Efficient Large-scale Trace Checking Using MapReduce*

Marcello M. Bersani[1], Domenico Bianculli[2], Carlo Ghezzi[1], Srđan Krstić[1] and Pierluigi San Pietro[1]
[1]DEEPSE group - DEIB - Politecnico di Milano, Milano, Italy
[2]SnT 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[1] 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: $\mathsf{G}(\mathsf{create} \rightarrow (\neg\mathsf{delete}\,\mathsf{U}_{=6\text{ years}}\ \mathsf{delete})$, where the operator U (called "*Until*") states that its right operand, the delete event, must occur in exactly six years (i.e., 189.2 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 ($\neg$delete) must hold until that happens. Operator G (called "*Globally*") states that property holds over the whole

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

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884832

[1]Also 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 Π 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_I is the metric "*Until*" operator: $\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \phi\, \mathsf{U}_I\, \phi$. Additional boolean and temporal operators can be derived using the usual conventions: "*Eventually*" is defined as $\mathsf{F}_I\phi \equiv \top \mathsf{U}_I \phi$; "*Globally*" is defined as $\mathsf{G}_I\phi \equiv \neg\mathsf{F}_I\neg\phi$. 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: $\mathsf{F}^K(\phi) \equiv \underbrace{\mathsf{F}\mathsf{F}\ldots\mathsf{F}}_{K \text{ times}}(\phi)$, with $\mathsf{F}^0(\phi) = \phi$. 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* τ, of length $|\tau| > 0$, is a sequence $\tau_0\tau_1 \ldots \tau_{|\tau|-1}$ of values $\tau_i \in \mathbb{R}$ such that $0 < \tau_i < \tau_{i+1}$ for each $0 \leq i < |\tau| - 1$, i.e., the sequence is *strictly monotonic*. A *word* σ over the alphabet 2^Π 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] $\omega = \omega_0\omega_1 \ldots \omega_{|\omega|-1}$ 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^Π and $\tau_0 \ldots \tau_{|\omega|-1}$ is a timed sequence. A pair ω_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 ω_i in the timed word ω, while τ_i refers to the *time instant* or *time-stamp* of the element ω_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^Π is a set of timed words over 2^Π. MTL_P semantics on timed words is given in Figure 1, where the point-based satisfaction relation $\models_P$ is defined with respect to a timed word (σ, τ), a position $i \in \mathbb{N}$, atom $p \in \Pi$, and MTL for-

$$
\begin{aligned}
(\sigma,\tau,i) &\models_P p \text{ iff } p \in \sigma_i \text{ for } p \in \Pi\\
(\sigma,\tau,i) &\models_P \neg\phi \text{ iff } (\sigma,\tau,i) \not\models_P \phi\\
(\sigma,\tau,i) &\models_P \phi \vee \psi \text{ iff } (\sigma,\tau,i) \models_P \phi \text{ or } (\sigma,\tau,i) \models_P \psi\\
(\sigma,\tau,i) &\models_P \phi\, \mathsf{U}_I\, \psi \text{ iff } \exists j.(i \le j < |\sigma| \text{ and } \tau_j - \tau_i \in I \text{ and}\\
&\quad (\sigma,\tau,j) \models_P \psi \text{ and } \forall k.(i < k < j \text{ then } (\sigma,\tau,k) \models_P \phi))
\end{aligned}
$$

Figure 1: $\mathsf{MTL_P}$ semantics on timed words.

mulae ϕ and ψ. Note that, due to the strictly monotonic definition of the timed sequence τ, the metric "*Next*" operator can be defined as $\mathsf{X}_I\phi \equiv \bot\, \mathsf{U}_{I-\{0\}}\, \phi$. $L_P(\phi)$ is a timed language defined by a formula ϕ when interpreted over the $\mathsf{MTL_P}$ semantics, i.e., $L_P(\phi) = \{(\sigma,\tau) \mid (\sigma,\tau,0) \models_P \phi\}$.

2.2 The MapReduce programming model

MapReduce [14] is a programming model, developed by Google, for processing and analyzing large data sets using a parallel, distributed infrastructure. The MapReduce programming model uses two user-defined functions, *map* and *reduce*, that are inspired by the homonymous functions that are typically found in functional programming languages. The *map* function receives a key-value pair associated with the input data and returns a set of intermediate key-value pairs; its signature is `map(k:K1,v:V1):list[(k:K2, v:V2)]`. The *reduce* function is applied to all the intermediate values that have the same intermediate key, in order to combine the derived data appropriately; its signature is `reduce(k:K2, list(v:V2)):list[v:V2]`. In the definitions above, `K1` and `K2` are types for keys and `V1` and `V2` are types for values.

Besides the actual programming model, MapReduce is also a framework that provides, in a transparent way to developers, parallelization, fault tolerance, locality optimization, and load balancing. The MapReduce framework is responsible for partitioning the input data, scheduling and executing the *Map* and *Reduce* tasks (also called *mappers* and *reducers*, respectively) on a cluster of available nodes, and for managing communication and data transfer (usually leveraging a distributed file system). More in detail, the execution of a MapReduce operation (called *job*) proceeds as follows. First, the system splits the input into blocks of a certain size and parses them using an *InputReader*, generating input key-value pairs. It then assigns each input key-value pair to a mapper, which processes it in parallel on the nodes of the distributed infrastructure. A mapper passes the set of key-value pairs to the *map* function, which generates a set of intermediate key-value pairs. Notice that each run of the *map* function is stateless, i.e., the transformation of a single key-value pair does not depend on any other key-value pair. The next phase is called *shuffle and sort*: it takes the intermediate data generated by each mapper, sorts them based on the intermediate key, divides these data into regions to be processed by reducers, and distributes these data on the nodes where the reducers will be executed. The division of intermediate data into regions is done by a *partitioning function*, which depends on the (user-specified) number of reducers and the key of the intermediate data. Each reducer executes the *reduce* function, which produces the output 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. Multiple MapReduce calls can be chained together to perform complex data processing.

Atoms:	$\{p\}$	$\{p\}$		$\{q\}$		$\{p,q\}$		$\{p,q\}$	$\{q\}$	$\{q\}$
Time-stamps:	1	2		4		6		8	9	10
Time instants:	1	2	3	4	5	6	7	8	9	10
p	$\top$	$\top$		$\bot$		$\top$		$\top$	$\bot$	$\bot$
$\mathsf{F}_{[3,7]}(p)$	$\top$	$\top$		$\top$		$\bot$		$\bot$	$\bot$	$\bot$

Figure 2: Evaluation of formula $\Phi = \mathsf{F}_{[3,7]}(p)$.

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 $\omega_i = (\sigma_i, \tau_i)$ corresponds to the i-th element of the trace, where atoms in σ_i represent all the events with time-stamp τ_i.

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[2]. 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 $\Phi = \mathsf{F}_{[3,7]}(p)$ and its evaluation on the following trace (represented as a timed word):* $(\{p\},1)$, $(\{p\},2)$, $(\{q\},4)$, $(\{p,q\},6)$, $(\{p,q\},8)$, $(\{q\},9)$, $(\{q\},10)$. *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 $\mathsf{F}_{[3,7]}(p)$ using point-based semantics. As shown in the last row of Figure 2, according to point-based semantics, formula $\mathsf{F}_{[3,7]}(p)$ holds at time instants 1, 2 and 4.*

For a formula of the form[3] $\mathsf{F}_{[a,b]}(p)$, 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 $\mathsf{F}_{[3,7]}(p)$ 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

[2]For example, if a "*Globally*" temporal operator is used.

[3]We consider the most general case in which MTL formulae can be arbitrarily nested. This means that a trace checking algorithm has to evaluate every subformula in every position of the trace. Nevertheless, more specific cases could use heuristics based on the actual values of the temporal intervals in the formula and hence reduce the number of positions in the trace in which the formula is evaluated.

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

Atoms:	$\{p\}$	$\{p\}$		$\{q\}$		$\{p,q\}$		$\{p,q\}$	$\{q\}$	$\{q\}$
Time-stamps:	1	2		4		6		8	9	10
Time instants:	1	2	3	4	5	6	7	8	9	10
p	$\top$	$\top$		$\bot$		$\top$		$\top$	$\bot$	$\bot$
$\mathsf{F}_{[3,4]}(p)$	$\bot$	$\top$		$\top$		$\bot$		$\bot$	$\bot$	$\bot$
$\mathsf{F}_{[0,3]}(p)$	$\top$	$\top$		$\top$		$\top$		$\top$	$\bot$	$\bot$
$\mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$	$\bot$	$\top$		$\top$		$\bot$		$\bot$	$\bot$	$\bot$
Φ'	$\bot$	$\top$		$\top$		$\bot$		$\bot$	$\bot$	$\bot$

Figure 3: Evaluation of formula $\Phi' = \mathsf{F}_{[3,4]}(p) \vee \mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$.

$$
\begin{aligned}
&(\sigma,\tau,t) \models_L p \text{ iff } \exists i.(0 \le i < |\sigma| \text{ and } t = \tau_i \text{ and } p \in \sigma_i)\\
&(\sigma,\tau,t) \models_L \neg\phi \text{ iff } (\sigma,\tau,t) \not\models_L \phi\\
&(\sigma,\tau,t) \models_L \phi \vee \psi \text{ iff } (\sigma,\tau,t) \models_L \phi \text{ or } (\sigma,\tau,t) \models_L \psi\\
&(\sigma,\tau,t) \models_L \phi\, \mathsf{U}_I\, \psi \text{ iff } \exists t'.(t' \ge t \text{ and } t' - t \in I \text{ and}\\
&\quad (\sigma,\tau,t') \models_L \psi \text{ and } \forall t''.(t < t'' < t' \text{ and } \exists i.(0 \le i < |\sigma| \text{ and}\\
&\qquad t'' = \tau_i) \text{ then } (\sigma,\tau,t'') \models_L \phi))
\end{aligned}
$$

Figure 4: MTL_L semantics on timed words.

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 $w = (\sigma,\tau) = (\{q\},1)(\{p\},7)$ and two MTL formulae $\psi_1 = \mathsf{F}_{=6}p$ and $\psi_2 = \mathsf{F}_{=3}\mathsf{F}_{=3}p$. 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: ψ_1 correctly evaluates to true, but ψ_2 to false. Indeed, in ψ_2 the outermost $\mathsf{F}_{=3}$ 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. Indeed, this 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 (σ,τ) and a time instant $t \in \mathbb{R}^+$; p is an atom and ϕ and ψ are MTL formulae. An MTL formula ϕ, 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 $\phi \equiv \psi_1\, \mathsf{U}_I\, \psi_2$ evaluates to false in case there are no positions in the interval I, due to the existential quantification on j (see Figure 1). Conversely, over the MTL_L semantics, the evaluation of ϕ depends on the evaluation of ψ_2. If the latter is an atom then formula ϕ also evaluates to false, because of the existential quantifier in the MTL_L semantics of atoms. However, if ψ_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 ϕ. Hereafter we refer to the MTL formulae interpreted over the MTL_L

semantics as "$\mathsf{MTL_L}$ formulae"; similarly, "$\mathsf{MTL_P}$ formulae" are MTL formulae interpreted over the $\mathsf{MTL_P}$ semantics.

Let $\mathbb{M}(\Pi)$ 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 $L_P(\phi)$ defined using some $\mathsf{MTL_P}$ formula ϕ can be defined using an $\mathsf{MTL_L}$ formula obtained after applying the translation $l2p : \mathbb{M}(\Pi) \rightarrow \mathbb{M}(\Pi)$ to ϕ, i.e., $L_p(\phi) = L_L(l2p(\phi))$ for any ϕ. The $l2p$ translation is defined as follows:

$$l2p(p) \equiv p, p \in \Pi; \quad l2p(\phi \vee \psi) \equiv l2p(\phi) \vee l2p(\psi)$$
$$l2p(\neg\phi) \equiv \neg l2p(\phi); \quad l2p(\phi \,\mathsf{U}_I\, \psi) \equiv l2p(\phi)\, \mathsf{U}_I\, (\varphi_{act} \wedge l2p(\psi))$$

where $\varphi_{act} \equiv a \vee \neg a$ for some $a \in \Pi$.

The goal of $l2p$ is to prevent the occurrence of *direct nesting* of temporal operators, i.e., to avoid the presence of (sub)formulae like $\mathsf{F}_{=3}\mathsf{F}_{=3}p$. 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 "existential" component of "*Until*"). The argument is conjuncted 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) over the alphabet $\Pi = \{a\}$. Over point-based semantics, $(\sigma, \tau, i) \models_P \varphi_{act} \equiv (\sigma, \tau, i) \models_P a \vee \neg a$ is true for any position i, since either a belongs to σ_i or not. However, the same does not hold for lazy semantics. According to lazy semantics, $(\sigma, \tau, t) \models_L \varphi_{act}$ is true only in those time instants t for which there exists i such that $\tau_i = t$ 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 $\omega = (\sigma, \tau)$, for any $i \geq 0$, the following equivalence (modulo $l2p$ translation) holds: $(\sigma, \tau, i) \models_P \phi$ iff $(\sigma, \tau, \tau_i) \models_L l2p(\phi)$.*

PROOF. See the extended version [9] of the paper. □

THEOREM 1. *Any timed language defined by an $\mathsf{MTL_P}$ formula can be defined by an $\mathsf{MTL_L}$ formula over the same alphabet.*

PROOF. By Lemma 1, for $i = 0$. □

Notice that the translation $l2p$ defines a syntactic MTL fragment where temporal or boolean combination of temporal-only operators cannot be directly nested. In this fragment $\mathsf{MTL_P}$ and $\mathsf{MTL_L}$ formulae define the same languages. However, if we consider the complete definition of MTL, without syntactic restrictions, the class of timed languages defined by $\mathsf{MTL_L}$ formulae strictly includes the class of languages defined by $\mathsf{MTL_P}$ 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 $\mathsf{MTL_L}$ formula that cannot be defined by any $\mathsf{MTL_P}$ formula.*

PROOF. Consider the language of timed words $L = \{(\sigma, \tau) \mid \exists i \exists j (i \leq j \wedge (\sigma, \tau, i) \models_L b \wedge (\sigma, \tau, j) \models_L c \wedge \tau_j \leq 2)\}$. L is defined by the $\mathsf{MTL_L}$ formula $\Phi = \Phi_1 \vee \Phi_2 \vee \Phi_3$, where $\Phi_1 = (\mathsf{F}_{(0,1)}b) \wedge (\mathsf{F}_{[1,2]}c) \vee (\mathsf{F}_{(0,1]}b) \wedge (\mathsf{F}_{(1,2]}c)$; $\Phi_2 = \mathsf{F}_{(0,1]}(b \wedge \mathsf{F}_{(0,1]}c)$ and $\Phi_3 = \mathsf{F}_{(0,1]}((\mathsf{F}_{(0,1)}b) \wedge (\mathsf{F}_{[1,1]}c))$. L cannot be defined by any $\mathsf{MTL_P}$ 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 $I \oplus J = \{i + j \mid \forall i \in I \text{ and } j \in J\}$.

LEMMA 2. *For any timed word (σ, τ) and $t \geq 0$,*

$$(\sigma, \tau, t) \models_L \mathsf{F}_I \mathsf{F}_J \phi \text{ iff } (\sigma, \tau, t) \models_L \mathsf{F}_{I \oplus J} \phi.$$

COROLLARY 1. *For any timed word (σ, τ) and $t, N \geq 0$,*

$$(\sigma, \tau, t) \models_L \mathsf{F}^N_{=K} \phi \text{ iff } (\sigma, \tau, t) \models_L \mathsf{F}_{=K \cdot N}.$$

LEMMA 3. *For any timed word (σ, τ) and $t \geq 0$,*

$$(\sigma, \tau, t) \models_L \mathsf{F}_I \phi \vee \mathsf{F}_J \phi \text{ iff } (\sigma, \tau, t) \models_L \mathsf{F}_{I \cup J} \phi, \text{ if } I \cap J \neq \emptyset.$$

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 "$\langle$" and "$\rangle$") 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* $\mathcal{L}_K$ of MTL formulae with respect to *parameter* K is the translation $\mathcal{L}_K : \mathbb{M}(\Pi) \rightarrow \mathbb{M}(\Pi)$ such that $\mathcal{L}_K(\mathsf{F}_{\langle a,b \rangle}\phi) =$

$$\begin{cases} \mathsf{F}_{\langle a,b \rangle}\mathcal{L}_K(\phi) & , b \leq K \\ \mathsf{F}^{\lfloor \frac{a}{K} \rfloor}_{=K}(\mathsf{F}_{\langle a \bmod K, b - \lfloor \frac{a}{K} \rfloor \cdot K \rangle}\mathcal{L}_K(\phi)) & , K < b \leq \lfloor \frac{a}{K} + 1 \rfloor \cdot K \\ \mathsf{F}^{\lfloor \frac{a}{K} \rfloor}_{=K}(\mathsf{F}_{\langle a \bmod K, K]}\mathcal{L}_K(\phi) \vee \mathsf{F}_{=K}(\mathcal{D}_F(\mathcal{L}_K(\phi), K, b - \lfloor \frac{a}{K} + 1 \rfloor \cdot K))) & , b > \lfloor \frac{a}{K} + 1 \rfloor \cdot K \end{cases}$$

where

$$\mathcal{D}_F(\psi, K, h) = \begin{cases} \mathsf{F}_{[0,h\rangle}\psi & , h \leq K \\ \mathsf{F}_{[0,K]}\psi \vee \mathsf{F}_{=K}(\mathcal{D}_F(\psi, K, h - K)) & , h > K. \end{cases}$$

The decomposition $\mathcal{L}_K$ considers three cases depending on the values of a, b, and K. In the first case we have $b \leq K$, 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 $b \leq \lfloor \frac{a}{K} + 1 \rfloor \cdot K \equiv b \leq \lfloor \frac{a}{K} \rfloor \cdot K + K$. The decomposition yields a formula of the form $\mathsf{F}^{\lfloor \frac{a}{K} \rfloor}_{=K}(\alpha)$, where

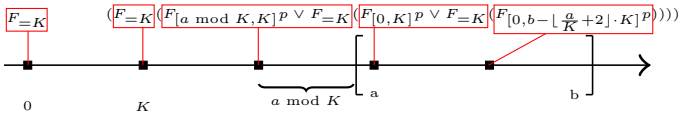


Figure 5: $\mathcal{L}_K$ decomposition of formula $\mathsf{F}_{[a,b]}p$.

$\alpha = \mathsf{F}_{[a \bmod K, b-\lfloor \frac{a}{K} \rfloor \cdot K]}\mathcal{L}_K(\phi)$ is equivalent to the input formula $\mathsf{F}_{[a,b]}(\phi)$ evaluated at time instant $\lfloor \frac{a}{K} \rfloor \cdot K$. Notice that according to Corollary 1, the argument α in $\mathsf{F}^{\lfloor \frac{a}{K} \rfloor}_{=K}(\alpha)$ is evaluated at time instant $\lfloor \frac{a}{K} \rfloor \cdot K$. The third case is characterized by $b > \lfloor \frac{a}{K} + 1 \rfloor \cdot K$.

We illustrate the decomposition of $\mathsf{F}_{[a,b]}p$ with $p \in \Pi$ by referring to the example in Figure 5, where the black squares divide the timeline into segments of length K. We refer to each position in the timeline pinpointed by a black square as a K-position. The big brackets enclose the interval $[a, b]$ relative to time instant 0. Moreover, we assume some values for a and K such that $\lfloor \frac{a}{K} \rfloor = 2$; hence, in the figure the position of a in the timeline is between the marks corresponding to $2K$ and $3K$. The application of $\mathcal{L}_K(\mathsf{F}_{[a,b]}p)$ returns the formula $F_{=K}(F_{=K}(F_{[a \bmod K,K]}p \vee F_{=K}(F_{[0,K]}p \vee F_{=K}(F_{[0,b-\lfloor \frac{a}{K}+2 \rfloor \cdot K]}p))))$, which is shown above the timeline, spanning through its length such that each subformula (highlighted in red) is written above the corresponding K-position where it is evaluated. Since $\lfloor \frac{a}{K} \rfloor = 2$ there are two subformulae of the form $\mathsf{F}_{=K}$ evaluated in the first two K-positions. Unlike the previous case, the interval $[a, b]$ is too big to allow for rewriting the input formula into another formula with a single F operator with bounded length. Hence, we use three subformulae: 1) $\mathsf{F}_{[a \bmod K,K]}p$ evaluated at the third K-position, 2) $\mathsf{F}_{[0,K]}p$ evaluated at the fourth K-position, and 3) $\mathsf{F}_{[0,b-\lfloor \frac{a}{K}+2 \rfloor \cdot K]}p$ evaluated at the fifth K-position; the last two subformulae are obtained from the definition of $\mathcal{D}_F$. Notice that if $K = 1$, the $\mathcal{L}_K$ decomposition boils down to the reduction of MTL to LTL.

THEOREM 3. *Given an* MTL_L *formula* ϕ*, a timed word* (σ, τ) *and a positive constant* K*, we have that:*

$$(\sigma, \tau, 0) \models_L \phi \text{ iff } (\sigma, \tau, 0) \models_L \mathcal{L}_K(\phi)$$

and the upper bound of every bounded interval in all temporal subformulae of $\mathcal{L}_K(\phi)$ *is less than or equal to* K*.*

PROOF. We can prove this statement by showing that $\mathcal{L}_K(\phi)$ can always be rewritten back as ϕ 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_L 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 the modifications (emphasized with gray boxes in Figure 6) applied to the original algorithm defined in [10] to support MTL_L semantics.

Additional notation. Let ϕ and ψ be MTL formulae. The set of all proper subformulae of ϕ is denoted with $\mathsf{sub}(\phi)$; notice that for atoms $p \in \Pi$, $\mathsf{sub}(p) = \emptyset$. The size of a formula ϕ, denoted $|\phi|$, is defined as the number of its non-proper subformulae, i.e., $|\phi| = |\mathsf{sub}(\phi)| + 1$. The set $\mathsf{sub}_a(\phi) = \{p \mid p \in \mathsf{sub}(\phi),\ \mathsf{sub}(p) = \emptyset\}$ is the set of atoms of formula ϕ. The set $\mathsf{sub}_d(\phi) = \{\alpha \mid \alpha \in \mathsf{sub}(\phi), \forall \beta \in \mathsf{sub}(\phi), \alpha \notin \mathsf{sub}(\beta)\}$ is called the set of all *direct subformulae* of ϕ; ϕ is called the *superformula* of all formulae in $\mathsf{sub}_d(\phi)$. The set $\mathsf{sup}_\psi(\phi) = \{\alpha \mid \alpha \in \mathsf{sub}(\psi), \phi \in \mathsf{sub}_d(\alpha)\}$ is the set of all subformulae of ψ that have formula ϕ as *direct subformula*. The *height* of ϕ, $h(\phi)$, is defined recursively as: $h(\phi) =$ if $(\phi \notin \Pi)$ then $\mathsf{max}\{h(\psi) \mid \psi \in \mathsf{sub}_d(\phi)\}+1$; else 1. For example, given the formula $\gamma = \mathsf{F}_{[2,4]}(a \wedge b)\ \mathsf{U}_{(30,100)}\ \neg c$, we have: $\mathsf{sub}(\gamma) = \{a, b, c, a \wedge b, \neg c, \mathsf{F}_{[2,4]}(a \wedge b)\}$ is the set of all proper subformulae of γ; $\mathsf{sub}_a(\gamma) = \{a, b, c\}$ is the set of atoms in γ; $\mathsf{sub}_d(\gamma) = \{\mathsf{F}_{[2,4]}(a \wedge b), \neg c\}$ is the set of direct subformulae of γ; $\mathsf{sup}_\gamma(a) = \mathsf{sup}_\gamma(b) = \{a \wedge b\}$ shows that the sets of superformulae of a and b in γ coincide; and the height of γ is 4, since $h(a) = h(b) = h(c) = 1$, $h(\neg c) = h(a \wedge b) = 2$, $h(\mathsf{F}_{[2,4]}(a \wedge b)) = 3$ and therefore $h(\gamma) = \mathsf{max}\{h(\mathsf{F}_{[2,4]}(a \wedge b)), h(\neg c)\} + 1 = 4$.

Overview. The algorithm takes as input a non-empty execution trace T and an MTL formula Φ and provides a verdict, indicating whether the trace satisfies the formula or not. Before the algorithm is used we assume that the execution infrastructure, i.e., the cluster of machines, is configured and running. We also assume that one can easily estimate through experimentation K_{cluster}, which is the largest time interval bound that can be used in a formula without triggering memory saturation in the cluster. This bound depends on the memory configuration of the node in the cluster with the least amount of memory available. Once we have this information, we can preprocess the input formula Φ, leveraging the theoretical results of Section 5. If the temporal operators in Φ have bounded intervals less than K_{cluster}, we apply the unmodified version of the original algorithm [10], which evaluates formulae over point-based semantics. Otherwise, we have to transform the original formula into an equivalent one that can be checked in a memory-efficient way. This transformation is achieved by first interpreting the input formula Φ over lazy semantics: to preserve its meaning, we apply the $l2p$ transformation. Afterwards, given the parameter K_{cluster}, we rewrite the formula using the $\mathcal{L}_{K_{\mathsf{cluster}}}$ decomposition (i.e., the $\mathcal{L}_K$ decomposition instantiated with parameter K_{cluster}) and obtain the formula $\Phi_L^{K_{\mathsf{cluster}}} = \mathcal{L}_{K_{\mathsf{cluster}}}(l2p(\Phi))$. Thanks to Theorem 3, this formula contains intervals no greater than K_{cluster} and is equivalent to Φ. The trace is modeled as a timed word with integer[4] time-stamps. We assume that the execution trace is saved in the distributed file system of the cluster on which the distributed algorithm is executed. This is a realistic assumption since in a distributed setting it is possible to collect logs, as long as there is a total order among the time-stamp induced by some clock synchronization protocol.

The trace checking algorithm processes the trace iteratively, through a sequence of MapReduce executions. The

[4]Since integers are isomorphic to a subset of real numbers, the theoretical results of the previous sections are still valid for integer time-stamps.

number of MapReduce iterations is equal to the height of the MTL formula Φ. 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 Φ 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 Φ. 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 (σ, τ) in T_k and every atom p occurring in the MTL formula Φ, 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 τ (obtained by evaluating the expression $p \in \sigma$) and the time-stamp τ. 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 Φ. For example, given $\Phi = (a \wedge b) \vee \neg a$, if the input reader returns a tuple $(a, (\top, 42))$, the mapper will associate it with formulae $a \wedge b$ and $\neg a$, emitting tuples $(a \wedge b, (a, \top, 42))$ and $(\neg a, (a, \top, 42))$. 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 $\psi \in \mathsf{sup}_\Phi(\phi)$.

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 $\mathsf{F}_{=K}$ —explicitly introduced by the $\mathcal{L}_K$ 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

```
1: function INPUT READER_{Φ,l}(T_k[])
2:   for all (σ, τ) ∈ T_k[] do
3:     for all p ∈ sub_a(Φ) do
4:       emit(p, (p ∈ σ, τ))
5:     end for
6:   end for
7: end function
```

(a) Input reader algorithm

```
1: function MAPPER_{Φ,K,l}((φ, (v, τ)))
2:   for all ψ ∈ sup_Φ(φ) do
3:     emit(ψ, (φ, v, τ))
       if lazy(ψ) then
         emit(ψ, (φ_act, ⊥, τ + K))
       end if
4:   end for
5: end function
```

(b) Mapper algorithm

```
1: function REDUCER^{F_I}_{Φ,l}(ψ, T[])
2:   val ← ⊥, win ← ∅
3:   for all (φ, v, τ) ∈ checkDup(T) do
4:     win ← win ∪ (φ, v, τ) if (v)
5:     while ⌈win⌉_τ − ⌊win⌋_τ ∉ 0 ⊎ I do
6:       win ← win \ argmax_τ(win)
7:     end while
8:     val ← ∃τ' ∈ {win}_τ : τ' − τ ∈ I
9:     emit(ψ, (val, τ))
10:  end for
11: end function
```

(c) Reducer for operator F_I

```
1: function REDUCER^{G_I}_{Φ,l}(ψ, T[])
2:   val ← ⊤, win ← ∅
3:   for all (φ, v, τ) ∈ checkDup(T) do
4:     win ← win ∪ (φ, v, τ) if (¬v)
5:     while ⌈win⌉_τ − ⌊win⌋_τ ∉ 0 ⊎ I do
6:       win ← win \ argmax_τ(win)
7:     end while
8:     val ← ∃τ' ∈ {win}_τ : τ' − τ ∈ I
9:     emit(ψ, (¬val, τ))
10:  end for
11: end function
```

(d) Reducer for operator G_I

Figure 6: Reader, Mapper and Reducer algorithms.

(see Figure 6b) to introduce one element at $\tau + K$ only when the parent formula ψ is of the form $\mathsf{F}_{=K}$; 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 φ_{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 Φ and the number of available reducers[5]. 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_I and the one for the metric "*Globally*" operator G_I. We refer the reader to our previous work [10] for the full description of all the reducer algorithms.

Figure 6c shows the algorithm for formulae of the form $\mathsf{F}_I\phi$. It uses an auxiliary boolean variable *val* and a queue *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[6]. 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 *win* keeps track of all the tuples with positive truth value that fall in the convex union[7] (denoted as $\uplus$) of the intervals $[0, 0]$ and I. This is ensured by the inner while loop, which compares the minimal ($\lfloor win \rfloor_\tau$) and maximal ($\lceil win \rceil_\tau$) time-stamp in the queue and keeps removing the maximal tuple

[5]This depends on the configuration of the cluster. Typically, the number of reducers is the number of nodes in the cluster multiplied by the number of cores available on each node.
[6]Sorting intermediate tuples is called secondary sorting and for simplicity we omit the implementation details.
[7]A convex union of intervals is defined as a convex hull of the union of the intervals.

l	Atoms: / Time-stamps:	$\{p\}$ 1	$\{p\}$ 2		$\{q\}$ 4		$\{p,q\}$ 6		$\{p,q\}$ 8	$\{q\}$ 9	$\{q\}$ 10				
	Time instants:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	p	$\top$	$\top$		$\bot$		$\top$		$\top$	$\bot$	$\bot$				
1	$\mathsf{F}_{[3,4]}(p)$	$\top$	$\top$		$\top$	$\top$	$\bot$		$\bot$	$\bot$	$\bot$				
	$\mathsf{F}_{[0,3]}(p)$	$\top$	$\top$		$\top$	$\top$	$\top$		$\top$	$\bot$	$\bot$				
	φ_{act}					$\bot$	$\not\bot$		$\not\bot$		$\not\bot$		$\bot$	$\bot$	$\bot$
2	$\mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$	$\top$	$\top$		$\top$	$\bot$	$\bot$		$\bot$	$\bot$	$\bot$		$\bot$	$\bot$	$\bot$
3	$\mathcal{L}_4(l2p(\Phi))$	$\top$	$\top$		$\top$	$\top$	$\bot$		$\bot$	$\bot$	$\bot$		$\bot$	$\bot$	$\bot$

Figure 7: Evaluation of the $\mathcal{L}_4(l2p(\Phi)) = \mathsf{F}_{[3,4]}(p) \vee \mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$ formula over $\mathsf{MTL_L}$ semantics.

($\mathsf{argmax}_\tau(win)$) until the loop condition is satisfied. The final truth value of $\mathsf{F}_I\phi$ depends on whether the queue *win* contains a tuple with a time-stamp τ' that is in the interval I. Notice that the size of the queue *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 Φ.

The reducer algorithm in Figure 6d implements the semantics of formulae of the form $\mathsf{G}_I\phi$. The code is similar to the one for the operator F_I. The only difference is that the queue *win* keeps track of all the tuples with negative truth value; hence, the truth value of $\mathsf{G}_I\phi$ depends on whether the queue *win* contains a tuple in the interval I that is a witness to the violation of $\mathsf{G}_I\phi$.

Examples of application of the algorithm. Let us use our algorithm to evaluate the formula Φ from Example 1 on the same trace using $\mathsf{MTL_P}$ semantics. In the *read* phase the algorithm parses the trace in parallel and creates the input tuples for the *map* phase. From the first element $(\{p\}, 1)$ the *Input Reader* creates only the tuple $(p, (\top, 1))$ since Φ refers only to atom p. Tuples $(p, (\top, 1))$, $(p, (\top, 2))$, $(p, (\bot, 4))$, $(p, (\top, 6))$, $(p, (\top, 8))$, $(p, (\bot, 9))$, $(p, (\bot, 10))$ are thus received by the *map* phase. The *Mapper* associates the formulae from the input tuples with their superformulae. In the case of tuple $(p, (\top, 1))$ it generates only tuple $(\mathsf{F}_{[3,7]}(p), (p, \top, 1))$ since $\mathsf{F}_{[3,7]}(p)$ is the only superformula of p. The *Reduce* phase, therefore, receives tuples
$(\mathsf{F}_{[3,7]}(p), (p, (\bot, 10)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\bot, 9)))$,
$(\mathsf{F}_{[3,7]}(p), (p, (\top, 8)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 6)))$,
$(\mathsf{F}_{[3,7]}(p), (p, (\bot, 4)))$, $(\mathsf{F}_{[3,7]}(p), (p, (\top, 2)))$,
$(\mathsf{F}_{[3,7]}(p), (p, (\top, 1)))$, all shuffled and sorted in descending order of their time-stamps. Since all the tuples have the same key, only one reducer is needed. The reducer applies the algorithm shown in Figure 6c and outputs the truth values of $\mathsf{F}_{[3,7]}(p)$ for every position in the trace:
$(\mathsf{F}_{[3,7]}(p), (\bot, 10))$, $(\mathsf{F}_{[3,7]}(p), (\bot, 9))$, $(\mathsf{F}_{[3,7]}(p), (\bot, 8))$,
$(\mathsf{F}_{[3,7]}(p), (\bot, 6))$, $(\mathsf{F}_{[3,7]}(p), (\top, 4))$, $(\mathsf{F}_{[3,7]}(p), (\top, 2))$,
$(\mathsf{F}_{[3,7]}(p), (\top, 1))$. Notice that the boolean values in the tuples correspond to the values in Figure 2 (row #4).

Assuming again that the memory requirement of keeping 8 positions is too demanding for our infrastructure we can now use parametric decomposition and lazy semantics to limit the upper bound of the interval in Φ to $K = 4$. We obtain formula $\mathcal{L}_4(l2p(\Phi)) = \mathsf{F}_{[3,4]}(p) \vee \mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$.

Let us evaluate formula $\mathcal{L}_4(l2p(\Phi))$ on the same trace from Example 1 over $\mathsf{MTL_L}$ semantics. Table 7 shows the truth values of the emitted tuples for every evaluated subformulae of $\mathcal{L}_4(l2p(\Phi))$. Since $h(\mathcal{L}_4(l2p(\Phi))) = 4$ the algorithm performs three iterations (whose index is indicated in the left-most column l). The truth values of the subformulae from the different iterations are separated by the horizontal dashed lines. In the first iteration the trace is parsed to obtain the truth values of atom p. After that, two reducers in parallel calculate the truth values of the $\mathsf{F}_{[0,3]}(p)$ and $\mathsf{F}_{[3,4]}(p)$ subformulae. In the second iteration the *Mapper* emits the additional φ_{act} tuples since the superformula is of the form $\mathsf{F}_{=4}$. The reducer evaluating formula $\mathsf{F}_{[4,4]}(\mathsf{F}_{[0,3]}(p))$ receives the tuples with the evaluation of $\mathsf{F}_{[0,3]}(p)$ and φ_{act}. The φ_{act} tuples with the crossed truth values are discarded because of the already existing $\mathsf{F}_{[0,3]}(p)$ tuples shown in the row above. Finally, in the third iteration we can see that the truth values $\mathcal{L}_4(l2p(\Phi))$ (circled in Figure 7) are the same (at all time instants in common) as the truth values of Φ shown in Figure 2.

7. EVALUATION

We have implemented our trace checking algorithm in the MTLMapReduce tool, which is publicly available [20]. The tool is implemented in Java and uses the Apache Spark framework [28,29], which supports iterative MapReduce applications in a better way than Apache Hadoop [2].

In this section we report on the evaluation of our tool, in terms of scalability and time/memory tradeoffs. More specifically, we evaluate our new trace checking algorithm by answering the following research questions:

RQ1: *How does the proposed algorithm scale with respect to the size of the time interval used in the formula to be checked?* (Section 7.2)

RQ2: *When compared to state-of-the-art tools, does the proposed algorithm have a better memory scalability with respect to the size of the time interval used in the formula to be checked?* (Section 7.2)

RQ3: *What are the time/memory tradeoffs of the proposed algorithm with respect to the decomposition parameter K?*(Section 7.3)

7.1 Evaluation settings

To evaluate our approach, we used six *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 *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 n and the number m of events (i.e., atoms) per trace element. The program generates a trace with n trace elements, 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 $e_1 = p$ 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 $\mathsf{G}_{[0,N]}q$ and $\mathsf{F}_{[0,N]}p$, parametrized by the value N of the bound of their time interval. Formula $\mathsf{F}_{[0,N]}p$ refers to atom p; notice that our trace generator guarantees that p is present in every trace element. Formula $\mathsf{G}_{[0,N]}q$ 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_I buffers all the elements where atom p is true; hence, when checking formula $\mathsf{F}_{[0,N]}p$, at any point in time the queue *win* will be at its maximum capacity. Dually, when checking formula $\mathsf{G}_{[0,N]}q$, 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 $\mathsf{G}_{[0,N]}q$ and $\mathsf{F}_{[0,N]}p$ 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 MonPoly [6], which was the best performing tool in the "offline monitoring" track of the first international Competition on Software for Runtime Verification [4] (CSRV 2014). MonPoly, 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 the one described in our previous work [10], to which we compare.

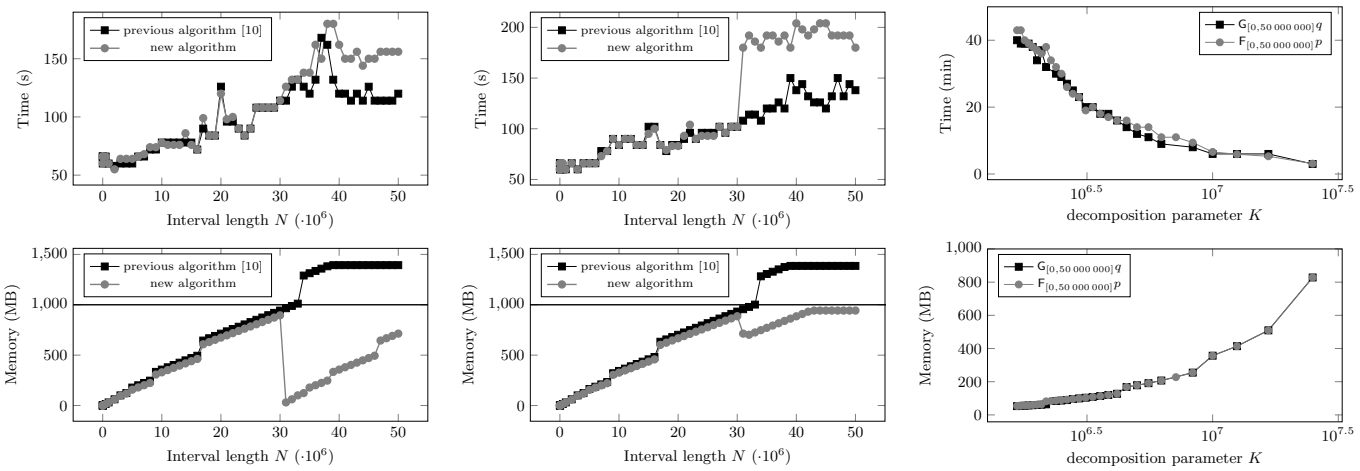
Plots in Figures 8a and 8b show the execution time and the memory usage required to check, respectively, formula $\mathsf{G}_{[0,N]}q$ and $\mathsf{F}_{[0,N]}p$, instantiated with different values of parameter N. Each data point is obtained by running the algorithm over the ten synthesized traces and averaging the results. The plots colored in black show the average time and memory usage of our previous algorithm [10], which applies $\mathsf{MTL}_{\mathsf{P}}$ semantics. The plots colored in gray represent the runs of our new algorithm that applies $\mathsf{MTL}_{\mathsf{L}}$ 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 1GB 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 1GB when the time interval N is higher than 30 000 000. Nevertheless, the proposed algorithm is on average 1.35x slower that 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 $\mathcal{L}_K$ 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 $\mathsf{G}_{[0,50\,000\,000]}q$ and $\mathsf{F}_{[0,50\,000\,000]}p$. These formulae are processed using the $\mathcal{L}_K$ decomposition, with values of K that are taken from $V = \{\frac{5 \cdot 10^7}{i} \mid i = 2, 3, \ldots\}$ 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 the 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 54.14MB 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



(a) Execution time (top) and memory usage (bottom) to check $\mathsf{G}_{[0,N]}q$

(b) Execution time (top) and memory usage (bottom) to check $\mathsf{F}_{[0,N]}p$

(c) Execution time (top) and memory usage (bottom) when varying parameter K

Figure 8: Scalability and time/memory tradeoffs for the proposed trace checking algorithm.

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 $\mathcal{L}_K$ decomposition of formulae, based on the configuration of the available cloud infrastructure.

10. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] Apache Software Foundation. Hadoop MapReduce. http://hadoop.apache.org/.

[3] B. Barre, M. Klein, M. Soucy-Boivin, P.-A. Ollivier, and S. Hallé. MapReduce for parallel trace validation of LTL properties. In *Proc. of RV 2012*, volume 7687 of *LNCS*, pages 184–198. Springer, 2012.

[4] E. Bartocci, B. Bonakdarpour, and Y. Falcone. First international competition on software for runtime verification. In *Proc of RV 2014*, volume 8734 of *LNCS*, pages 1–9. Springer, 2014.

[5] D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. Scalable offline monitoring. In *Proc of RV 2014*, volume 8734 of *LNCS*, pages 31–47. Springer, 2014.

[6] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. Monpoly: Monitoring usage-control policies. In *Proc. of RV 2011*, volume 7186 of *Lecture Notes in Computer Science*, pages 360–364, 2011.

[7] D. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu. Monitoring data usage in distributed systems. *IEEE Trans. Softw. Eng.*, 39(10):1403–1426, 2013.

[8] A. Bauer and Y. Falcone. Decentralised LTL monitoring. In *Proc of FM 2012*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.

[9] M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krstić, and P. San Pietro. Efficient large-scale trace checking using MapReduce. Extended version available online at http://arxiv.org/abs/1508.06613, 2015.

[10] D. Bianculli, C. Ghezzi, and S. Krstić. Trace checking of metric temporal logic with aggregating modalities using MapReduce. In *Proc. of SEFM 2014*, volume 8702 of *LNCS*, pages 144–158. Springer, 2014.

[11] D. Bianculli, C. Ghezzi, and P. San Pietro. The tale of SOLOIST: a specification language for service compositions interactions. In *Proc. of FACS 2012*, volume 7684, pages 55–72. Springer, 2012.

[12] P. Bouyer, F. Chevalier, and N. Markey. On the expressiveness of TPTL and MTL. *Information and Computation*, 208(2):97 – 116, 2010.

[13] A. Coen-Porisini, M. Pradella, and P. San Pietro. A finite-domain semantics for testing temporal logic specifications. In *Proc. of FTRTFT 1998*, volume 1486 of *LNCS*, pages 41–54. Springer, 1998.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[15] D. D'Souza and P. Prabhakar. On the expressiveness of MTL in the pointwise and continuous semantics. *International Journal on Software Tools for Technology Transfer*, 9(1):1–4, 2007.

[16] P. Faymonville, B. Finkbeiner, and D. Peled. Monitoring parametric temporal logic. In *Proc. of VMCAI 2014*, volume 8318 of *LNCS*, pages 357–375. Springer, 2014.

[17] M. Felder and A. Morzenti. Validating real-time systems by history-checking TRIO specifications. *ACM Trans. Softw. Eng. Methodol.*, 3(4):308–339, Oct. 1994.

[18] H.-M. Ho, J. Ouaknine, and J. Worrell. Online monitoring of metric temporal logic. In *Proc of RV 2014*, volume 8734 of *LNCS*, pages 178–192. Springer, 2014.

[19] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.

[20] S. Krstić. MTL-MapReduce. https://bitbucket.org/krle/mtlmapreduce.

[21] R. Medhat, Y. Joshi, B. Bonakdarpour, and S. Fischmeister. Parallelized runtime verification of first-order LTL specifications, 2014. Technical report.

[22] A. Morzenti, D. Mandrioli, and C. Ghezzi. A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14:521–573, October 1992.

[23] A. Mrad, S. Ahmed, S. Hallé, and E. Beaudet. Babeltrace: A collection of transducers for trace validation. In *Proc. of RV 2012*, volume 7687 of *LNCS*, pages 126–130. Springer, 2013.

[24] Public Law 104–191. Health Insurance Portability and Accountability Act of 1996 (HIPAA), 1996.

[25] G. Rosu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1), 2012.

[26] P. Thati and G. Rosu. Monitoring algorithms for metric temporal logic specifications. *Electr. Notes Theor. Comput. Sci*, 113:145–162, 2005.

[27] Wikipedia. Wikipedia page traffic statistics. http://aws.amazon.com/datasets/2596.

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI'12*, pages 2–2. USENIX Association, 2012.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of HotCloud 2010*. USENIX, 2010.