Efficient Large-scale Trace Checking Using MapReduce

Marcello M. Bersani¹, Domenico Bianculli², Carlo Ghezzi¹, Srđan Krstić¹ and Pierluigi San Pietro¹
¹DEEPS group - DEIB - Politecnico di Milano, Milano, Italy
²SnT Centre - University of Luxembourg, Luxembourg, Luxembourg
marcellomaria.bersani@polimi.it, domenico.bianculli@uni.lu, carlo.ghezzi@polimi.it, srdan.krstic@polimi.it, pierluigi.sanpietro@polimi.it

ABSTRACT
The problem of checking a logged event trace against a temporal logic specification arises in many practical cases. Unfortunately, known algorithms for an expressive logic like MTL (Metric Temporal Logic) do not scale with respect to two crucial dimensions: the length of the trace and the size of the time interval of the formula to be checked. The former issue can be addressed by distributed and parallel trace checking algorithms that can take advantage of modern cloud computing and programming frameworks like MapReduce. Still, the latter issue remains open with current state-of-the-art approaches.

In this paper we address this memory scalability issue by proposing a new semantics for MTL, called lazy semantics. This semantics can evaluate temporal formulae and boolean combinations of temporal-only formulae at any arbitrary time instant. We prove that lazy semantics is more expressive than point-based semantics and that it can be used as a basis for a correct parametric decomposition of any MTL formula into an equivalent one with smaller, bounded time intervals. We use lazy semantics to extend our previous distributed trace checking algorithm for MTL. The evaluation shows that the proposed algorithm can check formulae with large intervals, on large traces, in a memory-efficient way.

1. INTRODUCTION
Software systems have become more complex, distributed, and increasingly reliant on third-party functionality. The dynamic behavior of such systems makes traditional design-time verification approaches unfeasible, because they cannot analyze all the behaviors that can emerge at run time. For this reason, techniques like run-time verification and trace checking have become viable alternative for the verification of modern systems. While run-time verification checks the behavior of a system during its execution, trace checking is a post-mortem technique. In other words, to perform trace checking one must first collect and store relevant execution data (called execution traces or logs) produced by the system and then check them offline against the system specifications. This activity is often done to inspect server logs, crash reports, and test traces, in order to analyze problems encountered at run time. More precisely, trace checking is an automatic procedure for evaluating a formal specification over a trace of recorded events produced by a system. The output of the procedure is called verdict and states whether the system’s behavior conforms to its formal specification.

The volume of the execution traces gathered for modern systems increases continuously as systems become more and more complex. For example, an hourly page traffic statistics for Wikipedia articles collected over a period of seven months amounts to 320GB of data [27]. This huge volume of trace data challenges the scalability of current trace checking tools [7, 16, 18, 25, 26], which are centralized and use sequential algorithms to process the trace. One possible way to efficiently perform trace checking over large traces is to use a distributed and parallel algorithm, as done in [3, 5] and also in our previous work [10]. These approaches rely on the MapReduce framework [14] to handle the processing of large traces. MapReduce is a programming model and an underlying execution framework for parallel and distributed processing of large quantities of data stored on a cluster of different interconnected machines (or nodes). In [10] we proposed a MapReduce algorithm that checks very large execution traces against formal specifications expressed in metric temporal logic (MTL); the algorithm exploits the structure of the formula to parallelize its evaluation.

MTL [19] is a class of temporal logic used for the specification and verification of real-time systems. It extends the well-known “Until” temporal operator of the classic LTL with an interval that indicates the time distance within which the formula must hold. For example, the property “A covered entity […] must retain the documentation […] for 6 years from the date of its creation.” is a simplified version of a policy taken from the US HIPAA [24]. It can be expressed, for a particular document, as: G(create → (¬delete U[6 years] delete), where the operator U (called “Until”) states that its right operand, the delete event, must occur in exactly six years (i.e., 180.2 billion ms, assuming a millisecond granularity in the log) from the moment of creation (expressed with the event create). It also states that the left operand (¬delete) must hold until that happens. Operator G (called “Globally”) states that property holds over the whole

¹Work partially funded by National Research Fund, Luxembourg (FNR/P10/03); EC grant no. 644869 (EU H2020), DICE; Italian PRIN Project 2010LYA9RH_006.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884832

¹Also called trace validation [23] or history checking [17].
trace. In this logic, time can be expressed using either integer or real time-stamps. MTL specifications may express properties that refer to different parts of the trace or to large portions of the trace at once by using large time intervals. In the example above, to check if the “Until” subformula holds in a single position of the trace, the algorithm needs to consider a portion of the trace corresponding, in the worst case, to six years of logged data. To check the whole formula, this process needs to be performed for every position in the trace because of the outer “Globally” operator. Generally speaking, trace checking algorithms scan a trace and buffer the events that satisfy the temporal constraints of the formula. The buffer is incrementally updated as the trace is scanned and the algorithms incrementally provide verdicts for the positions for which they have enough information (to determine the verdict). The lower-bound for memory complexity of trace checking algorithms is known to be exponential in the numeric constants occurring in the MTL formula encoded in binary [26]. Therefore the strategy of buffering events creates a memory scalability issue for trace checking algorithms. This issue also affects distributed and parallel solutions, including our previous work [10]. Specifically, the memory scalability of a trace checking algorithm on a single cluster node depends exponentially on the numeric constants defining the bounds of the time intervals in the MTL formula to be checked.

The goal of this paper is to address this memory scalability issue by proposing a trace checking algorithm that exploits a new semantics for MTL, called lazy semantics. Unlike traditional point-based semantics [19], our lazy semantics can evaluate both temporal formulae and boolean combinations of temporal-only formulae at any arbitrary time instant, while it evaluates atomic propositions only at time-stamped positions of the trace. We propose lazy semantics because it possesses certain properties that allow us to decompose any MTL formula into an equivalent MTL formula where the upper bound of all time intervals of its temporal operators is limited by some constant. This decomposition plays a major role in the context of (distributed) trace checking of formulae with large time intervals. In practice, if we want to check a formula with a large time interval, applying the decomposition entails an equivalent formula, with smaller time intervals. This new formula can be checked in a more memory efficient way by using our new trace checking algorithm, which applies lazy semantics.

We show that lazy semantics does not hinder the expressive power of MTL: we prove that MTL interpreted over lazy semantics is strictly more expressive than MTL interpreted over point-based semantics. In other words, any MTL formula interpreted over point-based semantics can be rewritten using an MTL formula interpreted over lazy semantics. Moreover, there are MTL formulae interpreted over lazy semantics that do not have an equivalent formula that can be interpreted over point-based semantics. We have integrated lazy semantics and the modified distributed trace checking algorithm into our MTLMapReduce tool [20], implemented using the Apache Spark framework. The evaluation shows that the proposed approach can be used to check formulae with very large time intervals, on very large traces, while keeping a low memory footprint. This footprint is compatible with the available configuration of common cloud instances. Moreover, our tool performs better, in terms of memory scalability, than our previous implementation [10].

We have also assessed the time and memory tradeoffs of the algorithm with respect to the decomposition parameter.

In summary, the specific contributions of this paper are: 1) A new semantics for MTL, called lazy semantics; we prove that it is strictly more expressive than point-based semantics. 2) A parametric decomposition of MTL formulae into MTL formulae where the upper bound of all time intervals is limited by some constant; 3) A new trace checking algorithm that exploits lazy semantics and parametric decomposition, to check MTL formulae in a memory-efficient way; 4) The evaluation of the proposed algorithm in terms of memory scalability and time/memory tradeoffs.

The rest of the paper is structured as follows. Section 2 briefly introduces MTL interpreted over point-based semantics and the MapReduce programming model. Section 3 overviews our approach and motivates the need for lazy semantics and the parametric decomposition of MTL formulae. Lazy semantics is introduced in Section 4. Section 5 details the parametric decomposition of MTL formulae. Section 6 introduces our distributed trace checking algorithm that supports lazy semantics. Section 7 reports on the evaluation of our implementation. Section 8 surveys related work, while Section 9 concludes the paper.

2. PRELIMINARIES

2.1 Point-based Semantics for MTL

Let $I$ be any non-empty interval over $\mathbb{R}$ with endpoints in $\mathbb{N}$ and let $\Pi$ be a finite set of atomic propositions (or atoms). The syntax of MTL is defined by the following grammar, where $p \in \Pi$ and $U_{t}$ is the metric “Until” operator: $\phi ::= p \mid \neg \phi \mid \phi \lor \phi \mid U_{t} \phi \phi$. Additional boolean and temporal operators can be derived using the usual conventions: “Eventually” is defined as $F_{t} \phi \equiv U_{\omega} \phi$; “Globally” is defined as $G_{t} \phi \equiv \neg F_{t} \neg \phi$. We adopt the convention that an interval of the form $[i, i]$ is written as “$= i$”. The interval $[0, +\infty)$ in temporal operators is omitted for simplicity. We introduce the following shorthand notation: $F_{t}^{K} \phi \equiv F_{t} \cdots F_{t} \phi$, with $K$ times.

F_{t}^{K} \phi = \phi$. Hereafter we refer to point-based semantics for MTL as MTL$_{p}$ semantics.

**MTL$_{p}$ semantics.** We focus on the finite-word semantics of MTL, since we apply it to the problem of trace checking. A timed sequence $\sigma$, of length $|\sigma| > 0$, is a sequence $\sigma_{0} \ldots \sigma_{|\sigma|-1}$ of values $\tau \in \mathbb{R}$ such that $0 < \tau_{i} < \tau_{i+1}$ for each $0 \leq i < |\sigma| - 1$, i.e., the sequence is strictly monotonic. A word $\sigma$ over the alphabet $2^{\Pi}$ is a sequence $\sigma_{0} \sigma_{1} \ldots \sigma_{|\sigma|-1}$ such that $\sigma_{i} \in 2^{\Pi}$ for all $0 \leq i < |\sigma|$, where $|\sigma|$ denotes the length of the word. A timed word $[1] \omega = \omega_{0} \ldots \omega_{|\omega|-1}$ is a word over $2^{\Pi} \times \mathbb{R}$, i.e., a sequence of pairs $\omega_{i} = (\sigma_{i}, \tau_{i})$, where $\sigma_{0} \ldots \sigma_{|\omega|-1}$ is a word over $2^{\Pi}$ and $\tau_{0} \ldots \tau_{|\omega|-1}$ is a timed sequence. A pair $\omega_{i}$ is also called an element of the timed word. Moreover, notice that in this definition $i$ refers to a particular position of the element $\omega_{i}$ in the timed word $\omega$, while $\tau_{i}$ refers to the time instant or time-stamp of the element $\omega_{i}$. We abuse the notation and represent a timed word equivalently as a pair containing a word and a timed sequence of the same length, i.e., $\omega = (\sigma, \tau)$. A timed language over $2^{\Pi}$ is a set of timed words over $2^{\Pi}$. MTL$_{p}$ semantics on timed words is given in Figure 1, where the point-based satisfaction relation $|=_{p}$ is defined with respect to a timed word $(\sigma, \tau)$, a position $i \in N$, atom $p \in \Pi$, and MTL
multiple MapReduce calls can be chained together to perform available in several files, one for each used reducer. Multi-

 reductions and the key of the intermediate data. Each re-

ducer executes the reduce function, which produces the out-
put data. This output is appended to a final output file for
this reduce partition. The output of the MapReduce job is
available in several files, one for each used reducer. Mul-

tiple MapReduce calls can be chained together to perform
complex data processing.

\[(\sigma, \tau, i) \models p \land p \in \sigma, \text{ for } p \in \Pi\]
\[(\sigma, \tau, i) \models p \land \phi \land (\sigma, \tau, i) \not\models p \land \phi\]
\[(\sigma, \tau, i) \models p \land \phi \lor \psi \land (\sigma, \tau, i) \not\models p \land \phi \lor (\sigma, \tau, i) \models p \land \psi\]
\[(\sigma, \tau, i) \models p \land \phi \lor \psi \land (\sigma, \tau, j) \not\models i < j < \sigma \land \tau_{j} - \tau_{i} \in I \land (\sigma, \tau, j) \models p \land \psi \land \forall k.(i < k < j) \text{ then } (\sigma, \tau, k) \not\models p \land \phi)\]

Figure 1: MTL semantics on time words.

3. MOTIVATION AND OVERVIEW OF THE APPROACH

As mentioned in Section 1, trace checking is an automatic procedure for evaluating a formal specification over a trace of recorded events produced by a system. Since traces can be seen as a sequence of time-stamped elements (where each element records one or more events), we use timed words as abstract models of traces. Hence, a pair \(\omega_{i} = (\sigma, \tau)\) corresponds to the \(i\)-th element of the trace, where atoms in \(\sigma\) represent all the events with time-stamp \(\tau\).

Trace checking algorithms handle metric temporal operators by buffering elements of the trace. The time interval specified in the metric temporal formula to be checked determines the portion of the trace to be considered for emitting a verdict in a single position of the trace. Depending on the particular MTL formula to be checked, in the worst case this process needs to be repeated for every position in the trace.

What trace checking algorithms typically do is to keep the relevant portion of the trace in a buffer as they scan the trace. The buffer is updated incrementally while the algorithm scans and produces verdicts for the next elements in the trace. The procedure for updating the buffer consists of adding a newly-scanned element \(e\) of the trace and removing the elements whose time-stamps do not satisfy the temporal constraint of the formula to be checked, when evaluated with respect to the time-stamp of \(e\). Buffering elements presents a memory scalability issue if a metric temporal formula with a large time interval needs to be checked. Let us present an example to motivate the need for lazy semantics.

Example 1. Consider formula \(\Phi = F_{[3,7]}(p)\) and its evaluation on the following trace (represented as a timed word):

\[
\begin{align*}
\{p\}, 1 & \quad \{p\}, 2 & \quad \{q\}, 4 & \quad \{p, q\}, 6 & \quad \{p, q\}, 8 & \quad \{q\}, 9 & \quad \{q\}, 10
\end{align*}
\]

The timed word, shown in Figure 2, is defined over the set of atoms \(\Pi = \{p, q\}\); its length is 7 and it spans over 10 time units. The first two rows in the picture represent its atoms and time-stamps; the last two rows show, respectively, the evaluation of subformula \(p\) and formula \(F_{[3,7]}(p)\) using point-based semantics. As shown in the last row of Figure 2, according to point-based semantics, formula \(F_{[3,7]}(p)\) holds at time instants 1, 2 and 4.

For a formula of the form \(F_{[a,b]}(p)\), the algorithm needs to buffer, in the worst case (i.e., in case there exists an element at every time instant), at most \(b + 1\) elements. For example, to evaluate formula \(F_{[3,7]}(p)\) at time instant 2, in the worst case the algorithm will buffer 8 elements, i.e., all the elements whose time-stamp ranges from 2 to 9. The elements

\[\begin{array}{ccccccccc}
\text{Atoms:} & \{p\} & \{p\} & \{q\} & \{p, q\} & \{p, q\} & \{q\} & \{q\} \\
\text{Time-stamps:} & 1 & 2 & 4 & 6 & 8 & 9 & 10 \\
\text{Time instants:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}\]

Figure 2: Evaluation of formula \(\Phi = F_{[3,7]}(p)\).
with time-stamps ranging from 6 to 9 satisfy the time interval constraint of the formula; the others are kept for the evaluation of the formula at subsequent positions. Let us assume that the execution infrastructure could only store 5 elements in the buffer, because of the limited available memory. The worst-case requirement of keeping 8 elements in the buffer would then be too demanding for the infrastructure, in terms of memory scalability. To lower the memory requirement for the buffer we would need a formula with a smaller time interval and expressing the same property as $\Phi$. In other words, one might ask whether there is an MTL formula equivalent to $\Phi$ with all the intervals bounded by the constant 4 (and thus requiring to store at most $4 + 1 = 5$ elements in the buffer).

Let us consider formula $\Phi' = F_{[3,4]}(p) \lor F_{[4,5]}(F_{[0,3]}(p))$: a naïve and intuitive interpretation might lead us to think that it defines the same property as $\Phi$. Roughly speaking, instead of checking if $p$ eventually occurs within the entire [3, 7] time interval, $\Phi'$ checks if $p$ either occurs in the interval [3, 4] (as specified by subformula $F_{[3,4]}(p)$) or in the interval [0, 3] when evaluated exactly 4 time instants in the future (as specified by subformula $F_{[4,5]}(F_{[0,3]}(p))$). Figure 3 shows the evaluation of formula $\Phi'$ over the same trace used in Figure 2. As you can see, formula $\Phi'$ does not have the same evaluation as $\Phi$ on the same trace. More specifically, at time instant 1 $\Phi'$ is false while $\Phi$ is true (see the values circled in both figures). By analyzing the evaluation of $\Phi'$, one can notice that subformula $F_{[4,5]}(F_{[0,3]}(p))$ at time instant 1 refers to the value of $F_{[0,3]}(p)$ at time instant 5, which does not have a corresponding element in the trace. If there was an element at time instant 5, $F_{[0,3]}(p)$ would be true since $p$ holds at time instant 6.

The above example shows that the evaluation of temporal subformulae according to point-based semantics depends on the existence of certain elements in the trace. It also shows that point-based semantics is not suitable to support the intuitive decomposition of MTL formulae into equivalent ones with smaller time intervals, like the one from $\Phi$ to $\Phi'$ shown above. We maintain that this constitutes a limitation for the application of point-based semantics in the context of trace checking. Therefore, in this paper we propose a new, alternative semantics for MTL, called lazy semantics.

The main feature of lazy semantics is that it evaluates temporal formulae and boolean combinations of temporal-only formulae at any arbitrary time instant, regardless of the existence of the corresponding elements in the trace. The existence of the elements is only required when evaluating atoms. This feature allows us to decompose any MTL formula into an equivalent MTL formula in which the upper bound of all time intervals of its temporal operators is limited by some constant. Such a decomposition can be used as a preprocessing step of a trace checking algorithm, which can then run in a more memory-efficient way.

In the following sections we first introduce lazy semantics (Section 4) and formalize the notion of the decomposition exemplified above (Section 5). Afterwards, in Section 6 we describe the modifications to our previous trace checking algorithm [10], required to preprocess the formula and support lazy semantics.

### 4. LAZY SEMANTICS FOR MTL

The following example shows an anomalous case of MTL$^p$ semantics that lazy semantics for MTL (denoted as MTL$_L$ semantics) intends to remedy. Consider a timed word $w = ((\sigma, \tau), |\sigma|)$, $|\sigma| = 10$, $(\sigma, \tau) = \{(q_1), (p_1), (q_2), (p_2), (q_3), (p_3), (q_4), (p_4), (q_5), (p_5), (q_6), (p_6), (q_7), (p_7), (q_8), (p_8), (q_9), (p_9), (q_{10}), (p_{10})\}$. The intuitive meaning of the two formulae is the same: $p$ holds 6 time units after the origin, i.e., at time-stamp 7. However, when evaluated on $w$ using the MTL$^p$ semantics, the two formulae have different values: $\varphi_1$ correctly evaluates to true, but $\varphi_2$ to false. Indeed, in $\varphi_2$ the outermost $F_{=3}$ subformula is trivially false, because there is no position that is exactly 3 time instants in the future with respect to the origin. The two formulae, instead, are equivalent over the MTL$_L$ semantics, where they both evaluate to true. This, indeed, is true also over signal-based semantics [12]; however, signals are not very practical for monitoring and trace checking, which operate on logs that are best modeled as a sequence of individual time-stamped observations, i.e., timed words.

**MTL$_L$ semantics.** MTL$_L$ semantics on timed words is given in Figure 4, in terms of the satisfaction relation $\models_L$, with respect to a timed word $(\sigma, \tau)$ and a time instant $t \in \mathbb{R}^+$; $p$ is an atom and $\phi$ and $\psi$ are MTL formulae. An MTL formula $\phi$, when interpreted over MTL$_L$ semantics, defines a timed language $L_L(\phi) = \{(\sigma, \tau) | (\sigma, \tau, 0) \models_L \phi\}$. The main difference between MTL$^p$ and MTL$_L$ semantics is that MTL$^p$ evaluates formulae only at positions $i$ of a timed word, while MTL$_L$ inherits a feature of signal-based semantics, namely it may evaluate (non-atomic) formulae at any possible time instant $t$, even if there is no time-stamp equal to $t$. For example, according to the MTL$^p$ semantics, an “$\Until$” formula $\phi \equiv \psi_1 \lor \psi_2$ evaluates to false in case there are no positions in the interval $I$, due to the existential quantification on $\sigma$ (see Figure 1). Conversely, over the MTL$_L$ semantics, the evaluation of $\phi$ depends on the evaluation of $\psi_2$. If the latter is an atom then formula $\phi$ also evaluates to false, because of the existential quantifier in the MTL$_L$ semantics of atoms. However, if $\psi_2$ is a temporal formula or a boolean combination of temporal-only formulae (e.g., other “$\Until$” formulae), it will be evaluated in the part of the timed word that satisfies the interval of $\phi$. Hereafter we refer to the MTL formulae interpreted over the MTL$_L$
semantics as “MTL₁ formulae”; similarly, “MTLₚ formulae” are MTL formulae interpreted over the MTLₚ semantics.

Let M(Π) be the set of all formulae that can be derived from the MTL grammar shown in Section 2.1, using Π as the set of atoms. We show that any language Lₚ(ϕ) defined using some MTLₚ formula ϕ can be defined using an MTL₁ formula obtained after applying the translation 1p₂ : M(Π) → M(Π) to ϕ, i.e., Lₚ(ϕ) = Lₚ₂(Lₚ(ϕ)) for any ϕ. The 1p₂ translation is defined as follows:

$$1p₂(ϕ) ≡ p, p ∈ Π; \quad 1p₂(ϕ \lor ψ) ≡ 1p₂(ϕ) \lor 1p₂(ψ); \quad 1p₂(ϕ) ≡ \neg 1p₂(ϕ); \quad 1p₂(ϕ) ≡ 1p₂(ϕ) \cup_l (ϕ_{act} \land 1p₂(ϕ))$$

where $ϕ_{act} ≡ a \land \neg a$ for some $a ∈ Π$.

The goal of 1p₂ is to prevent the occurrence of direct nesting of temporal operators, i.e., to avoid the presence of (sub)formulae like $F₁₃F₄₅₆$. As discussed in the example above, nested temporal operators are interpreted differently over the two semantics. Direct nesting is avoided by rewriting the right argument of every “Until” (i.e., the “extensional component of “Until”). The argument is conjoined with a formula $ϕ_{act}$ that evaluates to true (over both semantics) if there exists a position in the underlying timed word; otherwise $ϕ_{act}$ evaluates to false. To explain this intuition, let us evaluate $ϕ_{act}$ over a timed word $(σ, τ, i)$ over the alphabet $Π = \{a\}$. Over point-based semantics, $(σ, τ, i) \models ϕ_{act} ≡ σ(i) \models ϕ \lor \neg a$ is true for any position $i$, since either $a$ belongs to $σ$ or not. However, the same does not hold for lazy semantics. According to lazy semantics, $(σ, τ, i) \models Lₚ ϕ_{act}$ is true only in those time instants $t$ for which there exists $i$ such that $τ = t$ and therefore exists the corresponding $σ(i)$ (to which $a$ can belong or not).

**Lemma 1.** Given an MTL formula $ϕ$ and a timed word $ω = (σ, τ)$, for any $i ≥ 0$, the following equivalence (modulo 1p₂ translation) holds: $(σ, τ, i) \models Lₚ ϕ$ iff $(σ, τ, i) \models Lₚ₂ Lₚ(ϕ)$.

**Proof.** See the extended version of [9] of the paper.

**Theorem 1.** Any timed language defined by an MTLₚ formula can be defined by an MTL₁ formula over the same alphabet.

**Proof.** By Lemma 1, for $i = 0$.

Notice that the translation $1p₂$ defines a syntactic MTL fragment where temporal or boolean combination of temporally only operators cannot be directly nested. In this fragment MTLₚ and MTL₁ formulae define the same languages. However, if we consider the complete definition of MTL, without syntactic restrictions, the class of timed languages defined by MTL₁ formulae strictly includes the class of languages defined by MTLₚ formulae. In other words, MTL interpreted over lazy semantics is strictly more expressive than MTL interpreted over point-based semantics; this result is established by the following theorem.

**Theorem 2.** There exists a timed language defined by some MTL₁ formula that cannot be defined by any MTLₚ formula.

**Proof.** Consider the language of timed words $L = \{(σ, τ) \mid \exists j(i ≤ j \land (σ, τ, i) \models L \land (σ, τ, j) \models L \land τ_j ≤ 2)\}$. $L$ is defined by the MTL₁ formula $ω = Φ₁ \lor Φ₂ \lor Φ₃$ where $Φ₁ = (F₀(0) \land b) \lor(F₀(0) \land \langle b, k \rangle) \lor(F₀(0) \land \langle b, k \rangle) \lor(F₀(0) \land \langle b, k \rangle)$ and $Φ₂ = F₀(0) \land F₀(0)$ and $Φ₃ = F₀(0) \land F₀(0)$ and $F₀(0) \land F₀(0)$ and $F₀(0) \land F₀(0)$. $L$ cannot be defined by any MTLₚ formula (see reference [12], proposition 6).

**5. PARAMETRIC DECOMPOSITION**

In this section we show that lazy semantics allows for a parametric decomposition of MTL formulae into MTL formulae where the upper bound of all intervals of the temporal operators is limited by some constant $K$ (the parameter of the decomposition). This structural characteristic will then be used in the trace checking algorithm presented thereafter.

We first introduce some notation and show some properties of lazy semantics that will be used to prove the correctness of the decomposition. We define the operator $\oplus$ over intervals in $\mathbb{R}$ with endpoints in $\mathbb{N}$ such that $I \oplus J = \{i + j \mid \forall i \in I \land j \in J\}$.

**Lemma 2.** For any timed word $(σ, τ)$ and $t ≥ 0$,

$$(σ, τ, t) \models Lₚ F_j F_j ϕ \iff (σ, τ, t) \models Lₚ F_j \oplus J, ϕ$$

**Corollary 1.** For any timed word $(σ, τ)$ and $T ≥ 0$,

$$(σ, τ, t) \models Lₚ Fₜ \lor Fₜ ϕ \iff (σ, τ, t) \models Lₚ Fₜ \oplus T, ϕ$$

**Lemma 3.** For any timed word $(σ, τ)$ and $t ≥ 0$,

$$(σ, τ, t) \models Lₚ Fₜ \lor Fₜ ϕ \iff (σ, τ, t) \models Lₚ Fₜ \oplus T, ϕ \iff I \cap J ≠ \emptyset$$

The proofs of the above corollary and lemmata are in the extended version [9] of the paper.

Hereafter, we focus on bounded MTL formulae, i.e., formulae where intervals are always finite. Notice that it is this class of formulae that causes memory scalability issues in trace checking algorithms. We present the parametric decomposition by referring to the bounded “Eventually” operator. The bounded “Until” and “Globally” operators can be expressed in terms of the bounded “Eventually” operator using the usual equivalences; moreover, we remark that the decomposition does not affect atoms and is applied recursively to boolean operators. We use angle brackets (symbols “(“ and “)”) in the definition of the decomposition to cover all four possible cases of open (denoted with round brackets) and closed (denoted with square brackets) intervals; the definition is valid for any instantiation of the symbols as long as they are consistently replaced on the right-hand side.

The decomposition $L_K$ of MTL formulae with respect to parameter $K$ is the translation $L_K : M(Π) → M(Π)$ such that $L_K(F_{[a,b]}ϕ) =$

$$\begin{cases}
F_{[a,b]} L_K(ϕ) & b ≤ K \\
F_{[a,K]}(F_{[a mod K,b mod K]}(K) \land K) & b ≤ \left\lfloor \frac{a}{K} + 1 \right\rfloor \land K \\
F_{[a,K]}(K \land ϕ) & b > \left\lfloor \frac{a}{K} + 1 \right\rfloor \land K \\
F_{[a,b]}(DF(L_K(ϕ), K, b - \left\lfloor \frac{a}{K} + 1 \right\rfloor)), K) & b > \left\lfloor \frac{a}{K} + 1 \right\rfloor \land K
\end{cases}$$

where

$$DF(ϕ, K, h) = \begin{cases}
F_{[0,h]}(K) & h ≤ K \\
F_{[0,K]}(K) \land ϕ \land F_{[a,h]}(K) & h > K
\end{cases}$$

The decomposition $L_K$ considers three cases depending on the values of $a, b, K$. In the first case we have $b ≤ K$, which means that the upper bound of the temporal interval $[a,b]$ in the input formula is smaller than $K$, therefore no decomposition is needed. The other two cases consider input formulae where $b > K$. The second case is characterized by $b ≤ \left\lfloor \frac{a}{K} + 1 \right\rfloor \land K \equiv b ≤ \left\lfloor \frac{a}{K} \right\rfloor \cdot K + K$. The decomposition yields a formula of the form $F_{[a,K]}(ϕ)$, where
The theoretical results presented in Section 5 can be applied to improve the memory scalability of the distributed trace checking algorithm based on the MapReduce programming model, and introduced by some of the authors in previous work [10]. Although the algorithm presented in [10] was designed to perform trace checking of properties written in SOLOIST [11] (an extension of MTL with aggregating temporal modalities), here we consider, without loss of generality (see [11]), only its MTL subset. In the rest of this section, after introducing some additional notation, we give an overview of the algorithm's execution flow, and detail the modifications (emphasized with gray boxes in Figure 6) applied to the original algorithm defined in [10] to support MTL\_\kappa semantics.

**Theorem 3.** Given an MTL\_\kappa formula \(\phi\), a timed word \((\sigma, \tau)\) and a positive constant \(K\), we have that:

\[
\left(\sigma, \tau, 0\right) \models L \phi \iff \left(\sigma, \tau, 0\right) \models L L\kappa \phi(\phi)
\]

and the upper bound of every bounded interval in all temporal subformulae of \(L\kappa \phi\) is less than or equal to \(K\).

Proof. We can prove this statement by showing that \(L\kappa \phi\) can always be rewritten back as \(\phi\) and vice versa using Lemmata 2 and 3. The complete proof is provided in the extended version [9] of this paper.

6. TRACE CHECKING MTL\_\kappa FORMULAE WITH MAPREDUCE

The theoretical results presented in Section 5 can be applied to improve the memory scalability of the distributed trace checking algorithm based on the MapReduce programming model, and introduced by some of the authors in previous work [10]. Although the algorithm presented in [10] was designed to perform trace checking of properties written in SOLOIST [11] (an extension of MTL with aggregating temporal modalities), here we consider, without loss of generality (see [11]), only its MTL subset. In the rest of this section, after introducing some additional notation, we give an overview of the algorithm’s execution flow, and detail
number of MapReduce iterations is equal to the height of the MTL formula $\Phi$. The first MapReduce iteration parses the input trace from the distributed file system, applies the map and reduce functions and passes the output (a set of tuples) to the next iteration. Each subsequent iteration $l$ (where $1 < l \leq h(\Phi)$) receives the set of tuples from iteration $l-1$ in the expected internal format (hence, parsing is performed only in the first iteration). The set of tuples contains all the positions where the subformulae of $\Phi$ of height $l-1$ hold. Note that the trace itself is a similar set, containing all the positions where the atoms (with height 1) hold. Based on the set it receives, the $l$-th iteration can then calculate all the positions where the subformulae of height $l$ hold. Each iteration consists of three phases: 1) read phase that reads and splits the input; 2) map phase that associates each formula with its superformula; and 3) reduce phase that applies the semantics of the appropriate subformula of $\Phi$. The final set of tuples represents all the positions where the input formula holds. Hence, producing the verdict is only a matter of checking if the input formula holds in the first position.

**Read phase.** The input reader component of the MapReduce framework is used in this phase; this component can process the input trace in a parallel way. The trace saved in a distributed file system is split into several blocks, replicated 3 times and distributed among the nodes. The MapReduce framework exploits this block-level parallelization both during the read and map phases. For example, the default block size of the Hadoop deployment is 64MB, which means that a 1GB trace is split in 16 parts and can be potentially processed using 16 parallel readers and mappers. However, if we executed the algorithm on 3 nodes with 4 cores each, we could process up to 12 blocks in parallel. The input reader is used only in the first iteration and can be seen as a parser that converts the trace into a uniform internal representation that is used in the subsequent iterations. As shown in Figure 6a, the $k$-th instance of the input reader handles the $k$-th block $T_k$ of the trace $T$. For each element $(\sigma, \tau)$ in $T_k$ and every atom $p$ occurring in the MTL formula $\Phi$, the reader emits a key-value pair of the form $(p, (p \in \sigma, \tau))$. The key is the atom $p$ itself, while the value is a pair consisting of the truth value of $p$ at time $\tau$ (obtained by evaluating the expression $p \in \sigma$) and the time-stamp $\tau$. The emit function incrementally builds the list of outgoing tuples.

**Map phase.** Each tuple generated by an input reader is passed to a mapper on the same node. Mappers associate the formula in the tuple with all its superformulae in $\Phi$. For example, given $\Phi = (a \land b) \lor \neg a$, if the input reader returns a tuple $(a, (T, 42))$, the mapper will associate it with formulae $a \land b$ and $\neg a$, emitting tuples $(a \land b, (a, T, 42))$ and $(\neg a, (a, T, 42))$. The mapper, shown in Figure 6b, receives tuples in the form $(\phi, (v, \tau))$ from the input reader and emits all tuples of the form $(\psi, (\phi, v, \tau))$ where $\psi \in \text{sup}_{p}\phi$.

To support lazy semantics, the algorithm needs to consider all the time instants where we want to evaluate the temporal operators. If any of these instants does not have a corresponding element in the trace, then the original algorithm would evaluate a formula to false. However, to support lazy semantics, we do not need to introduce an element in the trace for each time instant: we know a priori that only formulae of the form $F_{\neg K} \phi$ — explicitly introduced by the $L_K$ decomposition — may be evaluated incorrectly if the appropriate elements are not in the trace (see Figure 3). Therefore, we modify the algorithm for the mapper (see Figure 6b) to introduce one element at $\tau + K$ only when the parent formula $\psi$ is of the form $F_{\neg K} \phi$; this condition is captured by the lazy() predicate. The emitted tuple contains the tuple $(\varphi_{act}, \bot, \tau + K)$ as its value. In this tuple, the truth value of $\varphi_{act}$ is false by convention, to represent a non-existent trace element. Since the mapper is stateless and cannot check if a tuple exists at time instant $\tau + K$, it is the reducer’s responsibility to discard tuple $(\varphi_{act}, \bot, \tau + K)$ if there is already a tuple at $\tau + K$.

**Reduce phase.** The reducers exploit the information produced by the mappers to determine the truth values of the superformula at each position, i.e., reducers apply the appropriate MTL semantics for the operator used in the superformula. The total number of reducers running in parallel at the $l$-th iteration is the minimum between the number of subformulae with height $l$ in the input formula $\Phi$ and the number of available reducers. Each reducer calls an appropriate reduce function depending on the type of formula used as key in the received tuple. For space reasons we focus only on two algorithms: the one for the metric “Eventually” operator $F_1$ and the one for the metric “Globally” operator $G_1$. We refer the reader to our previous work [10] for the full description of all the reducer algorithms. Figure 6c shows the algorithm for formulæ of the form $F_1 \phi$. It uses an auxiliary boolean variable $val$ and a queue $\text{win}$. The algorithm receives the tuples in $T$ already sorted (in the shuffle and sort phase of the MapReduce framework) in descending order with respect to the time-stamps. These tuples are incrementally processed by the checkDup() function, which discards the tuples of the form $(\varphi_{act}, \bot, \tau)$ if tuples with the same time-stamp already exist. The queue $\text{win}$ keeps track of all the tuples with positive truth value that fall in the convex union (denoted as $B$) of the intervals $[0, 0]$ and $I$. This is ensured by the inner while loop, which compares the minimal $\lfloor \text{win} \rfloor_{\tau}$ and maximal $\lceil \text{win} \rceil_{\tau}$ time-stamp in the queue and keeps removing the maximal tuple.

---

Figure 6: Reader, Mapper and Reducer algorithms.

1: function INPUT READER, $g(T_k[])$
2: for all $(\sigma, \tau) \in T_k[]$ do
3: if all $p \in \text{sup}_{p}\phi$ then
4: emit(p, (p \in \sigma, \tau))
5: end for
6: end for
7: end function

(b) Mapper algorithm

1: function MAPPER, $K_1((\phi, (v, \tau)))$
2: for all $\psi \in \text{sup}_{p}\phi$ do
3: emit(\phi, (v, \tau))
4: end for
5: end function

(c) Reducer for operator $F_1$

1: function REDUCER, $F_1((\psi, T[]))$
2: val ← $\bot$, win ← $\emptyset$
3: for all $(\phi, v, \tau) \in \text{checkDup}(T[])$ do
4: win ← $\text{win} \cup (\phi, v, \tau)$ if $\psi \in \text{sup}_{p}\phi$
5: while $|\text{win}| > 0$ do
6: win ← $\text{win} \cap \text{sup}_{p}\phi$
7: end while
8: emit(\phi, $\text{val}$, \tau)
9: end for
10: end function
11: end function

(d) Reducer for operator $G_1$
(argmax_{\text{win}}) until the loop condition is satisfied. The final truth value of \( F p \) depends on whether the queue \text{win} contains a tuple with a time-stamp \( t' \) that is in the interval \( I \). Notice that the size of the queue \text{win} depends directly on the size of the interval \( I \); hence, the memory scalability of the algorithm on individual nodes depends on the size of the intervals in formula \( \Phi \).

The reducer algorithm in Figure 6d implements the semantics of formulae of the form \( G_{\mathbf{p}} \). The code is similar to the one for the operator \( F_{\mathbf{t}} \). The only difference is that the queue \text{win} keeps track of all the tuples with negative truth value; hence, the truth value of \( G_{\mathbf{p}} \) depends on whether the queue \text{win} contains a tuple in the interval \( I \) that is a witness to the violation of \( G_{\mathbf{p}} \).

### Examples of application of the algorithm.

Let us use our algorithm to evaluate the formula \( \Phi \) from Example 1 on the same trace using our algorithm to evaluate the formula \( \Phi \) from Example 1 over MTL\textsubscript{L} semantics.

### Evaluation settings

To evaluate our approach, we used six \textit{t2.micro} instances from the Amazon EC2 cloud-based infrastructure with a single CPU core and 1GB of memory each. We used the standard configuration for the HDFS distributed file system and the YARN data operating system. HDFS block size was set to 64 MB and block replication was set to 3. YARN was configured to allocate containers with memory between 512 MB and 1 GB, with 1 core. In all the executions, we limited the memory of our algorithm to 1 GB.

Measuring the actual memory usage of user-defined code in Spark-based applications requires to distinguish between the memory usage of the Spark framework itself and the one of user-defined code. This step is necessary since the framework may use the available memory to cache intermediate data to speed up computation. Hence, to measure the memory usage of the auxiliary data structures used by our algorithm (e.g., the \text{win} queue), we instrumented the code.

This instrumentation, which has a negligible overhead, monitors the memory usage of the algorithm’s data structures and reports the maximum usage for each run.

For the evaluation described in the next two subsections, we used synthesized traces. By using synthesized traces, we are able to control in a systematic way the factors, such as the trace length and the frequency of events, that impact on
the time and memory required for checking a specific type of formula. In particular, we evaluated our approach by triggering the worst-case scenario, in terms of memory scalability, for our trace checking algorithm. Such scenario is characterized by having the auxiliary data structures used by the algorithm always at their maximum capacity. To synthesize the traces, we implemented a trace generator program that takes as parameters the desired trace length and the number of events per trace element. The program generates a trace with a number of events, such that the $i$-th element (with $0 \leq i \leq n - 1$) has $i$ as time-stamp value. Each trace element has between 1 and $m$ events denoted as \(e_1, \ldots, e_m\), where $e_i = p$ and the other $m - 1$ events are randomly selected from the set of atoms \(\{p_2, \ldots, p_m\}\) using a uniform distribution. We generated ten traces, with $n$ set to 50 000 000 and $m$ set to 20; the average size of each trace, before saving it in the distributed file system, is 3.2 GB. These traces and the other artifacts used for the evaluation are available on the tool web site [20].

**7.2 Scalability**

The performance of our distributed trace checking algorithm with respect to the length of the trace and the size of the formula has been already investigated in our previous work [10]. The same conclusions regarding these two parameters apply also to the new algorithm, which uses lazy semantics. Therefore, in this section we only focus on evaluating the memory scalability of the new algorithm.

To address RQ1, we evaluate the memory usage of the algorithm for different sizes of the time interval used in the MTL formula to be checked. As discussed in Section 6, the largest time interval that does not trigger memory saturation in a cluster, depends on the memory configuration of the node in the cluster with the least amount of memory available. Hence, we evaluate the memory usage on a single node by using formulae of height 1; nevertheless, the map phase is still executed in parallel. We consider the two metric formulae \(G_{0,N}[q]\) and \(F_{0,N}[p]\), parametrized by the value $N$ of the bound of their time interval. Formula \(F_{0,N}[p]\) refers to atom $p$; notice that our trace generator guarantees that $p$ is present in every trace element. Formula \(G_{0,N}[q]\) refers to atom $q$; we configured our trace generator so that event $q$ is absent in all trace elements. These two formulae exercise the trace checking algorithm in its worst-case. Indeed, according to line 4 in Figure 6c, the reducer for $F_1$ buffers all the elements where atom $p$ is true; hence, when checking formula \(F_{0,N}[p]\), at any point in time the queue $win$ will be at its maximum capacity. Dually, when checking formula \(G_{0,N}[q]\), the absence of the event $q$ from the trace will force the algorithm to maintain the queue $win$ at its maximal capacity (line 4 in Figure 6d). As mentioned in section 3, our trace checking algorithm deals with MTL formulae in the most general case, therefore it evaluates formulae \(G_{0,N}[q]\) and \(F_{0,N}[p]\) at every position to allow for arbitrary nesting.

To address RQ2, we need a baseline for comparison. Among the non-distributed, non-parallel trace checking tools, the only tool supporting MTL and publicly-available is MON-POLY [6], which was the best performing tool in the “offline monitoring” track of the first international Competition on Software for Runtime Verification [4] (CSR V 2014). MON-POLY, when executed on the traces described above, produced a stack overflow error; hence, we could not use it for comparison. Among distributed and parallel approaches, the only tool supporting MTL and publicly-available is MTL-A and publicly-available is the one described in our previous work [10], to which we compare.

Plots in Figures 8a and 8b show the execution time and memory usage required to check, respectively, formula \(G_{0,N}[q]\) and \(F_{0,N}[p]\), instantiated with different values of parameter $N$. Each data point is obtained by running the algorithm over the ten synthesized traces and averaging the results. The plots colored in black show the average time and memory usage of our previous algorithm [10], which applies MTL semantics. The plots colored in gray represent the runs of our new algorithm that applies MTL semantics and decomposes all the formulae with time interval $N$ strictly greater than 30 000 000. The decomposition parameter $K = 30 000 000$ is the maximal value that our infrastructure can support before saturating its memory.

We answer RQ1 by observing the trend in the gray plots of Figures 8a and 8b: the proposed algorithm can check, on very large traces, formulae that use very large time intervals (up to 50 000 000), using at most 1 GB of memory and taking a reasonable time (at most 200s). To answer RQ2, the plots show that the proposed algorithm is more scalable in terms of memory usage than the algorithm from [10]. Indeed, for the evaluation of both formulae, the latter exhausts the memory bound of 1 GB when the time interval $N$ is higher than 30 000 000. Nevertheless, the proposed algorithm is on average 1.35x slower than the previous algorithm [10] when the time interval $N$ is higher than 30 000 000. This additional time is needed to process the new formula obtained through the $L_K$ decomposition.

**7.3 Time/memory tradeoffs**

As suggested above, the parametric decomposition used in the proposed trace checking algorithm leads to a reduced memory usage but increases the execution time. In this section we dig into and generalize this result by investigating the time/memory tradeoffs of our algorithm, with respect to the decomposition parameter $K$. More specifically, to address RQ3 we evaluate the execution time and the memory usage of the algorithm for different values of parameter $K$, when checking formulae \(G_{0,50 000 000}[q]\) and \(F_{0,50 000 000}[p]\). These formulae are processed using the $L_K$ decomposition, with values of $K$ that are taken from $V = \{2^{10^i} | i = 2, 3, \ldots\}$ representing an infinite harmonic series scaled by a $5 \cdot 10^7$ factor. By using set $V$, we can study the performance of the algorithm for different values of $K$ without exhaustively exploring its large domain. Since set $V$ is infinite, we put a threshold of one hour on the execution time.

The plots in Figure 8c show the execution time and memory usage to check the two formulae. Each data point is obtained by running the algorithm over the ten synthesized traces and averaging the results. The value of $K$ is represented in both plots on the $x$-axis using the logarithmic scale. The smallest value of $K$ that satisfies the execution time threshold is 1 666 666 (obtained from set $V$ with $i = 30$); for this value of $K$ the algorithm used 54.14 MB of memory and took 43 minutes to complete. The plots show that using a lower value for $K$ decreases the memory footprint of the algorithm. However, a lower value for $K$ also yields a longer execution time for the algorithm. This longer execution time is due to the fact that a lower value for $K$ increases the size (and the height) of the formula obtained after applying the $L_K$ decomposition. The increased height of the decomposed formula triggers more iterations of the
algorithm, yielding longer execution times. We answer RQ3 by stating that there is a tradeoff between time and memory, determined by the value of parameter $K$. A good balance between these two factors can be achieved when $K$ is set to the largest possible value supported by the infrastructure: in this way, it is possible to reduce the size of the decomposed formula without incurring a longer execution time for the algorithm. Nevertheless, our algorithm is completely parametric in $K$, allowing engineers to tune the algorithm to be either more time- or more memory-intensive, depending on the application’s requirements.

8. RELATED WORK

The approach presented in this paper is strictly related to work done in the areas of alternative semantics for metric temporal logics and of trace checking/run-time verification.

Alternative semantics for metric temporal logics. The work closest to our lazy semantics is the one in [15], which proposes an alternative MTL semantics, used to prove that signal-based semantics is more expressive than point-based semantics over finite words. Despite the similarity between the two semantics, the definition of the Until operator over our lazy semantics is more practical for the purpose of trace checking, since it requires the left subformula of an Until operator to hold in a finite number of positions. Reference [13] revises the model parametric semantics of the TRIO temporal logic [22], in order to overcome counterintuitive behaviors of bounded temporal operators on a finite temporal domain. The proposal shares the same intuition behind our definition of lazy semantics, but overall the two semantics are quite different (in particular, in the interpretation of bounded and unbounded temporal operators).

Trace checking/run-time verification. Several approaches for trace checking and run-time verification and monitoring of temporal logic specifications have been proposed in the last decade. The majority of them (see, for example, [7, 16, 18, 25, 26]) are centralized and use sequential algorithms to process the trace (or, in online algorithms, the stream of events). The centralized, sequential nature of these algorithms does not allow them either to process large traces or properties containing very large time bounds. In the last years there have been approaches for trace checking [5] and runtime verification [8, 21, 25] that rely on some sort of parallelization. However, they mostly focus on splitting the traces based on the data they contain, rather than on the structure of the formula. These approaches adopt first-order relations with finite domains to represent the events in the trace. The trace can then be split into several unrelated partitions based on the terms occurring in the relations. We consider these approaches orthogonal to ours, since we focus on the scalability with respect to the temporal dimension, rather than the data dimension. As for the specific application of MapReduce for trace checking, an iterative algorithm for LTL is proposed in [3]. Similarly to the algorithm presented in this paper and to our previous work [10], the algorithm in [3] performs iterations of MapReduce jobs depending on the height of the formula to check. However, it does not address the issue of memory consumption of the reducers. Moreover, the whole trace is kept in memory during the reduce phase, making the approach unfeasible for very large traces.

9. CONCLUSIONS AND FUTURE WORK

This work addresses the memory scalability issue that affects trace checking algorithms when dealing with temporal properties that use large time intervals. We have proposed an alternative, lazy semantics for MTL, whose properties allow for a parametric decomposition of any MTL formula into an equivalent MTL formula with bounded time intervals. As shown in the evaluation, such decomposition can be used to improve distributed trace checking algorithms, making them more memory-efficient and able to deal with both very large traces and very large time intervals.

A future research direction is to study lazy semantics with respect to the signal-based semantics for MTL. Another direction is the investigation of techniques for determining the most appropriate value for $K$ in the $L_K$ decomposition of formulae, based on the configuration of the available cloud infrastructure.
10. REFERENCES


