T U M

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

Runtime verification revisited

Oliver Arafat, Andreas Bauer, Martin Leucker, Christian Schallhart

TUM-I05
Oktober 05

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

Runtime verification revisited

Oliver Arafat, Andreas Bauer, Martin Leucker, and Christian Schallhart

Institut für Informatik, Technische Universität München
{arafat,baueran,leucker,schallha}@informatik.tu-muenchen.de

Abstract. In this paper, we address a typical obstacle in runtime verification of linear temporal logic (LTL) formulae: standard models of linear temporal logic are infinite traces, whereas run-time verification has to deal with only finite system behaviours. This problem is usually addressed by defining an LTL semantics for finite traces, which, however, does usually not fit well to the infinite trace semantics.
We define a 3-valued semantics (*true, false, inconclusive*) for LTL on finite traces that resembles the infinite trace semantics in a preferable manner. Furthermore, we describe how to construct, given an LTL formula, a (deterministic) finite state machine with three output symbols. This automaton reads finite traces and yields their 3-valued LTL semantics. Thus, it can directly be deployed for runtime verification.
Our concepts are first developed in the setting of LTL and then extended to the timed case for which a linear real-time logic, abbreviated as TLTL, is considered. Consequently, for a TLTL formula a monitor is constructed that operates over finite timed traces.
We have implemented the untimed setting and validated our whole approach by examining a real-world case study.

1 Introduction

Runtime verification [7] is becoming a popular tool to complement verification techniques such as model checking and testing. It is especially useful for black or gray-box systems where model checking is not applicable (directly), or, when systems have to be analysed that are beyond the capabilities of today's model checkers.

In a nutshell, runtime verification works as follows. A correctness property φ, usually formulated in some linear temporal logic, such as LTL [17], is given and a so called *monitor* that accepts all models for φ is automatically generated. The system under scrutiny as well as the generated monitor are then executed in parallel, such that the monitor observes the system's behaviour. System behaviour which violates property φ is then detected by the monitor and an according alarm signal returned.

In testing [5], one approach is to generate a test monitor that checks whether the application works correctly. Then, the system under test is executed with typical inputs and it is observed whether the monitor complains. Thus, the monitor generation as used for runtime verification is applicable in the domain of testing as well.

Various runtime verification approaches for LTL have been proposed already [14, 15, 13, 20]. However, the current approaches suffer—to our opinion—from the treatment of the following obstacle.

Notably, the semantics of LTL is defined over infinite (behavioural) traces whereas monitoring a running system allows an at most finite view. In consequence, various authors have proposed custom interpretations of LTL over finite traces using *weak* and *strong semantics*: the weak interpretation of a formula φ w. r. t. to a finite trace, denoted as u, is that if up to the point where u ends, "nothing has yet gone wrong", φ holds.

In the strong view, φ holds only if it evaluates to *true* within u. Eisner et al. give a good overview on the topic [9]. However, good examples can be found for each of the interpretations and—at the same time—also examples that the chosen approach is misleading.

As an alternative, it has also been proposed to restrict the syntax of LTL for runtime verification, such that formulae which may contain certain future obligations cannot be specified at all [11].

In this paper, we propose a simple, yet—as we find—convincing way to overcome this obstacle. Instead of trying to define a two-valued semantics for LTL on finite traces, we define a three valued semantics, using values *true*, *false*, and ?, where the latter denotes *inconclusive*. Given a finite string u and a formula φ, the truth values are defined as expected: if there is no continuation of u satisfying φ, the value is *false*. If every continuation of u satisfies φ, we go for *true*. Otherwise, we say ?, since the observations so far are just inconclusive to say either *true* or *false*.

We argue that it is important to work with three instead of two truth values: consider, for instanace, the property $G\neg p$ stating that no state satisfying p should occur. Clearly, when p is observed, the monitor should complain. As long as p does not hold, it is misleading to say that the formula is *true*, since the next observation might already violate the formula. On the other hand, consider the formula $\neg p\, U\, \mathit{init}$ stating that nothing bad, i. e., p should happen before the init function is called. If, indeed, the init function has been called and no p has been observed before, the formula is *true*, regardless what will happen in the future. For testing and verification, it is important to know whether some property is indeed *true* or whether the current observation is just inconclusive.

Thus, in this paper, we propose a 3-valued logic, LTL$_3$, which can be interpreted over finite traces based on the standard semantics of LTL for infinite trace.

Furthermore, we describe how to construct, given an LTL formula, a (deterministic) finite state machine with three output symbols. This automaton reads finite traces and yields their 3-valued LTL semantics. Thus, it can be directly deployed for runtime verification.

In contrast to many existing monitor generation procedures, our method is designed to yield a conclusive answer as early as possible. Consider, for example, the formula $XXX\mathit{false}$, saying that *false* should hold after three observations. Clearly, the formula is not satisfiable and no observation is needed to conclude *false*. However, typical procedures such as the one described in [15] are only able to complain after three observations. Thus, especially when testing some application, one might stop the observation process with not being informed about some violation although the current observations indicate a problem already.

Our concepts are first developed in the setting of LTL and then extended to the timed case for which a linear real-time logic, abbreviated as TLTL, is considered. We use TLTL, a logic introduced in [18], which, as argued in [8] can be considered a natural counterpart of LTL in the timed setting. Thus, for a TLTL formula a monitor is constructed which operates over finite timed traces. While the general scheme, as we show, is also applicable in the timed setting, the monitor construction is slightly more involved.

We have implemented the untimed setting and validated our approach examining a real-world case study. The monitor generator as well as exemplifying material is available as open-source at `http://runtime.in.tum.de/`, and an example is provided in the appendix of this paper.

Related work. Besides the work mentioned before, our approach is related to [12], where monitor generation based on LTL enriched with a freeze quantifier for time is carried out.

TLTL is event based, meaning that the system emits events when the system's state has changed. In [16] monitoring of continuous signals is considered, which is intrinsicly different to obverse discrete signal in a continuous time domain.

All of the work mentioned so far employs a 2-valued semantics.

Outline. After fixing preliminaries in the next section, we present the construction of the untimed setting in Section 3. In Section 4, we explain the method in the timed setting. We conclude the paper describing our current implementation.

2 Preliminaries

In this section, we briefly recall some formal definitions regarding LTL, infinite and finite automata, which are needed later on. For the remainder of this paper, let us fix a finite set AP of atomic propositions and let us define a finite alphabet $\Sigma = 2^{\mathrm{AP}}$. We write a_i for any single element of Σ, i.e., a_i is a possibly empty set of propositions taken from AP. Finite traces over Σ are elements of Σ^*, and are usually denoted by $u, u', u_1, u_2, \ldots$, whereas infinite traces are elements of Σ^ω, usually denoted by $w, w', w_1, w_2, \ldots$. For some trace $w = a_0 a_1 \ldots$, we denote by w^i the suffix $a_i a_{i+1} \ldots$.

The set of LTL formulae is inductively defined by the following grammar:

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

Let $i \in \mathbb{N}$ be a position. The semantics of LTL formulae is then defined inductively over infinite sequences $w = a_0 a_1 \ldots \in \Sigma^\omega$ as follows: $w, i \models \mathit{true}$, $w, i \models \neg\varphi$ iff $w, i \not\models \varphi$, $w, i \models p$ iff $p \in a_i$, $w, i \models \varphi_1 \vee \varphi_2$ iff $w, i \models \varphi_1$ or $w, i \models \varphi_2$, $w, i \models \varphi_1 U \varphi_2$ iff there exists $k \geq i$ with $w, k \models \varphi_2$ and for all l with $i \leq l < k$, $w, l \models \varphi_1$, and $w, i \models X\varphi$ iff $w, i+1 \models \varphi$. Further, let $w \models \varphi$, iff $w, 0 \models \varphi$.

For every LTL formula φ, its set of models, denoted by $\mathcal{L}(\varphi)$, is a regular set of infinite traces and can be described by a corresponding Büchi automaton.

Formally, a (nondeterministic) Büchi automaton (NBA), is represented by a tuple $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$, where Σ is a finite alphabet, Q is a finite non-empty set of states, $Q_0 \in Q$ is a set of initial states, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, and $F \subseteq Q$ is a set of accepting states.

A NBA is called *deterministic* iff for all $q \in Q$, $a \in \Sigma$, $|\delta(q, a)| = 1$, and $|Q_0| = 1$. We use DBA to denote a deterministic Büchi automaton.

A *run* of an automaton $\mathcal{A}$ on a word $w = a_1 \ldots \in \Sigma^\omega$ is a sequence of states and actions $\rho = q_0 a_1 q_1 \ldots$, where q_0 is an initial state of $\mathcal{A}$ and for all $i \in \mathbb{N}$ we have $q_{i+1} \in \delta(q_i, a)$. For a run ρ, let $\mathrm{Inf}(\rho)$ denote the states visited infinitely often. A run ρ of a NBA $\mathcal{A}$ is called *accepting* iff $\mathrm{Inf}(\rho) \cap F \neq \emptyset$. In other words, a Büchi-accepting run passes infinitely often through at least one final state.

A nondeterministic *finite automaton* (NFA) $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$ is one where Σ, Q, Q_0, δ, and F are defined as for a Büchi automaton, but which operates on finite words, denoted by Σ^*. A *run* of $\mathcal{A}$ on a word $w = a_1 \ldots a_n \in \Sigma^*$ is a sequence of states and actions $\rho = q_0 a_1 q_1 \ldots q_n$, where q_0 is an initial state of $\mathcal{A}$ and for all $i \in \mathbb{N}$ we have $q_{i+1} \in \delta(q_i, a)$. The run is called accepting if $q_n \in F$.

A NFA is called *deterministic* iff for all $q \in Q$, $a \in \Sigma$, $|\delta(q, a)| = 1$, and $|Q_0| = 1$. Again, we use DFA to denote a deterministic finite automaton.

Finally, let us recall the notion of a *Moore machine*, which is a finite state automaton enriched with an output alphabet and output function, formally denoted by a tuple $(\Sigma, Q, Q_0, \delta, \Delta, \lambda, F)$, where Σ is a finite alphabet, Q is a finite non-empty set of states, $Q_0 \in Q$ is a distinguished set of initial states, $\delta : Q \times \Sigma \to 2^Q$ is the partial transition relation, Δ is the output alphabet, $\lambda : Q \to \Delta$ the output function, and $F \subseteq Q$ is a set of final states.

The outputs of a Moore machine, defined by the function λ, are thus determined by the current state $q \in Q$ alone, rather than the input symbols. In the remainder, we will use FSM to abbreviate a finite (Moore) state machine.

3 Three-valued LTL in the untimed setting

3.1 Semantics

To overcome difficulties in defining an adequate boolean semantics for LTL on finite traces, we propose a 3-valued semantics. The intuition is as follows: in theory, we observe an infinite sequence w of some system. For a given formula φ, thus either $w \models \varphi$ or not. In practice, however, we can only observe a finite prefix u of w. Thus, we have to find some sensible semantic evaluation of φ with respect to a finite prefix u of some infinite trace w. Consequently, we let the semantics of u and φ be true, if $uw' \models \varphi$ for every possible future extension 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 u and φ 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 u itself is not sufficient to determine how φ will evaluate in any possible future which is prefixed with u.

Formally, we define our 3-valued semantics in terms of LTL$_3$ over the set of truth values $\mathbb{B}_3 = \{\bot, ?, \top\}$ as follows:

Definition 1 (3-valued semantics of LTL). *Let $u \in \Sigma^*$ denote a finite trace. The* truth value *of a LTL$_3$ formula φ w. r. t. u, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:*

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

3.2 A monitor procedure for LTL$_3$

In this section, we develop an automata-based decision procedure for LTL$_3$. More specifically, for a given formula $\varphi \in$ LTL$_3$, we construct a finite Moore state machine, $\bar{\mathcal{A}}^\varphi$ that reads finite traces $u \in \Sigma^*$ and outputs $[u \models \varphi]$, thus a value in $\mathbb{B}_3$.

For a NBA $\mathcal{A}$, we denote by $\mathcal{A}(q)$ the NBA that coincides with $\mathcal{A}$ except for Q_0, which is defined as $Q_0 = \{q\}$. Let $\varphi \in$ LTL for the rest of this section and let $\mathcal{A}^\varphi$ denote the NBA, which accepts all models of φ, and let $\mathcal{A}^{\neg\varphi}$ denote the NBA, which accepts all counter examples of φ. For these automata, we observe:

Lemma 1. *Let $\mathcal{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ denote the NBA such that $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$. For $u \in \Sigma^*$, let $\hat{\delta}(Q_0^\varphi, u) = \{q_1, \ldots, q_l\}$. Then*

$$[u \models \varphi] \neq \bot \textit{ iff } \exists q \in \{q_1, \ldots, q_l\} \textit{ such that } \mathcal{L}(\mathcal{A}^\varphi(q)) \neq \emptyset.$$

Lemma 2. *Let $\mathcal{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, F^{\neg\varphi})$ denote the NBA such that $\mathcal{L}(\mathcal{A}^{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. For $u \in \Sigma^*$, let $\hat{\delta}(Q_0^{\neg\varphi}, u) = \{q_1, \ldots, q_l\}$. Then*

$$[u \models \varphi] \neq \top \textit{ iff } \exists q \in \{q_1, \ldots, q_l\} \textit{ such that } \mathcal{L}(\mathcal{A}^{\neg\varphi}(q)) \neq \emptyset.$$

The correctness of the first lemma follows directly from the definition of Büchi automata and their acceptance, and the second lemma rephrases the first one by substituting $\neg\varphi$ for φ.

For $\mathcal{A}^{\varphi}$ and $\mathcal{A}^{\neg\varphi}$, we now define a function $\mathcal{F}^{\varphi} : Q^{\varphi} \to \mathbb{B}$, respectively $\mathcal{F}^{\neg\varphi} : Q^{\neg\varphi} \to \mathbb{B}$, assigning to each state q whether the language of the respective automaton starting in state q is not empty. Using $\mathcal{F}^{\varphi}$ and $\mathcal{F}^{\neg\varphi}$, we define two NFAs $\hat{\mathcal{A}}^{\varphi} = (\Sigma, Q^{\varphi}, Q_0^{\varphi}, \delta^{\varphi}, \hat{F}^{\varphi})$ and $\hat{\mathcal{A}}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ where

$$\hat{F}^{\varphi} = \{q \in Q^{\varphi} \mid \mathcal{F}^{\varphi}(q) = \top\} \qquad \hat{F}^{\neg\varphi} = \{q \in Q^{\neg\varphi} \mid \mathcal{F}^{\neg\varphi}(q) = \top\}$$

$\hat{\mathcal{A}}^{\varphi}$, resp. $\hat{\mathcal{A}}^{\neg\varphi}$, accept the finite traces u for which $[u \models \varphi]$ evaluates to $\neq \bot$ and, respectively, $\neq \top$.

Lemma 3. *Using the notation as before, we have for all $u \in \Sigma^*$:*

- *$u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi})$ iff $[u \models \varphi] \neq \bot$*
- *$u \in \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi})$ iff $[u \models \varphi] \neq \top$.*

Therefore, we can evaluate $[u \models \varphi]$ according to Lemma 3 as follows.

Lemma 4. *Using the notation as before, we have:*

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

The lemma yields a simple 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 4 to determine $[u \models \varphi]$. As a final step, we now define a (deterministic) FSM $\bar{\mathcal{A}}^{\varphi}$ that outputs for each finite string u its associated 3-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 the standard manner by the power-set construction. Now, we define the FSM in question as a product of $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$:

Definition 2 (Monitor $\bar{\mathcal{A}}^{\varphi}$ for a LTL-formula φ). *Let $\tilde{\mathcal{A}}^{\varphi} = (\Sigma, Q^{\varphi}, Q_0^{\varphi}, \delta^{\varphi}, \tilde{F}^{\varphi})$ and $\tilde{\mathcal{A}}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \tilde{F}^{\neg\varphi})$ be the DFAs which correspond to the two NFAs $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$ as defined for Lemma 3.*

Then we define the Monitor *$\bar{\mathcal{A}}^{\varphi} = \tilde{\mathcal{A}}^{\varphi} \times \tilde{\mathcal{A}}^{\neg\varphi}$ as FSM $(\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{L})$, where Σ is the finite input alphabet, $\bar{Q} = Q^{\varphi} \times Q^{\neg\varphi}$, $\bar{q}_0 = (q_0^{\varphi}, q_0^{\neg\varphi})$, $\bar{\delta}((q, q'), a) = (\delta^{\varphi}(q, a), \delta^{\neg\varphi}(q', a))$, and $\bar{L} : \bar{Q} \to \mathbb{B}_3$ is defined by*

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

We conclude by formulating the following theorem.

Theorem 1. *Let φ be a formula of LTL$_3$ and let $\bar{\mathcal{A}}^{\varphi} = (\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{L})$ be the corresponding monitor. Then, for all $u \in \Sigma^*$ the following holds:*

$$[u \models \varphi] = \bar{L}(\bar{\delta}(\bar{q}_0, u)).$$

We have summed up our entire construction again in Table 1.

	Input	$\varphi \in \text{LTL}_3$	
1	Formula	φ	$\neg\varphi$
2	NBA	$\mathcal{A}^\varphi$	$\mathcal{A}^{\neg\varphi}$
3	Emptiness per state	$\mathcal{F}^\varphi$	$\mathcal{F}^{\neg\varphi}$
4	NFA	$\hat{\mathcal{A}}^\varphi$	$\hat{\mathcal{A}}^{\neg\varphi}$
5	DFA	$\tilde{\mathcal{A}}^\varphi$	$\tilde{\mathcal{A}}^{\neg\varphi}$
6	FSM	$\bar{\mathcal{A}}$	

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

Complexity. Let us study the size of the resulting FSM. Consider Table 1: given φ, step 1 requires us to replicate φ and to negate it, i.e., it is linear in the original size. Step 2, the construction of the NBAs, causes an exponential blow-up in the worst-case. Steps 3 and 4, leading to $\hat{\mathcal{A}}^\varphi$ and $\hat{\mathcal{A}}^{\neg\varphi}$, do not change the size of the original automata. Then, computing the deterministic automata of step 5, might again require an exponential blow-up in size. In total the FSM of step 6 will have double exponential size with respect to $|\varphi|$.

Discussion. As an alternative to the proposed approach, we could have used the following procedure. For $\varphi \in \text{LTL}_3$, define a deterministic parity automaton. For a deterministic parity automaton, it is easy to define a labelling function $\bar{L}$ and to obtain a FSM as in Theorem 1. However, the solution proposed in this paper has some advantages over this alternative. Firstly, the size of a deterministic parity automaton is in $O(2^{2^{n \cdot \log n}})$ in the size of φ while the proposed solution is in $O(2^{2^n})$.

More importantly, it is easy to implement the determinisation of NFAs and the product for obtaining $\bar{\mathcal{A}}$ (steps 4–6) in an on-the-fly fashion, as described in detail in Section 5.

4 Three-valued LTL in the timed setting—TLTL

In this part, we extend the approach developed in the preceding section to the timed setting. Thus, the goal is to dynamically check real-time specifications formulated in a timed temporal logic. We use timed LTL (TLTL for short), a logic introduced in [18], in the form presented in [19].

The language expressible by a TLTL formula can be defined by *event-clock automata* [3], a subclass of *timed automata*. It was shown in [8] that TLTL corresponds exactly to the class of languages definable in first-order logic interpreted over timed words. Thus, it can be considered to be the natural counterpart of LTL for the timed setting. Given the translation to event-clock automata in the literature [19], it is promising to base our timed runtime verification approach on TLTL and event-clock automata.

4.1 Preliminaries

Let us fix an alphabet Σ of actions for the rest of this section. In the timed setting, every symbol $a \in \Sigma$ is associated with an *event-recording clock*, x_a, and an *event-predicting clock*, y_a. An (infinite) *timed word* w over the alphabet Σ is an (infinite) sequence of *timed events* $(a_0, t_0)(a_1, t_1)\ldots$ consisting of symbols $a_i \in \Sigma$, and non-negative numbers $t_i \in \mathbb{R}^{\geq 0}$, such that

1. for each $i \in \mathbb{N}$, $t_i < t_{i+1}$, (*strict monotonicity*)
2. for all $t \in \mathbb{R}^{\geq 0}$ there is an $i \in \mathbb{N}$ such that $t_i > t$. (*progress*)

Furthermore, for w as above, we call its sequence of actions (the projection to the first component) the *untimed word* of w, denoted by $ut(w)$.

To simplify notation, we abbreviate $(\Sigma \times \mathbb{R}^{\geq 0})$ by $T\Sigma$. Thus, a finite timed word is an element of $T\Sigma^*$ and the domain of infinite timed words is denoted by $T\Sigma^\omega$.

Given an (infinite) timed word w, the value of the event-recording clock variable x_a at position j of w equals $t_j - t_i$, where i represents the last position preceding j such that $a_i = a$. If no such position exists, then the value of x_a remains undefined, denoted by $\perp$. The event-predicting clock variable y_a then equals $t_j - t_i$, where j represents the next position after i such that $a_j = a$. If no such position exists, again, the variable remains undefined. The set of all event-clocks is denoted by $C_\Sigma = \{x_a, y_a \mid a \in \Sigma\}$. A *clock valuation function* over a timed word w, $\gamma_i : C_\Sigma \to \mathbb{R}^{\geq 0} \cup \{\perp\}$ assigns a positive real, or undefined value to each clock variable corresponding to position i. We abbreviate $\mathbb{R}^{\geq 0} \cup \{\perp\}$ by $T_\perp$.

A *clock constraint* compares a clock value to a natural number. Let $\Psi(C_\Sigma)$ denote the set of clock constraints over C_Σ. Formally, a clock constraint $\psi \in \Psi(C_\Sigma)$ is a conjunction of atomic formulae of the form $z \bowtie c$, where $z \in C_\Sigma$, $\bowtie \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{N}$. Given a clock constraint ψ and a clock valuation function γ, we write $\gamma \models \psi$ to denote that according to γ, constraint ψ is fulfilled, where $\perp \bowtie c$ for $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, \geq, >\}$ does not hold, and the remaining cases are defined in the expected manner.

4.2 Syntax and semantics of TLTL$_3$

Let Σ be a finite set of actions. A set of formulas φ of TLTL is defined by the grammar

$$\varphi ::= true \mid a \mid \lhd_a \in I \mid \rhd_a \in I \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\ U\ \varphi \mid X\varphi \quad (a \in \Sigma),$$

where $\lhd_a$ is the operator which measures the time elapsed since the last occurrence of a, and $\rhd_a$ the operator which predicts the next occurrence of a within a timed interval $I \in \mathcal{I}$. The set of intervals $\mathcal{I}$ contains intervals of the form (l, r), $[l, r)$, $(l, r]$, or $[l, r]$, where $l, r \in \mathbb{R}^{\geq 0} \cup \{\infty\}$. Without loss of generality, we assume $l < r$, and for intervals $(l, r]$, or $[l, r]$ that $r \neq \infty$. To simplify notation, we use $\llparenthesis$ and $\rrbracket$ for interval borders which can either be (or [, respectively),].

The semantics of TLTL formulae are defined inductively over infinite timed words $w \in T\Sigma^\omega$, where $w = (a_0, t_0)(a_1, t_1)$, and $i \in \mathbb{N}^{\geq 0}$ as follows:

$$\begin{array}{ll}
w, i \models true & \\
w, i \models \neg\varphi & \Leftrightarrow w, i \not\models \varphi \\
w, i \models a & \Leftrightarrow a(i) = a \\
w, i \models \lhd_a \in I & \Leftrightarrow \gamma_i(x_a) \in I \\
w, i \models \rhd_a \in I & \Leftrightarrow \gamma_i(y_a) \in I \\
w, i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow (w, t \models \varphi_1 \vee w, t \models \varphi_2) \\
w, t \models \varphi_1 U \varphi_2 & \Leftrightarrow \exists k \geq 0 : (w, k \models \varphi_2 \wedge \forall l : (0 \leq l < k \wedge w, l \models \varphi_1)) \\
w, i \models X\varphi & \Leftrightarrow w, i+1 \models \varphi
\end{array}$$

Further, let $w \models \varphi$, iff $w, 0 \models \varphi$.

Analogously to the untimed case, we now define a 3-valued semantics for TLTL, from this point onwards denoted as TLTL$_3$, as follows:

Definition 3. *Let $u \in T\Sigma^*$ denote a finite timed trace. The truth value of a TLTL$_3$ formula φ w. r. t. u, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:*

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

4.3 Event-clock automata

Given a finite set of clocks, C_Σ, we define an event-clock automaton as a finite state machine whose edges are annotated both with input symbols and with clock constraints as $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$, where Σ is a finite input alphabet, Q a finite set of states, $Q_0 \subseteq Q$ are initial states, $F \subseteq 2^Q$ is a set of accepting states (generalised Büchi acceptance condition), and $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times Q$ a set of transitions. An edge $e = (q, a, \psi, q')$ represents a transition from source state q upon symbol a to destination q', where the clock constraint ψ then specifies when this transition is enabled. For an event-clock automaton $\mathcal{A}$, let $K_\mathcal{A}$ denote the biggest constant appearing in some constraint of $\mathcal{A}$; we write K when $\mathcal{A}$ is clear from the context.

A *timed run* θ of an automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ over a timed word $w \in T\Sigma^\omega$ starting in (q_0, γ_0) is an infinite sequence of state-valuation tuples and transitions as follows: $(q_0, \gamma_0) \overset{\alpha_1}{\to} (q_1, \gamma_1) \overset{\alpha_2}{\to} \ldots$ with $q_i \in Q$, and γ_i being the evaluation function assigning for every element from Σ the value of the recording and predicting event clocks corresponding to α_i, where $\alpha_i \in T\Sigma$ is a timed event of the form $(a_i \in \Sigma, t_i \in \mathbb{R}^{\geq 0})$[1], and for all $i \geq 1$ there is a transition in E of the form $(q_{i-1}, a_i, \psi, q_i)$ such that $\gamma_i \models \psi$. $\mathcal{A}_{ec}$ accepts θ, iff for each $F_i \in F$, a state $q \in F_i$ exists such that q occurs infinitely often in θ.

γ_0 is *initial* (w.r.t. w) if $\gamma_0(x_a) = \bot$ and $\gamma_0(y_a) = i$ if $\alpha_i = (a, t_i)$ and $\gamma_0(y_a) = \bot$ if a does not occur in w. Then, the timed language accepted by $\mathcal{A}_{ec}$, denoted as $\mathcal{L}(\mathcal{A}_{ec})$, is the set of timed words for which an accepting run of $\mathcal{A}_{ec}$ exists starting in (q_0, γ_0), for some $q_0 \in Q_0$ and the initial γ_0.

For runtime verification predicting clock variables pose a problem, since information about the future occurrence of an action a is predicted, but this information is not available yet. We solve this problem by representing the value of some predicting clock variable *symbolically.*

A *symbolic clock valuation function* $\Gamma : C_\Sigma \to T_\bot \cup \mathcal{I}$ assigns a positive real, or undefined value to each recording clock variable and an *interval* or undefined value to each predicting clock variable. The interval constrains the possible values of a predicting variable. To simplify notation, we identify $\Gamma(y_a) = (l, r)$ with the constraint $y_a > l \wedge y_a < r$ (and similarly for borders [and]).

For a symbolic clock evaluation Γ, we define the following three operations: time elapse, reset, and conjunction. Given an elapsed time $t \in \mathbb{R}^{\geq 0}$, $\Gamma' = \Gamma + t$, where $\Gamma'(x_a) = \Gamma(x_a) + t$ and for $\Gamma(y_a) = [\![l, r]\!]$, we set $\Gamma'(y_a) = [\![l \dot{-} t, r - t]\!]$, where $\dot{-}$ yields at least 0. If $r - t < 0$, then Γ' is invalid. Γ reset by action a, denoted by $\Gamma\!\downarrow\! a$, sets $x_a = 0$ and removes all constraints on y_a, and we set $\Gamma'(y_a) = [0, \infty)$ and $\Gamma'(z_b) = \Gamma(z_b)$ for all $b \neq a$. The conjunction of Γ with constraint ψ yields $\Gamma' = \Gamma \wedge \psi$, where each predicting clock y_a is combined with the constraints of ψ which involve y_a, i. e., for $a \in \Sigma$, $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge\{y_a \bowtie c \subseteq \psi\}$. We call Γ' invalid, if for some y_a, $\Gamma'(y_a)$ is not satisfiable.

[1] Note that the sequence of γ_i is determined by the α_i and just listed for clarity.

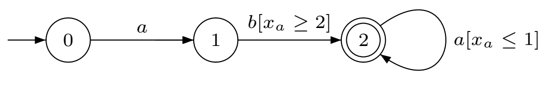

Fig. 1. Event-clock automaton A_1

Furthermore, a transition $(q, a, \psi, q') \in E$ is applicable to a pair (q, Γ), if the constraints $x_b \bowtie c$ in ψ are satisfied by Γ, for all $b \in \Sigma$, and $0 \in \Gamma(y_a)$. If $(q, a, \psi, q') \in E$ is applicable, then the corresponding successor of (q, Γ) is (q', Γ'), where $\Gamma' = (\Gamma{\downarrow}a) \wedge \psi$.

A *symbolic timed run* Θ of an automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ over a timed word $w \in T\Sigma^\omega$ starting in (q_0, Γ_0) is an infinite sequence of state-symbolic-valuation tuples and transitions as follows: $(q_0, \Gamma_0) \overset{\alpha_1}{\to} (q_1, \Gamma_1) \overset{\alpha_2}{\to} \ldots$ with $q_i \in Q$, and Γ_i being a symbolic valuation function, where for each $(q_{i-1}, \Gamma_{i-1}) \overset{(a_i,t_i)}{\to} (q_i, \Gamma_i)$, there exists some transition $(q_{i-1}, a_i, \psi, q_i)$ which is applicable to $(q_{i-1}, \Gamma_{i-1} + t_i)$ and (q_i, Γ_i) is the result of this application. The notion of acceptance for symbolic runs corresponds to that of runs, i. e., for each $F_i \in F$ there is some $q \in F_i$ occurring infinitely often.

We call Γ_0 *initial* if for $a \in \Sigma$, $\Gamma_0(x_a) = \perp$ and $\Gamma_0(y_a) = [0, \infty)$.

Theorem 2. *Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ be an event-clock automaton and $w \in T\Sigma^\omega$. Then, there is an accepting run on w starting in (q_0, γ_0) iff there is a symbolic accepting run on w starting in (q_0, Γ_0).*

The important fact about the previous theorem is that γ_0 is dependent on w (since each predicting clock y_a has to be initialised to match the first occurrence of a), while Γ_0 is independent of w. Thus, symbolic runs are a suitable device for runtime verification.

4.4 A monitor procedure for TLTL$_3$

We can assume that for some property φ as well as its negation, an event-clock automaton is given, accepting precisely the models respectively counterexamples of φ respectively $\neg\varphi$ (see [19] for details).

Looking at the scheme developed in the untimed 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 can be seen in Figure 1. 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 1 to 2 requires the clock x_a to be at least 2. This, however, restricts the loop in state 2 to be taken.

We therefore decided to work on the so-called region automaton (for alternatives see Remark 2 on page 11). Recall that K denotes the biggest constant occurring in some constraint of the event-clock automaton. Two clock valuations γ_1, γ_2 are in the same region, denoted by $\gamma_1 \equiv \gamma_2$ iff

- for all $z \in C_\Sigma$, $\gamma_1(z) = \perp$ iff $\gamma_2(z) = \perp$, and (*agreement on undefined*)
- for all $z \in C_\Sigma$, if $\gamma_1(z) \leq K$ or $\gamma_2(z) \leq K$, then $\lfloor \gamma_1(z) \rfloor = \lfloor \gamma_2(z) \rfloor$, and (*agreement on integral part*)
- for all $a \in \Sigma$, let $\langle \gamma(x_a) \rangle = \lceil x_a \rceil - \gamma(x_a)$ and $\langle \gamma(y_a) \rangle = \gamma(y_a) - \lfloor y_a \rfloor$. Then, for all $z_1, z_2 \in C_\Sigma$ with $\gamma_1(z_1) \leq K$ and $\gamma_2(z_2) \leq K$,
  - $\langle \gamma_1(z_1) \rangle = 0$ iff $\langle \gamma_2(z_1) \rangle = 0$
  - $\langle \gamma_1(z_1) \rangle \leq \langle \gamma_1(z_2) \rangle$ iff $\langle \gamma_2(z_1) \rangle \leq \langle \gamma_2(z_2) \rangle$. (*agreement on order of fractions*)

A *clock region* is an equivalence class of $\equiv$. We denote the set of all regions by $\mathcal{R}$.

The key property of the region equivalence is *stability* [2]: given state s and two equivalent valuation γ_1 and γ_2, then (s', γ') is an a-successor of (s, γ_1) iff it is one of (s, γ_2), too. By induction, this can be lifted to infinite runs. This yields:

Lemma 5. *Let $\mathcal{A}_{ec}$ be an event-clock automaton. Let q be some state of $\mathcal{A}_{ec}$ and γ_1, γ_2 two valuations with $\gamma_1 \equiv \gamma_2$. Let $\bar{w} \in \Sigma^\omega$. Then, there exists an accepting run on some infinite timed word $w_1 \in T\Sigma^\omega$ with $ut(w_1) = \bar{w}$ starting in (q, γ_1) iff there exists an accepting run on some infinite timed word $w_2 \in T\Sigma^\omega$ with $ut(w_2) = \bar{w}$ starting in (q, γ_2).*

Note that the so-called *zones equivalence* [1] is not stable. Thus, *zone automata*, while successfully used in model checking tools such as Uppaal [4], do not satisfy our needs.

For completeness, we give the translation of an event-clock automaton to a region automaton, as presented in [19], whose states actually serve their purpose in our approach, because of the previous lemma.

A clock region κ_2 is a *time successor* of a clock region κ_1, denoted by $\kappa_2 \in TS(\kappa_1)$, iff for all $\gamma \in \kappa_1$ there is some $t \in \mathbb{R}^{\geq 0}$ such that $\gamma + t \in \kappa_2$. Here, $\gamma' = \gamma + t$ is defined with $\gamma'(x_a) = \gamma(x_a) + t$ and $\gamma'(y_a) = \gamma(y_a) - t$. To simplify notation, let us fix an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$. The region automaton of $\mathcal{A}_{ec}$ is the (generalised) Büchi automaton $R(\mathcal{A}_{ec}) = (\Sigma^r, Q^r, Q^r_0, E^r, F^r)$, where

- $Q^r\{(l, \kappa, \zeta) \mid l \in Q, \kappa \in \mathcal{R}, \zeta \in \{t, d\}\}$ is the set of states
- $Q^r_0 = \{(l, \kappa, \zeta) \in Q^r \mid l \in Q_0, \forall a \in \Sigma, \kappa(x_a) = \bot, \zeta = d\}$ is the set of initial states
- $\Sigma^r = \Sigma \cup \{\epsilon\}$
- $E^r = E^r_d \cup E^r_t$ is the union of untimed and timed transitions, where
  - $E^r_d = \{((l_1, \kappa_1, t), (l_2, \kappa_2, d), a) \mid (l_1, a, \psi, l_2) \in E$ and $\exists \kappa_3$ s.t. $\kappa_1 = \kappa_3[y_a := 0], \kappa_2 = \kappa_3[x_a := 0],$ and $\kappa_3 \models \psi\}$
  - $E^r_t = \{((l, \kappa_1, d), (l, \kappa_2, t), \epsilon) \mid \kappa_2 \in TS(\kappa_1)\}$
- $F^r = \{F^r_i \mid F_i \in F\} \cup \{F_{x_a} \mid \lhd_a \in I \in Sub(\varphi)\} \cup \{F_{y_a} \mid \rhd_a \in I \in Sub(\varphi)\}$,
  - where for $F_i \in F$, $F^r_i = \{(l, \kappa, \zeta) \mid l \in F_i\}$
  - $F_{x_a} = \{(l, \kappa, \zeta \mid \forall \gamma \in \kappa\ \gamma(x_a) = 0 \vee \gamma(x_a) > c \vee \gamma(x_a) = \bot\}$
  - $F_{y_a} = \{(l, \kappa, \zeta \mid \forall \gamma \in \kappa\ \gamma(y_a) = 0 \vee \gamma(y_a) = \bot\}$

Note that the region automaton as defined here is a Büchi automaton and thus, the accepted language is a sequence of (untimed) words over Σ. Thus, it is easy to compute for every state, whether the accepted (untimed) language is empty or not. For every state (l, κ, ζ) with a non-empty language, stability now guarantees that there for each $\gamma \in \kappa$, there is some accepting run of the original event-clock automaton starting in (l, γ) for some timed word w. Dually, if the accepted language is empty, the underlying event-clock automaton has no accepting run starting in (l, γ) for any $\gamma \in \kappa$ and any w (Lemma 5).

We now describe a procedure that reads timed events and decides whether further events might yield an accepting run (satisfying the formula to check).

The procedure is based on the event-clock automaton as well as the region automaton. It follows the possible *symbolic* computations for the given input along the lines of the event-clock automaton. To decide, whether future events might contribute to an accepting run, the region automaton is consulted.

Let us fix an event-clock automaton $\mathcal{A}_{ec}$ and its region automaton $R(\mathcal{A}_{ec})$ for the moment. Let us consider the timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^\omega$. Recall that (a_0, t_0) actually means that the first action a_0 occurs at time t_0.

Let Γ_0 be the initial symbolic valuation of $\mathcal{A}_{ec}$ and l_0 one of the initial states of $\mathcal{A}_{ec}$. Now, for the first event (a_0, t_0), we compute the set of successors w.r.t. $\mathcal{A}_{ec}$. If this set is

	Input	$\varphi \in$ TLTL$_3$	
1	Formula	φ	$\neg\varphi$
2	ECA	$\mathcal{A}_{ec}^{\varphi}$	$\mathcal{A}_{ec}^{\neg\varphi}$
3	Region automaton	R^{φ}	$R^{\neg\varphi}$
4	Emptiness per state	$\mathcal{F}^{\varphi}$	$\mathcal{F}^{\neg\varphi}$
5	Monitor	$\bar{\mathcal{A}}$	

Table 2. The procedure for getting $[u \models \varphi]$ for a given $\varphi \in$ TLTL$_3$.

empty, the underlying formula is obviously violated. If not, each successor is a pair (l, Γ). This (l, Γ) now corresponds to a set of states in the region automaton. If *and only if* all of them accept the empty language, the underlying property is violated, which follows directly from Theorem 2 and Lemma 5. We continue with each successor state (l, Γ) for which a corresponding accepting state of $R(\mathcal{A}_{ec})$ exists, reading the input event.

Thus, the generated procedure keeps a set of possible state-symbolic valuation pairs that represent the possible current states of $\mathcal{A}_{ec}$ (giving credit to the non-deterministic nature of $\mathcal{A}_{ec}$). Furthermore, the transition table of $\mathcal{A}_{ec}$ and the states of $R(\mathcal{A}_{ec})$ enriched with emptiness per state information are stored as look-up tables.

Remark 1. To enhance the practical applicability of our approach, we adjust the procedure slightly: the formal framework described above requires the monitor to complain iff for some prefix $(a_0, t_0) \ldots (a_i, t_i)$ no accepting run exists. In particular, it is assumed that "a watch is consulted only when some action occurs". But the time transitions yielding the subsequent regions in the region automaton actually (often) constrain the possible occurrence of some future event a. For each current valuation Γ corresponding to a set of regions, we check in $R(\mathcal{A})$ the possible accepting time successors and compute a maximal time bound before some event has to occur to reach an accepting state. Thus, in practice, we can set a timer interrupt, when such a bound exists, and decide for rejection, when a timeout occurs before a suitable action has been read.

The overall monitor procedure for TLTL$_3$ is similar to the untimed case and summarised in Table 2. However, since we have to consider the region automaton (with emptiness per state information) together with the current clock valuation to compute the timed successor, we do not get an NFA neither can determinise to get a DFA (at least in a straightforward manner). We therefore propose for the overall monitor procedure to rely on $R(\mathcal{A}_{ec}^{\varphi})$ and $R(\mathcal{A}_{ec}^{\neg\varphi})$ in an on-the-fly manner, as described above.

Remark 2. We have used region automata to keep our presentation short and simple. The key property of our monitor construction, however, is *stability* of the region equivalence. Thus, our approach can be improved by taking a coarser stable partition of the underlying timed transition system instead of the region equivalence. Such partitions have been studied extensively in [21].

Complexity. We consider again Table 2, and observe that step 1 is constant. The region automaton of $\mathcal{A}_{ec}^{\varphi}$ (resp. $\mathcal{A}_{ec}^{\neg\varphi}$) is exponential with respect to the length of the underlying formula φ as well as the largest constant K appearing in φ. Following the different paths for some prefix (due to the non-determinism of the region automaton) might cause further exponential blow-up in space, in the worst case.

5 Implementation

In this section, we discuss how to implement our ideas in the untimed case as described in Sec. 3. We show how the actual monitors are automatically generated from a LTL formula, and give a brief overview over our current implementation, which basically consists of the monitor generator, referred to as Ltl2Fsm, and a logging framework, referred to as Diagnostics. Further, we describe how our techniques can be used to avoid a particular C++-pitfall; that is, to spawn threads before entering the main procedure of a program.

5.1 Efficient monitor code generation

In a nutshell, the approach to produce a monitor for a given LTL formula φ as described in Sec. 3 involves the following steps. First, for a given LTL formula φ, a corresponding NBA $\mathcal{A}^\varphi$ must be constructed. For this purpose, we employ the implementation described in [10] which, in most cases, yields very efficient automata containing only a minimal number of states.

Second, the NBA $\mathcal{A}^\varphi$ must be transformed into an NFA $\hat{\mathcal{A}}^\varphi$, which will then be determinised to yield the DFA $\tilde{\mathcal{A}}^\varphi$.

Since our automata are defined with respect to an alphabet $\Sigma = 2^{\mathrm{AP}}$, each symbol $a \in \Sigma$ is a finite set of atomic propositions. A set $a \subseteq \mathrm{AP}$ represents the assignment which evaluates a proposition $p \in \mathrm{AP}$ to *true* iff $p \in a$ holds. Thus, we can denote the transitions of our automata as tuple (s, Φ, s'), where s is the original state, Φ is a formula over the set of propositions AP, and s' is the new state. Such a transition (s, Φ, s') is enabled for a given alphabet symbol $a \subseteq \mathrm{AP}$, if Φ is satisfied by a.

Following Lemma 3 on page 5, we transform the corresponding NBA $\mathcal{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ into an NFA $\hat{\mathcal{A}}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, \hat{F}^\varphi)$, by checking for every state $q \in Q^\varphi$ whether the language accepted by $\mathcal{A}^\varphi(q)$ is empty or not (note, $\mathcal{A}^\varphi(q)$ uses q as initial state but is otherwise identical to $\mathcal{A}^\varphi$). $\mathcal{A}^\varphi(q)$ accepts an ω-word, iff, starting at q, a final state $q' \in F^\varphi$ can be reached which is a member of a non-trivial strongly connected component, i. e., there must be a loop which leads from q' back again to q'. This process is repeated for the negated formula $\neg\varphi$ in order to obtain the corresponding DFA, $\tilde{\mathcal{A}}^{\neg\varphi}$, and finally to obtain the FSM, $\bar{\mathcal{A}}$, with a cross-product construction (see Definition 2).

Typically, an explicit generation of the FSM $\bar{\mathcal{A}}$ causes a double exponential blow-up, firstly for building the NBA $\mathcal{A}^\varphi$, and secondly for computing the corresponding DFA $\tilde{\mathcal{A}}^\varphi$. For this reason, we deliberately decided against the explicit construction of two DFAs, $\tilde{\mathcal{A}}^\varphi$ and $\tilde{\mathcal{A}}^{\neg\varphi}$, in favour of an *implicit representation* of the FSM $\bar{\mathcal{A}}$ by means of two NFAs $\hat{\mathcal{A}}^\varphi$ and $\hat{\mathcal{A}}^{\neg\varphi}$. In other words, for each of the two NFAs, we generate a C++-class which implements the NFA-interface and thus, offers the following three methods:

- getSuccessors(s,a) takes a state s and a subset $a \subseteq \mathrm{AP}$ of atomic propositions and returns the set of successors reachable from s by a. That is, we check for every transition (s, Φ, s') in the transition table, whether a satisfies Φ and, if so, add s' to the result set.
- isFinal(s) returns **true** (or, **false**) if s is a final state (not a final state, correspondingly).
- initialStates () returns the set of initial states of the NFA.

Now, to determinise a NFA dynamically at runtime, and without explicitly storing its comprehensive look-up table, we define a class DFA similar to the interface above, which wraps a NFA object and provides the following methods:

- getSuccessor(s,a) takes a state s and a subset $a \subseteq \mathrm{AP}$ of atomic propositions and returns a single successor reachable from s by a.

- isFinal(s) returns **true** (or, **false**) if s is a final state (not a final state, correspondingly).
- initialState () returns the single initial state of the DFA.

A single state of the DFA corresponds to a set of states of the NFA. To implement the methods described above, a DFA object uses a reference to the corresponding NFA object in order to compute the state transitions of the DFA object in an on-the-fly manner, as shown below in a C++-inspired pseudo code.

```
StateSet DFA::initialState()
{
    return(nfa.initialStates());
}

StateSet DFA::getSuccessor(StateSet S, PropositionSet a)
{
    StateSet result;
    for_all s in S
        result.add(nfa.getSuccessors(s, a));
    return result;
}

bool DFA::isFinal(StateSet S)
{
    for_all s in S
        if(nfa.isFinal(s))
            return true;
    return false;
}
```

This code is independent of the formula φ and the underlying nfa-object, and is, therefore, implemented once manually rather than automatically generated. Finally, the FSM is implemented in a similar on-the-fly fashion: the constructor of our FSM-class takes two references which point to the DFA objects which implement $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ and stores them in the fields dfa_pos and dfa_neg. Furthermore, the FSM maintains the current state of the two DFAs in the fields state_pos and state_neg, respectively. The FSM-class then provides a method processInput which takes a subset a of propositions from AP and returns the current evaluation of the system trace with respect to φ:

```
BoolThree FSM::processInput(PropositionSet a)
{
    state_pos=dfa_pos.getSuccessor(state_pos,a);
    state_neg=dfa_neg.getSuccessor(state_neg,a);
    if(!dfa_pos.isFinal(state_pos)) return false;
    if(!dfa_neg.isFinal(state_neg)) return true;
    return ?;
}
```

5.2 Our implementation: Ltl2Fsm and Diagnostics

Our C++-implementation consists of two core parts. First, we developed the monitor generator LTL2FSM, which is based on the implicit FSM representation as described above. Secondly, we use DIAGNOSTICS which is part of the RUNTIME REFLECTION project. This framework allows to annotate C++-code in order to generate log records for events such as method entries and exits, unexpected exceptions, violated assertions, and passing of simple trace points. DIAGNOSTICS then allows to attach Loggers to the stream of log records. Each time, a log event described by a Record occurs, the log method of all registered Loggers is invoked in order to write the contents of the Record onto disk, to send it to a remote server, or to analyse the trace for erroneous behavior. In fact, DIAGNOSTICS uses a specific Logger to provide facilities for unit testing: the test verdict is determined by this specific Test_Logger which checks certain basic properties on the log stream, e. g., that all assertions in the tested code have been satisfied.

LTL2FSM takes a LTL formula φ and generates the two NFAs, which are used to parameterise a Fsm object, in order to obtain a Fsm object which serves as a monitor for the property φ. This Fsm object provides a method processInput, as described above, i. e., processInput takes a bit vector where each bit indicates whether the associated proposition is true or not, and returns one of the three semantical valuations **true**, ?, or **false**. Such a Fsm object can be easily integrated into an arbitrary framework that provides a stream of logging records. More specifically, the occurring log events must be used to update an abstract representation of the current system state, i. e., the bit vector which describes the propositions which hold in the current state.

To integrate Fsm with DIAGNOSTICS, we additionally provide a Logger named Monitor Wrapper, which adapts the Fsm to suit the DIAGNOSTICS framework: A MonitorWrapper maintains a local bit vector to represent the current abstract state. When the log method of the MonitorWrapper is invoked with a new Record, the protected method p_translate is called to update the bit vector according to the current Record. Then the processInput method of the underlying Fsm object is invoked with the updated bit vector. Again, like the Fsm class, we do not generate the MonitorWrapper class but provide it as library code. Only the two Nfa classes are generated specifically for each φ. Finally, to integrate the glue code, one has to derive a class from MonitorWrapper and provide an implementation of the p_translate method.

5.3 An example: the static initialisation order fiasco

In order to demonstrate the feasibility of our approach, we have used LTL2FSM and DIAGNOSTICS to check at runtime that no thread gets spawned before the program under scrutiny enters the main procedure. In C++ it is a particularly bad idea to spawn threads during the static initialisation, because of the so called *static initialisation order fiasco* [6]: all static objects of an executable are initialised before main is entered, however, their order is undefined. Thus if a thread is spawned before entering main, it is difficult to ensure that all resources necessary to synchronise the threads are already initialised, such as some globally available and statically initialised mutex object. The problem is an especially striking one when large applications are built from a number of frameworks which must remain independent from each other.

We used $\varphi \equiv (!\text{SPAN\_THREAD}\ U\ \text{ENTER\_MAIN})$ to specify that no thread should be spawned until the main procedure has been entered; with LTL2FSM, we generated the two NFA classes for $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$. To produce a log event when main is entered, we employed an annotation of DIAGNOSTICS which is provided to guard the entry and exit of procedures. To generate a log event when a thread is spawned, we wrapped the system call for creating a thread transparently such that an independent and unchanged library will automatically use the new wrapper instead of the original call (in our case pthread_create). DIAGNOSTICS provides a macro to emit such a wrapper at ease: the so-generated wrapper for pthread_create has the same signature as pthread_create itself, obtains the original procedure dynamically from `libpthread`, and generates a log message before and after calling the original procedure.

The source code to implement the monitor for the static initialisation order fiasco and to instrument pthread_create consists of slightly more than 100 lines of code, not counting the generated code. The complete manually written code is included in the appendix.

6 Conclusions

In this paper, we presented a three valued semantics for both timed and untimed LTL with respect to finite traces. The three-valued semantics resembles the infinite traces semantics of LTL more naturally, as we have argued.

Furthermore, we developed efficient monitor generation procedure for both logics that alert as soon as some prefix allows to do so.

We have already implemented the untimed setting. We integrated this monitor generation tool within a larger logging and unit testing framework. We have examined a standard C++ pitfall and provided a run-time verification solution to this problem, which is efficient in terms of both engineering overhead as well as runtime penalty.

References

1. R. Alur. Timed automata. In *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems*, 1998.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
3. R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273, 1999.
4. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: a tool suite for the automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1996.
5. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
6. M. Cline. `http://www.parashift.com/c++-faq-lite/`.
7. S. Colin and L. Mariani. *Run-Time Verification*, chapter 18. Volume 3472 of Broy et al. [5], 2005.
8. D. D'Souza. A logical characterisation of event clock automata. *Int. Journ. Found. Comp. Sci.*, 14(4):625–639, Aug. 2003.
9. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *CAV03*, volume 2725 of *LNCS*, pages 27–39, Boulder, CO, USA, July 2003. Springer.
10. C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating büchi automata. In O. H. Ibarra and Z. Dang, editors, *CIAA*, volume 2759 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2003.
11. D. Giannakopoulou and K. Havelund. Runtime analysis of linear temporal logic specifications. Technical Report 01.21, RIACS/USRA, 2001.
12. J. Håkansson, B. Jonsson, and O. Lundqvist. Generating online test oracles from temporal logic specifications. *Journ. Softw. Tools for Tech. Transf.*, 4(4):456–471, 2003.
13. K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *Electr. Notes Theor. Comp. Sci.*, 55(2), 2001.
14. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 135, Washington, DC, USA, 2001. IEEE Computer Society.
15. K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.
16. O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In Y. Lakhnech and S. Yovine, editors, *FORMATS/FTRTFT*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2004.
17. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, Oct. 31–Nov. 2 1977. IEEE Computer Society Press.

18. J.-F. Raskin and P.-Y. Schobbens. State clock logic: A decidable real-time logic. In O. Maler, editor, *HART*, volume 1201 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 1997.
19. J.-F. Raskin and P.-Y. Schobbens. The logic of event clocks - decidability, complexity and expressiveness. *Journ. of Autom. Lang. and Comb.*, 4(3):247–286, 1999.
20. V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV'05).* To be published in ENTCS, Elsevier, 2005.
21. S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

A Implementation

In this appendix, we briefly show the manually written sources which we used to apply our runtime verification approach to the static initialisation order fiasco. In Listing 1.1 you see the code to instrument the pthread_create library call. The macro DIAGNOSTICS_AUDIT_INSTRUMENT_C_CALL as used there emits the wrapper with the signature

```
int pthread_create(pthread_t *threadp,
                   pthread_attr_t const * attr,
                   void *(*start_routine) (void *),
                   void * arg) __THROW
```

which dynamically loads `"libpthread.so.0"` to obtain a pointer to the original pthread_create implementation. The wrapper generates a log message on entry and exit and calls the original implementation inbetween.

Listing 1.1. `pthread_create_wrapper.cpp`

```
#include <pthread.h>
#include <diagnostics/instrumentation.hpp>

extern "C" DIAGNOSTICS_AUDIT_INSTRUMENT_C_CALL
("libpthread.so.0",                         // library name
 int,                                       // result type
 pthread_create,                            // the name
 (pthread_t *threadp,                       // the arguments
  pthread_attr_t const * attr,              //    with types
  void *(*start_routine) (void *),
  void * arg),
 __THROW,                                   // the throw decl
 (threadp, attr, start_routine, arg),       // the arguments
 "");                                       // further output in
                                            // the log messages
```

Listing 1.2 and 1.3 shows the interface and implementation of the Siof_Monitor class. The class inherits from ::ltl2fsm::Monitor_Wrapper and is only implementing the constructor, destructor, and the p_translate method. The constructor first integrates the generated classes for the two NFAs, namely Pos_static_fiasco and Neg_static_fiasco. Secondly, the constructor sets the m_bit_vector to its initial state. The p_translate method interprets each logged Record_t in terms of a Bit_Vector_t: If a relevant log message occurs, it updates the bit_vector accordingly and returns **true**. Otherwise, it only returns **false**.

Listing 1.2. `siof_monitor.hpp`

```
#ifndef EXAMPLE_SIOF_MONITOR_HPP
#define EXAMPLE_SIOF_MONITOR_HPP

#include <ltl2fsm/monitor_code/Monitor_Wrapper.hpp>

class SIOF_Monitor : public ::ltl2fsm::Monitor_Wrapper
{
public:
    SIOF_Monitor();
    virtual ~SIOF_Monitor();
protected:
    virtual bool p_translate(Record_t const & record,
                             Bit_Vector_t & bit_vector);
};

#endif
```

Listing 1.3. `siof_monitor.cpp`

```
#include "siof_monitor.hpp"

#include <generated/Neg_static_fiasco.hpp>
#include <generated/Pos_static_fiasco.hpp>

#include <ltl2fsm/monitor_code/Fsm.hpp>
#include <ltl2fsm/monitor_code/Dfa.hpp>
#include <diagnostics/frame/record.hpp>

using namespace ltl2fsm;

SIOF_Monitor::SIOF_Monitor()  : Monitor_Wrapper
(new Fsm(new Dfa(new Pos_static_fiasco),
         new Dfa(new Neg_static_fiasco)), 2)
{
    // pthread_create is not called initially
    m_bit_vector[0]=false;
    // main starts not immediately
    m_bit_vector[1]=false;
}

SIOF_Monitor::~SIOF_Monitor()
{
}

#define WHAT_MAIN "PROCEDURE=\"int main()\""
#define WHAT_PC   "PROCEDURE=\"int pthread_create("

bool SIOF_Monitor::p_translate(Record_t const & record,
                               Bit_Vector_t & bit_vector)
{
    using namespace diagnostics;
    if(record.type()==TYPE_PROCEDURE_ENTER
       && record.str_what().find(WHAT_MAIN)==0) {
        bit_vector[1]=true;
        return true;
    }
    if(record.type()==TYPE_PROCEDURE_ENTER
       && record.str_what().find(WHAT_PC)==0) {
        bit_vector[0]=true;
        return true;
    }
    return false;
}
```

Finally, in Listing 1.4, we show a small main component which attaches a Siof_Monitor to the log stream. The call back ::diagnostics::set_initial_loggers provides an initial set of Logger objects, even before entering the main procedure – when the first log message occurs, DIAGNOSTICS is initialised and calls ::diagnostics::set_initial_loggers during this initialisation. The main procedure itself starts with an DIAGNOSTICS_PROD_PROCEDURE_GUARD annotation which generates a log message on entry and exit of this procedure. After entering main, we create a thread which starts with start_func and wait for its termination. At the end of main, we print the state of Siof_Monitor. In this case, since the first thread is spawned after main has been reached, the Siof_Monitor is in state **true**.

Listing 1.4. `main.cpp`

```
#include <diagnostics/annotations.hpp>
#include <diagnostics/configuration.hpp>
```

```

#include "siof_monitor.hpp"

#include <pthread.h>

static ltl2fsm::Monitor_Wrapper * siof_monitor;

DIAGNOSTICS_NAMESPACE_BEGIN;
void set_initial_loggers(::std::vector<Logger *> & loggers)
{
    loggers.push_back(siof_monitor=new SIOF_Monitor);
}
DIAGNOSTICS_NAMESPACE_END;

void* start_func(void*)
{
    DIAGNOSTICS_PROD_PROCEDURE_GUARD("");
    return NULL;
}

int main()
{
    DIAGNOSTICS_PROD_PROCEDURE_GUARD("");

    pthread_t tid;
    void **return_value;
    pthread_create(&tid, NULL, &start_func, NULL);
    pthread_join(tid,return_value);

    ::std::cout << siof_monitor->status() << ::std::endl;
    return 0;
}
```