Runtime Verification for LTL and TLTL

ANDREAS BAUER, NICTA and Australian National University
MARTIN LEUCKER, Technische Universität München
CHRISTIAN SCHALLHART, Technische Universität Darmstadt

This article 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 feasibility of the developed methodology is demontrated using a collection of real-world temporal logic specifications. Moreover, the presented approach is 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 *cosafety* 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 shown, possible.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Verification

Additional Key Words and Phrases: Assertion checkers, monitors, runtime verification

ACM Reference Format:
Bauer, A., Leucker, M., and Schallhart, C. 2011. Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol 20, 4, Article 14 (September 2011), 64 pages.
DOI = 10.1145/2000799.2000800 http://doi.acm.org/10.1145/2000799.2000800

1. 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 [Clarke et al. 1999] and testing.

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

---

This article is a revised and extended version of Bauer et al. [2006b].
Authors' address: A. Bauer, M. Leucker, and C. Schallhart; email: baueran@rsise.anu.edu.au; leucker@in.tum.de; schallhart@forsyte.de.

DOI 10.1145/2000799.2000800 http://doi.acm.org/10.1145/2000799.2000800

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 deals with the word problem, that is, the problem of whether a given word is included in some language.

Correctness properties in runtime verification specify all admissible individual executions of a system and may be expressed using a variety of different formalisms. These range from, for example, language oriented formalisms like extended regular expressions [Sen and Rosu 2003] or tracematches by the AspectJ team [Allan et al. 2005], to query-oriented languages (PQL, [Martin et al. 2005]) and rule-based approaches like Eagle [Barringer et al. 2004] and RuleR [Barringer et al. 2007]. Moreover, temporal logic-based formalisms, which are well-known from model checking, are also very popular in runtime verification, especially variants of linear temporal logic, such as LTL [Pnueli 1977] , as seen for example in [Giannakopoulou and Havelund 2001a, 2001b; Havelund and Rosu 2001, 2002, 2004; Stolz and Bodden 2006]. But also linear μ-calculus variants or past-time temporal logics that are not LTL-based are used, for example in D'Angelo et al. [2005] and, respectively, Kim et al. [2004].

Research in runtime verification is mainly focused on the *detection* of violations (or satisfactions) of correctness properties, for instance, by developing synthethis algorithms for monitors from high-level specifications. At the same time, most runtime verification systems allow the mitigation of the problem by executing suitable and user-defined code, for instance, as in the MAC system [Kim et al. 2002]. Similarly, *Monitor-oriented programming* [Chen and Rosu 2003; Chen and Roşu 2007] aims at a programming methodology that allows for the execution of code whenever monitors observe a violation of a given correctness property. *Runtime reflection* [Bauer et al. 2006a], on the other hand, is an architecture pattern that is applicable for systems in which monitors are enriched with a diagnosis and reconfiguration layer.

Runtime verification is closely related to the field of runtime monitoring, which has recent applications in monitoring web services [Barbon et al. 2006; Baresi et al. 2008; Robinson 2006] and fault monitoring, see Delgado et al. [2004] for an overview.

A large number different runtime verification systems have been developed in recent years. We only list their names with appropriate references as a further comparison of these tools is beyond the scope of this article: PQL [Martin et al. 2005], Tracematches [Allan et al. 2005], MOP [Chen and Roşu 2007], SASI [Erlingsson and Schneider 2000], Pal [Chaudhuri and Alur 2007], Eagle [Barringer et al. 2004], RuleR [Barringer et al. 2007], J-Lo [Bodden 2004; Stolz and Bodden 2006], PTQL [Goldsmith et al. 2005].

1.1 Runtime Verification versus Model Checking

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

(1) In model checking, all executions of a given system are examined to answer whether these satisfy a given correctness property φ, either explicitly or implicitly, when using symbolic techniques. This 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 Sistla and Clarke [1985] and Markey and Schnoebelen [2003].

(2) While model checking, especially in case of LTL, considers infinite traces, runtime verification deals with finite traces as nonidealized executions are necessarily finite.
(3) 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 develop a set of core techniques differing significantly from those used for model checking to enable runtime verification applications. For example, item (2) 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 [Kamp 1968]. However, this semantics as well as further alternatives given in Manna and Pnueli [1995] and Eisner et al. [2003] are not suitable for runtime verification, as we discuss in Section 1.3.

A popular branch of model checking is so-called *bounded model checking* [Biere et al. 1999, 2003]. The underlying insight, which is equally used in conformance testing [Chow 1978; Vasilevski 1973], is that for every finite-state system, an infinite trace must reach at least one state twice. Thus, if a finite trace reaches a state a second time, the trace can be extended to an infinite trace by taking the corresponding loop infinitely often. Likewise, considering all finite traces of length up-to the state-space plus one, one covers all possible loops of the underlying system without explicitly considering the system's model, for instance, by computing strongly connected components, as done in conventional model checking algorithms. This observation paired with the right adaption of LTL's semantics to finite traces allows to deduce a formula's validity with respect to infinite traces based on all finite executions up-to a certain length.

Clearly, similar ideas seem tempting for runtime verification as well. However, in runtime verification, in contrast to the standard approach taken in bounded model checking, an upper bound on the system's state space is typically not known. More importantly, the states of an observed execution usually do not reflect the system's state completely but only contain the value of certain variables of interest. Thus, seeing a state twice in an observed execution does not allow one to infer that the observed loop can be taken ad infinitum. Moreover, in runtime verification, one does not have any symbolic representation of all executions up-to a certain length, but deals with one concrete finite execution. This renders the LTL semantics used for bounded model checking inappropriate 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. Note that it is possible to automatically learn [Berg et al. 2003] and verify a system model, thereby applying model checking techniques to an a priori unknown system [Peled et al. 1999].

Furthermore, model checking suffers from the so-called *state explosion problem*, which terms the fact that analyzing 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, most applications of runtime verification are not practically limited by their memory requirements, since the necessary history information, although potentially unbounded, is commonly fairly small.

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, that is, 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 impose only a minimal penalty on the system's response time and memory footprint. Ideally, the monitor does not alter the running system in terms of its functional and non-functional behaviour.

1.2 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*. 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*. 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; that is, 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.

1.3 Monitoring of Discrete-Time Properties

In this article, we are only considering monitoring of properties that are specified in LTL (or later, in its timed counterpart TLTL). As Pnueli's LTL [Pnueli 1977] 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 summarized in the following rationale.

> Pnueli's LTL [Pnueli 1977] 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, we introduce LTL_3 as a lineartime temporal logic which shares the syntax with LTL but deviates in its semantics 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 semantics for LTL on finite traces ([Eisner et al. 2003; Manna and Pnueli 1995], see also Bauer et al. [2010] for their comparison), these use (only) two truth values. We strongly believe that only two truth values lead to misleading results in runtime verification. Consider the formula $\neg pUinit$ (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 Bauer et al. [2006b]. However, some essential concepts were defined by Kupferman and Vardi: In Kupferman and Vardi [2001] 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 nondeterministic 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 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, that is, 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 d'Amorim and Rosu [2005], 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 that complements the constructions carried out in d'Amorim and Rosu [2005] with a formal framework. Nevertheless, we feel that our constructions are more explicit in their semantical foundation and are therefore easier to understand. The feasibility of the developed methodology is demontrated using Dwyer et al.'s collection of temporal logic specifications on which the common specification patterns are based on Dwyer et al. [1999]. For the LTL properties among these formulae, almost all resulting monitors had a size less than 100, measured in terms of the number of states and transitions.

In this article, we further discuss which LTL_3 properties are *monitorable* at all. We follow the definition given by Pnueli and Zaks in Pnueli and Zaks [2006] 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* [d'Amorim and Rosu 2005] in monitors, which is similar yet not the same. Moreover, we recall the notion of safety and cosafety properties. We show that the popular belief that monitoring is suitable only for safety properties is misleading: The class of monitorable properties is richer than the union of safety and cosafety 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 Geilen [2001]. 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.

In general, in multivalued logics, a formula is not evaluated to either *true* or *false*, but one of several (truth) values. This allows for more precise assessment to which extent a formula is considered true. Such multivalued approaches have been considered for LTL, for example, in the context of abstraction [Chechik et al. 2001]. There, the semantics is defined for *infinite* traces and the resulting logics and model checking approaches are completely different from LTL_3. In contrast, we designed LTL_3 to specifically match the needs arising in runtime verification.

1.4 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 [1999]. TLTL, as argued by D'Souza [2003], can be considered a natural counterpart of LTL in the timed setting: He showed 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 [Kamp 1968] 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 [Alur et al. 1999].

So far, not many approaches for runtime verification of real-time properties have been given. Håkansson et al. [2003] study monitor generation based on LTL enriched with a freeze quantifier for time. In Thati and Rosu [2005], monitoring algorithms for metric temporal logic specifications are presented. Moreover, the very general rewriting-based approaces of RulerR [Barringer et al. 2007] and Eagle [Barringer et al. 2004] also allow the specification of realtime properties. In Tripakis [2002] and Bouyer et al. [2005], *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. Moreover, no three-valued semantics in the sense used here has been discussed.

TLTL is event-based, meaning that the system emits events when the system's state has changed. In Maler and Nickovic [2004] monitoring of continuous signals is

considered, which is intrinsically different to observing discrete signals in a continuous time domain.

1.5 Outline

In Section 2, 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. We demonstrate this approach with an example from concurrent C++-development practice and show its feasibility using the formulae upon which Dwyer et al.'s patterns are based on.

Section 3 analyzes the structure of the developed monitors and complements the notions of good and bad prefixes with *ugly* prefixes to characterise the instant when properties become *nonmonitorable*. Moreover, we discuss monitoring in the light of safety and cosafety properties and compare our work with ideas based on *informative* prefixes.

In Section 4, we expand our runtime verification approach to 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.

We draw our conclusion in Section 5.

2. THREE-VALUED LTL IN THE DISCRETE-TIME SETTING

In this section, we consider runtime verification for systems whose behavior 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 behavior 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 [Pnueli 1977], and, consequently, our developments in this section are for LTL specifications.

More precisely, the section is organized as follows.

— *Preliminaries* (*Section 2.1*). We recall standard LTL syntax and semantics (Definitions 2.1 and 2.2) altogether with standard nondeterministic Büchi automata (Definition 2.3) used to match words satisfying LTL properties.

— *Syntax and Semantics of LTL_3* (*Section 2.2*). Using LTL semantics as basis, we introduce a three-valued semantics to evaluate standard LTL formulae on finite words yielding the logic LTL_3 (Definition 2.4). Thereby, LTL_3 distinguishes three cases follows:

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

— *Monitor Construction for LTL_3* (*Section 2.3*). 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

words according to LTL_3 semantics: First, we describe how to utilize the Büchi automata $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$ accepting the infinite words that satisfy φ and $\neg\varphi$, respectively, to evaluate φ according to LTL_3 semantics (Lemma 2.5). Then, using this evaluation rule, we construct from $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$ the finite-state machine $\mathcal{M}^{\varphi}$ (Definition 2.6) and prove that it indeed implements the LTL_3 semantics of φ (Theorem 2.7). Note that the resulting monitor $\mathcal{M}^{\varphi}$ evaluates φ as predictively as possible, since once it can be decided that φ will remain either satisfied or unsatisfied, the monitor will provide this information immediately. Moreover, it can be minimized to obtain an FSM with a provably minimal number of states.

— *Example (Section 2.4).* We demonstrate this approach with an example from concurrent C++-development practice.

— *Evaluation (Section 2.5).* Taking the temporal logic formulae on which Dwyer et al.'s specification patterns are based upon as a starting point, we construct corresponding monitors and measure their resulting size. Thereby, we show the practical feasability of our monitoring methodology.

2.1 Preliminaries

For the remainder of this section, let AP be a finite set of atomic propositions and $\Sigma = 2^{\mathrm{AP}}$ a finite alphabet. We write a_i for any single element of Σ, that is, a_i is a possibly empty set of propositions taken from AP. Finite traces over Σ are elements of Σ^*, and are usually denoted by $u, v, u', v', u_1, v_1, u_2, \ldots$, whereas infinite traces are elements of Σ^{ω}, usually denoted by $w, w', w_1, w_2, \ldots$ 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 2.1 (*LTL formulae [Pnueli 1977]*). 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 2.2 (*LTL semantics [Pnueli 1977]*). 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 formulae inductively as follows

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

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 2.3 (*Nondeterministic Büchi automaton (NBA) [Büchi 1962]*). 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 nonempty set of states,
— $Q_0 \subseteq Q$ is a set of initial states,
— $\delta : Q \times \Sigma \to 2^Q$ is a transition function, and
— $F \subseteq Q$ is a set of accepting states.

We extend the transition function $\delta : Q \times \Sigma \to 2^Q$, as usual, to $\delta' : 2^Q \times \Sigma^* \to 2^Q$ 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 $w = a_0 a_1 \ldots \in \Sigma^\omega$ is a sequence of states and actions $\rho = q_0 a_0 q_1 a_1 q_2 \ldots$, 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 $\mathrm{Inf}(\rho)$ denote the states visited infinitely often. A run ρ of an NBA $\mathcal{A}$ is called *accepting* iff $\mathrm{Inf}(\rho) \cap F \neq \emptyset$.

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 $\rho = q_0 a_0 q_1 a_1 \ldots a_n q_{n+1}$, 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 us to apply determinization once we have converted all NBAs into NFAs, thereby yielding a deterministic finite-state machine. The resulting FSM can be minimized to obtain an FSM with a provably minimal number of states. Note that in many practical cases, the monitor will be based directly on the underlying nondeterministic automata and will be determinized on-the-fly using the power-set construction (cf. the discussion at the end of Section 2.3).

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

2.2 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. Thus, for a given formula φ, either $w \models \varphi$ holds or not. In practice, however, we can observe only 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 by ?, 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 2.1 on finite prefixes to obtain a truth value from the set $\mathbb{B}_3 = \{\bot, ?, \top\}$ as follows.

Definition 2.4 (*LTL_3 semantics*). Let $u \in \Sigma^*$ denote a finite word. The *truth value* of a LTL_3 formula φ with respect to u, denoted by $[u \models \varphi]$, 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 this 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 Kamp [1968], 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 well as further approaches given in Manna and Pnueli [1995] or in Eisner et al. [2003] inappropriate in our case.

2.3 Monitor Construction for LTL_3

Now we develop an automata-based monitoring procedure for LTL_3. More specifically, for a given formula $\varphi \in \mathrm{LTL}_3$, 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 $\varphi \in \mathrm{LTL}$ for the rest of this section, and let $\mathcal{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ denote the NBA that 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 [Vardi and Wolper 1986] and explained, for example in Vardi [1996]. 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 $\mathcal{L}(\mathcal{A}^\varphi(q)) \neq \emptyset$, that is, 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 [Tarjan 1972] or nested depth-first algorithms as examined in Schwoon and Esparza [2005]. 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 2.5 (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 with $\mathcal{L}(\mathcal{A}^{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. 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 2.4, $[u \models \varphi] = \top$.

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 2.5 to determine whether $[u \models \varphi]$. 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.

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 2.6 (*Monitor $\mathcal{M}^{\varphi}$ for an LTL_3 formula φ*). Let φ be an LTL formula and let $\hat{\mathcal{A}}^{\varphi}$ be the NFA, as defined above, indicating emptiness per state of φ's NBA. Moreover, let $\tilde{\mathcal{A}}^{\varphi} = (\Sigma, Q^{\varphi}, \{q_0^{\varphi}\}, \delta^{\varphi}, \tilde{F}^{\varphi})$ be a deterministic automaton with $\mathcal{L}(\tilde{\mathcal{A}}^{\varphi}) = \mathcal{L}(\hat{\mathcal{A}}^{\varphi})$ and let and $\tilde{\mathcal{A}}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, \{q_0^{\neg\varphi}\}, \delta^{\neg\varphi}, \tilde{F}^{\neg\varphi})$ be the analogously defined DFA for $\neg\varphi$.

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 the unique FSM 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.

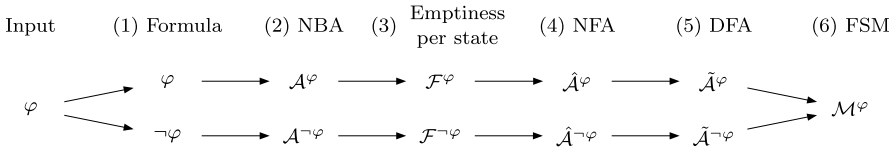


Fig. 1. The procedure for getting $[u \models \varphi]$ for a given φ.

THEOREM 2.7 LTL MONITOR CORRECTNESS. *Let φ be an LTL_3 formula 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 2.6 and Lemma 2.5 on the evaluation of LTL_3. □

Complexity. Consider Figure 1. Given φ, step 1 requires replication and negation of φ, that is, it is linear in the size of φ. 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 minimized with standard algorithms for FSMs Hopcroft [1971] 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 [Kupferman and Vardi 2001].

Thus, better complexity results in other approaches, like the one in Havelund and Rosu [2002], are due to one of the following reasons.

— First, one can use a fragment of LTL that is strictly less expressive than full LTL, that is, one gives up the possibility to specify certain properties and thereby rules out some complicated cases exercising the worst case complexity. Our construction yields an optimal monitor, independent of which fragment of LTL is considered.
— Second, it is possible to use a variant of LTL that 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 synchronizing automata, hereby trading the size of automaton with an increased computational overhead at runtime [Rosu and Bensalem 2006].

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", similar to the approach taken in d'Amorim and Rosu [2005].

To do so, our implementation employs the power-set construction, known from compiler construction [Aho et al. 1986], 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).

Although we could follow the whole construction on-the-fly, that is, build the NBAs from the LTL-formulae and do the emptiness check per state during monitoring, we experienced a huge and favorable difference in size between the pair of NBAs and the resulting monitoring FSMs, as described in Section 2.5. For the same reason, it seems infeasible to base our approach on more succinct *alternating Büchi automata*. Checking emptiness (per state) of an alternating Büchi automaton is PSPACE-complete and thus not affordable at runtime. Note that existing approaches for monitoring LTL properties that use alternating automata [Finkbeiner and Sipma 2004; Stolz and Bodden 2006] in fact check LTL properties in a different manner. As discussed in detail in Section 3.3, these procedures check for informative prefixes rather than bad prefixes.

2.4 Example

Now we discuss a simple but comprehensive real-world example in more detail, which also highlights most of the features already described. Note that other approaches [Allan et al. 2005; Bodden 2004; Chen and Roşu 2007], are able to deal with this example as well, yet, as we feel, in a less direct manner.

In a C++-program, all static objects of an executable are initialized before the `main` method is entered, however, their order is undefined, and their initialization is thus performed in a nondeterministic order [Stroustrup 2000]. In consequence, if threads get spawned before executing `main`, it is difficult to ensure that all resources necessary to synchronize those threads are already initialized, such as globally available and statically initialized mutex objects. This problem is generally known as the *static initialization order fiasco* ([Dewhurst 2002]). The "fiasco" is an especially complicated one when large applications are built from a number of different frameworks that must remain independent from each other.

Using our monitor generator with a C++ logging layer such as the APACHE SOFTWARE FOUNDATION'S library, LOG4CXX,[1] 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 initialization.

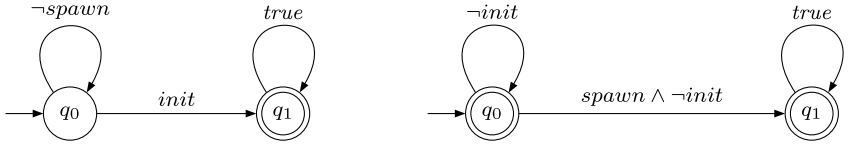
This example further illustrates the need for having three truth values, instead of two, when monitoring a running system.

— 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 in Fritz [2003], one obtains for φ, respectively $\neg\varphi$, the Büchi automata depicted in Figure 2.

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 2.5.

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



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

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

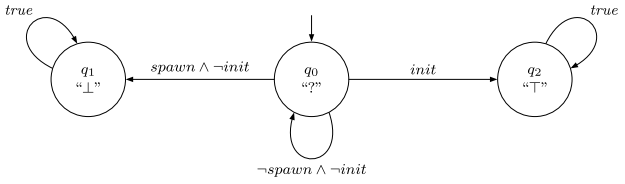


Fig. 3. The deterministic FSM $\mathcal{M}^{\varphi}$ for $\varphi \equiv \neg spawn\ U\ init$.

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 2.6, we construct an FSM as monitor for the static initialization 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, for instance, 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.

2.5 Evaluation

We have evaluated our approach on the basis of the frequently used software *specification patterns* introduced in Dwyer et al. [1999]. They represent a collection of patterns that occur commonly in the specification of concurrent and reactive systems, such as scoping (e.g., "some event P must hold in between two other events Q_1 and Q_2"), precedence (e.g., "an event P must always be preceded by an event Q"), or response (e.g., "an event P must always be followed by an event Q"). These patterns were derived from a comprehensive survey undertaken by the authors of that paper with the results being available online,[2] and at the time of writing consisting of 447 temporal logic specifications, of which 108 are labeled as being LTL formulae.

Starting point of our evaluation were these 108 formulae labeled as LTL formulae for which we then tried to generate monitors according to the procedure given above. To this end, we developed a "monitor generator," in the following referred to as the LTL_3 tools, which is available online under a public license.[3] It relies on version 1.1 of LTL2BA, which is Gastin's and Oddoux's translator for converting LTL formulae into corresponding Büchi automata [Gastin and Oddoux 2001]. From the 108 formulae, however, only 97 were actually usable in a sense that they were syntactically correct LTL formulae (as opposed to CTL or incomplete specifications).

[2]See `http://patterns.projects.cis.ksu.edu/`.
[3]See `http://LTL3tools.SourceForge.net/`.

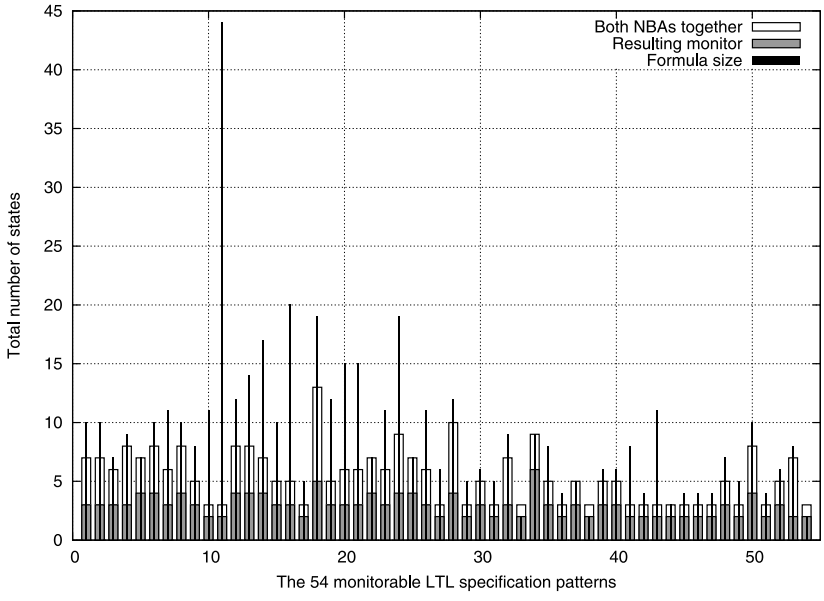


Fig. 4. Number of states of NBAs vs. number of states of monitors.

Generating a (minimized and deterministic) monitor for each of these 97 formulae using the LTL_3 tools, it turned out that for 43 formulae, the monitor consisted of a single ?-state which had one universal self-loop. These are formulae which can only be satisfied or violated by an infinite trace as compared to a finite one, hence the resulting monitor did not contain $\bot$ and $\top$ states in such cases. And as such, obviously, they cannot be sensibly monitored. (For a more detailed discussion of what constitutes a *monitorable property*, see also Section 3).

For the remaining 54 LTL formulae, respectively referred to as φ, Figure 4 shows the size of φ (measured in the standard way, i.e., nodes of the syntax tree), and the size of φ's monitor (measured in terms of the *number of states*) in comparison with the cumulative size of the two NBAs generated for φ and $\neg\varphi$ during monitor construction. Although, in the worst case, the monitor is double exponentially bigger than the size of the formula, the number of states of the resulting monitor is in the same range as the length of the formula. This suggests that the worst-case upper bound is no limiting factor for practical applications. With no exception, the resulting monitors are always smaller than the corresponding NBAs, and in some cases significantly smaller. The diagram also shows that the monitors contain only a seemingly small number of states in all of the cases, which is due to the specific formulae chosen by Dwyer et al. [1999], and not inherent to our method. However, as the formulae are taken from real world applications of temporal logic specification during systems design, we believe that this further emphasizes the suitability of our approach for monitoring design specifications also at runtime.

In practice, not only the states of a monitor have to be stored but also their transitions. Therefore, Figure 5 shows the overall size of the monitors built, that is, we now count the number of states *and transitions*. Note that for this purpose we have normalized the automata in the sense that all possible transitions for each state are explicitly represented and counted. In fact, the number of states in the previous diagram are now multiplied with the size of the alphabet underlying the formula.

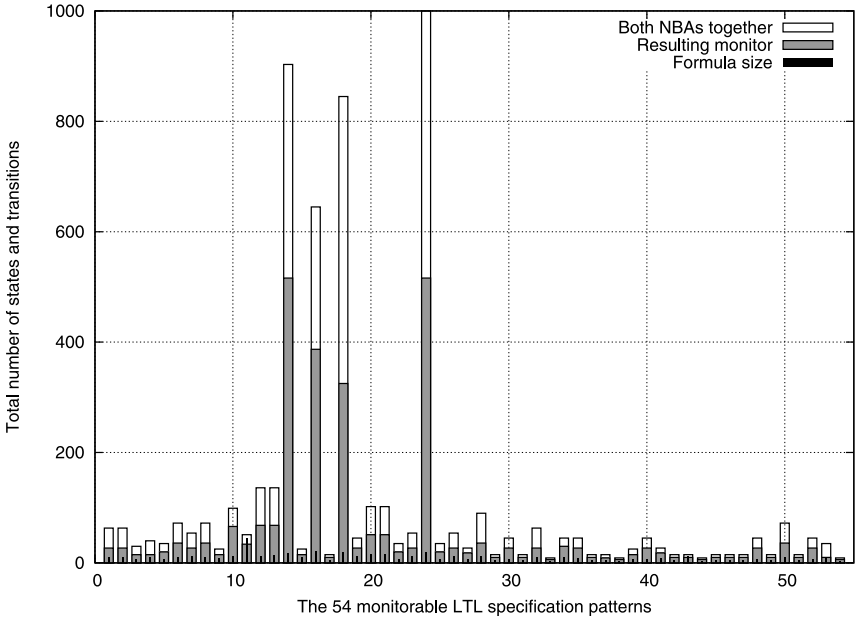


Fig. 5. Total size of NBAs vs. total size of monitors.

From the measurements described in this section, we draw the following conclusions. Using deterministic monitors does not seem to be a limiting factor for practically relevant specifications, despite the double exponential worst-case upper bound. Nevertheless, generating the monitor *on-the-fly* directly from the underlying NFA will typically reduce its memory requirements—*but* only if the corresponding NFAs are minimized, as our measurements indicate.

While minimization is an expensive operation, in can be done offline before deploying the monitor. However, if one computes even the underlying NBA on-the-fly (instead of precomputing the NBA and the NFA), it is necessary to perform emptyness-checks on-the-fly as well and to postpone the minimization of the (partially) generated NFA to runtime. The best trade-off between precompuation and on-the-fly computation has to be explored in the context of different application scenarios.

3. LTL_3 PUT INTO PERSPECTIVE

Let us compare the approach carried out in the previous section with some accomplishments in the literature. In Section 3.1, we compare LTL_3's semantics when faced with so-called *good* and *bad* prefixes [Kupferman and Vardi 2001]. It turns out that LTL_3's semantics identifies exactly good and bad prefixes. Bad prefixes may be used to derive the notion of *safety properties*. Consequently, in Section 3.2, we study monitoring for the subclass of (co)-safety properties [Alpern and Schneider 1987; Kupferman and Vardi 2001; Lamport 1977]. Moreover, we recall the notion [Pnueli and Zaks 2006] of *monitorable* properties and show that monitorable properties are more than just safety properties. A related notion, due to Kupferman and Vardi [2001], which is, in a sense, weaker than that of bad prefixes, is the one of *informative prefixes*. In Section 3.3, we explain the idea of informative prefixes and compare monitoring based on LTL_3's semantics to monitoring approaches based on informative prefixes.

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

3.1 Good/Bad Prefixes

Let us first recall the notion of *good* and *bad prefixes* as introduced in Kupferman and Vardi [2001].

Definition 3.1 (*Good/bad prefixes [Kupferman and Vardi 2001]*). Given a language $L \subseteq \Sigma^\omega$ of infinite words over Σ, we call a finite word $u \in \Sigma^*$

— 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 follows.

Remark 3.2 (*LTL_3 identifies good/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 d'Amorim and Rosu [2005] is to modify a given Büchi automaton, typically arising from a given LTL property, into a monitor that signals the occurrence of a minimal bad prefix. Thus, this construction yields a monitor that distinguishes two cases, namely $[u \models \varphi] = \bot$ and $[u \models \varphi] \neq \bot$. At the same time, d'Amorim and Rosu [2005] do not discuss the semantics of the resulting monitor in terms of a matching logical framework. LTL_3 can be understood as a logic which complements the constructions carried out in d'Amorim and Rosu [2005] 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 2.6), a good or bad prefix leads to a state, which outputs either $\top$ or $\bot$, and which is only looping back to itself. We call such a state a *trap*. For example, a monitor for Fp enters a trap $\top$ once the first state satisfying p occurs. Analogously, a monitor for Gp reaches a trap with $\bot$ once p becomes *false* the first time.

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 3.3 (*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.

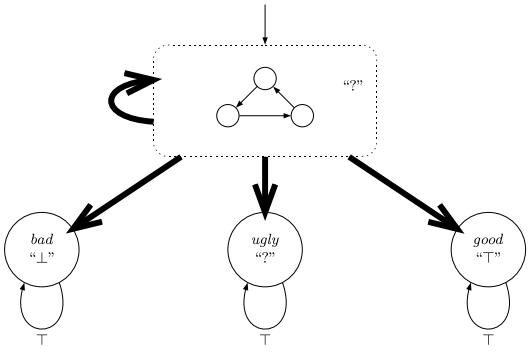


Fig. 6. The structure of the deterministic monitor.

We follow Pnueli and Zaks [2006] in calling a formula φ *nonmonitorable* with respect to a prefix u, if no $\bot$ or $\top$ verdict can be obtained. Using our terminology, we define the following.

Definition 3.4 ((*Non*)-*monitorable*). Let φ be an LTL-formula and $u \in \Sigma^*$. We call φ *nonmonitorable 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 nonmonitorable 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 6. 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 Section 2.5, we generated monitors for the LTL formulae of Dwyer et al.'s specification patterns. As mentioned before, 44 out of 97 formulae consisted of a single state emitting ?. Thus, the underlying properties are indeed nonmonitorable. For the remaining properties, it turns out that no trap outputting ? exists, so that the underlying properties are monitorable.

In d'Amorim and Rosu [2005], 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 6, both *ugly* and *good* are never-violate states, that is, in d'Amorim and Rosu [2005], 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 important to actually classify a prefix $u \in \Sigma^*$ as either good or ugly, 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 monitoring continuations of u.

Note that the notion of nonmonitorable fits well to LTL_3. In Bauer et al. [2007], however, we suggest a more precise semantics of LTL-formulae with respect to finite words allowing us 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 nonmonitorable properties can still be considered meaningful, but this discussion is beyond the scope of this article.

3.2 Safety and Cosafety Properties

The notion of bad and good prefixes was introduced in Kupferman and Vardi [2001] in the context of safety and cosafety languages as well as formulae, and which follows the formal definitions as given in Alpern and Schneider [1987]:

Definition 3.5 ((*Co-*)*Safety languages [Kupferman and Vardi 2001]*). 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 *cosafety 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 3.6 (*Safety/Cosafety property*). A formula $\varphi \in$ LTL is called a *safety property* (*cosafety property*), if its set of models $\mathcal{L}(\varphi)$ is a safety language (cosafety language, respectively).

Let us give some examples:

formula	safety	cosafety
Gp	•	
Fq		•
Xp	•	•
GFp		
$Xp \vee GFp$		
pUq		•

The definitions of safety and cosafety properties and languages immediately yield the following.

Remark 3.7 (*Safety/Cosafety properties are monitorable*). Each LTL formula that is safety or cosafety is also monitorable.

In other words, for a safety or a cosafety language L, there are no ugly prefixes and for a safety or cosafety formula φ, the monitor $\mathcal{M}^\varphi$ has no ugly trap. However, this property also holds for some nonsafety/cosafety properties.

LEMMA 3.8 (MONITORABLE IS MORE THAN SAFETY AND COSAFETY). *The class of monitorable LTL$_3$ properties is strictly larger than the union of safety and cosafety properties.*

PROOF. Consider, for example, $\varphi = ((p \vee q)Ur) \vee Gp$. 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 by $\ldots$, 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 cosafety 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 ???..., one could classify the property as satisfied. This follows the intuition that, if nothing has gone wrong for a "long time," the property to be checked is indeed satisfied. The proof of the previous lemma shows that such an understanding does not work for all monitorable properties. For example, both $ppp\ldots$ and $qqq\ldots$ do not reach a trap when monitoring $\varphi = ((p \vee q)Ur) \vee Gp$. Thus, the monitor $\mathcal{M}^\varphi$ will output the infinite sequence ???... 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 ???..., one cannot classify the property as satisfied or violated, as both $ppp\ldots$ and $qqq\ldots$ yield the same output sequence ????....

To summarize this discussion, we note that

— violations of safety properties are reported by monitoring procedures that check for finite prefixes of violating words, as well as
— successful validations of cosafety properties are reported by monitoring procedures that consider finite prefixes of satisfying words, but additionally,
— there are monitorable properties that are neither characterized by finite violating or finite satisfying prefixes.

3.3 Informative Prefixes

Kupferman and Vardi [2001], the authors have introduced the notion of *informative prefixes*. The idea is to consider prefixes of infinite words that "tell the whole story" of why a formula is (not) satisfied [Kupferman and Vardi 2001]. Consider, for example, the formula $Xfalse$. 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 $Xfalse$, 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 Kupferman and Vardi [2001], we consider LTL formulae in negation normal form, that is, the set of formulae defined by the following grammar:[4]

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

The semantics is defined as expected, for instance, 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, that is, 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 3.9 (*Informative prefix [Kupferman and Vardi 2001]*). 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 $\neg\varphi \in \ell(0)$, $\ell(n+1) = \emptyset$,

[4]While the ideas presented below can also be developed in the version of LTL defined in Section 2 (as done for example in Eisner et al. [2003]), we follow Kupferman and Vardi [2001] to simplify the presentation.

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 the following.

Remark 3.10 (*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, that is, there are formulae which have bad prefixes but no informative ones, as shown in the following examples.

Example 3.11 (*Informative prefixes*).

— Consider $\varphi = Gp$ and $u = pq$. Note that u is a bad prefix for Gp and that $\neg Gp = F\neg p$. Recall also that $\neg\varphi$ is a shorthand for the negation of φ in negation normal form. 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 $\varphi_2 = G(p \vee X\mathit{false})$ and $u = pq$ as before. As φ_2 is equivalent to Gp, u is still a bad prefix for φ_2. Note, $\neg\varphi_2 = F(\neg p \wedge X\mathit{true})$. Thus, some witness ℓ_2 should satisfy $F(\neg p \wedge X\mathit{true}) \in \ell_2(0)$. As p is satisfied in the first position of u, it has to hold that $\{F(\neg p \wedge X\mathit{true}), \neg p \wedge X\mathit{true}, X\mathit{true}\} \subseteq \ell_2(1)$. This implies that $\mathit{true} \in \ell_2(2) \neq \emptyset$. 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 $\varphi_3 = G(p \vee F\mathit{false})$ and $u = pq$. As φ_3 is equivalent to Gp, u is still a bad prefix for φ_3. Note, $\neg\varphi_2 = F(\neg p \wedge G\mathit{true})$. Now, having $G\mathit{true}$ as a subformula in some possible witness $\ell_3(i)$ requires $G\mathit{true}$ 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 [Lichtenstein and Pnueli 1985] and alternating-automata-based approach to model checking LTL formulae [Vardi 1996]: 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 Lichtenstein and Pnueli [1985].

In Kupferman and Vardi [2001], the notion of informativeness is used to classify safety properties into three distinct safety levels. A safety property $\varphi \in$ LTL is *intentional safety* if all its bad prefixes are informative, it is *accidental safety* if every word that violates φ has an informative prefix, and *pathological safety* if there is a word that violates φ which has no informative prefix.

For example, Gp is intentional safety, $G(p \vee X\mathit{false})$ is accidental safety, and $G(p \vee F\mathit{false})$ is pathological safety. All three formulae are equivalent; that is, they accept the same set of models. Interestingly, Eisner et al. [2003] give a semantical characterisation of informative prefixes in terms of a *weak* semantics of LTL on finite traces; however, for the discussion to come, we stick to the syntactical presentation. See [Bauer et al. 2010] for further details.

Obviously, the notions introduced before can be dualized towards the notion of *co*-informative prefixes with mappings that are witnesses of why a property φ is satisfied by considering informative prefixes of $\neg\varphi$. Thus, the discussion based on bad and good prefixes can be reconsidered in terms informative and coinformative prefixes.

Discussion. 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 intentional safety properties and not also for accidental or pathological safety properties. Then, of course, monitors identifying only informative prefixes suffice to report all bad prefixes. However, while Kupferman and Vardi [2001] provide a decision procedure for checking whether a formula is intentional safety, no conversion algorithm from nonintentional to intentional 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 intentional safety. However, it might be too hard and cumbersome for the user to carry out a translation manually, and not necessary when following our construction. Then the only downside may be that when reporting a bad but not informative prefix to a user, it could be harder to understand why the prefix is indeed bad.

The monitor synthesis algorithms [Geilen 2001; Håkansson et al. 2003] 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 Geilen [2001] and implicitly in Håkansson et al. [2003]). Kupferman and Vardi [2001] suggest that one could search for an informative prefix for φ as well as the negation of φ.

Because of Remark 3.10 (and its implicit dual), such a search procedure would stop on either some good or bad prefix; however, these prefixes have to be informative at the same time. Hence, in all the aforementioned 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. This may be because it is not informative "so far" or because there is not even any informative extension.

Monitors checking exclusively for informative bad prefixes, for instance, as in d'Amorim and Rosu [2005], are very useful in many practical settings. Such a monitor reports the violation of a given property, and by using the negation of the property in question, one can as well check for the satisfaction of the property. This is sufficient in many application scenarios and might be the most efficient solution in these cases. However, we believe that using bad prefixes instead of informative bad prefixes (and their duals) is an advantage in its own right, since bad prefixes allow our monitors to report, in contrast to informative bad prefixes, a property violation as early as possible.

Likewise, using two monitors, one checking for a bad prefix (based on the property) itself and another one for a good prefix (based on the formula's negations), it is possible to check for the satisfaction and violation of a given property as the same time and report results as early as possible. However, we want to point out that our combined monitors are smaller than the two underlying NFAs, as discussed in Section 2.5, unless these are minimized by sophisticated and expensive algorithms. Thus, our procedure provides an efficient alternative if satisfaction and violation need to be monitored at the same time.

4. THREE-VALUED LTL IN THE REAL-TIME SETTING—TLTL

Especially for embedded, safety critical systems it is important to check real-time properties. For such systems, one distinguishes between *event-triggered* and *time-triggered* systems [Kopetz 1991]. In this section, we consider runtime verification for event-triggered 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 behavior 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) system states 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 of 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—requiring a more complicated logic as starting point. By following an event-based approach, we avoid these issues entirely. Moreover, one 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.[5]

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 [1999]. TLTL, as argued by D'Souza [2003], can be considered a natural counterpart of LTL in the timed setting. Consequently, our developments in this section deal with TLTL specifications.

TLTL is very well suited for expressing simple yet typical time-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. But in most cases, a specification such as $XXXa$ occurs if one would like to express that the event a should occur after three time units regardless of how many other events have occurred in between.

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 the corresponding time stamps on the other hand. This renders TLTL especially suitable to specify properties of *asynchronous* systems.

We follow again the three-valued semantics motivated in the LTL section: This is the main difference from existing approaches of checking realtime properties as, for example, for Metric-interval Temporal Logic in Thati and Rosu [2005] or in RulerR [Barringer et al. 2007] and Eagle [Barringer et al. 2004].

We give a detailed outline of this section.

— *Preliminaries (Section 4.1).* We start out with *timed words* (Definition 4.1, following Alur and Dill [1994]) where each event is associated witht a non-negative and strictly monotone increasing time-stamp. Such timed words give rise to the definition of recording and predicting clocks which report, respectively, the time elapsed since the last occurrence of some event, and the time it will take until some event occurs for the next time. The evaluation rules for these clocks are given in terms of *clock evaluation functions* (Definition 4.2 [D'Souza 2003]) which determine the values of all clocks at the time of a corresponding event. This definition plays a crucial role for the TLTL semantics, as introduced later (Definition 4.12). But before we describe the TLTL semantics, we discuss the clock valuation function and its consequences: Most notable, the clock evaluation function associated with some event, does not depend on the current event but only on the current time stamp and on past and future events—ignoring the current event (Remark 4.3). In particular, if the current event is a, then the recording as well as the predicting clock for a refer,

[5]In contrast to TLTL, we presented LTL with a proposition-based alphabet in order to follow in both cases the most common form of presentation. Alternatively, we could have defined LTL in terms of events as well, because—in case of LTL—choosing either events or propositions is mostly a matter of notational convenience.

respectively, to the last preceding and first subsequent occurrence of a. This appears to be a natural choice in order to maximise the information available at each current time instant while it does not impose any further complications in case of infinite words. However, as we will see, in case of finite words which are evaluated incrementally, this choice bears a number of technical issues.
The constraints on the future expressed by predicting clocks make a symbolic representation of clock valuation functions necessary. To support such a symbolic representation, we first state the *noncoincidence* and *continuity* properties of clock valuation functions and describe some notation on clock valuation functions (Proposition 4.4). By not enforcing the noncoincidence property on clock valuation functions, one can relax the strict monotonicity of timed words to monotonicity (Remark 4.5). Then we recall *intervals* suitable to represent ranges of clock valuations (Definition 4.6 [D'Souza 2003]): These intervals can be closed, half-open, open, or identical to $\perp$, indicating that the corresponding event never occurred in the past or will never occur in the future, whereas ∞ is indicating that the event will occur eventually. Using intervals, we introduce *clock constraints* (Definitions 4.7 and 4.8) as foundation for symbolic evaluation needed later. A clock constraint associates with each recording and predicting clock an interval of possible evaluations. We state a number of further properties of clock constraints which we will use in the subsequent proofs (Remark 4.9 and Fact 4.10).
The material of Section 4.1 takes Alur and Dill [1994] and D'Souza [2003] as starting point to develop suitable definitions of clock constraints (Definitions 4.7 and 4.8) and a number of immediate properties (Remark 4.3, Proposition 4.4, Remark 4.9, and Fact 4.10) which are presumably all well-known, even if not stated explicitly in the literature.

— *Syntax and Semantics of TLTL$_3$ (Section 4.2).* We recall standard TLTL syntax and semantics (Definitions 4.11 and 4.12) according to D'Souza [2003] and present some TLTL properties (Example 4.13). Following the same rationale as for LTL$_3$, we introduce a three-valued semantics TLTL$_3$ for evaluating standard TLTL formulae on finite timed words (Definition 4.14). We discuss the evaluation rules of TLTL$_3$ (Example 4.15) using the same properties as before in Example 4.13.
— *Monitor Construction for TLTL$_3$ (Sections 4.3 throughout 4.6).* In Section 4.3, we continue with a *detailed overview* on the now more involved monitor construction, which includes *symbolic runs for event-clock automata* (Section 4.4), *emptiness check for symbolic states* (Section 4.5), and the *monitor procedure* itself (Section 4.6).
— *Platform Adaption (Section 4.7).* We conclude the real-time case on issues arising in adapting our monitor construction to specific platforms.

4.1 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 $(a_0, t_0)(a_1, t_1)\ldots$ of timed events $(\Sigma \times \mathbb{R}^{\geq 0})$.

Definition 4.1 (*Timed Word [Alur and Dill 1994]*). An (infinite) *timed word* w over the alphabet Σ is an (infinite) sequence $(a_0, t_0)(a_1, t_1)\ldots$ of *timed events* (a_i, t_i) consisting of symbols $a_i \in \Sigma$, and nonnegative 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*).

Please note that strict monotonicity can be relaxed to monotonicity, as discussed in Remark 4.5. But aside from this discussion on the difference in handling strictly monotone and monotone timed words, we always refer to the strict case. The *length* of

a timed word is denoted by $|w|$ 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, that is, every word in $T\Sigma^*$ satisfies strict monotonicity and every word in $T\Sigma^\omega$ satisfies both, strict monotonicity and 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, that is, 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.

Furthermore, for w as above, we call its sequence of events the *untimed word* of w, denoted by $\mathsf{ut}(w) = a_0 a_1 \dots$ and we write $\mathsf{ut}(\mathcal{L}) = \{\mathsf{ut}(w) \mid w \in \mathcal{L}\}$ 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 $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^* \cup T\Sigma^\omega$, an event $a \in \Sigma$, and an index i as arguments:

$$
\begin{array}{lll}
\mathsf{last}(w, a, i) = t_i - t_j & \text{iff} & a_j = a \text{ and } \quad 0 \leq j < i \\
 & \text{and} & a_k \neq a \text{ for all } \quad j < k < i \\
\mathsf{last}(w, a, i) = \perp & \text{iff} & a_j \neq a \text{ for all } \quad 0 \leq j < i \\
\mathsf{next}(w, a, i) = t_j - t_i & \text{iff} & a_j = a \text{ and } \quad i < j < |w| \\
 & \text{and} & a_k \neq a \text{ for all } \quad i < k < j \\
\mathsf{next}(w, a, i) = \perp & \text{iff} & a_j \neq a \text{ for all } \quad i < j < |w|
\end{array}
$$

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

Definition 4.2 (*Clock Valuation Function [D'Souza 2003]*). The *clock valuation function* γ_i over the timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^* \cup T\Sigma^\omega$ is a map $C_\Sigma \to T_\perp$ with $T_\perp = \mathbb{R}^{\geq 0} \cup \{\perp\}$ which assigns—corresponding to position i in w—a positive real or the undefined value $\perp$ to each clock variable such that the following holds:

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

The *set of clock valuation functions* over the clocks C_Σ is denoted by V_Σ (i.e., each $\gamma \in V_\Sigma$ is induced by some timed word).

Thus, γ_i describes the evaluation of the clocks in C_Σ at time t_i, but it ignores the fact that the event a_i: $\gamma_i(x_{a_i}) = \mathsf{last}(w, a_i, i)$ does *not* evaluate to 0 but refers to the ultimate 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 4.3 (*Blind Spot of Clock Valuation Functions*). For 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.

Definition 4.2 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 causes 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 handle this problem, we introduce in Section 4.4 *symbolic* timed runs (Definition 4.28). While their technical definition is somewhat involved, the basic idea behind their construction is straightforward: Instead of relying on precise information on the timing of future events, symbolic timed runs use *constraints on the permissible timing* of these future events. The runtime monitor maintains these constraints by advancing the time, incorporating transition guards, and adding information on occurred events. If the constraints become infeasible, then some timing constraint has been violated, while otherwise there still exists a (time-wise) consistent continuation.

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

PROPOSITION 4.4 (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 clocks $c \in C_\Sigma$ *and for all pairs of events* $a \neq a' \in \Sigma$, $\gamma(x_a) \neq \bot$ *implies* $\gamma(x_a) \neq \gamma(x_{a'})$ *and* $\gamma(y_a) \neq \bot$ *implies* $\gamma(y_a) \neq \gamma(y_{a'})$.
(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), noncoincidence, holds since there exist no two events $(a_i, t_i)(a_{i+1}, t_{i+1})$ with $t_i = t_{i+1}$ (given the strict monotonicity of the underlying word). Property (b), continuity, holds since the corresponding infinite word has an infinite supply of events (given the progress property of the underlying word). □

Remark 4.5 (*Relaxing Strict Monotonicity*). In Definition 4.1, we imposed strict monotonicity on timed words. However, it is possible to relax the strict monotonicity requirement to monotonicity, that is, to require $t_i \leq t_{i+1}$ instead of $t_i < t_{i+1}$ for all $i \in \mathbb{N}$. This holds true, since we do not rely on strict monotonicity *but only preserve* it—by means of the noncoincidence property of clock valuations as described in Proposition 4.4.

To shift our approach from strict monotonicity to monotonicity, one *needs to drop the noncoincidence requirement* in all checks for some $\gamma \in V_\Sigma$. Besides frequent occurrences in our proofs, such a check occurs only once in the final construction (we give the forward reference to avoid dispersing information on the issue throughout the paper): In step 3 of the procedure symb_step (Figure 9), we check for $\gamma \in V_\Sigma$ **with** $\gamma \models \Gamma_i$ **and** $\gamma \models \psi$. If one wants to preserve strict monotonicity, one needs to check whether

there exists a noncoincident and continuous γ satisfying $\gamma \models \Gamma_i$ **and** $\gamma \models \psi$—and if one wants to preserve simple monotonicity, one needs to check only whether there exists a continuous solution γ satisfying the same condition $\gamma \models \Gamma_i$ **and** $\gamma \models \psi$.

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

$$\begin{aligned} \gamma[c = v](c') &= \gamma(c') \quad \text{iff } c' \neq c \\ \gamma[c = v](c') &= v \qquad \text{iff } c' = c. \end{aligned}$$

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

$$\begin{aligned} (\gamma \pm \delta)(x_a) &= \gamma(x_a) \pm \delta \quad \text{iff } \gamma(x_a) \neq \bot \\ (\gamma \pm \delta)(x_a) &= \bot \qquad\qquad \text{iff } \gamma(x_a) = \bot \\ (\gamma \pm \delta)(y_a) &= \gamma(y_a) \mp \delta \quad \text{iff } \gamma(y_a) \neq \bot \\ (\gamma \pm \delta)(y_a) &= \bot \qquad\qquad \text{iff } \gamma(y_a) = \bot, \end{aligned}$$

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 $\mathbb{R}^{\geq 0}$ with integral boundary values and the singleton $\{\bot\}$.

Definition 4.6 (*Intervals [D'Souza 2003]*). The set $\mathcal{I}$ of *intervals* contains all intervals of the form $[(l, r)]$ where $[($ and $)]$ 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 $[(l, \infty)$ 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 $[t, t]$, as below in the definition of clock constraints: Each clock constraint Ψ_Σ over the clocks in C_Σ requires a subset of clocks in C_Σ to assume corresponding values in a respective interval $I \in \mathcal{I}$.

Definition 4.7 (*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 $\gamma \models \psi$.

Thus, if $\psi(c) = \mathsf{undef}$ for a clock $c \in C_\Sigma$, then ψ does not constrain the value of c, that is, 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 which meet Proposition 4.4: Every clock valuation function must satisfy noncoincidence and continuity, and hence each clock constraint $\psi \in \Psi_\Sigma$ must allow a noncoincident and continuous clock valuation function as solution.

Definition 4.8 (*Set of Clock Constraints*). The *set of clock constraints* on the clocks C_Σ is denoted by Ψ_Σ and contains all clock constraints ψ which have 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, that is, 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 4.9 (*Conjunction of Clock Constraints*). If the two clock constraints $\psi_0, \psi_1 \in \Psi_\Sigma$ are consistent, that is, 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

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

We require $\gamma \models \psi_i$ for $i = 0, 1$ in order to ensure $\psi_0 \wedge \psi_1$ to be defined and valid, since then $\gamma \models (\psi_0 \wedge \psi_1)$ holds and $\psi_0 \wedge \psi_1$ has indeed a noncoincident and continuous solution.

We 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 4.10 (*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 $v \in T_\bot$ 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 $v \in T_\bot$ $\gamma[c = v] \models \psi[c = \mathsf{undef}]$ holds.

4.2 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 D'Souza [2003] where it is called LTL_{ec}.

Definition 4.11 (*TLTL Formulae [D'Souza 2003]*). The set of *formulae φ of TLTL* is defined by the grammar

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

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 $true \; U \; \varphi$, and $G\varphi$ for $\neg(true \; U \; \neg\varphi)$. 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 4.12 (*Semantics of TLTL [D'Souza 2003]*). 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:

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

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 4.13 (*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 $t_{i+1} - t_i > 5$, that is, two adjacent occurrences of *alive* may be separated by an arbitrary period of time. Note that we could replace the interval $[0, 5]$ by $(0, 5]$ without affecting the meaning of the formula, since no clock can measure a zero distance to a previous or upcoming event.
— $G(\rhd_{alive} \in [0, 5])$ requires that from every given point in time, *alive* will occur within the next 5 time units. In contrast to the preceding example, in this case 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.

— $G(req \rightarrow \triangleright_{req} \notin [0,5])$ requires that two subsequent request events *req* are separated by strictly more than 5 time units.
— $G(actuator \rightarrow \triangleleft_{error} \in [\bot,\bot])$ states that if an *actuator* event occurs, then previously, no *error* has occurred (therefore, we could equivalently write $G(error \rightarrow G\neg actuator)$ in standard LTL).
— In our last example we consider a more involved property: Assume that a first request r_1 must be complemented by a second request r_2, before it is acknowledged with a reply *ack*. Assume moreover that the first request r_1 has to be answered within less than 2 time units and that the acknowledgement *ack* has to be sent out more than 1 time unit after the second request r_2. Allowing arbitrary and inconsequential intermediate events w at any time, we arrive at the following property:

$$\varphi = G\left(r_1 \rightarrow \left(\triangleright_{ack} \in [0,2) \wedge X\left(wU\left(r_2 \wedge \triangleright_{ack} \in (1,\infty) \wedge X\left(wUack\right)\right)\right)\right)\right)$$

Analogously to the discrete-time case, we now define a 3-valued semantics for TLTL, yielding the logic TLTL_3.

Definition 4.14 (*Semantics of TLTL*$_3$). Let $u \in T\Sigma^*$ be a finite timed word. The *truth value* of a TLTL_3 formula φ with respect to u, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:

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

In this 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 $u = (a_0,t_0)\ldots(a_i,t_i)$, 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.

Example 4.15 (TLTL_3 *Evaluation*). To illustrate the three-valued semantics, we discuss the evaluation of several properties from Example 4.13:

— $G(\neg alive \rightarrow \triangleright_{alive} \in [0,5])$ 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 ?.
— $\triangleright_{alive} \in [0,2]Udone$ 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 signaled 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 $\bot$.
— Consider the property φ relating r_1, r_2, and *ack* with

$$\varphi = G\left(r_1 \rightarrow \left(\triangleright_{ack} \in [0,2) \wedge X\left(wU\left(r_2 \wedge \triangleright_{ack} \in (1,\infty) \wedge X\left(wUack\right)\right)\right)\right)\right)$$

and observe that φ evaluates to $\bot$ if there is a request r_1 which is not answered in time by a subsequent request r_2 and an acknowledgement *ack*: In particular, the timing fails, if (a) more than 1 time unit passes between r_1 and r_2, (b) at least 2 time units pass between r_1 and *ack*, or (c) not more than 1 time unit passes between r_2 and *ack*. The conditions (b) and (c) correspond directly to the expressions $\rhd_{ack} \in [0, 2)$ and $\rhd_{ack} \in (1, \infty)$ respectively. Condition (a) is a consequence of (b) and (c): If r_2 occurs at least 1 time unit after r_1, then every *ack* respecting (b) must violate (c) and vice versa.

Importantly, the semantics of TLTL_3 require that we detect a constraint violation in case of (a) even before processing *ack* when the constraint violation becomes obvious. Otherwise φ is evaluated to ?.

4.3 Overview on TLTL_3 Monitoring

In this section, we outline our monitor construction for TLTL_3, which we make concrete 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 from a TLTL_3-property φ, we generate for φ and its negation $\neg\varphi$ the two event-clock automata $\mathcal{A}^{\varphi}_{ec}$ and $\mathcal{A}^{\neg\varphi}_{ec}$ [Raskin and Schobbens 1999]. These two automata accept the timed words satisfying and respectively violating φ. 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 has to deal with predicting clocks and has to use a more complex emptiness check. Both issues are addressed separately in Sections 4.4 and 4.5, respectively. Then in Section 4.6, having suitable techniques at hand, we build the final monitor, following closely the scheme used in the discrete-time setting.

Let us give a comprehensive outline of our construction.

— *Symbolic Runs of Event-Clock Automata (Section 4.4).*
As starting point for the monitoring procedure, we recall the definition of event-clock automata and their timed runs (definitions 4.16 and 4.17, respectively, following Alur et al. [1999]) over infinite words as suitable automaton model for TLTL (Theorem 4.18 [Raskin and Schobbens 1999]). 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 4.20) which describe the viable values for each predicting clock as clock constraint. Standard timed runs involve *more than one event* in each transition, and therefore we cannot develop an incremental symbolic execution directly upon timed runs. Consequently we introduce *incremental timed runs* (Definition 4.24) as an equivalent alternative (Proposition 4.25). Based on incremental timed runs, we develop a procedure to implement a transition symbolically (Figure 9) and prove that this procedure is indeed abstracting all concrete transitions (Lemma 4.31). Next, we define symbolic timed runs (Definition 4.28) 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 4.30) 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 4.31).

It would remain to show the converse, that is, that every symbolic timed run is concretized by a corresponding timed run. However, this is not the case as there are spurious symbolic timed runs which cannot be concretized (Proposition 4.32). But we can use a backward simulation argument to show that every individual symbolic transition has a corresponding concrete transition (Lemma 4.33). Applying such backward steps inductively, we can concretize a symbolic run to a *finite* concrete timed run (Lemma 4.34).

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 concretized and continued by an ordinary infinite timed run iff there exists a timed run over the concatenated infinite word (Theorem 4.35). This leads directly to a criterion for runtime verification (Corollary 4.36): 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 state and a symbolic clock valuation that gives rise to a nonempty language.

— *Emptiness Check for Symbolic States (Section 4.5).* 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 4.37) which work with any time-abstract bisimulation relation (Definition 4.38 Tripakis and Yovine [2001]), and the emptiness check based upon such automata (Theorem 4.39 [Alur and Dill 1994]), 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 4.40). Finally, we recall the region equivalence for event-clock automata [Alur et al. 1993; Alur and Dill 1994] as one particular instance of a bisimulation relation and show how to compute a set of regions which covers a given symbolic clock valuation. Note that the zone equivalence as used in model checking tools for timed systems such as Uppaal [Bengtsson et al. 1996; Behrmann et al. 2006] does not yield a bisimulation and is thus inapplicable in our setting.

— *A Monitor Procedure for TLTL$_3$ (Section 4.6).* 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.

4.4 Symbolic Runs of Event-Clock Automata

We first recall event-clock automata as the automata model suitable to match TLTL-properties: 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 respective edge.

Definition 4.16 (*Event-Clock Automaton [Alur et al. 1999]*). Let Σ be a finite alphabet and C_Σ the corresponding set of event-recording and event-predicting clocks. Then

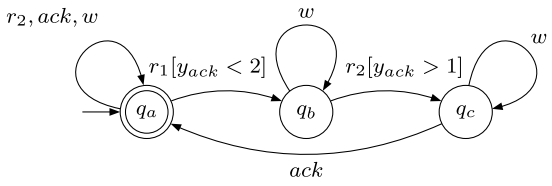


Fig. 7. Event clock automaton for φ from Example 4.19.

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.

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 4.17 (*Timed Run, following [Alur et al. 1999]*). Given an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$, a *timed run* θ of $\mathcal{A}_{ec}$ 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, that is, $q_0 \in Q_0$,
— each γ_i assumes values according to Definition 4.2 (thus γ_0 must be initial), and
— 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 Büchi acceptance condition instead of the standard one is due to the construction given in Raskin and Schobbens [1999].

THEOREM 4.18 (TLTL TO E.-C. AUTOMATA [RASKIN AND SCHOBBENS 1999]). *For each TLTL-property φ, there exists a constructible event-clock automaton $\mathcal{A}^\varphi_{ec}$ such that $\mathcal{L}(\mathcal{A}^\varphi_{ec}) = \{\sigma \in T\Sigma^\omega \mid \sigma \models \varphi\}$ holds. In the worst case, $\mathcal{A}^\varphi_{ec}$ has exponential size with respect to the length of φ.*

Example 4.19 (*Event-Clock Automaton*). Recall property

$$\varphi = G\left(r_1 \rightarrow \left(\triangleright_{ack} \in [0, 2) \wedge X\left(wU\left(r_2 \wedge \triangleright_{ack} \in (1, \infty) \wedge X\left(wUack\right)\right)\right)\right)\right)$$

introduced and discussed in Examples 4.13 and 4.15. A corresponding event-clock automaton $\mathcal{A}^\varphi_{ec}$ is given in Figure 7.

Because of Theorem 4.18, the question whether—given a timed prefix u and a TLTL-property φ—there exists an accepting infinite continuation σ, that is, $u\sigma \models \varphi$ holds, translates into the question whether a finite run of the event-clock automaton $\mathcal{A}^\varphi_{ec}$ over

u is possibly continued into an accepting run or not. Hence, from this point onwards, we will consider the latter question in order to turn TLTL-properties into executable monitoring procedures.

In case of runtime verification, transition guards which involve event-predicting clock variables impose 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 maintain only their conjunction—which is again a single clock constraint (see Remark 4.9). Note that a transition is only enabled, if all constraints on future events resulting from taking the transition are consistent and satisfiable.

Thus, when we symbolically execute an 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.

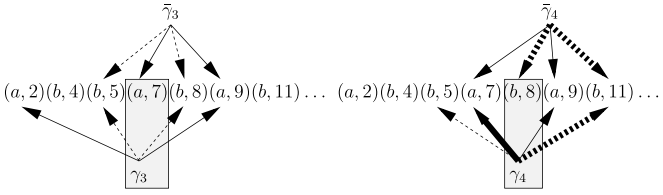
Definition 4.20 (*Symbolic Clock Valuation*). A symbolic clock valuation is a clock constraint $\Gamma \in \Psi_\Sigma$ where $\Gamma(x_a) = [l, l]$ with $l \in T_\perp$ holds for all event-recoding clocks.

Intuitively, a clock valuation function γ satisfies a symbolic clock valuation Γ, that is, $\gamma \models \Gamma$, 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 4.21 (*Initial Symbolic Clock Valuation*). A symbolic clock valuation Γ_0 over the clocks C_Σ is called the initial symbolic clock valuation iff $\Gamma_0(x_a) = [\perp, \perp]$ for all event-recoding clocks $x_a \in C_\Sigma$ and iff $\Gamma_0(y_a) = \mathsf{undef}$ for all event-predicting clocks $y_a \in C_\Sigma$.

We want to use symbolic runs in online monitoring, and therefore they must be computable incrementally, that is, we need to make transitions of the form $(q_i, \Gamma) \overset{(a_i,\delta_i)}{\to} (q_{i+1}, \Gamma')$ on processing some event a_i occurring $\delta_i = t_i - t_{i-1}$ time units after the preceding one. To support such symbolic transitions, we need a corresponding notion of *incremental* concrete transitions $(q_i, \gamma) \overset{(a_i,\delta_i)}{\to} (q_{i+1}, \gamma')$. As we will show in the following, the sequence of standard clock valuation functions $\gamma_0, \gamma_1, \ldots$ for a timed word $w = (a_0, t_0)(a_1, t_1)\ldots$ cannot be used in such an incremental approach (using $\gamma = \gamma_i$ and



i	(a_i, t_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	(a,2)	$\perp$	5	$\perp$	2	**0**	**5**	$\perp$	2
1	(b,4)	**2**	3	$\perp$	**1**	2	3	**0**	**1**
2	(b,5)	3	2	**1**	**3**	3	2	**0**	**3**
3	(a,7)	5	**2**	**2**	1	**0**	**2**	2	1
4	(b,8)	**1**	1	3	**3**	1	1	**0**	**3**

Fig. 8. Incremental and ordinary clock valuations.

$\gamma' = \gamma_{i+1}$) since γ_{i+1} depends not only on γ_i and the current timed event (a_i, t_i) but also on next subsequent event (a_{i+1}, t_{i+1}). Therefore, it is impossible to define a transition of the form $(q_i, \gamma) \overset{(a_i,\delta_i)}{\rightarrow} (q_{i+1}, \gamma')$ with standard clock valuation functions.

To overcome these difficulties, we introduce *incremental clock valuation functions* (Definition 4.22) which rely—as required—on the current event a_i and the time δ_i elapsed after the last processed event. Henceforth, we use incremental clock valuation functions as foundation for symbolic runs. To differentiate the incremental variant of concrete and symbolic timed runs from conventional runs, we denote them as $\bar{\gamma}$ and $\bar{\Gamma}$.

First, let us discuss sequences of standard clock valuation functions in detail. In Figure 8, we show a prefix of a timed word over the alphabet $\Sigma = \{a, b\}$. Hence, every clock valuation γ_i refers to 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 8 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, for instance, 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 8 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 move 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 4.2 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. This holds true, since all recording clocks x_a for $a \neq a_i$ and all predicting clocks y_a for $a \neq a_{i+1}$ only observe δ_{i+1} elapsing time units as we move from γ_i to γ_{i+1}—which amounts to the expression $(\gamma_i + \delta_{i+1})$ in Equation (1). To complete the transition from γ_i to γ_{i+1}, we must move the blind spot (Remark 4.3) from (a_i, t_i) to (a_{i+1}, t_{i+1}). Hence, γ_{i+1} must encounter a_i again, requiring the reset of x_{a_i}, while we have to ignore a_{i+1}, requiring us to reset $y_{a_{i+1}}$ to look beyond a_{i+1}.

Following Equation (1), 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 8. 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, that is, 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 by $\bar{\gamma}_i$. 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 4.22 (*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:

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

Thus, $\bar{\gamma}_i$ describes the values of the clocks from C_Σ *directly after* the timed event (a_i, t_i) occurred, that is, $\bar{\gamma}_i(x_{a_i}) = 0$. 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 8, we show the clock valuation functions γ_3 and γ_4 as a cursor that is placed upon some timed event, whereas the incremental clock valuation functions $\bar{\gamma}_3$ and $\bar{\gamma}_4$ are shown as a cursor that 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 8 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 check whether e is enabled, that is, $\gamma_i \models \psi$ holds, it must first compute γ_i with the following rule:

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

with $\delta_i = t_i - t_{i-1}$ for $i > 0$ and $\delta_0 = t_0$. To show that Equation (2) holds, we point out that $\mathsf{last}(w, a, i) = \mathsf{last}(w, a, i-1) + \delta_i$ holds for $a \neq a_{i-1}$ and $\mathsf{last}(w, a_{i-1}, i) = \delta_i$ holds for the remaining case $a = a_{i-1}$. Using Definitions 4.2 and 4.22, we obtain $\gamma_i(x_a) = \mathsf{last}(w, a, i) = \bar{\gamma}_{i-1}(x_a) + \delta_i$. Similarly, we have $\mathsf{next}(w, a, i) = \mathsf{next}(w, a, i-1) - \delta_i$ and hence $\gamma_i(y_a) = \mathsf{next}(w, a, i) = \bar{\gamma}_{i-1}(y_a) + \delta_i$ for $a \neq a_i$. The only remaining case is resolved by the reset of y_{a_i}.

If $\gamma_i \models \psi$ holds, the transition is enabled and we compute the next pair $(q_{i+1}, \bar{\gamma}_i)$ with

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

which follows directly from Definitions 4.2 and 4.22. Thus, we obtain from Equation (3) by expanding γ_i following Equation (2)

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

for the incremental computation of $\bar{\gamma}_i$. Equation (4) refers to future events beyond (a_i, t_i) only in terms of $\mathsf{next}(w, a_i, i)$. Hence, based on Equations (2) and (4), we define incremental transitions as follows.

Definition 4.23 (*Incremental Transition*). For an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$, two states $q, q' \in Q$, and two incremental clock valuation functions $\bar{\gamma}, \bar{\gamma}' \in V_\Sigma$, we write for the incremental transition from $(q, \bar{\gamma})$ to $(q', \bar{\gamma}')$

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

iff there exits a transition $e = (q, a, \psi, q') \in E$ and a time value $v \in T_\perp$ such that $\bar{\gamma} + \delta$ is defined (see the discussion after Remark 4.9), 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.

Since the sequence of incremental clock valuation functions starts with $\bar{\gamma}_{-1}$ the indices in incremental clock valuation functions $\bar{\gamma}_i$ are decremented by one as compared to their standard counterpart γ_i. The reason for not harmonising the indices is that γ_0 already incorporates information from the fist timed event (a_0, t_0), while $\bar{\gamma}_{i-1}$ truly precedes the first timed event (a_0, t_0). Thus, Equation (1) describes the transition from γ_i to γ_{i+1} on processing (a_i, t_i) (involving knowledge on the future event (a_{i+1}, t_{i+1})) while Equation (4) describes the transition from $\bar{\gamma}_{i-1}$ to $\bar{\gamma}_i$ on processing the same event (and only referring to the past time stamp t_{i-1}).

Timed runs pair each state q_i with the clock valuation function γ_i. Corresponding to each timed run, we define an *incremental timed run* which combines each state q_i with an incremental clock valuation function $\bar{\gamma}_{i-1}$. By using the *delayed* valuation function $\bar{\gamma}_{i-1}$, each pair $(q_i, \bar{\gamma}_{i-1})$ contains all information we can obtain immediately before processing (a_i, t_i).

Definition 4.24 (*Incremental Timed Run*). An incremental timed run $\bar{\theta}$ 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 $(q_0, \bar{\gamma}_{-1})(q_1, \bar{\gamma}_0) \ldots$ of state and incremental clock valuation pairs such that

— q_0 is an initial state, that is, $q_0 \in Q_0$,
— each $\bar{\gamma}_i$ adheres Definition 4.22 for $i \geq 0$ and $\bar{\gamma}_{-1}$ is initial, and
— $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{(a_i,\delta_i)} (q_{i+1}, \bar{\gamma}_i)$ for $\delta_0 = t_0$ and $\delta_i = t_i - t_{i-1}$ for $i > 0$.

PROPOSITION 4.25 (INCREMENTAL VERSUS STANDARD TIMED RUNS). *Let* $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ *be an automaton and* $w = (a_0, t_0)(a_1, t_1)\cdots \in T\Sigma^\omega$ *be a timed word. Then there exists a timed run* $\theta = (q_0, \gamma_0)(q_1, \gamma_1)\ldots$ *over* w *iff there exists a corresponding incremental timed run* $\bar{\theta} = (q_0, \bar{\gamma}_{-1})(q_1, \bar{\gamma}_0)\ldots$ *over* w.

PROOF. The proof has two parts: We first assume that θ exists and show that there exists a transition between each two subsequent pairs $(q_i, \bar{\gamma}_{i-1})$ and $(q_{i+1}, \bar{\gamma}_i)$ in $\bar{\theta}$. Since the initial pair $(q_0, \bar{\gamma}_{-1})$ is fixed, this proves already that $\bar{\theta}$ exists. Then we show the converse in the same manner, i.e., we show that for each two subsequent pairs (q_i, γ_i) and $(q_{i+1}, \bar{\gamma}_{i+1})$ there exists a corresponding enabled transition.

Assume that there exists a timed run $\theta = (q_0, \gamma_0)(q_1, \gamma_1)\ldots$ over w and consider a pair $(q_i, \bar{\gamma}_{i-1})$. We show that $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{(a_i,\delta_i)} (q_{i+1}, \bar{\gamma}_i)$ must hold: By Definition 4.17, we know that there must exist a transition $e = (q_i, a_i, \psi, q_{i+1}) \in E$ with $\gamma_i \models \psi$. Since $\gamma_i = (\bar{\gamma}_{i-1} + \delta_i)[y_{a_i} = \mathsf{next}(w, a_i, i)]$ (Equation (2)), we find that $(\bar{\gamma}_{i-1} + \delta_i)[y_{a_i} = v] \models \psi$ must hold for $v = \mathsf{next}(w, a_i, i)]$, matching Definition 4.23. Furthermore, the same definition requires $\bar{\gamma}_i = (\bar{\gamma}_{i-1} + \delta_i)[y_{a_i} = v][x_{a_i} = 0]$ which holds because of Equation (4).

Conversely, assume that there is an incremental timed run $\bar{\theta} = (q_0, \bar{\gamma}_{-1})(q_1, \bar{\gamma}_0)\ldots$ over w and consider an arbitrary pair (q_i, γ_i) from the timed run. We show that there is an enabled transition e leading from (q_i, γ_i) to (q_{i+1}, γ_{i+1}): From Definition 4.23, we know that $(\bar{\gamma}_{i-1} + \delta_i)[y_{a_i} = v] \models \psi$ must hold for some transition $e = (q_i, a_i, \psi, q_{i+1}) \in E$ and some time value v with $\bar{\gamma}_i = (\bar{\gamma}_{i-1} + \delta_i)[y_{a_i} = v][x_{a_i} = 0]$. By Equation (4), we find that $v = \mathsf{next}(w, a_i, i)$ must hold. But then $(\bar{\gamma}_{i-1} + \delta_i)[y_{a_i} = v] \models \psi$ and Equation (2) lead immediately to $\gamma_i \models \psi$. Hence the transition is e is enabled in (q, γ_i). □

To simplify notation, we expand the incremental 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 word $u = (a_0, t_0)\ldots(a_{k-1}, t_{k-1}) \in T\Sigma^k$ if there exists an incremental timed run over u that starts in $(q_i, \bar{\gamma}_{i-1})$ (instead of $(q_0, \bar{\gamma}_{-1})$) and ends in $(q_{i+k}, \bar{\gamma}_{i+k-1})$. 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 $\bar{\gamma}_{i-1}$, that is, the evaluations of the event-predicting clocks must match the occurring events in u.

In case of an infinite word $\sigma = (a_0, t_0)\cdots \in T\Sigma^\omega$, we write $(q_i, \bar{\gamma}_{i-1}) \xrightarrow{\sigma}$ if there exists an incremental timed run over σ which starts in $(q_i, \bar{\gamma}_{i-1})$. If the sequence of states $q_i, q_{i+1}\ldots$ 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 4.26 (*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 $\bar{\Gamma}_{-1}, \bar{\Gamma}_0, \ldots$

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 9. 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

```
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 ∈ V_Σ 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, 0]] ;
return (q_{i+1}, Γ̄_i) ;
end
```

Fig. 9. Procedure symb_step$((q_i, \bar{\Gamma}_{i-1}), \delta, e)$.

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.

Note that step 3) of symb_step requires that there exists a $\gamma \in V_\Sigma$ with $\gamma \models \Gamma_i \wedge \psi$. Since each $\gamma \in V_\Sigma$ must satisfy noncoincidence and continuity, see Proposition 4.4, this condition ensures that $\Gamma_i \wedge \psi$ satisfies these three properties as well. If one requires timed words to satisfy monotonicity instead of strict monotonicity, then noncoincidence can be dropped—requiring continuity only (see Remark 4.5).

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

LEMMA 4.27 (ABSTRACTING A TRANSITION). *Let $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^\omega$ be a timed word with the corresponding sequences of clock valuation functions $\gamma_0, \gamma_1, \ldots$ and incremental clock valuation functions $\bar{\gamma}_{-1}, \bar{\gamma}_0, \ldots$ 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 $\bar{\gamma}_{i-1} \models \bar{\Gamma}_{i-1}$, 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 that 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 $\bar{\gamma}_i \models \bar{\Gamma}_i$.

In the following, 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 4.10, 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 4.10, 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 4.10, $\bar{\gamma}_i = \gamma_i[x_{a_i} = 0] \models \Gamma'[x_{a_i} = 0] = \bar{\Gamma}_i$.

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 4.21 without modification, that is, we set $\bar{\Gamma}_{-1} = \Gamma_0$. Thus, we arrive at the following definition.

Definition 4.28 (*Symbolic Timed Run*). A *symbolic timed run* Θ of an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ over a given 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 4.20) for $i \geq 0$ and $\bar{\Gamma}_{-1}$ is the initial symbolic clock valuation ($\bar{\Gamma}_{-1} = \Gamma_0$ and following Definition 4.21).
— 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.

Example 4.29 (*Symbolic Timed Run*). Recall the property

$$\varphi = G\left(r_1 \rightarrow \left(\rhd_{ack} \in [0, 2) \wedge X\left(wU\left(r_2 \wedge \rhd_{ack} \in (1, \infty) \wedge X\left(wUack\right)\right)\right)\right)\right)$$

we introduced in Examples 4.13 and discussed subsequently. Observe that $[y_{ack} < 2]$ and $[y_{ack} > 1]$ are the only guards occurring in $\mathcal{A}^\varphi_{ec}$ (Figure 7) and that y_{ack} is the only clock evaluated therein. Thus, in the symbolic timed run Θ described below, we only consider the clock constraints for y_{ack}.

Given the finite timed word

$$u = (ack, 1), (ack, 1.5), (r_1, 2), (w, 2.1), (w, 3.2), (r_2, 3.3)$$

we construct the corresponding symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1}), \ldots (q_6, \bar{\Gamma}_5)$. This run Θ is uniquely identified since $\mathcal{A}_{ec}^{\varphi}$ is deterministic: The initial state of Θ is the pair $(q_a, \bar{\Gamma}_{-1})$ with $\bar{\Gamma}_{-1}(y_{ack}) = \mathsf{undef}$. On processing $(ack, 1)$ and $(ack, 1.5)$, we remain in the same state, i.e., $q_2 = q_1 = q_a$ and $\bar{\Gamma}_1(y_a) = \bar{\Gamma}_0(y_a) = \bar{\Gamma}_{-1}(y_a) = \mathsf{undef}$. When we receive $(r_1, 2)$, we reach $q_3 = q_b$. The transition to q_b is guarded with $[y_{ack} < 2]$. Hence we have to add the guard to the clock constraint and obtain $\bar{\Gamma}_2(y_a) = [0, 2)$. The event $(w, 2.1)$ does not change the state, that is, $q_4 = q_b$, nor does it require to incorporate another guard. However, time passes by with $\delta = t_3 - t_2 = 2.1 - 2 = 0.1$ time units and hence we have $\bar{\Gamma}_3(y_{ack}) = [0, 1.9)$. The same holds true for the next event $(w, 3.2)$, leading to $q_5 = q_b$ and $\bar{\Gamma}_4(y_{ack}) = [0, 0.8)$ after $\delta = t_4 - t_3 = 1.1$ time units.

Finally, an attempt to process $(r_2, 3.3)$ fails: In step 1) of symb_step (Figure 9) we obtain $\bar{\Gamma}'_4 = \bar{\Gamma}_4 + \delta$ with $\delta = t_5 - t_4 = 0.1$, that is, $\bar{\Gamma}'_4(y_{ack}) = [0, 0.7)$. Step 2) goes through as well, since $\bar{\Gamma}'_4(y_{r_2}) = \mathsf{undef}$, but step 3) causes a constraint violation: The guard on the transition from q_b to q_c (which is the only alternative on processing r_2) is guarded with $[y_{ack} > 1]$. Because Γ_5 is the conjunction of $\bar{\Gamma}'_4$ and the current guard, we obtain $\Gamma_5(y_{r_2}) = \bar{\Gamma}'_4(y_{r_2}) \cap (1, \infty) = [0, 0.7) \cap (1, \infty) = \emptyset$—which cannot be satisfied anymore and is therefore a constraint violation.

Summarized, the symbolic timed run Θ is determined as

$$\Theta = (q_a, \bar{\Gamma}_{-1}), (q_a, \bar{\Gamma}_{-1}), (q_a, \bar{\Gamma}_{-1}), (q_b, \bar{\Gamma}_2), (q_b, \bar{\Gamma}_3), (q_b, \bar{\Gamma}_4)$$

with $\bar{\Gamma}_2(y_{ack}) = [0, 2)$, $\bar{\Gamma}_3(y_{ack}) = [0, 1.9)$, and with $\bar{\Gamma}_4(y_{ack}) = [0, 0.8)$ before aborting with a **constraint_violation**.

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 4.30 (*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 4.27 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 4.31 (ABSTRACTING TIMED RUNS). *Let $\mathcal{A}_{ec}$ be an event-clock automaton with $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ 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 4.2) as well as a unique sequence $\bar{\gamma}_{-1}, \bar{\gamma}_0, \ldots$ of incremental clock valuation functions (Definition 4.22).

To construct Θ, we first set $\bar{\Gamma}_{-1} = \Gamma_0$ following Definition 4.28 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 4.27 is satisfied: $\bar{\gamma}_{-1} \models \bar{\Gamma}_{-1}$ holds and e_0 is an enabled transition. Consequently, Lemma 4.27 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 4.27 is satisfied and we obtain the the required symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_0) \dots$ inductively. □

At this point, we are tempted to show the converse of Lemma 4.31, that is, 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 4.32. *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 $\Theta = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_0) \dots$*

such that there exists no ordinary timed run $\theta = (q_0, \gamma_0)(q_1, \gamma_1) \dots$ over w which is based upon the same sequence of states $q_0, q_1 \dots$ 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 4.33. We finally prove in Lemma 4.34 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 concretize the run with a backward simulation. Then we show in Theorem 4.35 how to expand this prefix of a timed run into a suitable infinite timed run.

LEMMA 4.33 (CONCRETISING A SYMBOLIC TRANSITION). *If a symbolic transition $(q_i, \bar{\Gamma}_{i-1}) \xrightarrow{(a_i, \delta)} (q_{i+1}, \bar{\Gamma}_i)$ holds 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 (see Definition 4.23).*

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

For what follows, recall that all symbolic clock valuations $\bar{\Gamma}$ (Definition 4.20) evaluate all event-recording clocks x_a to intervals of the form $[l, l]$ for some value $l \in T_\bot$—precluding the cases $\bar{\Gamma}(x_a) = \mathsf{undef}$ and $\bar{\Gamma}(x_a) = [l, r]$ with $l < r$.

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 $\bar{\Gamma}_{i-1}(x_a) = [l, l]$ with $l \neq \bot$, 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 9) $0 \in (\bar{\Gamma}_{i-1}(y_{a_i}) + \delta)$ 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 $\bar{\gamma}_{i-1}(y_a) \in [(l \dot{-} \delta + \delta, r - \delta + \delta)] \subseteq [(l, r)] = \bar{\Gamma}_{i-1}(y_a)$.
— If $\bar{\Gamma}_{i-1}(y_a) = \mathsf{undef}$, then y_a remains unconstrained and cannot cause a constraint violation.

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. This implies that 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 $y_{a_i} = \mathsf{next}(\omega, a_i, i) = \bar{\gamma}_i(y_{a_i})$). 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, $\bar{\Gamma}_i(c) \subseteq \psi(c)$ holds and we therefore arrive at $\bar{\gamma}_i \not\models \bar{\Gamma}_i$ which contradicts the lemma statement.

□

Recall Equation (2) which allows us to compute γ_i from $\bar{\gamma}_{i-1}$. Below we need to compute γ_{i-1} from γ_i. Thus, we invert Equation (2) by first subtracting δ_i and then restoring the original value of y_{a_i} with $\mathsf{next}(\omega, a_i, i-1) = \delta_i$:

$$\bar{\gamma}_{i-1} = (\gamma_i - \delta_i)[y_{a_i} = \delta_i]. \tag{5}$$

LEMMA 4.34 (CONCRETIZING A FINITE SYMBOLIC RUNS). *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 for every finite symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1}) \dots (q_{i+1}, \bar{\Gamma}_i)$ and for every clock valuation function γ_{i+1} (given together with the corresponding timed event (a_{i+1}, t_{i+1})) satisfying $\bar{\gamma}_i \models \bar{\Gamma}_i$, there is a finite timed run $\theta = (q_0, \gamma_0) \dots (q_{i+1}, \gamma_{i+1})$ (adhering Definition 4.17) such that $\bar{\gamma}_j \models \bar{\Gamma}_j$ (as determined by Equation (5)) holds for all $-1 \leq j \leq i$.

PROOF. From (a_{i+1}, t_{i+1}) and γ_{i+1}, we can compute with Equation (5) the incremental clock valuation function $\bar{\gamma}_i$ as starting point for a backward simulation.

Since $\bar{\gamma}_i \models \bar{\Gamma}_i$ and $(q_i, \bar{\Gamma}_{i-1}) \xrightarrow{(a_i, \delta_i)} (q_{i+1}, \bar{\Gamma}_i)$ hold, we apply Lemma 4.33 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 4.33 inductively, we obtain $(q_0, \bar{\gamma}_{-1}) \dots (q_{i+1}, \bar{\gamma}_i)$.

The run $\bar{\theta}$ is an incremental timed run (Definition 4.24). Thus, we can use Equation (2) to compute from $\bar{\theta}$ the timed run $\theta = (q_0, \gamma_0)\dots(q_{i+1}, \gamma_{i+1})$, following Proposition 4.25 to match Definition 4.17 as required. □

THEOREM 4.35 (SYMBOLIC SIMULATION). *Let $u = (a_0, t_0)\dots(a_i, t_i) \in T\Sigma^*$ be a finite timed word and let $\sigma = (a_{i+1}, t_{i+1})(a_{i+2}, t_{i+2})\dots \in T\Sigma^\omega$ be an infinite continuation of u.*

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

(a) a finite symbolic timed run $\Theta = (q_0, \bar{\Gamma}_{-1})\dots(q_{i+1}, \bar{\Gamma}_i)$ over u,
(b) an infinite timed run $\theta = (q_{i+1}, \gamma_{i+1})(q_{i+2}, \gamma_{i+2})\dots$ 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 $v \in T_\perp$.

PROOF. Assume $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$ holds. Then there exists an accepting timed run $\theta' = (q_0, \gamma_0)(q_1, \gamma_1)\dots$ over $u\sigma$. Thus, by Lemma 4.31, there exists a symbolic timed run $\Theta' = (q_0, \bar{\Gamma}_{-1})(q_1, \bar{\Gamma}_1)\dots$ over $u\sigma$ with $\bar{\gamma}_i \models \bar{\Gamma}_i$ for all $i > 0$ as well. We take the prefix $\Theta = (q_0, \bar{\Gamma}_{-1})\dots(q_{i+1}, \bar{\Gamma}_i)$ of Θ' and the suffix $\theta = (q_{i+1}, \gamma_{i+1})(q_{i+2}, \gamma_{i+2})\dots$ of θ' to meet conditions (a) and (b) of the lemma statement, respectively. Condition (c) is satisfied since Lemma 4.31 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 $\theta' = (q_0, \gamma_0)(q_1, \gamma_1)\dots$ over $u\sigma$ to show $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$. To do so, we take the timed run $\theta = (q_{i+1}, \gamma_{i+1})(q_{i+2}, \gamma_{i+2})\dots$ over σ as suffix in θ'. We complete θ' with the prefix $(q_0, \gamma_0)\dots(q_{i+1}, \gamma_{i+1})$ according to Lemma 4.34 using u, Θ, γ_{i+1}, and (a_{i+1}, t_{i+1}), which completes the proof as well. □

Rereading the statement of Theorem 4.35 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 nonempty, that is, which has a concretization $(q_{i+1}, \bar{\gamma}_i)$ with a nonempty continuation language. This is exactly the statement of Corollary 4.36.

COROLLARY 4.36 (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 4.35 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 4.35 to find $u\sigma \in \mathcal{L}(\mathcal{A}_{ec})$. □

In both, Theorem 4.35 and its Corollary 4.36, 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 continuous solution (see Definition 4.8). To ensure $\bar{\gamma}_i \in V_\Sigma$, we need

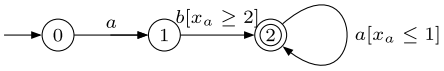


Fig. 10. Event-clock automaton.

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 4.4.

On the other hand, if one wants to relax the strict monotonicity requirement on timed words to simple monotonicity, one needs to drop the non-coincidence check on all checks for some $\bar{\gamma}_i \in V_\Sigma$, cf. Remark 4.5.

4.5 Emptiness Check for Symbolic States

Taking Corollary 4.36 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, as exemplified by the automaton shown in Figure 10. 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* [Alur and Dill 1994]. 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 $[\bar{\gamma}]_{\approx_R} = \{\bar{\gamma}' \in V_\Sigma \mid \bar{\gamma}' \approx_R \bar{\gamma}\}$ 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. Note that the second condition renders zone-based approaches inapplicable in our setting.

In the following, 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 [Alur et al. 1999], 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 4.37 (*Quotient Automaton, following [Alur et al. 1999]*).[6]
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

— $Q/_{\approx}$ as the set of states with $Q/_{\approx} = \{(q, [\bar{\gamma}]_{\approx}) \mid q \in Q \text{ and } \bar{\gamma} \in V_\Sigma\}$,
— $Q_0/_{\approx}$ as the set of initial states $\{(q, [\bar{\gamma}_{-1}]_{\approx}) \mid q \in Q_0 \text{ and } \bar{\gamma}_{-1} \in V_\Sigma \text{ being initial}\}$,
— $E/_{\approx}$ 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 2.3 and the subsequent discussion), where we use $F/_{\approx} = \{F_i/_{\approx} \mid F_i \in F\}$ for $F_i/_{\approx} = \{(q, [\bar{\gamma}]_{\approx}) \mid q \in F_i \text{ and } \bar{\gamma} \in V_\Sigma\}$.

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 [Schwoon and Esparza 2005].

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 4.38 (*Time-Abstract Bisimulation [Tripakis and Yovine 2001]*). 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 $\mathsf{ut}(\mathcal{L}(\mathcal{A}_{ec}))$ [Alur et al. 1999; Tripakis and Yovine 2001]. 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 $(q, [\bar{\gamma}]_{\approx})$, that is, 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 4.39 (EMPTINESS CHECK WITH BISIMULATION [ALUR AND DILL 1994]). *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 $\mathcal{L}(\mathcal{A}_{ec}/_{\approx}(q, [\bar{\gamma}]_{\approx})) = \emptyset$.*

[6]The automata we define here as quotient automata are denoted by region automata $Reg_{\cong}(A)$ in Alur et al. [1999]. More precisely, in Alur et al. [1999], 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 Alur et al. [1999] and our own definition yield the same automata.

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

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

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 4.40 (EMPTINESS CHECK FOR SYMBOLIC RUNS). *Assume that* $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ *is 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

$$\mathsf{empty}_{\mathcal{A}_{ec}}(q, \bar{\Gamma}) = \bigwedge_{[\bar{\gamma}]_\approx \in \mathsf{cover}_\approx(\bar{\Gamma})} T[q, [\bar{\gamma}]_\approx] \,. \tag{6}$$

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 [Alur and Dill 1994] $\approx_R$ which is a time-abstract bisimulation with finite index and to show how to compute $\mathsf{cover}_{\approx_R}(\bar{\Gamma})$ for $\approx_R$.

In the following, 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 $\lfloor \bar{\gamma}(y_a) \rfloor$).

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 [Alur and Dill 1994]:

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

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

where K_c is the largest integer compared with the clock c,

— 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 that 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 4.40 and which uses the scheme previously described for computing $\mathsf{cover}_{\approx}(\bar{\Gamma})$.

Example 4.41 (*Region Automaton*). Reconsider the event-clock automaton $\mathcal{A}^{\varphi}_{ec}$ for the property

$$\varphi = G\left(r_1 \rightarrow \left(\triangleright_{ack} \in [0, 2) \wedge X\left(w U \left(r_2 \wedge \triangleright_{ack} \in (1, \infty) \wedge X\left(w U ack\right)\right)\right)\right)\right)$$

shown in Figure 7 and discussed in Example 4.19. We continue this example and construct the corresponding region automaton $\mathcal{A}^{\varphi}_{ec}/_{\approx_R}$ depicted in Figure 11. For the sake of brevity, we will write $\approx$ instead of $\approx_R$.

As the event-clock automaton $\mathcal{A}^{\varphi}_{ec}$ only deals with a single predicting clock y_{ack}, we consider the region automaton likewise for regions determined only by y_{ack}, which are $\bot, 0, (0, 1), 1, (1, 2), 2, > 2$. Thus, the region automaton $\mathcal{A}^{\varphi}_{ec}/_{\approx}$ has 21 states, namely one for each of these seven regions and for each of the individual states q_a, q_b, and q_c of the underlying event-clock automaton $\mathcal{A}^{\varphi}_{ec}$. In Figure 11, we show all these states, leaving all unreachable states grayed out. The automaton shows for example that reading request r_1 and predicting the acknowledgement to be within 1 second will not allow to read a subsequent request r_2 event.

Example 4.42 (*Emptiness Check*). We continue Example 4.41 and compute the table $T[q, [\bar{\gamma}]_{\approx_R}]$ as preparation for the emptiness check according to Equation (6). Observe that an accepting run must pass one of the q_a-based states infinitely often. This is only possible from $(q_a, [\bar{\gamma}]_{\approx_R})$ with $[\bar{\gamma}]_{\approx_R} \neq \bot$, from $(q_b, (1, 2))$, and from $(q_c, 0), \ldots, (q_c, (1, 2))$. For these cases, $T[q, [\bar{\gamma}]_{\approx_R}]$ evaluates to *false* (since the continuation language is not empty) and to *true* otherwise.

Recall from Example 4.29 the trace

$$u = (ack, 1), (ack, 1.5), (r_1, 2), (w, 2.1), (w, 3.2), (r_2, 3.3)$$

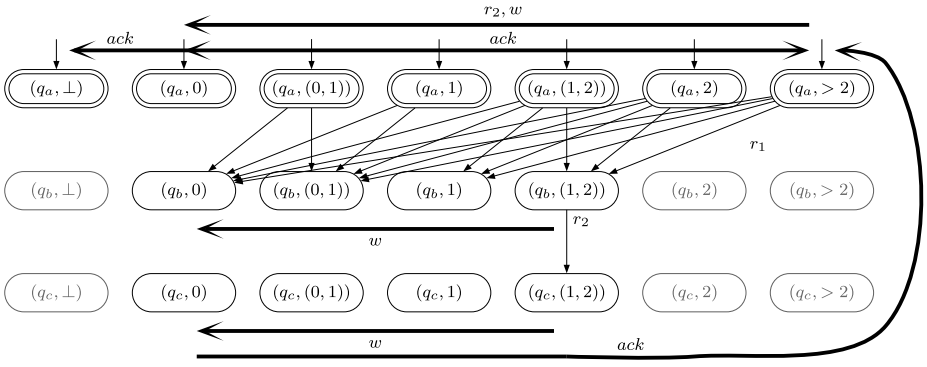


Fig. 11. Region automaton for φ from Example 4.41.

and the corresponding symbolic run

$$\Theta = (q_a, \bar{\Gamma}_{-1}), (q_a, \bar{\Gamma}_{-1}), (q_a, \bar{\Gamma}_{-1}), (q_b, \bar{\Gamma}_2), (q_b, \bar{\Gamma}_3), (q_b, \bar{\Gamma}_4), \textbf{constraint\_violation}$$

with $\bar{\Gamma}_2(y_{ack}) = [0, 2)$, $\bar{\Gamma}_3(y_{ack}) = [0, 1.9)$, and with $\bar{\Gamma}_4(y_{ack}) = [0, 0.8)$: After reading the third event $(r_1, 2)$, we reach $(q_b, \bar{\Gamma}_2)$.

To perform the emptiness check at this state, we observe $\bar{\Gamma}_2(y_{ack}) = [0, 2)$ and compute $\mathsf{cover}_{\approx}(\bar{\Gamma}_2) = \{0, (0, 1), 1, (1, 2)\}$, that is, $(q_b, \bar{\Gamma}_2)$ represents all states in the second row of Figure 11. While we have $T[q_b, [\bar{\gamma}]_{\approx}] = \mathit{true}$ for $[\bar{\gamma}]_{\approx} = \{0, (0, 1), 1\}$, we find $T[q_b, (1, 2)] = \mathit{false}$, and hence $\mathsf{empty}_{\mathcal{A}^{\varphi}_{ec}}(q_b, \bar{\Gamma}_2)$ evaluates to false, i.e., the set of potential satisfying continuations is nonempty. The fourth event $(w, 2.1)$ brings us to $(q_b, \bar{\Gamma}_3)$ with $\bar{\Gamma}_3(y_{ack}) = [0, 1.9)$ and $\mathsf{cover}_{\approx}(\bar{\Gamma}_3) = \mathsf{cover}_{\approx}(\bar{\Gamma}_2)$. Note that the region $(1, 2)$ is not contained in $[0, 1.9)$ but is still necessary to cover $[0, 1.9)$. Again, the so far read trace is potentially continued into an accepted word.

But upon reading the next event $(w, 3.2)$, we reach $(q_b, \bar{\Gamma}_4)$ with $\bar{\Gamma}_4(y_{ack}) = [0, 0.8)$ and $\mathsf{cover}_{\approx}(\bar{\Gamma}_4) = \{0, (0, 1)\}$. Since $T[q_b, 0] = T[q_b, (0, 1)] = \mathit{true}$ we obtain $\mathsf{empty}_{\mathcal{A}^{\varphi}_{ec}}(q_b, \bar{\Gamma}_4) = \mathit{true}$, that is, the so far read trace cannot be continued into an accepted word anymore.

This case demonstrates that the emptiness check anticipates indirect constraint violations before $\mathsf{symb\_step}$ is able to detect these violations; recall Example 4.29, where the constraint violation became obvious only one event later.

4.6 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.

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 12, 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 4.4 and 4.5: $\mathsf{monitor}_{\mathcal{A}_{ec}}$ 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 4.36 taken from Section 4.4. The runtime

```
procedure monitor_{A_ec}(a, δ)
begin
{ ---------------------------------------- }
{ step 1: initialisation (first call only) }
if first_time then
   P := {(q, Γ̄) | q ∈ Q_0 ∧ Γ̄ is initial} ;

{ ---------------------------------------- }
{ step 2: symbolic step }
P' := {(q', Γ̄') |   (q, Γ̄) ∈ P
                  ∧ e = (q, a, ψ, q') ∈ E
                  ∧ (q', Γ̄') = symb_step((q, Γ̄), δ, e) } ;
P := P' ;

{ ---------------------------------------- }
{ step 3: emptiness check }
if ⋀_{(q,Γ̄)∈P} empty_{A_ec}(q, Γ̄) then
   return ⊥
else
   return ?
end
```

Fig. 12. Procedure $\mathsf{monitor}_{\mathcal{A}_{ec}}(a, \delta)$.

```
procedure monitor_φ(a, δ)
begin
{ ---------------------------------------- }
{ step 1: symbolic step }
r_φ := monitor_{A^φ_ec}(a, δ) ;
r_¬φ := monitor_{A^¬φ_ec}(a, δ) ;

{ ---------------------------------------- }
{ step 2: compute verdict }
if r_φ = ⊥ then return ⊥ ;
if r_¬φ = ⊥ then return ⊤ ;
return ? { note: r_φ = r_¬φ = ? }
end
```

Fig. 13. Procedure $\mathsf{monitor}_{\varphi}(a, \delta)$.

verification criterion involves an emptiness check for symbolic timed runs, which is in turn implemented according to Corollary 4.40 taken from Section 4.5.

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 13, 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.

Example 4.43. We briefly consider the monitor for

$$\varphi = G\left(r_1 \rightarrow \left(\rhd_{ack} \in [0, 2) \wedge X\left(w U\left(r_2 \wedge \rhd_{ack} \in (1, \infty) \wedge X\left(w U ack\right)\right)\right)\right)\right)$$

As discussed in Example 4.15, φ may always be violated at some future point in time. Hence, the monitoring result for φ will always be either $\bot$ or ?. The corresponding construction of $\mathsf{monitor}_{\mathcal{A}_{ec}^{\neg\varphi}}(\cdot,\cdot)$ will always return ?—and there is no need to actually call it in procedure $\mathsf{monitor}_{\varphi}(a,\delta)$. On the other hand, $\mathsf{monitor}_{\mathcal{A}_{ec}^{\varphi}}(a,\delta)$ is built from the event-clock automaton discussed in Example 4.19 and the table for the emptiness check as given in Example 4.42.

So far, we cannot provide measurements for timed monitors as in the discrete-time case. However, we tend to generalize the previous examples as follows.

— Similar as for LTL (as witnessed by Dwyer's collection of properties), we expect practically relevant TLTL formulae to be short and range over a small set of clocks.
— Similar as for LTL, we expect the resulting event-clock automata only rarely to be exponentially larger than the underlying formula.
— Regarding the emptiness check, note that it is only of interest, whenever time-constraints are potentially overlapping. For the formulae given in Example 4.13, this is only the case for the last one. Moreover, we expect, as in Example 4.42, the region automata to have a simple structure. Thus, using representations like difference bound matrices (DBMs, [Dill 1989]) or discrete representations as for example the ones introduced in Gollu et al. [1994] and Grinchtein and Leucker [2008] together with BDDs [Bryant 1985] should allow for compact representations.

Thus, while practical experience has still to be gained for the continuous-time case, we expect our method to work well for typical examples.

4.7 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, that is, 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 4.40 remains unaffected as well.

Summarized, 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, that is, 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 behavior 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 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.

5. CONCLUSIONS

In this article, 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.

Runtime verification deals with finite runs and asks a suitable LTL semantics on finite traces: In our understanding of runtime verification, we consider a finite trace as an incrementally observed finite prefix of an unknown infinite trace. The uncertainty of continuation of the trace may then cause a correctness property to evaluate to either *true*, *false* or *inconclusive*, depending on the observered prefix and the verified property. We proposed the three-valued logic LTL_3, whose semantics is designed correspondingly.

For formulae of this logic, 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 cosafety properties.

We examplified our methodolgy using Dwyer's specification patterns, for which we run through the monitor generation. As the resulting deterministic monitors are of reasonable sizes (mostly less than 100 states and transitions), our approach proves to be practically feasible.

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. Using specifically constructed 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. 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.

APPENDIX

A. EVALUATION DATA

This appendix provides further details and the exact data we used in our evaluation of the synthesized monitors presented in Section 2.5. The following table lists 108 formulae taken from Dwyer et al.'s comprehensive collection of temporal logic specifications that were tagged as LTL, and from which 97 corresponded to well-formed LTL formulae. For each formula φ, we generated a deterministic monitor $\mathcal{M}_\varphi$ via first constructing a Büchi automaton for φ, that is, $\mathcal{A}_\varphi$, and for $\neg\varphi$, that is, $\mathcal{A}_{\neg\varphi}$. The following table lists each such formula φ, its size, and the sizes of $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$. For $\mathcal{M}_\varphi$, we list the number of states emitting $\top$, $\bot$, and ?, denoted by #$\top$, #$\bot$, and, respectively, #?. Moreover, we list the size of the resulting monitor. The number of states of an automaton $\mathcal{M}$ is listed as $|\mathcal{M}|_s$, while the number of states plus transitions is denoted by $|\mathcal{M}|$. The monitorable properties are numbered N1–N54 and depicted in Figures 4 and 5.

$\lvert\varphi\rvert$	$\lvert\mathcal{A}_\varphi\rvert_s$	$\lvert\mathcal{A}_\varphi\rvert$	$\lvert\mathcal{A}_{\neg\varphi}\rvert_s$	$\lvert\mathcal{A}_{\neg\varphi}\rvert$	$\lvert\mathcal{A}_\varphi\rvert+\lvert\mathcal{A}_{\neg\varphi}\rvert$	#$\top$	#$\bot$	#?	$\lvert\mathcal{M}_\varphi\rvert_s$	$\lvert\mathcal{M}_\varphi\rvert$	monitorable?
PATTERN: Constrained Response-chain 2-1											
LTL: []((call_Enqueue(d1) & (!return_Dequeue(d1) U call_Top_Down) -> <>(call_Top_Down & <>call_P(d1,*)))											
13	9	153	4	68	221	0	0	1	1	17	non monitorable
PATTERN: Constrained 3-2 Response Chain											
LTL: []((call_Enqueue(d1) & (!return_Dequeue(d1) U call_Bottom_Up) -> <>(call_Bottom_Up & <>call_P(d1,*)))											
13	9	153	4	68	221	0	0	1	1	17	non monitorable
PATTERN: Constrained 3-2 Response Chain											
LTL: []((call_Enqueue(d1) & (!return_Dequeue(d1) U (call_Enqueue(d2) & (!return_Dequeue(d1) & !return_Dequeue(d2) U call_Top_Down)))) -> <>(call_Top_Down & <>(call_P(d1) & <>call_P(d2))))											
24	44	5676	9	1161	6837	0	0	1	1	129	non monitorable
PATTERN: Constrained 2-1 Response Chain											
LTL: []((call_Enqueue & (!return_Dequeue U call_Empty) -> <>(call_Empty & <>return_Empty(false)))											
13	9	153	4	68	221	0	0	1	1	17	non monitorable
PATTERN: Existence											
LTL: []((call_Enqueue(d1) & <>return_Empty(true)) -> (!return_Empty(true) U return_Dequeue(d1)))											
10	4	36	3	27	63	0	1	2	3	27	monitorable—**N1**
PATTERN: 2 Bounded Existence											
LTL: []((call & <>open) -> ((!atfloor & !open) U (open | ((atfloor & !open) U (open | ((!atfloor & !open) U (open | ((atfloor & !open) U (open | (!atfloor U open))))))))))											
40	16	144	8	72	216	0	0	1	1	9	non monitorable
PATTERN: Response											

LTL: []((OpenNetworkConnection -> []((NetworkError -> <>ErrorMessage))											
8	3	27	3	27	54	0	0	1	1	9	non monitorable
PATTERN: Existence											
LTL: [](PopServerConnected & <>PlacedinMailboxes) -> (!PlacedinMailboxes U MessagesTransfered))											
10	4	36	3	27	63	0	1	2	3	27	monitorable—**N2**
PATTERN: Existence											
LTL: <>QueuedMailSent -> (!QueuedMailSent U SMTPServerConnected)											
7	4	20	2	10	30	1	1	1	3	15	monitorable—**N3**
PATTERN: Existence											
LTL: [](!MailboxSelected) | <>(MailboxSelected & <>MailboxWindowOpen))											
9	5	25	2	10	35	0	0	1	1	5	non monitorable
PATTERN: Existence											
LTL: [](!MessageReady) | <>(MessageReady & <>MessageinOutbox))											
9	5	25	2	10	35	0	0	1	1	5	non monitorable
PATTERN: Existence											
LTL: [](!MessageDragged) | <>(MessageDragged & <>MessageMoved))											
9	5	25	2	10	35	0	0	1	1	5	non monitorable
PATTERN: Response											
LTL: [](Error -> [](ErrorPopup -> <>ResponsetoPopup))											
8	3	27	3	27	54	0	0	1	1	9	non monitorable
PATTERN: Response											
LTL: [](MessageTransferred -> [](PlacedinMailboxes -> <>MarkasUnread))											
8	3	27	3	27	54	0	0	1	1	9	non monitorable
PATTERN: Existence											
LTL: [](!POPServerMessageDelete) | <>(POPServerMessageDelete & <>PlacedinMailboxes)											
9	5	25	3	15	40	1	0	2	3	15	monitorable—**N4**
PATTERN: Absence											
LTL: (<>call_Execute) -> ((!call_doWork) U call_Execute)											
7	4	20	3	15	35	1	1	2	4	20	monitorable—**N5**
PATTERN: Absence											
LTL: [](return_Execute && (<>call_Execute)) -> ((!call_doWork) U call_Execute))											
10	4	36	4	36	72	0	1	3	4	36	monitorable—**N6**
PATTERN: Absence											
LTL: [](call_doResults:i -> ((!call_doResults:j) U (return_doResults:i || [](!call_doResults:j))))											
11	3	27	3	27	54	0	1	2	3	27	monitorable—**N7**
PATTERN: Absence											
LTL: [](call_doResults:i && <>return_doResults:i) -> ((!call_doResults:j) U return_doResults:i))											
10	4	36	4	36	72	0	1	3	4	36	monitorable—**N8**
PATTERN: Response											
LTL: [](return_resultlock.wait:i -> <>call_resultlock.signal:i)											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Response											
LTL: [](call_doWork:i -> <>return_doWork:i)											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Filter=GlobalResponse-¿GlobalResponse											
LTL: [](call_doResult:i -> <>return_doResult:i) -> [](return_resultlock.wait:i -> <>call_resultlock.signal:i)											
11	5	85	4	68	153	0	0	1	1	17	non monitorable

PATTERN: Absence											
LTL: [](call_Execute -> ((!return_Execute) U (return_doWork(done==true):i || ... || return_doResult(done==true):i || ... || workCountEQzero || [](!return_Execute))))											
											not well formed LTL
PATTERN: Precedence											
LTL: ((! return_Execute) U (return_pool.Complete || [] (!return_Execute)))											
8	3	15	2	10	25	1	1	1	3	15	monitorable—**N9**
PATTERN: Precedence											
LTL: (!return_pool.Finished) U (return_doWork(done==true):i || ... || return_doResult(done==true):i || ... || [] (!return_pool.Finished))											
											not well formed LTL
PATTERN: Absence											
LTL: []!(call_Create:i || call_Destroy:i || call_Input:i || call_Execute:i || call_GetResults:i)											
11	1	33	2	66	99	0	1	1	2	66	monitorable—**N10**
PATTERN: Filter=GlobalAbsence+GlobalPrecedence											
LTL: ([]!(call_Create:i || call_Destroy:i || call_Input:i || call_Execute:i || call_GetResults:i)) -> ((!return_pool.Finished) U (return_doWork(done==true):i || ... || return_doResult(done==true):i || ... || [] (!return_pool.Finished)))											
											not well formed LTL
PATTERN: Response											
LTL: [](txB1==0 -> <>txB1==1); [](rxBufferSem1==0 -> <>rxBufferSem1==1); [](rxBCS1==0 -> <>rxBCS1==1);											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Universal											
LTL: []((state0 & !state1 & !state2 & !state3) | (!state0 & state1 & !state2 & !state3) | (!state0 & !state1 & state2 & !state3) | (!state0 & !state1 & !state2 & state3))											
44	1	17	2	34	51	0	1	1	2	34	monitorable—**N11**
PATTERN: Absence											
LTL: []((state0 & <>state1) -> !(state2 || state3) U state1) []((state3 & <>state2) -> !(state0 || state1) U state2)											
12	4	68	4	68	136	0	1	3	4	68	monitorable—**N12**
PATTERN: Absence											
LTL: []((state1 & <>(state0 || state2)) -> !state3 U (state0 || state2))											
14	4	68	4	68	136	0	1	3	4	68	monitorable—**N13**
PATTERN: Absence											
LTL: <>connect -> (!(disconnect || poke || send || blockingSend || receive || blockingReceive) U connect)											
17	4	516	3	387	903	1	1	2	4	516	monitorable—**N14**
PATTERN: Absence											
LTL: []((connect && <>disconnect) -> !connect U disconnect)											
10	2	10	3	15	25	0	1	2	3	15	monitorable—**N15**
PATTERN: Absence											
LTL: []((disconnect && <>connect) -> !(disconnect || poke || send || blockingSend || receive || blockingReceive) U connect)											
20	2	258	3	387	645	0	1	2	3	387	monitorable—**N16**
PATTERN: Universal											
LTL: [](blockingSendBeforeSendPlace -> !Connected)											
5	1	5	2	10	15	0	1	1	2	10	monitorable—**N17**
PATTERN: Response											

LTL: [](((register_a1_e1 && <> unregister_a1_e1) -> (after_notify_artists_e1 -> ((!(notify_artists_e1 || notify_artists_e2) && !unregister_a1_e1) U notify_client_event_a1_e1)) U unregister_a1_e1)											
19	8	520	5	325	845	0	1	4	5	325	monitorable—**N18**
PATTERN: Precedence											
LTL: !(notify_client_event_a1_e1 || notify_client_event_a2_e1) U (notify_artists_e1 || []!(notify_client_event_a1_e1 || notify_client_event_a2_e1))											
12	3	27	2	18	45	1	1	1	3	27	monitorable—**N19**
PATTERN: Absence											
LTL: [](notify_artists_e1 -> (!(notify_client_event_a1_e2 || notify_client_event_a2_e2) U (notify_artists_e2 || []!(notify_client_event_a1_e2 || notify_client_event_a2_e2))))											
15	3	51	3	51	102	0	1	2	3	51	monitorable—**N20**
PATTERN: Absence											
LTL: [](e1_szEQ0 -> (!(notify_client_event_a1_e1 || notify_client_event_a2_e1) U (e1_szGT0 || []!(notify_client_event_a1_e1 || notify_client_event_a2_e1))))											
15	3	51	3	51	102	0	1	2	3	51	monitorable—**N21**
PATTERN: Absence											
LTL: <>register_a1_e1 -> (!notify_event_a1_e1 U register_a1_e1)											
7	4	20	3	15	35	1	1	2	4	20	monitorable—**N22**
PATTERN: Absence											
LTL: [](unregister_a1_e1 -> (!notify_client_event_a1_e1 U (register_a1_e1 || []!notify_client_event_a1_e1)))											
11	3	27	3	27	54	0	1	2	3	27	monitorable—**N23**
PATTERN: Constrained Response-Chain (3-1)											
LTL: <>term -> [](((register_a1_e1 && (!unregister_a1_e1 U (register_a2_e1 && (!unregister_a1_e1 && !unregister_a2_e1) U notify_artists_e1))) -> <>(notify_artist_e1 & (!notify_client_event_a2_e1 U notify_client_event_a1_e1)))											
26	16	4112	19	4883	8995	0	0	1	1	257	non monitorable
PATTERN: Absence											
LTL: [](e1_szEQ2 & (after_register_a1_e1 | after_register_a2_e1) -> (!(register_a2_e1 || register_a1_e1)) U (e1_szLT2 || []!(register_a2_e1 || register_a1_e1)))											
19	3	387	6	774	1161	0	1	3	4	516	monitorable—**N24**
PATTERN: Absence											
LTL: <>register_a1_e1 -> (!unregister_a1_e1 U register_a1_e1)											
7	4	20	3	15	35	1	1	2	4	20	monitorable—**N25**
PATTERN: Absence											
LTL: [](after_unregister_a1_e1 -> (!unregister_a1_e1 U (register_a1_e1 || []!unregister_a1_e1)))											
11	3	27	3	27	54	0	1	2	3	27	monitorable—**N26**
PATTERN: Universal											
LTL: [](term -> (e1_szEQ0 && e2_szEQ0))											
6	1	9	2	18	27	0	1	1	2	18	monitorable—**N27**
PATTERN: Response											
LTL: (register_a1_e1 -> (!term U unregister_a1_e1)) U (term | [](!term))											
12	7	63	3	27	90	1	1	2	4	36	monitorable—**N28**
PATTERN: Response											
LTL: AG(landingButi.pressed -> AF(lift.floor=i & lift.door=open))											
7	2	18	2	18	36	0	0	1	1	9	non monitorable
PATTERN: Response											
LTL: AG(landingButi.pressed -> AF(lift.floor=i & lift.door=open & lift.direction=down))											
9	2	34	2	34	68	0	0	1	1	17	non monitorable
PATTERN: Response											

LTL: AG(liftButi.pressed -> AF(floor=i & door=open))											
7	2	18	2	18	36	0	0	1	1	9	non monitorable
PATTERN: Unknown											
LTL: AG(floor=3 & idle & door=closed -> EG(floor=3 & door=closed))											
											not well formed LTL
PATTERN: Response (constrained variation)											
LTL: AG(floor=2 & direction=up & liftBut5.pressed -> A[direction=up U floor=5])											
											not well formed LTL
PATTERN: Response											
LTL: AG(liftBut3.pressed & !lift.empty -> AF((floor=3 & dor=open) | lift.empty))											
12	2	34	2	34	68	0	0	1	1	17	non monitorable
PATTERN: Unknown											
LTL: AG(floor=4 & idle -> E[idle U floor=1])											
											not well formed LTL
PATTERN: Response (constrained variation)											
LTL: AG(up & liftBut2.pressed & !liftBut3.pressed -> A[!(floor=3 & doors=open) U ((floor=2 & door=open) | !up)])											
											not well formed LTL
PATTERN: Response (constrained variation)											
LTL: AG(floor=3 & overloaded -> A[floor=3 U !overloaded])											
											not well formed LTL
PATTERN: Unknown											
LTL: AG(door=open & !overloaded -> E[!overloaded U door=closed])											
											not well formed LTL
PATTERN: Absence											
LTL: AG(!(overloaded & door=closed))											
5	1	5	2	10	15	0	1	1	2	10	monitorable—**N29**
PATTERN: Constrained Response											
LTL: [] (p -> q U r)											
6	2	18	3	27	45	0	1	2	3	27	monitorable—**N30**
PATTERN: Absence											
LTL: [] !(cr1 && cr2)											
5	1	5	2	10	15	0	1	1	2	10	monitorable—**N31**
PATTERN: Response											
LTL: [] (up -> (<> down))											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Response											
LTL: [] (p -> <>q)											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Unknown											
LTL: [] (b -> ([] !np_ && <>r))											
9	3	27	4	36	63	0	1	2	3	27	monitorable—**N32**
PATTERN: Existence											
LTL: <> bp											
2	2	6	1	3	9	1	0	1	2	6	monitorable—**N33**
PATTERN: Unknown											
LTL: []<> (a -> (c U d))											
7	4	36	2	18	54	0	0	1	1	9	non monitorable
PATTERN: Unknown											
LTL: []<> (a -> <> d))											
6	4	20	2	10	30	0	0	1	1	5	non monitorable

PATTERN: Response											
LTL: [] ((p1 -> <>p2) && (p3 -> <>p4)) -> [](p5 -> <>p6)											
16	6	390	10	650	1040	0	0	1	1	65	non monitorable
PATTERN: Unknown											
LTL: (p && X q) || (q && X p)											
9	4	20	5	25	45	1	1	4	6	30	monitorable—**N34**
PATTERN: Unknown											
LTL: <> ((a && b) && b U c)											
8	3	27	2	18	45	1	0	2	3	27	monitorable—**N35**
PATTERN: Existence											
LTL: <> (p U q)											
4	2	10	1	5	15	1	0	1	2	10	monitorable—**N36**
PATTERN: Existence											
LTL: <>[] p											
3	2	6	2	6	12	0	0	1	1	3	non monitorable
PATTERN: Always											
LTL: [] (p U !p)											
5	2	6	2	6	12	0	0	1	1	3	non monitorable
PATTERN: Universal											
LTL: []<>f											
3	2	6	2	6	12	0	0	1	1	3	non monitorable
PATTERN: Universal											
LTL: [](f -> []f)											
5	2	6	3	9	15	0	1	2	3	9	monitorable—**N37**
PATTERN: Universal											
LTL: []<>aftsCC											
3	2	6	2	6	12	0	0	1	1	3	non monitorable
PATTERN: Existence											
LTL: <>aftsCC											
2	2	6	1	3	9	1	0	1	2	6	monitorable—**N38**
PATTERN: Absence											
LTL: [](aftsCC -> !<>aftsDR)											
6	2	10	3	15	25	0	1	2	3	15	monitorable—**N39**
PATTERN: Response											
LTL: [](aftsCR -> <>aftsCC)											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Existence											
LTL: aftsCR -> <>(aftsCC || aftsDR)											
6	3	27	2	18	45	1	0	2	3	27	monitorable—**N40**
PATTERN: Filter=GlobalUniversal+GlobalAbsence											
LTL: ([]<>aftsDR) -> !<>aftsCC											
7	4	20	3	15	35	0	0	1	1	5	non monitorable
PATTERN: Response											
LTL: [](p -> <>[]q)											
6	3	15	4	20	35	0	0	1	1	5	non monitorable
PATTERN: Response											
LTL: [](P(0) -> <>Q(0)) && ... && [](P(m) -> <>Q(m))											
											not well formed LTL
PATTERN: Universal											
LTL: [](pressing -> (!arm1_in_press && !arm2_in_press))											
8	1	9	2	18	27	0	1	1	2	18	monitorable—**N41**

PATTERN: Universal											
LTL: []((P(0) -> Q(0)) && ... && (P(m) -> Q(m)))											
											not well formed LTL
PATTERN: Response											
LTL: []((State=INIT && request) -> <>(State=WORK))											
7	2	18	2	18	36	0	0	1	1	9	non monitorable
PATTERN: Response											
LTL: []((State=WORK -> <>(State=INIT))											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Response Chain 1-2											
LTL: []((State=INIT && request) -> <>(!State=INIT && (!State=INIT -> <>(State=INIT))))											
14	5	25	3	15	40	0	0	1	1	5	non monitorable
PATTERN: Universal											
LTL: []((EndOfCommunication -> BufferEmpty)											
4	1	5	2	10	15	0	1	1	2	10	monitorable—**N42**
PATTERN: Absence											
LTL: []((Synch(to_medium) -> !Synch(to_medium) U (Broadcast | []!Synch(to_medium)))											
11	1	5	2	10	15	0	1	1	2	10	monitorable—**N43**
PATTERN: Absence											
LTL: []!data==id											
3	1	3	2	6	9	0	1	1	2	9	monitorable—**N44**
PATTERN: Universal											
LTL: []((safe -> one_top)											
4	1	5	2	10	15	0	1	1	2	10	monitorable—**N45**
PATTERN: Universal											
LTL: []((safe -> bounded_height)											
4	1	5	2	10	15	0	1	1	2	10	monitorable—**N46**
PATTERN: Universal											
LTL: []((safe -> unique_ids)											
4	1	5	2	10	15	0	1	1	2	10	monitorable—**N47**
PATTERN: Response											
LTL: []((merging -> <>safe)											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Absence											
LTL: []((merging -> !(new_id U safe))											
7	2	18	3	27	45	0	1	2	3	27	monitorable—**N48**
PATTERN: Universal											
LTL: []((SignalFreeWay && !LevelCrossingClosed)											
5	1	5	2	10	15	0	1	1	2	10	monitorable—**N49**
PATTERN: Absence											
LTL: []((PDcontrolX && <>EndCycle) -> !PDcontrolContraX U EndCycle)											
10	4	36	4	36	72	0	1	3	4	36	monitorable—**N50**
PATTERN: Response											
LTL: []((SignalFreeWay && LevelCrossingClosed) -> <>(complete_RESTORE-AUTOMATICMODE && LevelCrossingClosed))											
9	2	18	2	18	36	0	0	1	1	9	non monitorable
PATTERN: Universal											
LTL: []((return_Achieving_Task -> not_subscriber(this,p.memory_property))											
4	1	5	2	10	15	0	1	1	2	10	monitorable—**N51**
PATTERN: Response											
LTL: []((task1_property_broken -> <>task1_terminated)											

5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Response											
LTL: [](task1_property_broken -> <>(task1_terminated || task1_property_repaired))											
7	2	18	2	18	36	0	0	1	1	9	non monitorable
PATTERN: Response											
LTL: [](RequestedRegisterImpl[i] -> <>ServerRegistered[i])											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Existence											
LTL: [](RequestGetIOR[i]=j -> (ServerRegistered[j] U GetIOR(i)))											
6	2	18	3	27	45	0	1	2	3	27	monitorable—**N52**
PATTERN: Response											
LTL: [](ClientSend[i] -> <>ClientRecv[i])											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Response											
LTL: [](ClientSend[i] -> <>ServerRecv[i])											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Response											
LTL: [](ClientAskResult[i] -> <>ClientGetResult[i])											
5	2	10	2	10	20	0	0	1	1	5	non monitorable
PATTERN: Unknown											
LTL: [](TestImplKey[j]=ReceivedKey W !ServerRegistered[j])											
8	4	20	3	15	35	0	1	1	2	10	monitorable—**N53**
PATTERN: Universal											
LTL: [](ReceivedInteger = (SentInteger + 1))											
2	1	3	2	6	9	0	1	1	2	6	monitorable—**N54**

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their very detailed, insightful, and helpful comments.

REFERENCES

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles and Techniques and Tools*. Addison-Wesley.

ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L. J., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. Adding trace matching with free variables to aspectj. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 345–364.

ALPERN, B. AND SCHNEIDER, F. B. 1987. Recognizing safety and liveness. *Distrib. Comput. 2*, 3, 117–126.

ALUR, R. AND DILL, D. L. 1994. A theory of timed automata. *Theor. Comput. Sci. 126*, 2, 183–235.

ALUR, R., COURCOUBETIS, C., AND DILL, D. L. 1993. Model-checking in dense real-time. *Inform. Computat. 104*, 1, 2–34.

ALUR, R., FIX, L., AND HENZINGER, T. A. 1999. Event-clock automata: a determinizable class of timed automata. *Theor. Comput. Sci. 211*, 1–2, 253–273.

BARBON, F., TRAVERSO, P., PISTORE, M., AND TRAINOTTI, M. 2006. Run-time monitoring of instances and classes of web service compositions. In *Proceedings of the International Conference on Web Services (ICWS)*. IEEE Computer Society, 63–71.

BARESI, L., GUINEA, S., AND PASQUALE, L. 2008. Towards a unified framework for the monitoring and recovery of bpel processes. In *Proceedings of the Workshop on Testing, Analysis, and Verification of Web Services and Applications (TAV-WEB)*. T. Bultan and T. Xie Eds., ACM, 15–19.

BARRINGER, H., GOLDBERG, A., HAVELUND, K., AND SEN, K. 2004. Program monitoring with ltl in eagle. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*.

BARRINGER, H., RYDEHEARD, D. E., AND HAVELUND, K. 2007. Rule systems for run-time monitoring: From eagle to ruler. In *International Workshop on Runtime Verification (RV)*. Lecture Notes in Computer Science, vol. 4839, 111–125.

BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2006a. Model-based runtime analysis of reactive distributed systems. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 243–252.

BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2006b. Monitoring of real-time properties. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. S. Arun-Kumar and N. Garg Eds., Lecture Notes in Computer Science, vol. 4337, Springer-Verlag, 260–272.

BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2007. The good, the bad, and the ugly—but how ugly is ugly? In *Proceedings of the International Workshop on Runtime Verification (RV)*. Lecture Notes in Computer Science, vol. 4839, Springer-Verlag, 126–138.

BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2010. Comparing LTL semantics for runtime verification. *J. Logic Comput. 20*, 3, 651–674.

BEHRMANN, G., DAVID, A., LARSEN, K. G., HÅKANSSON, J., PETTERSSON, P., YI, W., AND HENDRIKS, M. 2006. Uppaal 4.0. In *Proceedings of the International Conference on the Quantitative Evaluaiton of Systems (QEST)*. IEEE Computer Society, 125–126.

BENGTSSON, J., LARSEN, K. G., LARSSON, F., PETTERSSON, P., AND YI, W. 1996. UPPAAL: a tool suite for the automatic verification of real-time systems. In *Hybrid Systems III*. R. Alur, T. A. Henzinger, and E. D. Sontag Eds., Lecture Notes in Computer Science, vol. 1066., Springer-Verlag, 232–243.

BERG, T., JONSSON, B., LEUCKER, M., AND SAKSENA, M. 2003. Insights to Angluin's learning. In *Proceedings of the International Workshop on Software Verification and Validation (SVV)*. Electronic Notes in Theoretical Computer Science, vol. 118, 3–18.

BIERE, A., CIMATTI, A., CLARKE, E. M., AND ZHU, Y. 1999. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. R. Cleaveland Ed., Lecture Notes in Computer Science, vol. 1579, Springer, 193–207.

BIERE, A., CIMATTI, A., CLARKE, E., STRICHMAN, O., AND ZHU, Y. 2003. Bounded model checking. In *Advances in Computers*, vol. 58. Academic Press, 118–149.

BODDEN, E. 2004. A lightweight ltl runtime verification tool for java. In *OOPSLA Companion*. J. M. Vlissides and D. C. Schmidt Eds., ACM, 306–307.

BOUYER, P., CHEVALIER, F., AND D'SOUZA, D. 2005. Fault diagnosis using timed automata. In *Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS)*. V. Sassone Ed., Lecture Notes in Computer Science, vol. 3441, Springer, 219–233.

BRYANT, R. E. 1985. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 688–694.

BÜCHI, J. 1962. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science*. 1–11.

CHAUDHURI, S. AND ALUR, R. 2007. Instrumenting c programs with nested word monitors. In *Proceedings of the International SPIN Workshop*. D. Bosnacki and S. Edelkamp Eds., Lecture Notes in Computer Science, vol. 4595, Springer, 279–283.

CHECHIK, M., DEVEREUX, B., AND GURFINKEL, A. 2001. Model-checking infinite state-space systems with fine-grained abstractions using spin. In *Proceedings of the International SPIN Workshop*. M. B. Dwyer Ed., Lecture Notes in Computer Science, vol. 2057, Springer, 16–36.

CHEN, F. AND ROSU, G. 2003. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electron. Notes Theor. Comput. Sci. 89*, 2.

CHEN, F. AND ROŞU, G. 2007. MOP: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 569–588.

CHOW, T. S. 1978. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Engin 4*, 3, 178–187.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. MIT Press, Cambridge, MA.

D'AMORIM, M. AND ROSU, G. 2005. Efficient monitoring of omega-languages. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 3576, 364–378.

D'ANGELO, B., SANKARANARAYANAN, S., SÁNCHEZ, C., ROBINSON, W., FINKBEINER, B., SIPMA, H. B., MEHROTRA, S., AND MANNA, Z. 2005. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME)*. 166–174.

DELGADO, N., GATES, A. Q., AND ROACH, S. 2004. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Engin. 30*, 12, 859–872.

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

DILL, D. L. 1989. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*. J. Sifakis Ed., Lecture Notes in Computer Science, vol. 407, Springer, 197–212.

D'SOUZA, D. 2003. A logical characterisation of event clock automata. *Int. J. Found. Comput. Sci. 14*, 4, 625–639.

DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, Computer Society Press, 411–420.

EISNER, C., FISMAN, D., HAVLICEK, J., LUSTIG, Y., MCISAAC, A., AND CAMPENHOUT, D. V. 2003. Reasoning with temporal logic on truncated paths. In *Proceedings of the International Conference Computer Aided Verification (CAV)*. W. A. H. Jr. and F. Somenzi Eds., Lecture Notes in Computer Science, vol. 2725, Springer, 27–39.

ERLINGSSON, Ú. AND SCHNEIDER, F. B. 2000. Sasi enforcement of security policies: A retrospective. *Proceedings of the DARPA Information Survivability Conference and Exposition 2*, 1287.

FINKBEINER, B. AND SIPMA, H. 2004. Checking finite traces using alternating automata. *Form. Meth. Syst. Des. 24*, 2, 101–127.

FRITZ, C. 2003. Constructing Büchi automata from linear temporal logic using simulation relations for alternating büchi automata. In *Proceedings of the International Conference on Implementation and Application of Automata (CIAA)*. O. H. Ibarra and Z. Dang Eds., Lecture Notes in Computer Science, vol. 2759, Springer, 35–48.

GASTIN, P. AND ODDOUX, D. 2001. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer-Aided Verification*. Lecture Notes in Computer Science, vol. 2102.

GEILEN, M. 2001. On the construction of monitors for temporal logic properties. *Electron. Notes Theor. Comput. Sci. 55*, 2.

GIANNAKOPOULOU, D. AND HAVELUND, K. 2001a. Automata-based verification of temporal properties on running programs. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 412–416.

GIANNAKOPOULOU, D. AND HAVELUND, K. 2001b. Runtime analysis of linear temporal logic specifications. Tech. rep. 01.21, RIACS/USRA.

GOLDSMITH, S., O'CALLAHAN, R., AND AIKEN, A. 2005. Relational queries over program traces. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 385–402.

GOLLU, A., PURI, A., AND VARAIYA, P. 1994. Discretization of timed automata. In *Proceedings of the 33rd IEEE Conference on Decision and Control*. 957–958.

GRINCHTEIN, O. AND LEUCKER, M. 2008. Network invariants for real-time systems. *Form. Asp. Comput. 20*, 619–635.

HÅKANSSON, J., JONSSON, B., AND LUNDQVIST, O. 2003. Generating online test oracles from temporal logic specifications. *J. Softw. Tools Technol. Transfer 4*, 4, 456–471.

HAVELUND, K. AND ROSU, G. 2001. Monitoring Java Programs with Java PathExplorer. *Electron. Notes Theor. Comput. Sci. 55*, 2.

HAVELUND, K. AND ROSU, G. 2002. Synthesizing Monitors for Safety Properties. In *Proceedings of the Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 2280, 342–356.

HAVELUND, K. AND ROSU, G. 2004. Efficient monitoring of safety properties. *J. Softw. Tools Technol. Transfer 6*, 2, 158–173.

HOPCROFT, J. 1971. An n log n algorithm for minimizing states in a finite automation. In *Theory of Machines and Computations*. 189–196.

KAMP, H. W. 1968. Tense logic and the theory of linear order. Ph.D. thesis, University of California, Los Angeles.

KIM, M., KANNAN, S., LEE, I., SOKOLSKY, O., AND VISWANATHAN, M. 2002. Computational analysis of run-time monitoring/fundamentals of java-mac. *Electron. Notes Theor. Comput. Sci. 70*, 4.

KIM, M., VISWANATHAN, M., KANNAN, S., LEE, I., AND SOKOLSKY, O. 2004. Java-mac: A run-time assurance approach for java programs. *Form. Meth. Syst. Des. 24*, 2, 129–155.

KOPETZ, H. 1991. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond*. A. I. Karshmer and J. Nehmer Eds., Lecture Notes in Computer Science, vol. 563, Springer, 87–101.

KUPFERMAN, O. AND VARDI, M. Y. 2001. Model checking of safety properties. *Form. Meth. Syst. Des. 19*, 3, 291–314.

LAMPORT, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Engin. 3*, 2, 125–143.

LICHTENSTEIN, O. AND PNUELI, A. 1985. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Conference on Principles of Programming Languages (POPL)*. ACM, New York, 97–107.

MALER, O. AND NICKOVIC, D. 2004. Monitoring temporal properties of continuous signals. In *Proceedings of the Joint International Conferences on Formal Modelling and Analysis of Timed Systems/Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems (FORMATS/FTRTFT)*. Y. Lakhnech and S. Yovine Eds., Lecture Notes in Computer Science, vol. 3253, Springer, 152–166.

MANNA, Z. AND PNUELI, A. 1995. *Temporal Verification of Reactive Systems: Safety*. Springer, New York.

MARKEY, N. AND SCHNOEBELEN, P. 2003. Model checking a path. In *Proceedings of the Conference on Concurrency Theory (CONCUR)*. Lecture Notes in Computer Science, vol. 2761, 248–262.

MARTIN, M. C., LIVSHITS, V. B., AND LAM, M. S. 2005. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 365–383.

PELED, D., VARDI, M., AND YANNAKAKIS, M. 1999. Black box checking. In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV)*. Kluwer, 225–240.

PNUELI, A. 1977. The temporal logic of programs. In *Proceedings of the Symposium on the Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, 46–57.

PNUELI, A. AND ZAKS, A. 2006. PSL model checking and run-time verification via testers. In *Proceedings of the International Symposium on Formal Methods (FM)*. J. Misra, T. Nipkow, and E. Sekerinski Eds., Lecture Notes in Computer Science, vol. 4085, Springer, 573–586.

RASKIN, J.-F. 1999. Logics, automata and classical theories for deciding real-time. Ph.D. thesis, Namur, Belgium.

RASKIN, J.-F. AND SCHOBBENS, P.-Y. 1999. The logic of event clocks—decidability, complexity and expressiveness. *J. Autom. Lang. Combina. 4*, 3, 247–286.

ROBINSON, W. 2006. A requirements monitoring framework for enterprise systems. *Require. Engin. 11*, 1, 17–41.

ROSU, G. AND BENSALEM, S. 2006. Allen linear (interval) temporal logic - translation to LTL and monitor synthesis. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. T. Ball and R. B. Jones Eds., Lecture Notes in Computer Science, vol. 4144, Springer, 263–277.

SCHWOON, S. AND ESPARZA, J. 2005. A note on on-the-fly verification algorithms. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 3440, 174–190.

SEN, K. AND ROSU, G. 2003. Generating optimal monitors for extended regular expressions. *Electron. Notes Theor. Comput. Sci. 89*, 2.

SISTLA, A. P. AND CLARKE, E. M. 1985. Complexity of propositional temporal logics. *J. ACM 32*, 733–749.

STOLZ, V. AND BODDEN, E. 2006. Temporal assertions using AspectJ. *Electron. Notes Theor. Comput. Sci. 144*, 4, 109–124.

STROUSTRUP, B. 2000. *The C++ Programming Language* Special Ed. Addison-Wesley, Boston, MA.

TARJAN, R. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1*, 2, 146–160.

THATI, P. AND ROSU, G. 2005. Monitoring algorithms for metric temporal logic specifications. *Electron. Notes Theor. Comput. Sci. 113*, 145–162.

TRIPAKIS, S. 2002. Fault diagnosis for timed automata. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*. W. Damm and E.-R. Olderog Eds., Lecture Notes in Computer Science, vol. 2469, Springer, 205–224.

TRIPAKIS, S. AND YOVINE, S. 2001. Analysis of timed systems using time-abstracting bisimulations. *Form. Meth. Syst. Des. 18*, 1, 25–68.

VARDI, M. Y. 1996. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure Versus Automata*. Lecture Notes in Computer Science, vol. 1043, 238–266.

VARDI, M. Y. AND WOLPER, P. 1986. An automata-theoretic approach to automatic program verification. In *Proceedings of the Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 332–345.

VASILEVSKI, M. 1973. Failure diagnosis of automata. *Cybernetic 9*, 4, 653–665.

Received September 2008; revised March 2009, July 2009; accepted August 2009