbolic timed runs, we have to define the initial symbolic clock valuation $\bar{\Gamma}_{-1}$. By inspecting $\bar{\gamma}_{-1}$, we find that we can use the initial symbolic clock valuation as given by Definition 37 without modification, i.e., we set $\bar{\Gamma}_{-1} = \Gamma_0$. Thus, we arrive at the following definition:

Definition 41 (Symbolic Timed Run) *A* symbolic timed run Θ *of an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ over an infinite timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^\omega$ is a sequence of pairs $(q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_0) \ldots$ such that the following conditions are met:*

- *$q_i \in Q$ holds for all $i \geq 0$ and $q_0 \in Q_0$ holds for the starting state.*
- *$\bar{\Gamma}_i \in \Psi_\Sigma$ is a symbolic clock valuation (Definition 36) for $i \geq i$ and $\bar{\Gamma}_{-1}$ is the initial symbolic clock valuation ($\bar{\Gamma}_{-1} = \Gamma_0$ and following Definition 37).*
- *For all $0 \leq i$ and with $\delta_0 = t_0$ and $\delta_i = t_i - t_{i-1}$, $(q_i, \bar{\Gamma}_{i-1}) \xrightarrow{a_i, \delta_i} (q_{i+1}, \bar{\Gamma}_i)$ holds.*

Since symb_step does not receive any information beyond the currently processed timed event (a_i, t_i), no information on future events beyond the already observed prefix $(a_0, t_0) \ldots (a_i, t_i)$ is necessary to compute a prefix of a symbolic timed run.

Remark 42 (Symbolic Timed Runs are not Previsionary) *To compute a prefix $(q_0, \bar{\Gamma}_{-1}) \ldots (q_{i+1}, \bar{\Gamma}_i)$ of a symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_0) \ldots$ for a prefix $u = (a_0, t_0) \ldots (a_i, t_i)$ of an infinite timed word $w = (a_0, t_0)(a_1, t_1) \ldots$, no information beyond u is necessary.*

Hence, symbolic timed runs are feasible as a tool for (online) runtime verification where we are provided with an incrementally expanded finite prefix of some system trace. But beyond its feasibility as a technique, it remains to prove that symbolic timed runs are semantically adequate as an abstraction of all possible concrete behaviours.

Lemma 40 is the first step towards that goal, where we show that each concrete transition can be abstracted into a corresponding symbolic transition. In the next lemma, we expand this statement to entire timed runs.

Lemma 43 (Abstracting Timed Runs) *Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ be an event-clock automaton and let $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^\omega$ be an infinite timed word with a timed run $\theta = (q_0, \gamma_0)(q_1, \gamma_1) \ldots$*

Then there exists a symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_0) \ldots$ over w which is based upon the same sequence of states $q_0, q_1 \ldots$ as θ.

Moreover $\bar{\gamma}_i \models \bar{\Gamma}_i$ holds for the sequence $\bar{\gamma}_{-1}, \bar{\gamma}_0, \ldots$ of incremental clock valuation functions as determined by w for all $i \geq -1$.

Proof: The infinite timed word w determines a unique sequence $\gamma_0, \gamma_1, \ldots$ of clock valuation functions (Definition 21) as well as a unique sequence $\bar{\gamma}_{-1}, \bar{\gamma}_0, \ldots$ of incremental clock valuation functions (Definition 38).

To construct Θ, we first set $\bar{\Gamma}_{-1} = \Gamma_0$ following Definition 41 and obtain immediately $\bar{\gamma}_{-1} \models \bar{\Gamma}_{-1}$. Since there exists a timed run $\theta = (q_0, \gamma_0)(q_1, \gamma_1) \ldots$, there exists for each $i \geq 0$ an enabled transition $e_i = (q_i, a_i, \psi, q_{i+1})$ facilitating the transition from (q_i, γ_i) to (q_{i+1}, γ_{i+1}). But then, the condition to apply Lemma 40 is satisfied: $\bar{\gamma}_{-1} \models \bar{\Gamma}_{-1}$ holds and e_0 is an enabled transition. Consequently, Lemma 40 yields the pair

$$(q_1, \bar{\Gamma}_0) = \mathsf{symb\_step}((q_0, \bar{\Gamma}_{-1}), \delta_0, e_0)$$

with $\bar{\gamma}_0 \models \bar{\Gamma}_0$. Then again, the condition to apply Lemma 40 is satisfied and we obtain the the required symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_0) \ldots$ inductively. ∎

At this point, we are tempted to show the converse, i.e., that each symbolic timed run gives rise a corresponding ordinary timed run. However, this is not the case: If we take some transition with a guard $\psi(y_a) = [0, \infty)$, then it is required that the event a occurs eventually in the future (in fact, such a guard is equivalent to Fa in standard LTL). But if a never occurs again, then this misbehaviour remains undetected by symb_step. On the other hand, at the concrete level of ordinary timed runs, if a never occurs again, we have $\gamma_i(y_a) = \bot$ and the transition with the guard $\psi(y_a) = [0, \infty)$ is not enabled

at the concrete level. Thus, not every symbolic timed run has a corresponding ordinary timed run—leading to the following proposition:

Proposition 44 *There exists an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ and an infinite timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^\omega$ with a symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_0) \ldots$ such that there exists no ordinary timed run $\theta = (q_0, \gamma_0)(q_1, \gamma_1) \ldots$ over w which is based upon the same sequence of states $q_0, q_1 \ldots$ as Θ.*

Nevertheless, we can show that each symbolic transition yields a corresponding concrete transition. To show this, we need to resort to a backward simulation argument, leading to Lemma 45. Using this lemma, we finally prove in Theorem 46 that every finite prefix of a symbolic timed run has a corresponding finite prefix of a timed run: We choose a suitable clock valuation function for the last pair of the symbolic timed run and concretise the run with a backward simulation. Then we show in Theorem 46 how to expand this prefix of a timed run into a suitable infinite timed run.

Lemma 45 (Concretising a Symbolic Transition) *Let $(q_i, \bar{\Gamma}_{i-1}) \xrightarrow{(a_i,\delta)} (q_{i+1}, \bar{\Gamma}_i)$ via a transition $e = (q_i, a_i, \psi, q_{i+1})$. Then for all $\bar{\gamma}_i \models \bar{\Gamma}_i$ and for all*

$$\bar{\gamma}_{i-1} = (\bar{\gamma}_i - \delta)[y_{a_i} = \delta][x_{a_i} = \bar{\Gamma}_{i-1}(x_{a_i})]$$

the following two conditions hold:

- $\bar{\gamma}_{i-1} \models \bar{\Gamma}_{i-1}$
- $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{(a_i,\delta)} (q_{i+1}, \bar{\gamma}_i)$ *holds via the transition e.*

Proof: We show the first claim by contradiction and thereupon prove the second claim using the first one.

Assume $\bar{\gamma}_{i-1} \models \bar{\Gamma}_{i-1}$ does not hold. Then one of the following four cases must arise—which we drive into a contradiction individually:

- $\bar{\gamma}_{i-1}(x_{a_i}) \notin \bar{\Gamma}_{i-1}(x_{a_i})$: In the definition of $\bar{\gamma}_{i-1}$ in the lemma statement, we use $[x_{a_i} = \bar{\Gamma}_{i-1}(x_{a_i})]$ and hence we always have $\bar{\gamma}_{i-1}(x_{a_i}) \in \bar{\Gamma}_{i-1}(x_{a_i})$.
- $\bar{\gamma}_{i-1}(x_a) \notin \bar{\Gamma}_{i-1}(x_a)$ for $a \neq a_i$: Since $a \neq a_i$ holds, the constraints on x_a are only affected by the elapsing time. We distinguish two cases:
  - If $\bar{\Gamma}_{i-1}(x_a) = [\bot, \bot]$, then $\bar{\Gamma}_i(x_a) = [\bot, \bot]$ as well and hence $\bar{\gamma}_i(x_a) = \bot$ must hold. But then we have $\bar{\gamma}_{i-1}(x_a) = \bot$ and therefore we find $\bar{\gamma}_{i-1}(x_a) = \bot \in [\bot, \bot] = \bar{\Gamma}_{i-1}(x_a)$.
  - If $\bar{\Gamma}_{i-1}(x_a) = [l, l]$, then $\bar{\Gamma}_i(x_a) = [l + \delta, l + \delta]$ holds such that $\bar{\gamma}_i(x_a) = l + \delta$ must hold. Then we obtain $\bar{\gamma}_{i-1}(x_a) = l$ and arrive at $\bar{\gamma}_{i-1}(x_a) = l \in [l, l] = \bar{\Gamma}_{i-1}(x_a)$.
- $\bar{\gamma}_{i-1}(y_{a_i}) \notin \bar{\Gamma}_{i-1}(y_{a_i})$: Because of the check in step 2) of symb_step (see Figure 6) $0 \in (\bar{\Gamma}_{i-1}(y_{a_i}) + \delta)$ must hold since otherwise, symb_step would have reported a constraint violation. Hence we have $\delta \in \bar{\Gamma}_{i-1}(y_{a_i})$. On the other hand, the definition of $\bar{\gamma}_{i-1}$ in the statement of the lemma resets y_{a_i} to δ and consequently, $\bar{\gamma}_{i-1}(y_{a_i}) \in \bar{\Gamma}_{i-1}(y_{a_i})$.
- $\bar{\gamma}_{i-1}(y_a) \notin \bar{\Gamma}_{i-1}(y_a)$ for $a \neq a_i$: In the case of $a \neq a_i$, the constraints on y_a are only affected by the elapsing time leading to the following two cases:
  - If $\bar{\Gamma}_{i-1}(y_a) = [\bot, \bot]$, then $\bar{\Gamma}_i(y_a) = [\bot, \bot]$ as well. Thus, $\bar{\gamma}_i(y_a) = \bot$ must hold resulting in $\bar{\gamma}_{i-1}(y_a) = \bot$ such that $\bar{\gamma}_{i-1}(y_a) = \bot \in [\bot, \bot] = \bar{\Gamma}_{i-1}(y_a)$ holds.
  - If $\bar{\Gamma}_{i-1}(y_a) = [\![l, r]\!]$, then $\bar{\Gamma}_i(y_a) = [\![l \dot{-} \delta, r - \delta]\!]$ holds and consequently $\bar{\gamma}_i(y_a)$ must be chosen from $[\![l \dot{-} \delta, r - \delta]\!]$. But then we have $\bar{\gamma}_{i-1}(y_a) \in [\![l \dot{-} \delta + \delta, r - \delta + \delta]\!] \subseteq [\![l, r]\!] = \bar{\Gamma}_{i-1}(y_a)$.

Since the constraints for each individual clock are satisfied, we know that $\bar{\gamma}_{i-1} \models \bar{\Gamma}_{i-1}$ holds.

Assume $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{(a_i,\delta)} (q_{i+1}, \bar{\gamma}_i)$ does not hold. Then the transition $e = (q_i, a_i, \psi, q_{i+1})$ is not enabled at $(q_i, \bar{\gamma}_{i-1})$. Following Equation (2), we have $\gamma_i = (\bar{\gamma}_{i-1} + \delta)[y_{a_i} = \bar{\gamma}_i(y_{a_i})]$ (since $y_{a_i} = \mathsf{next}(w, a_i, i) = \bar{\gamma}_i(y_{a_i})$). Since $\gamma_i \not\models \psi$, one of the following two cases must arise:

- $\gamma_i(x_{a_i}) \notin \psi(x_{a_i})$: Since $\gamma_i(x_{a_i}) = \bar{\Gamma}_{i-1}(x_{a_i}) + \delta$, it follows that $\bar{\Gamma}_{i-1} + \delta$ and ψ are inconsistent. But then, symb_step reports in step 3) a constraint violation.
- $\gamma_i(c) \notin \psi(c)$ for an event-recoding or event-predicting clock $c \neq x_{a_i}$: Since $\bar{\gamma}_i = \gamma_i[x_{a_i} = 0]$ we have $\bar{\gamma}_i(c) = \gamma_i(c)$ and hence $\bar{\gamma}_i(c) \notin \psi(c)$, i.e., $\bar{\gamma}_i \not\models \psi$. Because of step 3) in symb_step, $\bar{\Gamma}_i \Rightarrow \psi$ holds and we therefore arrive at $\bar{\gamma}_i \not\models \bar{\Gamma}_i$ which contradicts the lemma statement.

■

Theorem 46 (Symbolic Simulation) *Let $u = (a_0, t_0) \ldots (a_i, t_i) \in T\Sigma^*$ be a finite timed word and let $\sigma = (a_{i+1}, t_{i+1})(a_{i+2}, t_{i+2}) \ldots \in T\Sigma^\omega$ be an infinite continuation of u.*

The infinite timed word $u\sigma$ is accepted by an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$, i.e., $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$, iff there exists

(a) *a finite symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1}) \ldots (q_{i+1}, \bar{\Gamma}_i)$ over u,*
(b) *an infinite timed run $\theta = (q_{i+1}, \gamma_{i+1})(q_{i+2}, \gamma_{i+2}) \ldots$ starting at (q_{i+1}, γ_{i+1}) and accepting σ, and*
(c) *an incremental clock valuation function $\bar{\gamma}_i \in V_\Sigma$ with $\bar{\gamma}_i \models \bar{\Gamma}_i$ such that $\gamma_{i+1} = (\bar{\gamma}_i + \delta)[y_{a_{i+1}} = v]$ holds for some $\delta \in \mathbb{R}^{\geq 0}$ and some $v \in T_\bot$.*

Proof: Assume $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$ holds. Then there exists an accepting timed run $\theta' = (q_0, \gamma_0)(q_1, \gamma_1) \ldots$ over $u\sigma$. Thus, by Lemma 43, there exists a symbolic timed run $\Theta' = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_1) \ldots$ over $u\sigma$ with $\bar{\gamma}_i \models \bar{\Gamma}_i$ for all $i > 0$ as well. We take the prefix $\Theta = (q_0, \bar{\Gamma}_{-1}) \ldots (q_{i+1}, \bar{\Gamma}_i)$ of Θ' and the suffix $\theta = (q_{i+1}, \gamma_{i+1})(q_{i+2}, \gamma_{i+2}) \ldots$ of θ' to meet conditions (a) and (b) of the lemma statement, respectively. Condition (c) is satisfied since Lemma 43 ensures that $\bar{\gamma}_i \models \bar{\Gamma}_i$ holds and since $\bar{\gamma}_i$ and γ_{i+1} are being determined mutually consistently by $u\sigma$.

Assume Θ, θ, and $\bar{\gamma}_i$ exist as required in conditions (a) to (c). Then we construct an accepting infinite timed run $\theta' = (q_0, \gamma_0)(q_1, \gamma_1) \ldots$ over $u\sigma$ to show $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$. To do so, we take the timed run $\theta = (q_{i+1}, \gamma_{i+1})(q_{i+2}, \gamma_{i+2}) \ldots$ over

σ as suffix in θ'. It remains to construct the connected prefix $(q_0,\gamma_0)\dots(q_{i+1},\gamma_{i+1})$ of θ'.

We have $\bar{\gamma}_i \models \bar{\Gamma}_i$ and $(q_i,\bar{\Gamma}_{i-1}) \xrightarrow{(a_i,\delta_i)} (q_{i+1},\bar{\Gamma}_i)$ and hence we can apply Lemma 45 to obtain $\bar{\gamma}_{i-1}$ such that $(q_i,\bar{\gamma}_{i-1}) \xrightarrow{(a_i,\delta_i)} (q_{i+1},\bar{\gamma}_i)$ with $\bar{\gamma}_{i-1} \models \bar{\Gamma}_{i-1}$. By applying Lemma 45 inductively, we obtain $(q_0,\bar{\gamma}_{-1})\dots(q_{i+1},\bar{\gamma}_i)$. Using Equation (2), we finally obtain $(q_0,\gamma_0)\dots(q_{i+1},\gamma_{i+1})$, as required. ∎

Rereading the statement of Theorem 46 in abstract terms, the theorem states that a finite prefix u can be continued to an infinite word $u\sigma$, iff u has a symbolic timed run Θ which ends in a pair $(q_{i+1},\bar{\Gamma}_i)$ which is non-empty, i.e., which has a concretisation $(q_{i+1},\bar{\gamma}_i)$ with a non-empty continuation language. This is exactly the statement of Corollary 47 below.

Corollary 47 (Runtime Verification Criterion) *Let $u = (a_0,t_0)\dots(a_i,t_i) \in T\Sigma^*$ be a finite timed word and let $\mathcal{A}_{ec} = (\Sigma,Q,Q_0,E,F)$ be an event-clock automaton.*

Then there exists an infinite continuation $\sigma \in T\Sigma^\omega$ of u with $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$ iff there exists a finite symbolic timed run $\Theta = (q_0,\bar{\Gamma}_{-1})\dots(q_{i+1},\bar{\Gamma}_i)$ over u and an incremental clock valuation function $\bar{\gamma}_i \in V_\Sigma$ with $\bar{\gamma}_i \models \bar{\Gamma}_i$ such that $\mathcal{L}(\mathcal{A}_{ec}(q_{i+1},\bar{\gamma}_i)) \neq \emptyset$.

Proof: Assume σ with $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$ exists. Then we apply Theorem 46 to obtain Θ and $\bar{\gamma}_i$.

Assume Θ and $\bar{\gamma}_i$ exist. Since the language accepted from $(q_{i+1},\bar{\gamma}_i)$ is non-empty, there must exist an infinite continuation $\sigma \in T\Sigma^\omega$ with $(q_{i+1},\bar{\gamma}_i) \xrightarrow{\sigma}\downarrow$, i.e., there exits a sequence $(q_{i+1},\bar{\gamma}_i)(q_{i+2},\bar{\gamma}_{i+1})\dots$ accepting σ. Using Equation (2), we obtain a corresponding timed run $\theta = (q_{i+1},\gamma'_{i+1})(q_{i+2},\gamma'_{i+2})\dots$ and apply Theorem 46 to find $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$. ∎

In both, Theorem 46 and its Corollary 47, we need to find a suitable concrete and suitable incremental clock valuation function $\bar{\gamma}_i \in V_\Sigma$ of some symbolic clock constraint $\bar{\Gamma}_i \in \Psi_\Sigma$, i.e., $\bar{\gamma}_i \models \bar{\Gamma}_i$ must hold and $\bar{\gamma}_i$ must give rise to some infinite and accepting continuation σ. We note that symb_step ensures $\bar{\Gamma}_i \in \Psi_\Sigma$ and therefore, $\bar{\Gamma}_i$ has a non-coincident and a continuous solution (see Remark 26). To ensure $\bar{\gamma}_i \in V_\Sigma$, we need to make sure that $\bar{\gamma}_i$ also satisfies these properties. Otherwise $\bar{\gamma}_i$ would prescribe a sequence of timed events, which is *not* a timed word, see Proposition 23.

E. Emptiness Check for Symbolic States

Taking Corollary 47 as starting point, we discuss in this section how to determine for a given event-clock automaton $\mathcal{A}_{ec} = (\Sigma,Q,Q_0,E,F)$ and a corresponding pair $(q,\bar{\Gamma})$ whether there exists an incremental clock valuation function $\bar{\gamma} \in V_\Sigma$ with $\bar{\gamma} \models \bar{\Gamma}$ such that $\mathcal{L}(\mathcal{A}_{ec}(q,\bar{\gamma})) \neq \emptyset$.

Thus, we develop in this section a procedure $\mathsf{empty}_{\mathcal{A}_{ec}}(q,\bar{\Gamma})$ which returns *true* iff for all $\bar{\gamma} \in V_\Sigma$ with $\bar{\gamma} \models \bar{\Gamma}$ it holds that $\mathcal{L}(\mathcal{A}_{ec}(q,\bar{\gamma})) = \emptyset$.

Looking at the scheme developed in the discrete-time setting, we are now tempted to check for every state q of the event-clock automaton, whether the language accepted from state q is empty. However, this would yield wrong conclusions,

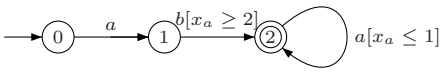


Fig. 7. Event-clock automaton

as exemplified by the automaton shown in Figure 7. While the language accepted in state 2 is non-empty and, despite, state 2 is reachable, the automaton does not accept any word when starting in state 0. The constraint when passing from state 1 to 2 requires the clock x_a to evaluate to at least 2. This, however, prevents the self-loop in state 2 from being enabled.

Thus, to implement the emptiness check, the event-clock automaton itself is too coarse as an abstraction of the infinite statespace spawned by the states of the automaton and the clock valuation functions.

The standard technique to determine the emptiness of an event-clock automaton (and of timed automata in general) relies on the translation of event-clock automata into *region automata* [48]. A region automaton is an ordinary (generalised) Büchi automaton whose states are pairs $(q,[\bar{\gamma}]_{\approx_R})$ where q is a state of the original event-clock automaton and $[\bar{\gamma}]_{\approx_R}$ is a *clock region*. A clock region $[\bar{\gamma}]_{\approx_R} = \{\bar{\gamma}' \in \mathrm{V}_\Sigma \mid \bar{\gamma}' \approx_R \bar{\gamma}\}$ is an equivalence class of incremental clock valuation functions in V_Σ determined by the *region equivalence* $\approx_R$.

However, the region equivalence is just *one possible choice* to implement the emptiness check. Every other equivalence relation $\approx$ over V_Σ meeting the following three conditions is suitable for that purpose: (1) the relation has finite index, (2) it is a bisimulation, and (3) each incremental symbolic clock valuation (as they are used in symbolic timed runs) equals the union of a set of equivalence classes $[\bar{\gamma}]_\approx$. From these three conditions, only the third one is specific to our approach.

Below, we introduce the relevant definitions underlying these three conditions. Then we formulate the emptiness check as used in this paper and prove its correctness. Finally, for the sake of completeness, we recall the region equivalence for event-clock automata [30], as one possible choice for a suitable equivalence relation.

We start with the definition of the quotient Büchi automaton of an event-clock automaton according to an equivalence relation on incremental clock valuation functions:

Definition 48 (Quotient Automaton, following [30]) [7] *For an event-clock automaton $\mathcal{A}_{ec} = (\Sigma,Q,Q_0,E,F)$ and an equivalence relation $\approx$ on incremental clock valuation functions, we define the* quotient automaton $\mathcal{A}_{eq}/_\approx = (\Sigma,Q/_\approx,Q_0/_\approx,E/_\approx,F/_\approx)$ *as a generalised Büchi automaton with*

- *$Q/_\approx$ being the set of states which is defined with $Q/_\approx = \{(q,[\bar{\gamma}]_\approx) \mid q \in Q$ and $\bar{\gamma} \in \mathrm{V}_\Sigma\}$,*

[7] The automata we define here as quotient automata are denoted with region automata $Reg_\cong(A)$ in [30]. More precisely, in [30], region automata are not defined directly but in terms of labelled transition systems. In the definition of these labelled transition systems, the authors use incremental clock valuation functions—but without explicitly stating the change from ordinary to incremental clock valuation functions. Nevertheless, the definition in [30] and our own definition yield the same automata.

- $Q_0/_\approx$ *as the set of initial states with* $Q_0/_\approx = \{(q, [\bar{\gamma}_{-1}]_\approx) \mid q \in Q_0$ *and* $\bar{\gamma}_{-1} \in V_\Sigma$ *being initial*$\}$,
- $E/_\approx$ *which is the set of transitions, where we define* $((q, [\bar{\gamma}]_\approx), (q', [\bar{\gamma}']_\approx), a) \in E/_\approx$ *iff there exist* $\bar{\beta} \in [\bar{\gamma}]_\approx$ *and* $\bar{\beta}' \in [\bar{\gamma}']_\approx$ *such that* $(q, \bar{\beta}) \xrightarrow{a,\delta} (q', \bar{\beta}')$ *holds for some* $\delta \in \mathbb{R}^{\geq 0}$*, and with*
- $F/_\approx$ *as the set of accepting state sets (generalised Büchi acceptance, recall Definition 3 and the subsequent discussion), where we use* $F/_\approx = \{F_i/_\approx \mid F_i \in F\}$ *for* $F_i/_\approx = \{(q, [\bar{\gamma}]_\approx) \mid q \in F_i$ *and* $\bar{\gamma} \in V_\Sigma\}$.

Note that the automaton $\mathcal{A}_{eq}/_\approx$ is an ordinary (generalised) Büchi automaton and hence, we can check the emptiness of the language accepted from a particular state $(q, [\bar{\gamma}]_\approx) \in Q/_\approx$ of $\mathcal{A}_{eq}/_\approx$ in the same way as in the discrete-time case [38].

We thus need to show that such a check is sufficient in our setting. For this purpose, the employed equivalence relation needs to satisfy the key property of being a time-abstract bisimulation:

Definition 49 (Time-Abstract Bisimulation, following [50]) *An equivalence relation* $\approx$ *is a* time-abstract bisimulation *for an automaton* $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$*, iff* $(q, \bar{\gamma}_1) \xrightarrow{a,\delta_1} (q', \bar{\gamma}_1')$ *for two states* $q, q' \in Q$*, two incremental clock valuations* $\bar{\gamma}_1, \bar{\gamma}_1' \in V_\Sigma$*, an event* $a \in \Sigma$*, and a delay* $\delta_1 \in \mathbb{R}^{\geq 0}$ *implies that for every equivalent incremental clock valuation* $\bar{\gamma}_2 \approx \bar{\gamma}_1$*, there exists another incremental clock valuation* $\bar{\gamma}_2' \approx \bar{\gamma}_1'$ *and a delay* $\delta_2 \in \mathbb{R}^{\geq 0}$ *such that* $(q, \bar{\gamma}_2) \xrightarrow{a,\delta_2} (q', \bar{\gamma}_2')$ *holds.*

If an equivalence relation $\approx$ is a time-abstract bisimulation with finite index for an automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ which accepts the timed language $\mathcal{L}(\mathcal{A}_{ec}) \subseteq T\Sigma^\omega$, then the corresponding quotient automaton $\mathcal{A}_{ec}/_\approx$ accepts the corresponding untimed language $\mathsf{ut}(\mathcal{L}(\mathcal{A}_{ec}))$ [30], [50]. Hence, given a pair $(q, \bar{\gamma})$, we can check whether the language $\mathcal{L}(\mathcal{A}_{ec}(q, \bar{\gamma}))$ accepted by $\mathcal{A}_{ec}$ continuing from $(q, \bar{\gamma})$ is empty or not by performing the emptiness check on $\mathcal{A}_{ec}/_\approx$ for the state $(q, [\bar{\gamma}]_\approx)$, i.e., by checking $\mathcal{L}(\mathcal{A}_{ec}/_\approx(q, [\bar{\gamma}]_\approx)) = \emptyset$, where $\mathcal{A}_{ec}/_\approx(q, [\bar{\gamma}]_\approx)$ is the automaton identical to $\mathcal{A}_{ec}/_\approx$ except for the set of initial states which is changed to $\{(q, [\bar{\gamma}]_\approx)\}$.

Theorem 50 (Emptiness Check with Bisimulation [48]) *Let* $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ *be an event-clock automaton and let the relation* $\approx$ *be a time-abstract bisimulation for* $\mathcal{A}_{ec}$*. Then, for a state* $q \in Q$ *and an incremental clock valuation function* $\bar{\gamma} \in V_\Sigma$*,* $\mathcal{L}(\mathcal{A}_{ec}(q, \bar{\gamma})) = \emptyset$ *iff* $\mathcal{L}(\mathcal{A}_{ec}/_\approx(q, [\bar{\gamma}]_\approx)) = \emptyset$*.*

Next, we describe a way to perform the emptiness check for a pair $(q, \bar{\Gamma})$ as it occurs in symbolic timed runs. To do so, we compute a (minimal) set $\mathsf{cover}_\approx(\bar{\Gamma})$ of equivalence classes such that

$$\left\{\bar{\gamma} \in V_\Sigma \mid \bar{\gamma} \models \bar{\Gamma}\right\} = \bigcup_{[\bar{\gamma}]_\approx \in \mathsf{cover}_\approx(\bar{\Gamma})} [\bar{\gamma}]_\approx$$

holds. Then, the untimed language accepted from $(q, \bar{\Gamma})$ (i.e., $\bigcup \mathcal{L}(\mathcal{A}_{ec}(q, \bar{\gamma}))$ for $\bar{\gamma} \models \bar{\Gamma}$) is determined with

$$\mathsf{ut}\left(\bigcup_{\bar{\gamma} \models \bar{\Gamma}} \mathcal{L}(\mathcal{A}_{ec}(q, \bar{\gamma}))\right) = \bigcup_{[\bar{\gamma}]_\approx \in \mathsf{cover}_\approx(\bar{\Gamma})} \mathcal{L}(\mathcal{A}_{ec}/_\approx(q, [\bar{\gamma}]_\approx)) \,,$$

yielding a way to implement the procedure $\mathsf{empty}_{\mathcal{A}_{ec}}(q, \bar{\Gamma})$ which returns *true* iff for all $\bar{\gamma} \in V_\Sigma$ with $\bar{\gamma} \models \bar{\Gamma}$, $\mathcal{L}(\mathcal{A}_{ec}(q, \bar{\gamma})) = \emptyset$ holds, as stated in the following corollary:

Corollary 51 (Emptiness Check for Symbolic Runs) *Let* $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ *be an event-clock automaton and let the relation* $\approx$ *be a time-abstract bisimulation for* $\mathcal{A}_{ec}$*.*

Then for a state $q \in Q$ *and a symbolic clock valuation* $\bar{\Gamma} \in \Psi_\Sigma$*, we have* $\mathsf{empty}_{\mathcal{A}_{ec}}(q, \bar{\Gamma}) = true$ *iff*

$$\bigcup_{[\bar{\gamma}]_\approx \in \mathsf{cover}_\approx(\bar{\Gamma})} \mathcal{L}(\mathcal{A}_{ec}/_\approx(q, [\bar{\gamma}]_\approx)) = \emptyset$$

holds.

This leads to the following procedure for the emptiness check for the event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ upon the equivalence relation $\approx$:

- **Precomputation:** Generate the quotient automaton $\mathcal{A}_{ec}/_\approx$ and determine for each state $(q, [\bar{\gamma}]_\approx)$ of $\mathcal{A}_{ec}/_\approx$ whether $\mathcal{L}(\mathcal{A}_{ec}/_\approx(q, [\bar{\gamma}]_\approx))$ is empty or not. Store the result in a look-up table T with $T[q, [\bar{\gamma}]_\approx] = true$ if $\mathcal{L}(\mathcal{A}_{ec}/_\approx(q, [\bar{\gamma}]_\approx)) = \emptyset$ and $false$ otherwise.
- **Emptiness Check:** To answer the emptiness check for a pair $(q, \bar{\Gamma})$, compute

$$\mathsf{empty}_{\mathcal{A}_{ec}}(q, \bar{\Gamma}) = \bigwedge_{[\bar{\gamma}]_\approx \in \mathsf{cover}_\approx(\bar{\Gamma})} T[q, [\bar{\gamma}]_\approx] \,.$$

  Then the language accepted from $(q, \bar{\Gamma})$ by $\mathcal{A}_{ec}$ is empty, iff $\mathsf{empty}_{\mathcal{A}_{ec}}(q, \bar{\Gamma})$ returns $true$.

It remains to recall the region equivalence $\approx_R$ which is a time-abstract bisimulation with finite index and to show how to compute $\mathsf{cover}_{\approx_R}(\bar{\Gamma})$ for $\approx_R$.

Below, we use the following abbreviation for the fractional period of time to pass by until a clock value changes its integral part: For all event-recoding clocks $x_a \in C_\Sigma$, we set $\langle \bar{\gamma}(x_a) \rangle = \lceil \bar{\gamma}(x_a) \rceil - \bar{\gamma}(x_a)$ (the time until $\bar{\gamma}(x_a)$ reaches $\lceil \bar{\gamma}(x_a) \rceil$) and for all event-predicting clocks $y_a \in C_\Sigma$, we set $\langle \bar{\gamma}(y_a) \rangle = \bar{\gamma}(y_a) - \lfloor \bar{\gamma}(y_a) \rfloor$ (the time until $\bar{\gamma}(y_a)$ reaches $\lfloor \bar{\gamma}(y_a) \rfloor$).

Definition 52 (Region Equivalence [51], [48]) *Let* K *be the biggest constant occurring in some constraint of an event-clock automaton* $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$*. Then we define the* region equivalence *relation* $\approx_R$ *on incremental clock valuations in* V_Σ *for* $\mathcal{A}_{ec}$*, such that* $\bar{\gamma}_1, \bar{\gamma}_2 \in V_\Sigma$ *are equivalent, symbolically* $\bar{\gamma}_1 \approx_R \bar{\gamma}_2$*, iff all the following conditions hold:*

- *(agreement on undefined)*
  for all $z \in C_\Sigma$*,* $\bar{\gamma}_1(z) = \bot$ *iff* $\bar{\gamma}_2(z) = \bot$
- *(agreement on integral part)*
  for all $z \in C_\Sigma$*, if* $\bar{\gamma}_1(z) \leq K$ *or* $\bar{\gamma}_2(z) \leq K$*, then* $\lfloor \bar{\gamma}_1(z) \rfloor = \lfloor \bar{\gamma}_2(z) \rfloor$
- *(agreement on fraction's order)*
  For all $z_1, z_2 \in C_\Sigma$ *with* $\bar{\gamma}_1(z_1) \leq K$ *and* $\bar{\gamma}_2(z_2) \leq K$*,*

procedure $\mathsf{monitor}_{\mathcal{A}_{ec}}(a,\delta)$
begin
{ ---------- }
{ step 1: initialisation (first call only) }
if **first_time** **then**
P := $\{(q,\bar{\Gamma}) \mid q \in Q_0 \wedge \bar{\Gamma}$ is initial$\}$;

{ ---------- }
{ step 2: symbolic step }
P' := $\{(q',\bar{\Gamma}') \mid (q,\bar{\Gamma}) \in P \wedge e=(q,a,\psi,q') \in E \wedge (q',\bar{\Gamma}') = \mathsf{symb\_step}((q,\bar{\Gamma}),\delta,e)\}$;
P := P' ;

{ ---------- }
{ step 3: emptiness check }
if $\bigwedge_{(q,\bar{\Gamma})\in P} \mathsf{empty}_{\mathcal{A}_{ec}}(q,\bar{\Gamma})$ **then**
return $\bot$
else
return ?
end

Fig. 8. Procedure $\mathsf{monitor}_{\mathcal{A}_{ec}}(a,\delta)$

- $\langle\bar{\gamma}_1(z_1)\rangle = 0$ *iff* $\langle\bar{\gamma}_2(z_1)\rangle = 0$
- $\langle\bar{\gamma}_1(z_1)\rangle \le \langle\bar{\gamma}_1(z_2)\rangle$ *iff* $\langle\bar{\gamma}_2(z_1)\rangle \le \langle\bar{\gamma}_2(z_2)\rangle$.

To construct $\mathsf{cover}_{\approx_R}(\bar{\Gamma})$, we use an equivalent description of the regions $[\bar{\gamma}]_{\approx_R}$ given as a set of constraints assembled according to the following rules [48]:

- For every clock $c \in C_\Sigma$ choose exactly one constraint from the set

$$\begin{array}{ll} \mathsf{choice}(c) = & \\ \{\bar{\gamma}(c) = v \mid v = \bot, 0, 1, \ldots, K\} & \text{type (1)} \\ \cup \quad \{v-1 < \bar{\gamma}(c) < v \mid v = 1,\ldots,K\} & \text{type (2)} \\ \cup \quad \{\bar{\gamma}(c) > K\} & \text{type (3)} \end{array}$$

- and for each pair of clocks $c \neq c' \in C_\Sigma$ which are both restricted by a type (2) constraint, choose additionally one constraint of the form

$$\langle\bar{\gamma}(c)\rangle \bowtie \langle\bar{\gamma}(c')\rangle \text{ with } \bowtie \in \{<,=,>\} \ . \quad \text{type (4)}$$

Hence, to compute $\mathsf{cover}_{\approx_R}(\bar{\Gamma})$, we have to find all constraint sets which obey these two rules and which are consistent with $\bar{\Gamma}$. First, we note that $\bar{\Gamma}$ does not impose any constraint between the values of two distinct clocks and therefore, $\bar{\Gamma}$ does not restrict the choice of type (4) constraints in $\mathsf{cover}_{\approx_R}(\bar{\Gamma})$. Consequently, to compute $\mathsf{cover}_{\approx_R}(\bar{\Gamma})$, we determine for each clock c the subset $\mathsf{choice}_{\bar{\Gamma}}(c) \subseteq \mathsf{choice}(c)$ of constraints which are consistent with $\bar{\Gamma}$. Then, $\mathsf{cover}_{\approx_R}(\bar{\Gamma})$ consists exactly of those regions $[\bar{\gamma}]_{\approx_R}$ which are determined by constraints chosen from the restricted set $\mathsf{choice}_{\bar{\Gamma}}()$.

This concludes our algorithm to compute $\mathsf{empty}_{\mathcal{A}_{ec}}(q,\bar{\Gamma})$ which is based upon the equivalence given in Corollary 51 and which uses the scheme described above for computing $\mathsf{cover}_{\approx}(\bar{\Gamma})$.

F. A Monitor Procedure for TLTL$_3$

We are now ready to present a monitor procedure for checking TLTL properties according to the three-valued semantics.

procedure $\mathsf{monitor}_{\varphi}(a,\delta)$
begin
{ ---------- }
{ step 1: symbolic step }
r_φ := $\mathsf{monitor}_{\mathcal{A}^{\varphi}_{ec}}(a,\delta)$;
$r_{\neg\varphi}$:= $\mathsf{monitor}_{\mathcal{A}^{\neg\varphi}_{ec}}(a,\delta)$;

{ ---------- }
{ step 2: compute verdict }
if $r_\varphi = \bot$ **then return** $\bot$;
if $r_{\neg\varphi} = \bot$ **then return** $\top$;
return ? { note: $r_\varphi = r_{\neg\varphi} =$? }
end

Fig. 9. Procedure $\mathsf{monitor}_{\varphi}(a,\delta)$

We symbolically execute the event-clock automaton $\mathcal{A}^{\varphi}_{ec}$ and check the emptiness for each reached pair consisting of a state and a symbolic clock valuation. In Figure 8, we show the procedure $\mathsf{monitor}_{\mathcal{A}_{ec}}(a,\delta)$ used to process a timed word $w = (a_0,t_0)(a_1,t_1)\ldots$ event-wise. After reading an event (a_i,t_i) (given as an event $a = a_i$ and a delay $\delta = t_i - t_{i-1}$ for $i > 0$ and $\delta = 0$ for $i = 0$), $\mathsf{monitor}_{\mathcal{A}_{ec}}(a,\delta)$ returns $\bot$ if the prefix $u = (a_0,t_0)\ldots(a_i,t_i)$ cannot be continued infinitely with a $\sigma \in T\Sigma^\omega$ such that the underlying event-clock automaton $\mathcal{A}_{ec}$ would accept $u\sigma$. Note that P is a global variable keeping track of the currently reached set of symbolic states.

In the implementation of $\mathsf{monitor}_{\mathcal{A}_{ec}}$, we combine the results of Sections IV-D and IV-E: $\mathsf{monitor}_{\mathcal{A}_{ec}}$ executes in parallel all symbolic timed runs which match the observed prefix and checks for the existence of possible continuations—according to the runtime verification criterion, as stated in Corollary 47 taken from Section IV-D. The runtime verification criterion involves an emptiness check for symbolic timed runs, which is in turn implemented according to Corollary 51 taken from Section IV-E.

Similar as in the discrete-time setting, given a property φ we run two versions of this monitor procedure in parallel, namely one for φ and another one for $\neg\varphi$. Then we combine the results of these two evaluations following directly the semantics of TLTL$_3$ to obtain the final verdict.

In Figure 9, we show the monitor procedure $\mathsf{monitor}_{\varphi}(a,\delta)$ for a TLTL$_3$-property φ. $\mathsf{monitor}_{\varphi}(a,\delta)$ also reads a finite prefix event-wise in terms of an event a and a delay δ and returns either $\bot, \top$, or ?, as determined by the semantics of TLTL$_3$.

G. Platform Adaption

In this section we discuss two practical issues arising in an implementation of the scheme laid out in the preceding sections, namely the *representation of time values* and the *detection of deadline expirations.* The problem of representing time values arises as we use real values for time values throughout the construction whereas we cannot represent reals with infinite precision. The problem of deadline expiration detection originates in the fact that our monitoring procedure is only reacting to incoming events, i.e., if an event is overdue,

this is not detected until another event is processed. Below we discuss both issues.

Representing Time Values: We based our construction on timed words involving non-negative real numbers as time stamps. But in any practical case, the occurring time stamps will be rational numbers, mostly expressed as counters with respect to a fixed denominator determined by some clock frequency. The correctness of our approach in such a setting relies on the following two properties of our monitor construction:

- *Monitor computations are precision independent.* Our monitor construction manipulates time values only in terms of additions, subtractions, comparisons, and assignments of integers. Since any rational- or integer-based time representation is closed under these operations, the system-wide used type for time values is sufficient for monitor-internal use as well.
- *Monitor generation is precision independent.* The generated monitor itself remains unaffected by the required precision for processing time stamps—only the type for representing time stamps must be chosen appropriately. If the region equivalence is used for the emptiness check, then the precomputed table $T[q, [\bar{\gamma}]_{\approx}]$ following Corollary 51 remains unaffected as well.

Summarised, to adapt our approach for a given system, it is only necessary to use the system's type for time values throughout the generated monitor.

Detection of Deadline Expirations: Our construction ensures that if a finite prefix u cannot be continued into an infinite word $u\sigma$ satisfying some TLTL-property φ, then the monitor $\mathsf{monitor}_\varphi$ will detect this fact immediately, i.e., for u of minimal length. However, in case of timed words, the lack of events is an input in itself. For example, if an event a is required by φ to occur within 4 seconds, then a quiescence of 6 seconds is meaningful with respect to our property φ which cannot be satisfied anymore. But $\mathsf{monitor}_\varphi$ will only detect the expired deadline, once the next event is being processed by $\mathsf{monitor}_\varphi$. There are three principal choices for dealing with this issue:

- *No further precaution.* In some cases, the behaviour as provided by $\mathsf{monitor}_\varphi$ is sufficient and hence no further provisions are necessary.
- *Statically scheduled interrupts.* If it is enough to detect an expired deadline within a certain period of time, then one can use an interrupt to send a special event to $\mathsf{monitor}_\varphi$ at a fixed rate, which is only used for checking deadline violations.
- *Dynamically scheduled interrupts.* Alternatively, we can compute in $\mathsf{symb\_step}$ the very next deadline to occur in monitoring φ and $\neg\varphi$. Then one can dynamically set a timeout interrupt for this minimal period of time and send an special event to $\mathsf{monitor}_\varphi$.

In any case, it is a simple matter to implement the desired detection of deadline expirations for the timed monitor, given that the corresponding interrupt types are provided by the target platform.

V. Conclusions

In this paper, we presented a runtime verification approach for properties expressed either in lineartime temporal logic (LTL) or timed lineartime temporal logic (TLTL), suitable for monitoring discrete-time and real-time systems, respectively.

Before introducing our technical approach, we discussed the relationship of runtime verification with model checking and testing in depth, thereby identifying its distinguishing features.

In contrast to LTL (TLTL), runtime verification deals with finite runs, thus asking for an LTL semantics on finite traces. We proposed a three-valued semantics: In our understanding of runtime verification, we consider a finite trace as an incrementally observed finite prefix of an unknown infinite trace, causing correctness properties to evaluate to either *true*, *false* or *inconclusive*.

For LTL_3, a conceptually simple monitor generation procedure is given, which is *optimal* in two respects: First, the size of the generated deterministic monitor is *minimal*, and, second, the monitor identifies a continuously monitored trace as either satisfying or falsifying a property *as early as possible*. Subsequently, we related our approach with existing techniques. Thereby, we identified the *monitorable properties* as *strictly* containing safety and co-safety properties.

For the real-time logic TLTL, we started with an analogous definition of a three-valued semantics. The resulting monitor construction, however, is technically much more involved. Automata for TLTL employ so-called *event recording* and *event predicting* clocks. Since in runtime verification the future of a trace is not known, such predicting clocks are difficult to handle. Introducing *symbolic* clock valuations, we were able to mimic the general approach as taken in the discrete-time case for constructing real-time monitors.

In this paper, we laid out the foundation for discrete-time and real-time monitoring of LTL and respectively TLTL properties. It remains to put these foundations into practice, and, as our long term goal, into common practice. For the discrete-time case, we have already implemented a prototype showing the feasibility of our approach, while an implementation for the real-time case remains to be done as part of future work.

References

[1] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, ser. Lecture Notes in Computer Science, S. Arun-Kumar and N. Garg, Eds., vol. 4337. Springer-Verlag, Dec. 2006.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[3] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-based Testing of Reactive Systems*, ser. Lecture Notes in Computer Science. Springer, 2005, vol. 3472.

[4] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*. Providence, Rhode Island: IEEE Computer Society Press, Oct. 31–Nov. 2 1977, pp. 46–57.

[5] K. Havelund and G. Rosu, "Monitoring Java Programs with Java PathExplorer," *Electr. Notes Theor. Comp. Sci.*, vol. 55, no. 2, 2001.

[6] D. Giannakopoulou and K. Havelund, "Runtime analysis of linear temporal logic specifications," RIACS/USRA, Tech. Rep. 01.21, 2001.

[7] ——, "Automata-based verification of temporal properties on running programs." in *ASE*. IEEE Computer Society, 2001, pp. 412–416.

[8] K. Havelund and G. Rosu, "Synthesizing Monitors for Safety Properties," in *Tools and Algorithms for Construction and Analysis of Systems*, 2002, pp. 342–356.

[9] ——, "Efficient monitoring of safety properties," *Journ. Softw. Tools for Tech. Transf.*, 2004.

[10] V. Stolz and E. Bodden, "Temporal Assertions using AspectJ," in *Proceedings of the 5th International Workshop on Runtime Verification (RV'05)*, ser. Electr. Notes Theor. Comput. Sci., vol. 144, no. 4. Elsevier, 2006, pp. 109–124.

[11] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "LOLA: runtime monitoring of synchronous systems," in *TIME*. IEEE Computer Society, 2005, pp. 166–174.

[12] C. Prisacariu and G. Schneider, "A formal language for electronic contracts," in *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, ser. LNCS, vol. 4468. Paphos, Cyprus: Springer, June 2007, pp. 174–189.

[13] A. Bauer, M. Leucker, and C. Schallhart, "Model-based runtime analysis of reactive distributed systems," in *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, Apr. 2006, pp. 243–252.

[14] A. P. Sistla and E. M. Clarke, "Complexity of propositional temporal logics," *Journal of the ACM*, vol. 32, pp. 733–749, 1985.

[15] N. Markey, "Past is for free: On the complexity of verifying linear temporal properties with past," in *Proceedings of the 9th International Workshop on Expressiveness in Concurrency (EXPRESS'02)*, ser. Electronic Notes in Theoretical Computer Science, U. Nestmann and P. Panagaden, Eds., vol. 68, no. 2. Brno, Czech Republic: Elsevier Science Publishers, Aug. 2002, pp. 87–104. [Online]. Available: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/NM-express2002.pdf

[16] H. W. Kamp, "Tense logic and the theory of linear order," Ph.D. dissertation, University of California, Los Angeles, 1968.

[17] T. Berg, B. Jonsson, M. Leucker, and M. Saksena, "Insights to Angluin's learning," in *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003)*, ser. Electronic Notes in Theoretical Computer Science, vol. 118, Dec. 2003, pp. 3–18. [Online]. Available: http://www.sciencedirect.com/science?_ob=MImg&_imagekey=B75H1-4FFN49V-18-1&_cdi=13109&_user=616147&_orig=browse&_coverDate=02%2F01%2F2005&_sk=998819999&view=c&wchp=dGLbVlz-zSkzV&md5=1cd6a1fcf1599d96b97a2e45a181c0e5&ie=/sdarticle.pdf

[18] D. Peled, M. Vardi, and M. Yannakakis, "Black box checking," in *Proc. FORTE/PSTV*. Kluwer, 1999, pp. 225–240. [Online]. Available: http://www.cs.rice.edu/~vardi/papers/pstv99.ps.gz

[19] A. Pretschner and M. Leucker, "Model-based testing - a glossary," in *Model-Based Testing of Reactive Systems*, ser. Lecture Notes in Computer Science, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., vol. 3472. Springer, 2004, pp. 607–609.

[20] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. New York: Springer, 1995.

[21] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout, "Reasoning with temporal logic on truncated paths." in *CAV*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 27–39.

[22] O. Kupferman and M. Y. Vardi, "Model checking of safety properties." *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001.

[23] M. d'Amorim and G. Rosu, "Efficient monitoring of omega-languages." in *CAV*, ser. Lecture Notes in Computer Science, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 364–378.

[24] A. Pnueli and A. Zaks, "PSL model checking and run-time verification via testers." in *FM*, ser. Lecture Notes in Computer Science, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 573–586.

[25] M. Geilen, "On the construction of monitors for temporal logic properties." *Electr. Notes Theor. Comput. Sci.*, vol. 55, no. 2, 2001.

[26] M. Chechik, B. Devereux, and A. Gurfinkel, "Model-checking infinite state-space systems with fine-grained abstractions using spin." in *SPIN*, ser. Lecture Notes in Computer Science, M. B. Dwyer, Ed., vol. 2057. Springer, 2001, pp. 16–36.

[27] A. Gruler, M. Leucker, and K. Scheidemann, "Modelling and verifying software product lines," TU München, Tech. Rep., 2007, to appear.

[28] J.-F. Raskin, "Logics, automata and classical theories for deciding real-time," Ph.D. dissertation, Namur, Belgium, 1999.

[29] D. D'Souza, "A logical characterisation of event clock automata," *Int. Journ. Found. Comp. Sci.*, vol. 14, no. 4, pp. 625–639, Aug. 2003.

[30] R. Alur, L. Fix, and T. A. Henzinger, "Event-clock automata: a determinizable class of timed automata," *Theor. Comp. Sci.*, vol. 211, no. 1-2, pp. 253–273, 1999.

[31] J. Håkansson, B. Jonsson, and O. Lundqvist, "Generating online test oracles from temporal logic specifications," *Journ. Softw. Tools for Tech. Transf.*, vol. 4, no. 4, pp. 456–471, 2003.

[32] S. Tripakis, "Fault diagnosis for timed automata." in *FTRTFT*, ser. Lecture Notes in Computer Science, W. Damm and E.-R. Olderog, Eds., vol. 2469. Springer, 2002, pp. 205–224.

[33] P. Bouyer, F. Chevalier, and D. D'Souza, "Fault diagnosis using timed automata." in *FoSSaCS*, ser. Lecture Notes in Computer Science, V. Sassone, Ed., vol. 3441. Springer, 2005, pp. 219–233.

[34] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals." in *FORMATS/FTRTFT*, ser. Lecture Notes in Computer Science, Y. Lakhnech and S. Yovine, Eds., vol. 3253. Springer, 2004, pp. 152–166.

[35] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Symposium on Logic in Computer Science (LICS'86)*. Washington, D.C., USA: IEEE Computer Society Press, June 1986, pp. 332–345.

[36] M. Y. Vardi, *An Automata-Theoretic Approach to Linear Temporal Logic*, ser. Lecture Notes in Computer Science. New York, NY, USA: Springer, 1996, vol. 1043, pp. 238–266.

[37] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[38] S. Schwoon and J. Esparza, "A note on on-the-fly verification algorithms," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005, pp. 174–190.

[39] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," *Theory of Machines and Computation*, pp. 189–196, 1971.

[40] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Form. Methods Syst. Des.*, vol. 19, no. 3, pp. 291–314, 2001.

[41] G. Rosu and S. Bensalem, "Allen linear (interval) temporal logic - translation to LTL and monitor synthesis," in *CAV*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 263–277.

[42] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles and Techniques and Tools*. Addison-Wesley, 1986.

[43] B. Stroustrup, *The C++ Programming Language*, special ed. Boston, MA, USA: Addison-Wesley, 2000.

[44] S. Dewhurst, *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[45] C. Fritz, "Constructing Büchi automata from linear temporal logic using simulation relations for alternating büchi automata." in *CIAA*, ser. Lecture Notes in Computer Science, O. H. Ibarra and Z. Dang, Eds., vol. 2759. Springer, 2003, pp. 35–48.

[46] A. Bauer, M. Leucker, and C. Schallhart, "The good, the bad, and the ugly—but how ugly is ugly?" in *Proceedings of the 7th International Workshop on Runtime Verification (RV'07)*, ser. Lecture Notes in Computer Science, vol. 4839. Vancouver, Canada: Springer-Verlag, Dec. 2007, pp. 126–138.

[47] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. New York: ACM, Jan. 1985, pp. 97–107.

[48] R. Alur and D. L. Dill, "A theory of timed automata." *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[49] J.-F. Raskin and P.-Y. Schobbens, "The logic of event clocks—decidability, complexity and expressiveness." *Journ. of Autom. Lang. and Comb.*, vol. 4, no. 3, pp. 247–286, 1999.

[50] S. Tripakis and S. Yovine, "Analysis of timed systems using time-abstracting bisimulations." *Formal Methods in System Design*, vol. 18, no. 1, pp. 25–68, 2001.

[51] R. Alur, C. Courcoubetis, and D. L. Dill, "Model-checking in dense real-time," *Information and Computation*, vol. 104, no. 1, pp. 2–34, 1993.