Technische Universiteit
Eindhoven
University of Technology

Checking Metric Temporal Logic with TRACE

Martijn Hendriks, Marc Geilen, Amir R. B. Behrouzian, Twan Basten,
Hadi Alizadeh, Dip Goswami

ES Reports
ISSN 1574-9517

ESR-2016-01
April 11, 2016

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Checking Metric Temporal Logic with TRACE

Martijn Hendriks*, Marc Geilen†, Amir R. B. Behrouzian†, Twan Basten†*, Hadi Alizadeh†, Dip Goswami†
*Embedded Systems Innovation by TNO, Eindhoven, The Netherlands
†Eindhoven University of Technology, Eindhoven, The Netherlands

***Abstract*—Execution traces, time-stamped sequences of events, provide a general, domain-independent, view on the behavior of systems. They enable analysis of metrics such as latency, pipeline depth and throughput. Often, however, it is not clear what such metrics exactly mean and ad hoc methods are used to compute them. Metric Temporal Logic (MTL) can be used to address this issue: it enables the formal specification of quantitative properties on execution traces. We thus have added an MTL checking capability to the TRACE tool, which is a tool for viewing and analyzing execution traces [1]. We use a recursive memoization algorithm that generates concise explanations of the truth value of the given MTL formula. These explanations can be visualized in the TRACE viewer to aid interpretation by the user.**

***Keywords*-Metric temporal logic, execution trace, informative prefix, explanation, visualization**

I. INTRODUCTION

The design of embedded systems requires careful analysis of trade-offs between properties such as cost, quality and performance. Many of these kind of system level metrics are hard to predict, which is why modeling and prototyping play an important role in the design process. Besides quantification of the system-level metrics of interest, models and prototypes can also be used to generate *execution traces*. These are time-stamped sequences of events and form a general and unified view on the behavior of systems and models of these systems. They can be used to investigate details of (unexpected) behavior, or to indentify bottlenecks or optimization opportunities. The TRACE tool has been built to visualize execution traces as Gantt charts. Recently, several generic analysis methods have also been added [1], [2].

Although terms as "latency" and "throughput" are often used to indicate system qualities that can be extracted from execution traces, it is often not clear what is exactly meant. Formalisms for property specification with a well-defined syntax and semantics alleviate this problem. Metric temporal logic (MTL) [3] is such a formalism and it is suitable for execution traces, since it enables the expression of a wide variety of quantitative real-time properties for time-stamped event sequences. In this paper we explain how we have added the capability to interpret MTL formulas on execution traces to the TRACE tool.

Related work & contribution: Gantt charts as used by the TRACE tool for visualization are omnipresent and appear, e.g., in project planning ([4]), embedded-systems analysis [5], and model checking tools [6]. There is also a plethora of domain specific visualization tools for execution traces through other means than Gantt charts, e.g., [7], [8].

Checking the validity of an MTL formula is a fundamental problem in model-checking and runtime verification. Tableau methods construct an automaton from the formula that accepts exactly those traces that are models of the formula. Such an automaton can then be used to verify properties of systems using model checking techniques, see, e.g., [9], [10], [11]. Runtime monitoring of formulas is investigated, e.g., in [12], [13], [14], [15], [16]. These techniques usually instrument source code with code that emits time-stamped events. The runtime stream of emitted time-stamped events is then checked, one event at a time, against the formula, which often takes the form of an automaton. These techniques are on-line and do not require the whole execution trace to be stored. Our situation is somewhat different from both the model-checking and the runtime verification settings in the sense that we have access to a single static finite execution trace in the TRACE tool. This execution trace does not grow anymore, but it can be (but does not need to be!) a prefix of a longer trace that just has not been recorded. In order to be able to deal with both situations (prefix or complete trace) and to generate a concise explanation of the truth value of the property, we have chosen to develop a straightforward recursive algorithm with memoization. It is directly based on the neutral, strong and weak MTL satisfaction relations defined in [16]. The algorithm is easy to implement and uses no complicated, error-prone data structures. The memoization allows us to construct an explanation of the truth value of the formula. This explanation can be used in the visualization to give feedback to the user. Our algorithm does not compute the truth values of all subformulas in all states in the trace as, e.g., the dynamic programming approaches that are suggested in [16], [17] do, but only the relevant ones. This makes the explanation more concise.

II. METRIC TEMPORAL LOGIC

In the remainder of this paper we assume the context of sets of states S, a set $\mathbf{AP}$ of atomic propositions, and a labeling function $l : S \to 2^{\mathbf{AP}}$ that assigns to a state $s \in S$ atomic propositions that are true in that state. Furthermore, we let $\mathbb{R}$ denote the set of real numbers.

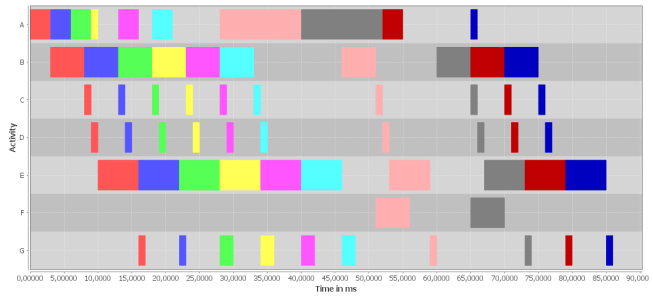

Figure 1: A TRACE Gantt-chart view of an execution in which 10 objects are processed.

A *trace* is a possibly infinite sequence $s_0, s_1, \ldots$ of states in S. A *timed trace* is a possibly infinite sequence of tuples of $S \times \mathbb{R}$, say $\rho = (s_0, t_0), (s_1, t_1), \ldots$, such that $t_i \leq t_{i+1}$. In the context of a timed trace, we call the state-time tuples also "states", and ρ_i refers to (s_i, t_i).

The syntax of MTL formulas ranged over by ϕ is inductively defined as follows:

$$\phi \;:=\; true \,|\, p \,|\, \phi \wedge \phi \,|\, \neg\phi \,|\, \phi\, \mathbf{U}_I\, \phi \quad p \in \mathbf{AP}$$

where $I \subseteq [0, \infty)$ is an interval (open, closed, or half open) on $\mathbb{R}$ where the end points are defined by real numbers or ∞. We let Φ denote the set of all MTL formulas. We assume the existence of comparison relations that check whether a real number is larger or smaller than all elements in the interval.

LTL is a fragment of MTL in which all intervals are of the form $[0, \infty)$. In this case, the intervals are omitted in the notation. Below we define the satisfaction relation $\models$ for MTL.

Definition 1 (MTL semantics)**.** *Let $\rho = (s_0, t_0), (s_1, t_1), \ldots$ be an infinite timed trace. The satisfaction relation on the elements $\rho_i = (s_i, t_i)$ is defined inductively as follows:*

- $\rho_i \models true$
- $\rho_i \models p$ *iff* $p \in l(s_i)$
- $\rho_i \models \phi_1 \wedge \phi_2$ *iff* $\rho_i \models \phi_1$ *and* $\rho_i \models \phi_2$
- $\rho_i \models \neg\phi$ *iff* $\rho_i \not\models \phi$
- $\rho_i \models \phi_1 \mathbf{U}_I \phi_2$ *iff* $\exists_j$ *s.t.* $j \geq i$, $\rho_j \models \phi_2$, $t_j - t_i \in I$, *and* $\rho_k \models \phi_1$ *for all* $i \leq k < j$

Note that formulas are interpreted relative to timed states and not to arbitrary moments in time. We say that a timed trace ρ satisfies an MTL formula ϕ, denoted by $\rho \models \phi$, if and only if $\rho_0 \models \phi$. Some useful and well-known abbreviations are the following: $\mathbf{F}_I \phi = true \mathbf{U}_I \phi$ (finally ϕ), and $\mathbf{G}_I \phi = \neg \mathbf{F}_I \neg \phi$ (globally ϕ). The satisfaction relation $\models$ can also be defined for finite traces by restricting the scope of the existential quantifier for the until operator to the trace length. We denote the finite satisfaction relation by $\models_f$.

Example 1. *We consider a pipelined processing system consisting of seven tasks, A – G, that work on a stream of input objects. Through the use of the* OCTOSIM *discrete-event simulator [18] we have access to finite timed traces of this system. For instance, Fig. 1 shows a Gantt-chart representation, using the* TRACE *tool, for processing of 10 input objects. Atomic propositions in this setting are of the form N, N^v or N^v_i, where N is the name of the task, $v \in \{s, e\}$ indicates whether it is the start event of the task or the end event, and i is the object number. For instance, G^e_0 specifies the end of task G for the first object in the stream, and G^e refers to end events of tasks with name G. A number of useful MTL properties that can be used to analyze timed traces of the system for, e.g., 1000 input objects, are shown below (object ids range from 0 to 999).*

The first property formalizes that the first object (with id 0) has been completely processed within 25 time units.

$$\mathbf{F}_{[0,25]} G^e_0 \tag{1}$$

The second property formalizes that the total execution time is at most 6500 time units.

$$\mathbf{F}_{[0,6500]} G^e_{999} \tag{2}$$

The third property is a variation of the second property that, however, does not refer to the object identifier 999. Instead, a temporal operator is used to ensure that there will be no more end events of activity G after 6500 time units.

$$\mathbf{F}_{[0,6500]} \mathbf{G}_{(0,\infty)} \neg G^e \tag{3}$$

The fourth property formalizes that the per object processing time is at most 70 time units.

$$\bigwedge_{i=0}^{999} \mathbf{G}\big(A^s_i \Rightarrow \mathbf{F}_{[0,70]} G^e_i\big) \tag{4}$$

The fifth property formalizes that the throughput is at least $10/65$ in every window of 10 consecutive end events of task G.

$$\bigwedge_{i=0}^{989} \mathbf{G}\big(G^e_i \Rightarrow \mathbf{F}_{[0,65]} G^e_{i+10}\big) \tag{5}$$

The sixth property formalizes that the pipeline is always at least four objects deep, except for the start and end areas in the trace (note that this is an LTL property).

$$\bigwedge_{i=9}^{990} \mathbf{G}\big(A^s_{i+3} \Rightarrow \mathbf{F} G^e_i\big) \tag{6}$$

The seventh property formalizes that after any end event of task G, another end event of task G happens within 3 and 15 time units.

$$\mathbf{G}(G^e \Rightarrow \mathbf{F}_{[3,15]} G^e) \tag{7}$$

These examples illustrate the flexibility and expressive power of MTL. The formalism allows us to define what we exactly mean with, e.g., pipeline depth, latency and throughput.

In the present paper we assume that we have a finite execution trace of some system. This trace can be obtained from a real system, but also, for instance, from a discrete-event simulation model. We distinguish two situations: (i) the trace represents the full execution of some process, or (ii) the trace is a prefix of some ongoing, possibly infinite, process. An example of the first situation is the execution trace of an image processing pipeline that processes 10 images and then is done. In this case, we can apply the MTL semantics for finite traces. An example of the second situation is a part of an execution obtained from a running web server. In this case, however, application of the finite MTL semantics is not appropriate, because there is an unknown extension of the trace that can affect the truth value of the property. For this situation, we use the notion of *informative prefixes* [19] which is explained in more detail in the next section.

III. Informative prefixes

Consider a finite prefix ρ of some timed trace and an MTL formula ϕ. Then we say that ρ is a *bad* prefix if and only if every extension of ρ dissatisfies ϕ. Dually, ρ is a *good* prefix if and only if every extension of ρ satisfies ϕ. A *neutral* prefix is neither good nor bad. Intuitively, an *informative* prefix tells the whole story about the (dis)satisfaction of an MTL formula [19]. For instance, the prefix $(p,0),(p,1),(p,2),(q,3)$ is bad for $\mathbf{G}p$ and it is also informative. The prefix is also bad for $\mathbf{F}(p \wedge \neg p)$, but not informative because the dissatisfaction for any extension depends on the unsatisfiability of $p \wedge \neg p$. This information is not to be found in the prefix itself.

Below we follow the approach of [16] (which uses the truncated semantics of [20]) to define strong and weak satisfaction relations for MTL formulas and timed traces, $\models^+$ and $\models^-$ respectively. If $\rho \models^+ \phi$ then for any finite extension of ρ, say ρ', holds that $\rho' \models_f \phi$. Thus, the finite prefix proves the satisfaction of the formula and it is stable. We say that ρ is an *informative good prefix* for ϕ. On the other hand, $\rho \not\models^- \phi$ implies that no finite extension of ρ exists that models ϕ. In that case, ρ is an *informative bad prefix* for ϕ. Otherwise, we say that ρ is non-informative.

Definition 2. *Let $\rho = (s_0,t_0),\ldots,(s_n,t_n)$ be a finite prefix of a timed trace. The strong and weak satisfaction relations $\models^+$ and $\models^-$ on the elements $\rho_i = (s_i,t_i)$ are defined inductively as follows:*

- $\rho_i \models^+ \mathit{true}$
- $\rho_i \models^+ p$ *iff* $p \in l(s_i)$
- $\rho_i \models^+ \phi_1 \wedge \phi_2$ *iff* $\rho_i \models^+ \phi_1$ *and* $\rho_i \models^+ \phi_2$
- $\rho_i \models^+ \neg\phi$ *iff* $\rho_i \not\models^- \phi$
- $\rho_i \models^+ \phi_1 \mathbf{U}_I \phi_2$ *iff* $\exists_{i\le j\le n}\ \rho_j \models^+ \phi_2$, $t_j - t_i \in I$, *and* $\rho_k \models^+ \phi_1$ *for all* $i \le k < j$

and

- $\rho_i \models^- \mathit{true}$
- $\rho_i \models^- p$ *iff* $p \in l(s_i)$
- $\rho_i \models^- \phi_1 \wedge \phi_2$ *iff* $\rho_i \models^- \phi_1$ *and* $\rho_i \models^- \phi_2$
- $\rho_i \models^- \neg\phi$ *iff* $\rho_i \not\models^+ \phi$
- $\rho_i \models^- \phi_1 \mathbf{U}_I \phi_2$ *iff either (1)* $\exists_{i\le j\le n}\ \rho_j \models^- \phi_2$, $t_j - t_i \in I$, *and* $\rho_k \models^- \phi_1$ *for all* $i \le k < j$ *or (2)* $t_n - t_i < sup(I)$ *and* $\rho_k \models^- \phi_1$ *for all* $i \le k \le n$

Note that the definitions for $\neg$ constructs use the dual satisfaction relation, and that the definition of $\models^-$ for the until operator consists of a disjunction that includes the case in which the end of the trace is reached.

Proposition 1 ([16])**.** *Let ρ be a finite timed trace and let i be a position in that trace. Then $\rho_i \models^+ \phi \Rightarrow \rho_i \models_f \phi$ and $\rho_i \models_f \phi \Rightarrow \rho_i \models^- \phi$.*

Proposition 2 ([16])**.** *Let ρ be a prefix of some finite timed trace ρ'. Then $\rho_i \models^+ \phi \Rightarrow \rho'_i \models^+ \phi$ and $\rho_i \not\models^- \phi \Rightarrow \rho'_i \not\models^- \phi$.*

IV. An algorithm for informative prefixes

In this section we present an algorihm that computes the informative value of a finite prefix of some timed trace for a given formula by straightforward application of the semantics in Def. 2.

Let $\rho = (s_0,t_0),\ldots,(s_n,t_n)$ be a finite prefix of a timed trace, let $\phi = \phi_1 \mathbf{U}_I \phi_2$ and let $0 \le i \le n$. We define the conditions $C_{11}(i,j)$ and $C_1(i,m)$ as follows:

$$\begin{aligned} C_{11}(i,j) &\triangleq \forall_{i\le k<j}\ \rho_k \models^+ \phi_1 \\ C_1(i,m) &\triangleq \exists_{i\le j\le m}\big(\rho_j \models^+ \phi_2 \wedge t_j - t_i \in I \wedge C_{11}(i,j)\big) \end{aligned}$$

By definition, $\rho_i \models^+ \phi$ if and only if $C_1(i,n)$ holds. We have a similar approach for $\models^-$. The condition for $\rho_i \not\models^- \phi_1 \mathbf{U}_I \phi_2$ can be written as:

$$\Big(\forall_{i\le j\le n}\big(\rho_j \not\models^- \phi_2 \vee t_j - t_i \notin I \vee \exists_{i\le k<j}\ \rho_k \not\models^- \phi_1\big)\Big) \wedge \Big(t_n - t_i \ge sup(I) \vee \exists_{i\le k\le n}\ \rho_k \not\models^- \phi_1\Big)$$

We extract the following conditions:

$$\begin{aligned} C_{22}(i,j) &\triangleq \exists_{i\le k<j}\ \rho_k \not\models^- \phi_1 \\ C_2(i,m) &\triangleq \forall_{i\le j\le m}\big(\rho_j \not\models^- \phi_2 \vee t_j - t_i \notin I \vee C_{22}(i,j)\big) \\ C_3(i,m) &\triangleq t_n - t_i \ge sup(I) \vee \exists_{i\le j\le m}\ \rho_j \not\models^- \phi_1 \end{aligned}$$

By definition, $\rho_i \not\models^- \phi_1 \mathbf{U}_I \phi_2$ if and only if both $C_2(i,n)$ and $C_3(i,n)$ hold.

Below we introduce an algorithm that decides whether the prefix of some infinite timed trace is an informative good prefix, an informative bad prefix, or a non-informative prefix. To this end, we use standard three-valued logic over the values $\{T,F,U\}$, where T stands for true, F for false and

Algorithm 1: $getOrCompute(\rho, \phi, i, v)$

```
1 result ← v(φ, i)
2 if result = ? then
3 |   result ← compute(ρ, φ, i, v)
4 |   v(φ, i) ← result
5 return result
```

U for unknown:

$$\begin{array}{llll} \neg T = F & T \wedge T = T & U \wedge T = U & F \wedge T = F \\ \neg F = T & T \wedge U = U & U \wedge U = U & F \wedge U = F \\ \neg U = U & T \wedge F = F & U \wedge F = F & F \wedge F = F \end{array}$$

The algorithm consists of two mutually recursive procedures, shown in Alg. 1 and Alg. 2. Both have four arguments: A finite timed trace $\rho = (s_1, t_1), \ldots, (s_n, t_n)$, an MTL formula ϕ, a state index $1 \leq i \leq n$, and a table $v : \Phi \times \mathbb{N} \hookrightarrow \{T, F, U, ?\}$ ($v(\phi, i) =?$ means that *compute* has not yet been called for formula ϕ and index i).

The user calls $getOrCompute(\rho, \phi, 0, v_0)$ in order to compute the truth value of formula ϕ on trace ρ, where v_0 initially maps all subformula-state combinations to ?. The $getOrCompute$ function is a wrapper around the *compute* function, and it performs memoization of computed values using the lookup table. The main *compute* algorithm in Alg. 2 uses the conditions defined above in a straightforward loop for the case of the temporal operator. When the call to $getOrCompute(\rho, \phi, 0, v_0)$ returns, the table v_0 contains the values that have been computed in order to reach the conclusion. This serves as a basis for the visualization of the explanation of the formula.

Above we have noted that in certain cases we do not need to interpret the finite part as a prefix. In that case we can use the finite MTL semantics as introduced in Sec. II. The difference between the truncated semantics in Def. 2 and the finite semantics in Def. 1 is in the case for the temporal operator that includes a disjunction for the situation at the end of the prefix in Def. 2 but not in Def. 1. Condition $C_3(i, m)$ deals with this case and hence removing c_3 from Alg. 2 results in an algorithm for $\models_f$.

We have the following invariants:

Invariant 1. *At the beginning of loop iteration j (line 16) of Alg. 2, it holds that if $getOrCompute(\rho, \psi, j, v) = T \Leftrightarrow \rho_j \models^+ \psi$ and $getOrCompute(\rho, \psi, j, v) = F \Leftrightarrow \rho_j \not\models^- \psi$ for both $\psi = \phi_1$ and $\psi = \phi_2$, then $c_{11} = C_{11}(i, j)$ and $c_{22} = C_{22}(i, j)$.*

Proof: It trivially holds for the first loop iteration with $j = i$ because c_{11} is initialized to *true*, and c_{22} is initialized to *false*. Now assume that this holds up to loop iteration $j = m - 1$, and consider that loop iteration that updates the values to get to iteration m. We consider four cases.

First, $c_{11} = false$ and by line 22 it remains *false*. An earlier loop iteration $k < m$ has changed the value from c_{11} from *true* to *false*. By the IH, $getOrCompute(\rho, \phi_1, k) \neq T$ and therefore $\rho_k \not\models^+ \phi_1$. Therefore we also have that $C_{11}(i, m) = false$.

Second, $c_{11} = true$. By line 22 and the IH we have that $getOrCompute(\rho, \phi_1, k) = T$ and thus $\rho_k \models^+ \phi_1$ for all $i \leq k < m - 1$. The iteration for $j = m - 1$ can update c_{11} in two ways. First, $getOrCompute(\rho, \phi_1, m - 1) = T$ in which case c_{11} stays *true* up to the start of iteration m. By the IH, $\rho_{m-1} \models^+ \phi_1$ and thus $C_{11}(i, m) = true$. Second, $getOrCompute(\rho, \phi_1, m - 1) \neq T$ in which case c_{11} becomes *false* at the start of iteration m. We thus have by the IH that $\rho_{m-1} \not\models^+ \phi_1$ and clearly $C_{11}(i, m) = false$.

Third, $c_{22} = true$ and by line 23 it remains *true*. An earlier loop iteration $k < m$ has changed the value from c_{22} from *false* to *true*. By the IH, $getOrCompute(\rho, \phi_1, k) = F$ and therefore $\rho_k \not\models^- \phi_1$. Therefore we also have that $C_{22}(i, m) = true$.

Algorithm 2: $compute(\rho, \phi, i, v)$

```
1  if φ = true then
2  |   return T
3  else if φ ∈ AP then
4  |   if φ ∈ l(s_i) then
5  |   |   return T
6  |   else
7  |   |   return F
8  else if φ = ¬φ_1 then
9  |   return ¬getOrCompute(ρ, φ_1, i, v)
10 else if φ = φ_1 ∧ φ_2 then
11 |   return getOrCompute(ρ, φ_1, i, v) ∧
   |   getOrCompute(ρ, φ_2, i, v)
12 else
13 |   // We have φ = φ_1 U_I φ_2
14 |   c_1 ← false, c_11 ← true
15 |   c_2 ← true, c_22 ← false, c_3 ← t_n − t_i ≥ sup(I)
16 |   for j = i to n do
17 |   |   r_1 ← getOrCompute(ρ, φ_1, j, v)
18 |   |   r_2 ← getOrCompute(ρ, φ_2, j, v)
19 |   |   c_1 ← c_1 ∨ (r_2 = T ∧ t_j − t_i ∈ I ∧ c_11)
20 |   |   c_2 ← c_2 ∧ (r_2 = F ∨ t_j − t_i ∉ I ∨ c_22)
21 |   |   c_3 ← c_3 ∨ (r_1 = F)
22 |   |   c_11 ← c_11 ∧ (r_1 = T)
23 |   |   c_22 ← c_22 ∨ (r_1 = F)
24 |   if c_1 then
25 |   |   return T
26 |   else if c_2 ∧ c_3 then
27 |   |   return F
28 |   else
29 |   |   return U
```

Fourth, $c_{22} = \mathit{false}$. By line 23 and the IH we have that $\mathit{getOrCompute}(\rho, \phi_1, k) \neq F$ and thus $\rho_k \models^- \phi_1$ for all $i \leq k < m-1$. The iteration for $j = m-1$ can update c_{22} in two ways. First, $\mathit{getOrCompute}(\rho, \phi_1, m-1) \neq F$ in which case c_{22} stays false up to the start of iteration m. By the IH, $\rho_{m-1} \models^- \phi_1$ and thus $C_{22}(i,m) = \mathit{false}$. Second, $\mathit{getOrCompute}(\rho, \phi_1, m-1) = F$ in which case c_{22} becomes true at the start of iteration m. We thus have by the IH that $\rho_{m-1} \not\models^- \phi_1$ and clearly $C_{22}(i,m) = \mathit{true}$. ■

Invariant 2. *At the end of line 23 in Alg. 2 in loop iteration j it holds that if $\mathit{getOrCompute}(\rho, \psi, j, v) = T \Leftrightarrow \rho_j \models^+ \psi$ and $\mathit{getOrCompute}(\rho, \psi, j, v) = F \Leftrightarrow \rho_j \not\models^- \psi$ for both $\psi = \phi_1$ and $\psi = \phi_2$, then $c_1 = C_1(i,j)$, $c_2 = C_2(i,j)$ and $c_3 = C_3(i,j)$.*

Proof: Suppose that $c_1 = \mathit{false}$ at the end of line 23. In that case c_1 had been false before line 19. The IH gives us that $C_1(i, j-1) = \mathit{false}$. Furthermore, line 19 gives that $r_2 \neq T$ or $t_j - t_i \notin I$ or $c_{11} = \mathit{false}$. Combination with the IH and Inv. 1 makes that $C_1(i,j) = \mathit{false}$.

Suppose that $c_1 = \mathit{true}$ at the end of line 23. Than either c_1 holds before line 19, or $r_2 = T$ and $t_j - t_i \in I$ and $c_{11} = \mathit{true}$. In either case, combination with the IH and Inv. 1 makes that $C_1(i,j) = \mathit{true}$.

The proofs for c_2 and c_3 are similar. ■

Theorem 1. *Let ρ be a finite timed trace of length n, let ϕ be an MTL formula, and let $0 \leq i \leq n$. Then we have that $\mathit{getOrCompute}(\rho, \phi, i, v_0) = T \Leftrightarrow \rho_i \models^+ \phi$, and also that $\mathit{getOrCompute}(\rho, \phi, i, v_0) = F \Leftrightarrow \rho_i \not\models^- \phi$.*

Proof: We prove the theorem by induction to the structure of ϕ. It is straightforward to verify that the theorem holds for the leaf cases that ϕ equals true or is an atomic proposition.

($\models^+, \Rightarrow$) Suppose that $\mathit{getOrCompute}(\rho, \phi, i, v) = T$ and that the theorem holds for all subformulas of ϕ. The cases for *true* and the atomic proposition are straightforward. We then distinguish three cases. First, $\phi = \neg\phi_1$. Then line 9 has returned T. By definition, we thus have that $\mathit{getOrCompute}(\rho, \phi_1, i, v) = F$. By the IH, we then have $\rho_i \not\models^- \phi_1$, which proves this case. Second, $\phi = \phi_1 \wedge \phi_2$. In this case, line 11 has returned T. By definition, we thus have that $\mathit{getOrCompute}(\rho, \phi_1, i, v) = T$ and $\mathit{getOrCompute}(\rho, \phi_2, i, v) = T$. Using the IH and applying the definition of $\models^+$ proves this case. Third, $\phi = \phi_1 \mathbf{U}_I \phi_2$. In this case, line 25 has returned T, which implies that c_1 holds. Our IH enables us to apply Inv. 2 to conclude that $C_1(i,n)$ holds and thus that $\rho_i \models^+ \phi$.

($\models^+, \Leftarrow$) Suppose that $\rho_i \models^+ \phi$ and that the theorem holds for all subformulas. The cases fore *true* and the atomic proposition are straightforward. First, consider the case in which $\phi = \neg\phi_1$. Then we have that $\rho_i \not\models^- \phi_1$. By the IH, we then have that $\mathit{getOrCompute}(\rho, \phi_1, i, v) = F$. Therefore, line 9 of the algorithm returns G. Next, consider the case in which $\phi = \phi_1 \wedge \phi_2$. Then we have by that $\rho_i \models^+ \phi_1$ and $\rho_i \models^+ \phi_2$. By the IH, the checks in line 11 both return G, and by definition $G \wedge G = G$. Therefore, line 11 returns G. Finally, consider the case in which $\phi = \phi_1 \mathbf{U}_I \phi_2$. Then we have that $C_1(i,n)$ holds. Using the IH we can apply Inv. 2 to conclude that at then end of the loop $c_1 = \mathit{true}$. Therefore, G is returned in line 25.

($\models^-, \Rightarrow$) Suppose that $\mathit{getOrCompute}(\rho, \phi, i, v) = F$ and that the theorem holds for all subformulas of ϕ. The cases fore *true* and the atomic proposition are straightforward. We then distinguish three cases. First, $\phi = \neg\phi_1$. Then line 9 has returned F. By definition, we thus have that $\mathit{getOrCompute}(\rho, \phi_1, i, v) = T$. By the IH, we then have $\rho_i \models^+ \phi_1$, which by definition gives that $\rho_i \not\models^- \neg\phi_1$ which proves this case. Second, $\phi = \phi_1 \wedge \phi_2$. In this case, line 11 has returned F. By definition, we thus have that $\mathit{getOrCompute}(\rho, \phi_1, i, v) = B$ or $\mathit{getOrCompute}(\rho, \phi_2, i, v) = B$. Using the IH gives us that $\rho_i \not\models^- \phi_1$ or $\rho_i \not\models^- \phi_2$. Application of the definition of $\models^-$ for the $\wedge$ case then gives that $\rho_i \not\models^- \phi$. Third, $\phi = \phi_1 \mathbf{U}_I \phi_2$. In this case, line 27 has returned F. This means that both c_2 and c_3 held after the last iteration of the loop. By the IH and Inv. 2 we can conclude that both $C_2(i,n)$ and $C_3(i,n)$ hold, and therefore $\rho_i \not\models^- \phi$.

($\models^-, \Leftarrow$) Suppose that $\rho_i \not\models^- \phi$ and that the theorem holds for all subformulas. The cases fore *true* and the atomic proposition are straightforward. First, consider the case in which $\phi = \neg\phi_1$. Then we have that $\rho_i \models^+ \phi_1$. By the IH, we then have that $\mathit{getOrCompute}(\rho, \phi_1, i, v) = T$. Therefore, line 9 of the algorithm returns B. Next, consider the case in which $\phi = \phi_1 \wedge \phi_2$. Then we have by that $\rho_i \not\models^- \phi_1$ or $\rho_i \not\models^- \phi_2$. By the IH, at least one of the checks in line 11 returns B, and by definition line 11 thus returns B. Finally, consider the case in which $\phi = \phi_1 \mathbf{U}_I \phi_2$. Then we have that both $C_2(i,n)$ and $C_3(i,n)$ hold. Using the IH we can apply Inv. 2 to conclude that at then end of the loop $c_2 = \mathit{true}$ and $c_3 = \mathit{true}$. Therefore, B is returned in line 27 (note that we need that $C_2(i,n) \wedge C_3(i,n) \Rightarrow \neg C_1(i,n)$ which is equivalent to saying $\rho_i \not\models^- \phi \Rightarrow \rho_i \not\models^+ \phi$, which follows from Prop. 1). ■

Theorem 2. *Let ρ be a finite timed trace and let ϕ be an MTL formula. The time complexity of the call to $\mathit{getOrCompute}(\rho, \phi, 0, v_0)$ is $\mathcal{O}(|\phi| \cdot |\rho|^2)$, where $|\phi|$ denotes the size of the formula, and $|\rho|$ is the length of the trace.*

Proof: Let us consider the calltree of the initial call to $\mathit{getOrCompute}(\rho, \phi, i, v_0)$. Note that a $\mathit{getOrCompute}$ node has either no child, or a *compute* child. A *compute* node has between 0 and $2 \cdot |\rho|$ $\mathit{getOrCompute}$ children.

The number of calls to *compute* is not greater than $|\rho| \cdot |\phi|$, because once a table entry is computed, any later reference

uses the memoized value from the table (see Alg. 1). We thus have at most $|\rho| \cdot |\phi|$ *compute* nodes in the calltree.

Now consider the subtree which contains all *compute* nodes and in which every leaf is a *compute* node. Because of the regular structure we have that this subtree contains at most $|\rho| \cdot |\phi|$ *getOrCompute* nodes. From a leaf node (which is a *compute* node), there are at most $2 \cdot |\rho|$ calls to *getOrCompute* (in case of a temporal operator). These calls, however, do not result in further calls by construction (because these would necessarily be to *compute*). Thus, the number of *getOrCompute* nodes in the calltree is at most $|\rho| \cdot |\phi| \cdot 2 \cdot |\rho|$.

The table v can be implemented by a two-dimensional array, and we can, before the start of the main algorithm, annotate all subformulas with an appropriate index to access the array. We therefore assume constant-time read and write access to v. Now observe that the time complexity of a *getOrCompute* call, excluding potential further calls, is constant: $\mathcal{O}(1)$. The time complexity of a *compute* call, excluding potential further calls, is linear in the number of children: $\mathcal{O}(|\rho|)$. Therefore, the total time complexity equals $\mathcal{O}(2 \cdot |\phi| \cdot |\rho|^2 + |\phi| \cdot |\rho|^2) = \mathcal{O}(|\phi| \cdot |\rho|^2)$. ■

We have applied two optimizations to Alg. 2 to (i) improve the performance, and (ii) to reduce the number of computed entries in the lookup table v to make the explanation for the user more concise. The first optimization consists of lazy evaluation of conjunction arguments. When one of the conjuncts is F, then the other one does not need to be computed. When both conjuncts are unknown, we first compute the one with the smallest formula depth (based on the number of nested temporal operators). See Alg. 3.

The second optimization consists of bounding rules for the temporal operator. The evaluation of the temporal operator in Alg. 2 is written for clarity. Clearly, the fact that the loop always iterates to the end of the trace is not very efficient. We can use the following observations to short-circuit the loop in several cases:

- Once c_1 becomes *true*, it remains *true*. Therefore, the return statement guarded by c_1 can be pulled into the loop. Furthermore, the evaluation of r_2 can wait until after this if-then construct.
- Similarly, once c_{22} or c_3 becomes *true* it remains *true*. This also holds for $t_j - t_i > \mathit{sup}(I)$, which implies $t_j - t_i \notin I$. This gives that if at the end of the loop both c_2 and c_3 hold, and this is the last iteration or c_{22} holds or $t_j - t_i > \mathit{sup}(I)$ holds, we have that in the next iterations (if any), c_2 (and c_3) also hold, and we can return F.
- If c_1 is *false* now, and c_{11} is *false* or $t_j - t_i > \mathit{sup}(I)$ then c_1 will remain *false*. Furthermore, if c_2 is *false* then it will remain *false*. This implies that the cases to return T or F will not be satisfied, and we can return U.

Algorithm 3: Optimization of $\mathit{compute}(\rho, \phi_1 \wedge \phi_2, i, v)$

```
if v(φ1, i) =? ∧ v(φ2, i) =? then
    if depth(φ1) ≤ depth(φ2) then
        v(φ1, i) ← compute(ρ, φ1, i, v)
    else
        v(φ2, i) ← compute(ρ, φ2, i, v)
if v(φ1, i) = F ∨ v(φ2, i) = F then
    return F
if v(φ1, i) =? then
    v(φ1, i) ← compute(ρ, φ1, i, v)
else if v(φ2, i) =? then
    v(φ2, i) ← compute(ρ, φ2, i, v)
return v(φ1, i) ∧ v(φ2, i)
```

Algorithm 4: Optimization of $\mathit{compute}(\rho, \phi_1 \mathbf{U}_I \phi_2, i, v)$

```
c1 ← false, c11 ← true
c2 ← true, c22 ← false, c3 ← t_n − t_i ≥ sup(I)
for j = i to n do
    r2 ← getOrCompute(ρ, φ2, j, v)
    c1 ← c1 ∨ (r2 = T ∧ t_j − t_i ∈ I ∧ c11)
    if c1 then
        return T
    r1 ← getOrCompute(ρ, φ1, j, v)
    c2 ← c2 ∧ (r2 = F ∨ t_j − t_i ∉ I ∨ c22)
    c3 ← c3 ∨ (r1 = F)
    c11 ← c11 ∧ (r1 = T)
    c22 ← c22 ∨ (r1 = F)
    if (j = n ∨ c22 ∨ t_j − t_i ≥ sup(I)) ∧ c2 ∧ c3 then
        return F
    if (¬c11 ∨ t_j − t_i ≥ sup(I)) ∧ ¬c1 ∧ ¬c2 then
        return U
return U
```

Algorithm 4 shows this optimization.

V. Implementation in Trace

A. The Trace tool

The Trace tool [1] is a generic visualization tool for activities and their resource usage. It is centered around the concept of a *claim*, which represents an activity on a single resource. A claim has a start and end time stamp, a resource, and a number of configurable attributes (e.g., the name of the activity), which are represented by a key-value mapping of attribute names to attribute values. A set of claims is visualized by Trace as a Gantt chart. The tool offers many features, e.g., filtering, zooming, grouping, coloring based on attribute values, and to do analysis such as critical-path analysis and difference analysis [2].

B. MTL in TRACE

To specify MTL formulas in the TRACE context, we first define both states and atomic propositions as key-value mappings. We say that a state defined by mapping M_1 satisfies atomic propositions defined by mapping M_2 if and only if $M_2 \subseteq M_1$. Second, we convert a claim c with start time-stamp t_s and end time-stamp t_e to two state-time tuples:

- $(m \cup \{mtl \mapsto s\}, t_s)$ and
- $(m \cup \{mtl \mapsto e\}, t_e)$,

where m is the key-value mapping given by the attributes of the claim and its values. Note that the additional *mtl* element in the mapping is used to indicate the start or the end of the claim. Third, we order the set of all state-time tuples associated with a set of claims according to their time-stamps, which gives a timed trace. Note, however, that we arbitrarily order events with identical time-stamps even though different orderings are generally distinguishable by MTL formulas. This is a consequence of the fact that the TRACE input format does not support information on the order of events with an identical time-stamp.

Besides the core constructs of MTL given in Sec. II, we have added the logical operators *or* and *implies* and the temporal operators *globally* (represented by `G`) and *finally* (represented by `F`) for convenience to the expression language. Furthermore, we have added a top-level conjunction construct in combination with integer expressions to be able to conveniently specify a set of formulas. An example that contains many of these ingredients is Prop. 5 in Ex. 1, which looks as follows in the syntax of the TRACE MTL expression language:

```
/\(i=0...989)
  G (
      {name=G,id=i,mtl=e}
            =>
       F_[0,65] {name=G,id=(i+10),mtl=e}
    )
```

The top-level conjunction instantiates the following formula (which starts with element $\mathbf{G}$) for every value of $0 \leq i \leq 989$. It thus creates 990 formulas which express that the distance between the end of activity G with id i and the end of activity G with id $i + 10$ is at most 65 for all $0 \leq i \leq 989$. This construct is very convenient when claims are associated, e.g., with an object identifier, image number, etc.

C. User interface

Figure 2 shows the ECLIPSE IDE with the TRACE plugin installed. The window has (1) a project explorer view of the files in the workspace, (2) a number of TRACE toolbar items, (3) the main Gantt-chart view, (4) the MTL explanation view, and (5) a concrete explanation of the property being analysed overlayed on the Gantt-chart view. In this case, the Gantt chart visualizes a (part of a) run of the system from Ex. 1 for 1000 objects while Prop. 7 is being analysed.

The project explorer associates files with an *mtl* extension with the MTL dialog. Double clicking an mtl file when a trace is open, opens the MTL dialog with the contents of the *mtl* file. The MTL dialog has several configuration options: (i) whether to apply it to the set of filtered claims or to the whole set of claims, (ii) whether to interpret the trace as a prefix or not, (iii) whether to generate explanations of computed values. If the OK button of the MTL dialog is pressed, the MTL specification is checked against the current trace.

We generate explanations in two forms. First, the claims that are relevant for the truth value of the formula can be highlighted. This is a rather straightforward visualization based on a marking of states and their claims during the run of the algorithm. Nevertheless, it is often very useful and allows us to zoom into relevant parts of the trace quickly for diagnosis. The second form consists of an annotation of (part of) the time axis with the truth values of all subformulas of the formula that is checked. Also this annotation is constructed on-the-fly during the run of the algorithm. This annotation allows the user to trace the result according to the semantics. Figure 2 shows the user interface after checking Prop. 7 and after visualizing the second type of explanation. Below the time axis are the three subformulas of the implication. A red bar means that the property is not satisfied in any state in that time interval, and a green bar means that it is satisfied. A blue bar indicates that the property may or may not be satisfied by an arbitrary extension of this prefix. For this property, the key is that the implication `{name=G,mtl=e} => F_[3,15]{name=G,mtl=e}` holds for every end event of G but the last one. However, an extension of the trace could have more end events of task G within the indicated interval. Therefore, `F_[3,15]{name=G,mtl=e}` may or may not be satisfied by the last state in the prefix, hence the blue marking.

D. Performance and scalability

Despite the time complexity of $\mathcal{O}(|\phi| \cdot |\rho|^2)$ we believe that the algorithm is practically feasible. For instance, Props. 1–3 and 7 can be checked for a trace of $25,000$ objects (approximately 10^6 states) within a second on a regular laptop computer. (Note that the trace is an informative good prefix for Props. 1 and 2, and non-informative for Props. 3 and 7.) Properties 4–6 can be checked in approximately 15 seconds for a trace of 2500 objects (approximately 10^5 states). The 2500 separate formulas per property are checked in parallel by our implementation using all available CPU cores.

Finally, we note that the worst-case quadratic complexity of our algorithm is not observed for the properties above, but it can be observed when checking formulas such as the following pathological example: $\mathbf{GGGG}true$.

Figure 2: A TRACE view in the ECLIPSE Integrated Development Environment.

VI. CONCLUSIONS

Metric Temporal Logic provides a flexible mechanism to specify quantitative properties of execution traces. We have shown how we have added an MTL checking capability to the TRACE tool using a recursive memoization algorithm that can generate concise explanations of the truth value of the given MTL formula. These explanations can be visualized in the TRACE viewer to aid the user in understanding. The combination of a generic and flexible visualization layer for execution traces and non-trivial, domain independent, analysis capabilities such as critical-path analysis, trace difference analysis and now MTL checking, that the TRACE tool provides, is something we have not seen elsewhere. The TRACE tool is available through http://trace.esi.nl/.

ACKNOWLEDGEMENT

This research is supported by the ARTEMIS joint undertaking under grant agreement no 621439 (ALMARVI).

REFERENCES

[1] Embedded Systems Innovation by TNO, "TRACE website," http://trace.esi.nl.

[2] M. Hendriks, J. Verriet, T. Basten, B. Theelen, M. Brassé, and L. Somers, "Analyzing execution traces – critical-path analysis and distance analysis," *Submitted to STTT*, 2015.

[3] R. Alur and T. Henzinger, "Real-time logics: complexity and expressiveness," *Information and Computation*, vol. 104, pp. 390–401, 1993.

[4] Atlassian, "Gantt plugins for JIRA," https://marketplace.atlassian.com/search?application=jira&q=gantt.

[5] National Instruments, "Profiling VI Execution With the LabVIEW Desktop Execution Trace Toolkit," http://www.ni.com/white-paper/8083/en/.

[6] G. Behrmann et al., "Uppaal 4.0," in *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, ser. QEST '06. IEEE Computer Society, 2006.

[7] Intel, "Intel VTune Amplifier 2016," https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[8] Microsoft, "Windows Performance Analyzer," https://msdn.microsoft.com/en-us/library/hh448170.aspx.

[9] R. Alur, T. Feder, and T. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM*, vol. 43, no. 1, Jan. 1996.

[10] M. Geilen, "An improved on-the-fly tableau construction for a real-time temporal logic," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 2725. Springer Berlin Heidelberg, 2003.

[11] D. Ničković and N. Piterman, "From mtl to deterministic timed automata," in *Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems*, vol. 6246. Springer-Verlag, 2010.

[12] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime assurance based on formal specifications," in *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[13] D. Drusinsky, "The temporal rover and the atg rover," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, 2000, pp. 323–330.

[14] K. Havelund and G. Roşu, "Testing linear temporal logic formulae on finite execution traces," Tech. Rep., 2001.

[15] P. Thati and G. Roşu, "Monitoring algorithms for metric temporal logic specifications," *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 145–162, 2005.

[16] H. Ho, J. Ouaknine, and J. Worrell, "Online monitoring of metric temporal logic," in *Runtime Verification*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8734.

[17] N. Markey and J.-F. Raskin, "Model checking restricted sets of timed paths," *Theoretical Computer Science*, vol. 358, no. 2, 2006.

[18] Hendriks, M. et al., "A blueprint for system-level performance modeling of software-intensive embedded systems," *International Journal on Software Tools for Technology Transfer*, 2014.

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

[20] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout, "Reasoning with temporal logic on truncated paths," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2725.