Hallé and Soucy-Boivin *Journal of Cloud Computing: Advances, Systems and Applications* (2015) 4:8
DOI 10.1186/s13677-015-0032-x

Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

MapReduce for parallel trace validation of LTL properties

Sylvain Hallé* and Maxime Soucy-Boivin

Abstract

We present an algorithm for the automated verification of Linear Temporal Logic formulæ on event traces using an increasingly popular cloud computing framework called MapReduce. The algorithm can process multiple, arbitrary fragments of the trace in parallel, and compute its final result through a cycle of runs of MapReduce instances. Experimentation on a variety of cloud-based MapReduce frameworks, including Apache Hadoop, show how complex LTL properties can be validated in reasonable time in a completely distributed fashion. Compared to the classical LTL evaluation algorithm, results show how the use of a MapReduce framework can provide an interesting alternative to existing trace analysis techniques, performance-wise, under favourable conditions.

Introduction

Over the recent years, the volume and complexity of interactions between information systems has been steadily increasing. Large amounts of data are gathered about these interactions, forming a trace of events, also called a *log*, that can be stored, mined, and audited. Web servers, operating systems, database engines and business processes of various kinds all produce event logs, crash reports, test traces or dumps in some format or another.

One possible use of such a log is to perform *trace validation*: given a specification of the expected or agreed-upon interaction (or inversely, of invalid behaviour), the trace of actions recorded at runtime can then be searched automatically for patterns satisfying or violating that specification. The specification generally relates events to some sequence of actions, method calls or events: the validity of each event cannot be assessed individually, but must rather be evaluated according to the event's position with respect to surrounding events, both before and after. As we shall see in Section 'Trace validation use cases', there exists a variety of scenarios where event traces are subject to sequencing constraints, and the use of a language such as Linear Temporal Logic represents a reasonable mean of expressing these constraints formally.

*Correspondence: shalle@acm.org
Laboratoire d'informatique formelle, Département d'informatique et de mathématique, Université du Québec à Chicoutimi, Chicoutimi (Québec) G7H 2B1, Canada

Various solutions have been proposed in the past to automate the task of trace validation [1-6], either based on temporal logic or other kinds of formal specifications. While these solutions allow the expression of intricate relationships between events in a log, the scalability of many of them is jeopardized by the growing amount of data generated by today's systems. Recently, the advent of *cloud computing* has been put forward as a potential remedy to this problem, in particular for the tasks of process discovery and conformance checking [7]. By allowing the distributed processing of data spread across a network of commodity hardware, cloud computing opens the way to dramatic improvements in the performance of many applications.

Given the growing amount of collected trace data and the observed move towards distributed computing infrastructures, it is crucial that existing trace validation methodologies be ported to the cloud paradigm. However, the prospect of parallel processing of temporal constraints in general, and LTL formulæ in particular, is held back precisely because of the sequential nature of the properties to verify: since the validity of an event may depend on past and future events, the handling of parts of the trace in parallel and independent processes seems to be disqualified at the onset. A review of available solutions in Section 'Related work' observes, perhaps unsurprisingly, that most existing trace validation tools are based on algorithms that do not take advantage of parallelism, while those that do offer very limited specification

languages where sequential relationships between events are excluded.

The present paper addresses this issue by presenting a *parallelizable* algorithm for the automated validation of LTL properties in event traces. The algorithm uses a recent and popular execution framework, called MapReduce [8], which is described in Section 'An overview of MapReduce'. MapReduce provides an environment particularly suitable to the breaking up of a task into small, independent processes that can be distributed across multiple nodes in a network, and is currently being used in large-scale applications such as the Google search engine for the computation of the PageRank index [9]. The algorithm, detailed in Section 'LTL trace validation with MapReduce', exploits this framework by splitting the original property into subformulæ that can be evaluated separately through cycles of MapReduce jobs.

The algorithm has been implemented in two distinct MapReduce environments: MrSim and Apache Hadoop. Experiments were conducted on evaluating sample LTL properties on traces of up to 10 million events, and compare their running time with a state-of-the-art, classical trace analyzer for LTL. Results from these experiments, described in Section 'Experimental results', show that the algorithm offers similar or better performance when mappers and reducers are executed in a purely sequential fashion. This first result confirms that the proposed algorithm is not intrinsically penalizing a user over existing solutions.

However, our experiments also reveal that, with parallelism turned on, the same batch of MapReduce jobs offers lesser performance, slowing down the process by as much as two orders of magnitude. This seems to suggest that LTL trace analysis is indeed a fundamentally linear process, and that attempts at distributing its computation are offset by the cost of parallel management (threads, inter-process communication, etc.).

To the best of our knowledge, the present work is the first application and analysis of MapReduce for the verification of temporal logic properties on event traces. The approach presents an interesting alternative to existing tools when the formula to verify is below a certain complexity threshold, in cases where LTL with past operators is required, or when the trace to analyze is fragmented across multiple computing nodes.

Trace validation use cases

We shall first recall basic concepts related to the validation of event traces in various contexts. For the needs of this paper, an *event trace* $m_0 m_1 \ldots$, noted $\overline{m}$, represents a sequence of events over a period of time. Each event is an individual entity, made of one or more parameter-value pairs of arbitrary names and types. The schema (that is, the number and names of each parameter in each event) is not assumed to be known in advance, or even to be consistent across all events.

Constraints on event sequences: linear temporal logic

Given an event trace, one is then interested in expressing properties or *constraints* that must be fulfilled either by individual events or sequences thereof. Given an event trace $\overline{m}$ and some constraint φ, we denote by $\overline{m} \models \varphi$ the fact that the trace satisfies the constraint. A variety of formal languages are available to describe constraints of different kinds; one of them is a logical formalism called Linear Temporal Logic (LTL), whose syntax is described in Figure 1. The basic building blocks of LTL formulæ are *propositional variables* p, q, $\ldots$, expressing Boolean conditions on particular messages of the trace. In the present context, each propositional variable is an assertion of the form parameter = value, which evaluates to true if the equality holds for the current message, and to false otherwise.

The complete semantics of LTL is given in Figure 2. One can evaluate when a trace $\overline{m}$ satisfies a given formula φ, written as $\overline{m} \models \varphi$, by giving conditions to be evaluated recursively on the structure of the formula. On top of propositional variables, LTL allows *Boolean connectives* $\vee$ (or), $\wedge$ (and), $\neg$ (not), bearing their usual meaning and *temporal operators* to express constraints on the sequence of events. The temporal operator **G** means "globally"; the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every event of the trace, starting from the current event. The operator **F** means "eventually"; the formula $\mathbf{F}\,\varphi$ is true if φ holds for some future event of the trace. The operator **X** means "next"; it is true whenever φ holds in the next event of the trace. Finally, the **U** operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if φ holds for all events until some event satisfies ψ. We also define $\varphi\,\mathbf{V}\,\psi$ as $\neg(\neg\varphi\,\mathbf{U}\,\neg\psi)$ and $\varphi\,\mathbf{W}\,\psi$ as $(\varphi\,\mathbf{U}\,\psi) \vee \mathbf{G}\,\varphi$. [a]

Two concepts bear particular importance in this paper. Given some operator $\star$ and a formula φ of the form $\star\varphi'$ or $\varphi' \star \psi$, expressions φ' and ψ are called the *direct subformulæ* of φ. Subformulæ form a partial ordering; we will denote as $\varphi' \prec \varphi$ the fact that φ' is a direct subformula of φ. The *depth* of a formula φ, noted $\delta(\varphi)$, is then defined as the maximum number of nested subformulæ it contains. For example, the expression $\mathbf{G}\,(p \wedge \mathbf{F}\,q)$ is of depth

$$\varphi := p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \mathbf{G}\,\varphi \mid \mathbf{F}\,\varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi$$

Figure 1 The syntax of linear temporal logic.

$$
\begin{aligned}
\overline{m} \models \neg\varphi &\equiv \overline{m} \not\models \varphi \\
\overline{m} \models \varphi \wedge \psi &\equiv \overline{m} \models \varphi \text{and } \overline{m} \models \psi \\
\overline{m} \models \varphi \vee \psi &\equiv \overline{m} \models \varphi \text{or } \overline{m} \models \psi \\
\overline{m} \models \varphi \rightarrow \psi &\equiv \overline{m} \not\models \varphi \text{or } \overline{m} \models \psi \\
\overline{m} \models \mathbf{X}\,\varphi &\equiv \overline{m}^1 \models \varphi \\
\overline{m} \models \mathbf{G}\,\varphi &\equiv m_0 \models \varphi \text{and } \overline{m}^1 \models \mathbf{G}\,\varphi \\
\overline{m} \models \mathbf{F}\,\varphi &\equiv m_0 \models \varphi \text{or } \overline{m}^1 \models \mathbf{F}\,\varphi \\
\overline{m} \models \varphi\,\mathbf{U}\,\psi &\equiv m_0 \models \psi \text{or both} \\
&\quad\; m_0 \models \varphi \text{and } \overline{m}^1 \models \varphi\,\mathbf{U}\,\psi \\
\overline{m} \models \forall_\pi x : \varphi &\equiv \text{for all } c \in Dom(\pi, m_0), \overline{m} \models \varphi[x/c] \\
\overline{m} \models \exists_\pi x : \varphi &\equiv \text{for some } c \in Dom(\pi, m_0), \overline{m} \models \varphi[x/c]
\end{aligned}
$$

Figure 2 The semantics of LTL operators, where $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ are LTL formulæ, $\overline{\boldsymbol{m}}$ is an event trace $\boldsymbol{m_0, m_1, m_2, \ldots}$ and $\overline{\boldsymbol{m}}^{\boldsymbol{i}}$ designates its suffix $\boldsymbol{m_i, m_{i+1}, \ldots}$.

3, and its set of proper subformulæ is $\{p \wedge \mathbf{F}\,q, p, \mathbf{F}\,q, q\}$. For a set of subformulæ S, we will say that φ is a (direct) *superformula* of ψ if $\varphi, \psi \in S$ and $\psi \prec \varphi$.

A use-case scenario

There exists a variety of scenarios where constraints on event traces can be modelled as LTL properties. This issue has gained considerable importance in the past decade with the advent of anti-fraud regulation such as the Sarbanes-Oxley Act (SOX) [10] or the Payment Card Industry Data Security Standard (PCI) [11], which require some form of storage and analysis of log files, such as database transaction history. We shall describe in the following a number of such scenarios described in past literature.

As a simple example, we recall an earlier work where a bookstore business process was modelled as a set of constraints in a language called DecSerFlow [12], [p.34]. The workflow is initiated by a customer placing an order (event place_c_order). This customer order is sent to and handled by the bookstore (event handle_order). The bookstore then transfers the order of the desired book to a publisher. If the bookstore receives a negative answer, it decides to either search for an alternative publisher or to reject the customer order (event c_reject). If the bookstore searches for an alternative publisher, a new bookstore order is sent to another publisher, etc. If the customer receives a negative answer (event rec_decl), then the workflow terminates. If the bookstore receives a positive answer (activity c_accept), the customer is informed (event rec_acc) and the bookstore continues processing the customer order.

From this workflow, the authors identify sequencing relationships between the various events that must be enforced for a valid transaction to take place. For example:

1. A customer order must eventually be acknowledged by the bookstore
2. Event rec_acc cannot occur unless some place_c_order has been seen previously

These relationships, expressed in a graphical notation called DecSerFlow, can be translated into equivalent LTL formulæ.

LTL can then be used to formalize the these properties. For example, the first property above becomes

$$\mathbf{G}\,(\text{place\_c\_order} \rightarrow \mathbf{F}\,(\text{rec\_acc}))$$

Similarly, the second can be expressed as:

$$(\neg\text{rec\_acc})\,\mathbf{U}\,\text{place\_c\_order}$$

We also mention that the same techniques used for LTL business process compliance can be reused for the verification of web service interface contracts [13], the detection of network intrusions in web server logs [14], and the analysis of system events produced by spacecraft hardware during testing [5].

Related work

Existing solutions for the validation of event traces can be split into two categories. On one side are formal trace validation tools, mostly experimental or academic, offering a rich input language but for which no parallel processing

algorithms are available; on the other side lie distributed log analysis products whose input language and validation capabilities are relatively limited.

Formal trace analysis

A first category of tools is made of so-called "formal" trace analyzers. Complex sequential patterns of events are expressed using a rich, mathematically-based notation such as finite-state machines, temporal logic or Petri nets. Algorithms are then developed to process these specifications and automatically check that some trace satisfies the given pattern.

In this realm, a wide variety of techniques have been developed for different purposes. When the specifications are written as temporal logic formulæ, algorithms can manipulate the expressions symbolically, and progressively rewrite the original specification as the trace is being read; the pattern is violated when this rewriting process transforms the specification into a contradiction. This idea has been implemented in two independent tools, respectively based on the Maude engine [15] and the Java programming language [13].

An alternate approach consists of storing the events into a database, and to transform the sequential patterns into an equivalent database query. This has been experimented with traditional relational databases and SQL [1], and more recently using XML databases and the XQuery language [2]. The database approach has also been followed, to some degree, by the Monpoly tool [3], which associates to each event in the trace a set of conditions on its values.

ProM [4] is an open-source environment aimed at the mining of patterns in large sets of log data. Among the many plugins developed for ProM, one can find a tool for the automated verification of LTL formulæ on process logs. Also worthy of mention are Logscope [5] and RuleR [6], which use their own input language loosely based on logic and finite-state machines.

However, none of the aforementioned tools is reported to offer parallel processing capabilities, and in particular the leveraging of cloud-based infrastructures, such as MapReduce, to that end. On the contrary, [16] uses parallelism by sharing the truth values of common atomic propositions of a past-time LTL (ptLTL) among multiple, low-hardware footprint micro-CPU cores. The evaluation of LTL properties has been offloaded to multiple GPUs in [17,18], in the latter case by first reducing the LTL properties to Büchi automata. The term "parallelism" has also been used in [19] in the limited context of executing the monitor of a temporal logic specification in a separate thread from the program being observed. However, these approaches are CPU or GPU-based, and do not attempt to leverage the MapReduce framework.

Distributed trace analysis

The second category of related work comprises so-called "log analysis" solutions. Most products in that category are commercial software aimed at the filtering of event data (such as database or server logs) to search for the presence of specific patterns. Notable examples include Snare [20], ManageEngine [21] or Splunk [22]; even operating systems such as Windows provide viewing and filtering capabilities for internal events. These tools can be seen as refined variants of the well-known "Grep" function, which performs pattern matching over an input file and returns lines corresponding to some regular expression. It can be seen as a (somehow limited) form of log analysis that can be used to return parts of an event trace corresponding to a filter expression. Indeed, such mechanism has also been proposed as the basis of trace validation tools in the past [23].

"Distributed Grep" [24] is the name given to the parallel version of this procedure, where the input file is split into chunks that can be processed independently. For each line ℓ read from an input file chunk, the Map function emits a tuple $\langle \ell, \emptyset \rangle$ if it matches a given pattern; the Reduce function just copies the supplied intermediate data to the output. As a matter of fact, most log analysis tools that leverage cloud infrastructures speed up their data processing using essentially this procedure. Of these aforementioned solutions, some offer the possibility to distribute the processing of filtering functions across multiple nodes in a network. This includes a log analysis tool called that p3 has been developed to analyze packet traces in the cloud using Apache's Hadoop distributed computing environment [25].

A problem arises, however, when one wants to query an event trace using a more articulate query language than single-line regular expressions. Linear Temporal Logic is a prime illustration of this problem: if p and q define single event patterns, a temporal expression like $\mathbf{G}\,(p \rightarrow \mathbf{X}\,q)$ validates whether an event that satisfies p is always immediately followed by an event that satisfies q. Events (or lines) are no longer compared individually, but rather with respect to their sequential relationship. The main hypothesis of the aforementioned techniques, namely that event processing can be done individually, no longer holds. If the two lines of some temporal pattern are stored on different chunks of the trace, and processed by independent parallel threads, the sequential relationship will be missed.

Therefore, in all the aforementioned solutions, the filtering process is generally limited to single events taken in isolation. For example, it is possible to obtain the list of all events satisfying some criterion on the event's attributes or to compute aggregate numerical statistics on events collected (such as total throughput, average delay, etc.), but not to fetch events in relation with other events, or satisfying some sequence or temporal pattern. Similarly, p3

can only be used to perform simple filtering on individual instances, or to compute aggregate numerical statistics on events collected (such as total throughput, average delay, etc.).

A close cousin to the approach presented in this paper has been exposed by Bauer and Falcone [26]. In this setting, multiple components in a system each observe a subset of some global event trace. Given an LTL property φ, their goal is to create sound formulæ derived from φ that can be monitored on each local trace, while minimizing inter-component communication. However, this work assumes that the projection of the global trace upon each component is well-defined and known in advance. Moreover, all components consume events from the trace synchronously, such that the distribution of monitoring does not result in a speed-up of the whole process.

An overview of MapReduce

Since the emergence of the concept of cloud computing a few years ago, a variety of distributed computing environments have been released. One notable proponent is MapReduce, a framework introduced by Google in 2004 for the processing of large amounts of data [8]. It is one of the forerunners of the so-called "NoSQL" trend, which has seen the development and rising popularity of alternative data processing schemes steering away from mainstream relational databases.

Figure 3 summarizes the schematics of MapReduce. Data processing starts by the reading of some piece of data (typically an input file) by an Input Reader, whose task is to convert the input stream into a set of tuples. Each tuple is a key-value pair, denoted $\langle q_i, v\rangle$, where both keys and values can be of arbitrary types.

As Figure 3 shows, multiple instances of the Input Reader can run in parallel, and typically process separate fragments of the input data simultaneously. The tuples produced by the Input Reader are then sent one by one to a Mapper, whose task is to convert each input tuple $\langle q_i, v\rangle$ into some output tuple $\langle k_i, v'\rangle$. The processing is stateless —that is, each tuple must be transformed independently of any previously-seen tuple, and regardless of the order in which tuples are received. For an input tuple, the Mapper may as well decide not to produce any output tuple.

The pool of tuples from all Mapper instances then goes through a shuffling step; all tuples with the same key are grouped and dispatched to the same instance of Reducer. Therefore, a Reducer that receives a tuple $\langle k_i, v\rangle$ is guaranteed to receive all other tuples $\langle k_i, v'\rangle$ for that same key k_i. For the sake of clarity, we can safely assume that each Reducer instance receives the tuples for exactly one key; we can hence parameterize each such instance with the key it has been assigned.

Contrarily to the Mapper, the Reducer receives its input tuples at once, and is hence allowed to iterate through and retain information about previously seen tuples. Again, the Reducer's task is to read the input tuples, and produce as output one final set of tuples of the form $\langle t_i, v\rangle$. This set of tuples can then be read, and formatted back to some output format by an Output Writer.

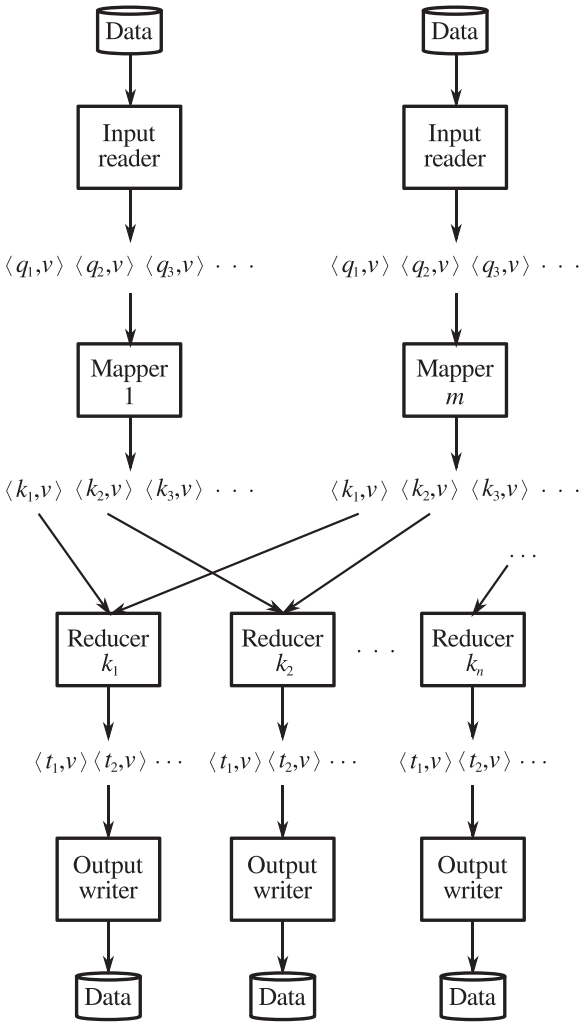

Figure 3 The different steps of MapReduce data processing.

In some cases, the Input Reader and Mapper may be fusioned into a single processing step, as is the case for the Reducer and Output Writer. Moreover, some definitions of MapReduce also imply that tuples are sorted according to their value before being fed to the Reducer, although we do not assume such sorting in the present paper.

Popular frameworks such as Google's or Apache's Hadoop [27] provide an environment and code libraries allowing one to write data processing tasks as MapReduce jobs. It generally suffices to write the (Java or Python) code for the Map and Reduce phases of the processing, compile it and send it to the nodes of the cloud infrastructure.

One can see from this simple description that the keys and values produced by a processing step need not be (and generally are not) the same for input and output. In the same way, there is no fixed relationship between the number of tuples read and the number of tuples sent out; a Mapper or Reducer processing some tuple may return zero, one, or even more than one tuple as output.

Moreover, it is possible to chain multiple MapReduce phases. It suffices to take the output of the Reducers as input for a subsequent cycle of Mappers. Google's PageRank algorithm is computed through three MapReduce phases, the second of which is repeated until convergence of some numerical value is reached [9]. The algorithm for Mappers and Reducers differs from phase to phase.

Although the MapReduce scheme is arguably less natural than a classical, linear program to an inexperienced developer, its architecture presents one key advantage: once a problem has been correctly split into Map and Reduce jobs, scaling up the processing to multiple nodes in the cloud becomes straightforward. Indeed, multiple Input Readers can simultaneously take care of a separate chunk of the input data. Then, since the Map step processes each tuple regardless of any past or future tuple, an arbitrary number of Mappers can process the tuples generated by the Input Readers in parallel. Similarly, the processing done by each Reducer only requires access to tuples of the same key, which entails that up to one Reducer per key can run in parallel. All in all, the whole processing chain greatly decreases the number of steps that require to be done in sequence. A good review of MapReduce's pros and cons can be found in Lee et al. [28].

LTL trace validation with MapReduce

Despite the potential parallelism brought about by the use of the MapReduce paradigm, the fundamental question of whether LTL trace validation is parallelizable remained open until very recently. We have already shown that, if one is to leverage distributed cloud frameworks for LTL querying of event traces, simple mechanisms such as Distributed Grep and their derivatives cannot be used directly.

Kuhtz and Finkbeiner showed in 2009 that LTL path checking belongs to the complexity class $\mathsf{AC}^1(\mathsf{logDCFL})$ [29]; this result entails that the process can be efficiently split by evaluating entire blocks of events in parallel. Rather than sequentially traversing the trace, their work considers the circuit that results from "unrolling" the formula over the trace. However, while the evaluation of this unrolling can be done in parallel, a specific type of Boolean circuit requires to be built in advance, which depends on the length of the trace to evaluate. Moreover, the formal demonstration of the result shows that, while a fixed number of gates of this circuit can be contracted in parallel at each step of the process, the algorithm itself requires a shared and global access to the trace from every parallel process. As such, it does not lend itself directly to distributed computing frameworks.

We take an alternate approach, and describe in this section an algorithm that performs LTL trace validation on event traces directly using the MapReduce computing paradigm. The algorithm evaluates an LTL formula in an iterative fashion. At the first iteration, all the states where ground terms are true are evaluated. In the next iteration, these results are used to evaluate all subformulæ directly using one of those ground terms. More generally, at the end of iteration i of the process, the events where all subformulæ of depth i hold are computed. It follows that, in order to evaluate an LTL formula of depth n, the algorithm will require exactly n MapReduce cycles. Each MapReduce cycle effectively acts as a form of temporal tester [30] processing a trace made of the evaluation of lower-level testers.

This does not mean, however, that the event trace must be read as many times. In fact, the input trace is entirely read only once, at the first iteration of the procedure. Afterwards, only sequential numbers referring to those events need to be passed between mappers and reducers. The contents of the original trace never need to be consulted ever again.

The system is described by providing details on each component of the MapReduce algorithm described in Figure 3. We suppose that every instance of the process (Input Reader, Mapper, Reducer, Output Writer) are parameterized by the formula to verify φ, and the length of the trace, ℓ.

We will illustrate the workings of this algorithm through a simple example, by considering the formula

$$\varphi \equiv \mathbf{G}\,(\neg c \vee \mathbf{F}\,(a \vee b))$$

evaluated on the trace a,c,a,d,c,d,b.

Trace format and input reader

The Input Reader is responsible for the processing of a set of events from the trace to read and the generation of a first set of key-value tuples from that set. We assume that each event is sequentially numbered, or that its position in the whole trace can be easily computed otherwise. For some event e, we will refer by $\#(e)$ this event's sequential number.

The Input Reader, whose algorithm is given in Figure 4a, iterates through each event of the trace chunk, and evaluates on each event the ground terms present in φ. For each propositional variable a and each event e, it outputs a tuple $\langle a, (i, 0)\rangle$ where i is the event's sequential number in the trace. The ground terms of a formula φ are computed using the function $\mathrm{atom}(\varphi)$.

```
Procedure Input Reader_{φ,ℓ}(chunk)
  A[] := atoms(φ)
  For each e in chunk do
    i := #(e)
    For each a in A do
      If e ⊨ a then
        output ⟨a, (i, 0)⟩
      End if
    End
  End
```

(a) Pseudo-code for the LTL Input Reader

```
Procedure Mapper_{φ,ℓ}(⟨ψ, (n, i)⟩)
  If i ≤ δ(ψ)
    S[] := superformulæ(φ, ψ)
    For each ξ in S do
      output ⟨ξ, (ψ, n, i + 1)⟩
    End If
  End
End If
```

(b) Pseudo-code for the LTL Mapper.

Figure 4 Pseudo-code for the LTL Input Reader **(a)** and Mapper **(b)**.

At the first iteration of the process on our sample formula, the InputR eader (or multiple input readers) process the trace and generates the first set of tuples:

$$\langle a,(0,0)\rangle, \langle a,(2,0)\rangle, \langle b,(6,0)\rangle, \langle \neg c,(0,0)\rangle,$$
$$\langle \neg c,(2,0)\rangle, \langle \neg c,(3,0)\rangle, \langle \neg c,(5,0)\rangle, \langle \neg c,(6,0)\rangle$$

One should remark that this initial processing step does not require that the trace be located on a single node, or even that each node's fragment consist of blocks of successive events. As long as each event can be placed in some total order (such as the value of a global, shared clock), any number of nodes can host any subset of the trace. This is particularly useful if event collection and storage is performed in a distributed fashion.

Mapper

The Mapper takes as input tuples of the form $\langle \psi, (n,i)\rangle$, either from the Input Reader or from the output of a previous MapReduce cycle. Each such tuple reads as "the process is at iteration i, and subformula ψ is true on event n". One can see, in particular, how the tuples returned by the Input Reader express this fact for ground terms of the formula to verify.

The Mapper, shown in Figure 4b, is responsible for lifting these results, computed for some ψ, up into every formulæ ψ' of which ψ is a *direct* subformula (these are obtained using the function superformulæ(φ,ψ)). For example, if the states where p is true have been computed, then these results can be used to determine the states where $\mathbf{F}\,p$ is true. To this end, the Mapper takes every tuple $\langle \psi,(n,i)\rangle$, and will output a tuple $\langle \psi',(\psi,n,i+1)\rangle$, where ψ is a subformula of ψ'. This tuple reads "the process is at iteration $i+1$, subformula ψ is true on event n, and this must be used to evaluate ψ'". In the definition of the reducer, ξ stands for whatever subformula the input tuple is build from.

On our example, the tuples produced by the Input Reader at the previous step are sent to mappers which produce the following output tuples:

$$\langle a\vee b,(a,0,1)\rangle, \langle a\vee b,(a,2,1)\rangle, \langle a\vee b,(b,6,1)\rangle, \langle \neg c\vee \mathbf{F}(a\vee b),(\neg c,0,1)\rangle,$$
$$\langle \neg c\vee \mathbf{F}(a\vee b),(\neg c,2,1)\rangle, \langle \neg c\vee \mathbf{F}\,(a\vee b),(\neg c,3,1)\rangle, \langle \neg c\vee \mathbf{F}(a\vee b),(\neg c,5,1)\rangle,$$
$$\langle \neg c\vee \mathbf{F}\,(a\vee b),(\neg c,6,1)\rangle$$

Reducer

The mappers are mostly used to prepare results from the last iteration to be used for the current iteration. In contrast, each instance of the reducer performs the actual evaluation of one more layer of the temporal formula to verify. After the shuffling step, each individual instance of the reducer receives all generated tuples of the form $\langle \psi',(\psi,n,i)\rangle$ for some formula ψ', and where ψ is a direct subformula of ψ'. Hence, the reducer is given information on all the event numbers for which ψ' holds, and is asked to compute the states where ψ holds based on this information. This task can then be decomposed depending on the top-level connective in ψ'. The algorithm for each reducer is shown in Figure 5.

When the top-level formula to evaluate is $\mathbf{X}\,\psi$, the events that satisfy the formula are exactly those immediately preceding an event where ψ holds. Consequently, the reducer iterates through its input tuples of $\langle \mathbf{X}\,\psi,(\psi,n,i)\rangle$ and produces for each one an output tuple $\langle \mathbf{X}\,\psi,(n-1,i)\rangle$.

When the top-level formula to evaluate is $\mathbf{F}\,\psi$, the events that satisfy the formula are exactly those for which some event in the future is such that ψ holds. The corresponding reducer iterates through the input tuples and computes the highest event number c for which ψ holds. All events preceding c satisfy $\mathbf{F}\,\psi$. Consequently, the reducer generates as output all tuples of the form $\langle \mathbf{F}\,\psi,(k,i)\rangle$, for each $k\in[\,0,c]$.

The reducer for $\neg\psi$ iterates through all tuples and stores in a Boolean array whether $e_i \models \psi$ for each event i in the trace. It then outputs a tuple $\langle \neg\psi,(k,i)\rangle$ for all event numbers k that were not seen in the input. The reducer for $\mathbf{G}\,\psi$ proceeds in reverse. It first iterates through all tuples in the same way. If we let c be the index of the last event for which ψ does not hold, the reducer will then output all tuples $\langle \mathbf{G}\,\psi,(k,i)\rangle$ for $k\in[\,c+1,\ell]$. This indeed corresponds to all events for which $\mathbf{G}\,\psi$ holds.

```
Procedure Reducer_{φ,ℓ}(F ψ, tuples[])
  m := -1
  For each ⟨F ψ, (ξ, n, i)⟩ in tuples do
    If n > m then m := n
  End
  For k from 0 to m do
    output ⟨F ψ, (k, i)⟩
  End

Procedure Reducer_{φ,ℓ}(¬ψ, tuples[])
  For each ⟨¬ψ, (ξ, n, i)⟩ in tuples do
    s[n] := ⊤
  End
  For k from 0 to ℓ do
    If s[k] ≠ ⊤ then
      output ⟨¬ψ, (k, i)⟩
    End If
  End

Procedure Reducer_{φ,ℓ}(G ψ, tuples[])
  For each ⟨G ψ, (ξ, n, i)⟩ in tuples do
    s[n] := ⊤
  End
  For k from ℓ to 0 do
    If s[k] ≠ ⊤ break
    output ⟨G ψ, (k, i)⟩
  End

Procedure Reducer_{φ,ℓ}(ψ ∨ ψ', tuples[])
  For each ⟨ψ ∨ ψ', (ξ, n, i)⟩ in tuples do
    If δ(ψ ∨ ψ') ≠ i then
      output ⟨ξ, (n, i)⟩
    Else
      output ⟨ψ ∨ ψ', (n, i)⟩
    End If
  End

Procedure Reducer_{φ,ℓ}(X ψ, tuples[])
  For each ⟨X ψ, (ξ, n, i)⟩ in tuples do
    output ⟨X ψ, (n − 1, i)⟩
  End

Procedure Reducer_{φ,ℓ}(ψ ∧ ψ', tuples[])
  For each ⟨ψ ∧ ψ', (ξ, n, i)⟩ in tuples do
    If δ(ψ ∧ ψ') ≠ i then
      output ⟨ξ, (n, i)⟩
      s_ξ[n] := ⊤
    End If
    If s_ψ[n] := ⊤ and s_ψ'[n] := ⊤ then
      output ⟨ψ ∧ ψ', (n, i)⟩
    End If
  End

Procedure Reducer_{φ,ℓ}(ψ U ψ', tuples[])
  For each ⟨ψ U ψ', (ξ, n, i)⟩ in tuples do
    If δ(ψ U ψ') ≠ i then
      output ⟨ξ, (n, i)⟩
    End If
    s_ξ[n] := ⊤
  End
  b := ⊥
  For k from ℓ to 0 do
    If s_ψ'[n] = ⊤ then
      output ⟨ψ U ψ', (k, i)⟩
      b := ⊤
    Else If s_ψ[n] := ⊤ and b = ⊤ then
      output ⟨ψ U ψ', (k, i)⟩
    Else
      b := ⊥
    End If
  End
```

Figure 5 Pseudo-code for the LTL Reducers.

The case of binary connectives $\vee$ and $\wedge$ is slightly more delicate. Special care must be taken to persist tuples whose result will be used in a later iteration. Consider the case of formula $(\mathbf{F}\,p) \wedge q$. The states where ground terms p and q hold will be computed by the Input Reader at iteration 0. However, although q is a direct subformula of $(\mathbf{F}\,p) \wedge q$, one has to wait until iteration 2 to combine it to $\mathbf{F}\,p$, evaluated at iteration 1. More precisely, a tuple $\langle \psi \star \psi', (\psi, n, i)\rangle$ can only be evaluated at iteration $\delta(\psi \star \psi')$; in all previous iterations, tuples $\langle \psi, (n, i)\rangle$ must be put back in circulation. The first condition in both reducers' algorithm takes care of this situation.

Otherwise, when the top-level formula to evaluate is $\psi \vee \psi'$, the reducer outputs a tuple $\langle \psi \vee \psi', (n, i)\rangle$ whenever it reads input tuples $\langle \psi \vee \psi', (\psi, n, i)\rangle$ or $\langle \psi \vee \psi', (\psi', n, i)\rangle$. When the top-level formula is $\psi \wedge \psi'$, the reducer must memorize event numbers n for which it has read tuples $\langle \psi \wedge \psi', (\psi, n, i)\rangle$ and $\langle \psi \wedge \psi', (\psi', n, i)\rangle$, and outputs $\langle \psi \wedge \psi', (n, i)\rangle$ as soon as it has seen both. The last case to consider is that of a formula of the form $\psi\,\mathbf{U}\,\psi'$. The reducer first iterates through all its input tuples and memorizes the event numbers for which ψ holds, and those for which ψ' holds. It then proceeds backwards from the last event of the trace, and outputs $\langle \psi\,\mathbf{U}\,\psi', (n, i)\rangle$ for some state n if ψ' holds for n, or if ψ holds for n and there exists an uninterrupted sequence of states leading to a state n' for which ψ' holds. This last information is handled through the Boolean variable b.

On our example formula, the reducer for $a \vee b$ will receive the first three tuples and output $\langle a \vee b, (0, 1)\rangle$, $\langle a \vee b, (2, 1)\rangle$, $\langle a \vee b, (6, 1)\rangle$. Since the iteration number is 1, and the depth of $\neg c \vee \mathbf{F}\,(a \vee b)$ is 3, the reducer for $\neg c \vee \mathbf{F}\,(a \vee b)$ will simply re-output the tuples

$$\langle \neg c, (0, 1)\rangle, \langle \neg c, (2, 1)\rangle, \langle \neg c, (3, 1)\rangle, \langle \neg c, (5, 1)\rangle, \langle \neg c, (6, 1)\rangle$$

As one can see, the tuples produced by each reducer is of the form $\langle \psi, (n, i)\rangle$, carrying the exact same meaning as those originally produced by the Input Reader, albeit for formulæ of greater depth. Therefore, the result of one MapReduce cycle can be fed back as input of a new cycle; as we have seen, it takes exactly $\delta(\varphi)$ such cycles to completely evaluate some LTL formula φ.

Output writer

At the end of the last MapReduce cycle, one is left with tuples $\langle \varphi, (n, \delta(\varphi))\rangle$. These represent all event numbers n such that $\overline{m}^n \models \varphi$. The output writer, shown in Figure 6, translates the last set of tuples into the truth value of the formula to evaluate. By the semantics of LTL, an event trace satisfies the formula φ if $\overline{m}^0 \models \varphi$. Hence the output

```
Procedure Output Writer_{φ,ℓ}(tuples[])
  For each ⟨φ, (n, i)⟩ in tuples do
    If n = 0 then
      output "Formula is true"
      Break
    End if
  End
  output "Formula is false"
```

Figure 6 Pseudo-code for the LTL output writer.

writer simply writes "true" if $\langle \varphi, (0, \delta(\varphi)) \rangle$ is found, and false otherwise.

A complete example

For the sake of completeness, the remaining iterations of MapReduce processing on our example formula are given below.

Iteration 2

The tuples produced by the first round of Reducers are sent to mappers for a second cycle, producing:

$$\langle \mathbf{F}(a \vee b), (a \vee b, 0, 2) \rangle, \langle \mathbf{F}(a \vee b), (a \vee b, 2, 2) \rangle, \langle \mathbf{F}(a \vee b), (a \vee b, 6, 2) \rangle,$$
$$\langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 0, 2) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 2, 2) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 3, 2) \rangle,$$
$$\langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 5, 2) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 6, 2) \rangle.$$

The reducer for $\mathbf{F}(a \vee b)$ will produce:

$$\langle \mathbf{F}(a \vee b), (0, 2) \rangle, \langle \mathbf{F}(a \vee b), (1, 2) \rangle, \langle \mathbf{F}(a \vee b), (2, 2) \rangle, \langle \mathbf{F}(a \vee b), (3, 2) \rangle,$$
$$\langle \mathbf{F}(a \vee b), (4, 2) \rangle, \langle \mathbf{F}(a \vee b), (5, 2) \rangle, \langle \mathbf{F}(a \vee b), (6, 2) \rangle$$

while the reducer for $\neg c \vee \mathbf{F}(a \vee b)$ will again re-output:

$$\langle \neg c, (0, 2) \rangle, \langle \neg c, (2, 2) \rangle, \langle \neg c, (3, 2) \rangle, \langle \neg c, (5, 2) \rangle, \langle \neg c, (6, 2) \rangle.$$

Iteration 3

The tuples are sent into the penultimate cycle; the mappers will produce:

$$\langle \neg c \vee \mathbf{F}(a \vee b), (\mathbf{F}(a \vee b), 0, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\mathbf{F}(a \vee b), 1, 3) \rangle,$$
$$\langle \neg c \vee \mathbf{F}(a \vee b), (\mathbf{F}(a \vee b), 2, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\mathbf{F}(a \vee b), 3, 3) \rangle,$$
$$\langle \neg c \vee \mathbf{F}(a \vee b), (\mathbf{F}(a \vee b), 4, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\mathbf{F}(a \vee b), 5, 3) \rangle,$$
$$\langle \neg c \vee \mathbf{F}(a \vee b), (\mathbf{F}(a \vee b), 6, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 0, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 2, 3) \rangle,$$
$$\langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 3, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 5, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (\neg c, 6, 3) \rangle.$$

The reducer for $\neg c \vee \mathbf{F}(a \vee b)$ will output:

$$\langle \neg c \vee \mathbf{F}(a \vee b), (0, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (1, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (2, 3) \rangle,$$
$$\langle \neg c \vee \mathbf{F}(a \vee b), (3, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (4, 3) \rangle, \langle \neg c \vee \mathbf{F}(a \vee b), (5, 3) \rangle,$$
$$\langle \neg c \vee \mathbf{F}(a \vee b), (6, 3) \rangle.$$

Iteration 4

For the last iteration, the mappers produce

$$\langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (\neg c \vee \mathbf{F}(a \vee b), 0, 4) \rangle, \langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (\neg c \vee \mathbf{F}(a \vee b), 1, 4) \rangle,$$
$$\langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (\neg c \vee \mathbf{F}(a \vee b), 2, 4) \rangle, \langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (\neg c \vee \mathbf{F}(a \vee b), 3, 4) \rangle,$$
$$\langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (\neg c \vee \mathbf{F}(a \vee b), 4, 4) \rangle, \langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (\neg c \vee \mathbf{F}(a \vee b), 5, 4) \rangle,$$
$$\langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (\neg c \vee \mathbf{F}(a \vee b), 6, 4) \rangle$$

The reducer for $\mathbf{G}(\neg c \vee \mathbf{F}(a \vee b))$ computes the output tuples

$$\langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (0, 4) \rangle, \langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (1, 4) \rangle \langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (2, 4) \rangle$$
$$\langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (3, 4) \rangle \langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (4, 4) \rangle \langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (5, 4) \rangle$$
$$\langle \mathbf{G}(\neg c \vee \mathbf{F}(a \vee b)), (6, 4) \rangle$$

Finally, since event number 0 is part of the tuple set, the output writer concludes that the formula is true for the trace considered.

From this simple example, one can see how multiple input readers can process separate chunks of the same event trace independently. As a matter of fact, each input reader does not even require that its chunk contains sets of consecutive events, as long as each event is properly numbered according to its sequential position. In addition, as a side effect of fitting the problem into the MapReduce framework, an arbitrary number of parallel Mapper instances can be used to process a set of input tuples at each cycle. Similarly, up to one Reducer per key (that is, one per subformula) can run in parallel in the Reduce phase of a cycle.

LTL with past

The use of the present algorithm for LTL validation provides a number of interesting side effects. The most notable one is that the evaluation of LTL *past* operators can be obtained mostly "for free". For instance the translation of operator $\mathbf{Y}$ (the past version of $\mathbf{X}$) simply amounts to replacing $n-1$ by $n+1$ in the definition of the reducer for $\mathbf{X}$. A similar reasoning can be applied for the remaining past temporal operators, like $\mathbf{P}$ (or $\mathbf{F}^{-1}$), $\mathbf{H}$ ($\mathbf{G}^{-1}$) or $\mathbf{S}$ ($\mathbf{U}^{-1}$). Furthermore, unusual operators such as the $\mathbf{C}$ ("chop") modality [31] can also be defined easily with their custom reducer.

Experimental results

To illustrate the concept and evaluate its feasibility, we implemented the algorithm described earlier in two different MapReduce frameworks, and compared it to an existing trace analysis tool called BeepBeep, which uses a classical, non-parallel algorithm. Experiments were then conducted to compare their running time on the same set of traces; the point of the comparison is to get an intuition whether using MapReduce can provide an improvement over existing techniques, performance-wise.

Sample properties

To assess the running time of the MapReduce validation algorithm, we built a dataset consisting of traces of randomly-generated events, with each event being made of up to ten random parameters, labelled $p_0, \ldots p_9$, each carrying five possible values. Each trace has a length between 1 and 100,000 events, and 500 such traces were produced. In total, this dataset amounts to more than one gigabyte of randomly-generated event data.

Four properties, with increasing complexity, were verified on these traces. Property #1 is $\mathbf{G}\, p_0 \neq 0$, and simply asserts that in every event, parameter p_0, when present, is never equal to 0. Property #2 is $\mathbf{G}\,(p_0 = 0 \rightarrow \mathbf{X}\, p_1 = 0)$: it expresses the fact that whenever $p_0 = 0$ in some event, the next event is such that $p_1 = 0$. Property #3 is a generalization of Property #2:

$$\forall x \in [0, 9] : \mathbf{G}\,(p_0 = x \rightarrow \mathbf{X}\, p_1 = x)$$

This property asserts that whatever value taken by p_0 will be taken by p_1 in the next event. The universal and existential quantifiers are meant as a shorthand notation; the actual LTL formula to be validated is the logical conjunction of the previous template for all possible values of x between 0 and 9, and reads

$$(\mathbf{G}\,(p_0 = 0 \rightarrow \mathbf{X}\, p_1 = 0)) \wedge (\mathbf{G}\,(p_0 = 1 \rightarrow \mathbf{X}\, p_1 = 1)) \ldots$$

Finally, Property #4 checks that *some* parameter p_m alternates between two possible values; this is true when the value of p_m in the current event is the same as the value two events from the current one, and is written:

$$\exists m \in [0, 9] : \forall x \in [0, 9] : \mathbf{G}\,(p_m = x \rightarrow \mathbf{X}\,\mathbf{X}\, p_m = x)$$

Again, the quantifiers are meant as a shorthand.

Execution environments

Our execution environment for the validation of properties consists of a virtual cluster inside a Solaris 11.1 server, equipped with 24 GB of RAM. One reason for the choice of Solaris is its possibility to create isolated environments, called *zones*, without the need for a full-fledged virtual machine. Each zone has its own resource controller, and communicates with the rest of the environment only through a connection using virtual network interfaces. This makes it easy to create and manage computing nodes.

The practice of using Solaris zones to create a Hadoop cluster is well known and documented [32]. Advantages of using such an architecture include fast provision of new cluster members using the zone cloning feature, very high network throughput between the zones for data node replication, optimized disk I/O utilization, and secure data at rest using ZFS encryption. In our setup, the cluster is made of five nodes: a master "name-node" whose task is to manage Hadoop jobs, three "data-nodes" that perform the actual Map and Reduce operations, and a backup of the name node that can resume its job should a failure occur. It is out of the scope of this paper to describe in detail the architecture of the cluster; we followed the basic setup steps contained in documentation from Oracle and available online [32].

We now proceed to describe the trace analysis tools that were run on the cluster. It shall be noted that only Hadoop requires a multi-node (and hence multi-zone) setup. The remaining tools were executed in the default, "global" zone of the system.

BeepBeep

The first tool included in our survey is a runtime monitor we developed in earlier work called BeepBeep [33].[b] BeepBeep receives a stream of events produced by some application or process, and constantly analyzes it against a specification given beforehand. When the stream of events deviates from what the specification stipulates, a signal is sent which can then be piped into another program for further processing. It can also work in offline mode and analyze a pre-recorded trace of events taken from a file.

Although BeepBeep accepts as input an LTL expression, its processing uses a completely different algorithm from the one described in this paper. This algorithm is *not* based on MapReduce: it is sequential, and requires a single process to analyze each event of the trace one by one in their proper order.

A recent benchmark has showed that BeepBeep provides performance in the average of a large number of other trace validation solutions [34]. It was included in our analysis as the baseline case, being representative of the kind of performance that classical solutions provide. It will hence be possible to compare the running time of our proposed MapReduce solution and measure any actual benefits in terms of performance.

MrSim

The second environment we used for the comparison is a hybrid between sequential trace processing and distributed MapReduce, called MrSim. MrSim [35] is a simple implementation of MapReduce in Java, intended for a pedagogical illustration of the programming model. It originates from frustrating experiences using other frameworks, which require a lengthy and cumbersome setup before running even the simplest example. In most cases those examples are entangled with technical considerations (distributed file system, network configuration) that distract from learning the MapReduce programming model itself.

MrSim aims at providing a simple framework to create and test MapReduce jobs with a minimal setup, using straightforward implementations of all necessary concepts. This entails some purposeful limitations to the

system: for example, it is not optimized in any way. In counterpart, MrSim offers interesting features from a pedagogical point of view: it runs out of the box, and the centralized processing makes it easy to perform step-by-step debugging of a MapReduce job.

MrSim can run in two modes. In sequential mode, the actual coordination of Map and Reduce jobs is done locally on a single machine using a single-thread implementation of MapReduce: the data source is fed tuple by tuple to the mapper, the output tuples are collected, split according to their keys, and each list is sent to the reducer, again in a sequential fashion. As such, this sequential workflow reproduces exactly the processing done by MapReduce environments, without the distribution of computation. This was done on purpose, so that the running time of each mapper and reducer instance could be easily measured.

In multi-threaded mode, Map and Reduce jobs of the same iteration each run in a distinct thread provided by a thread dispatcher. The dispatcher is instantiated with a parameter n specifying the maximum number of concurrent threads. The first n Map and Reduce jobs that execute are immediately given a thread; remaining jobs in excess of n, if any, are put into a waiting queue for one of the existing threads to terminate. In terms of performance, the operation in sequential mode is equivalent to multi-threaded when $n = 1$ (barring some light thread management overhead), as was confirmed by a set of preliminary setup tests.

A first observation that can be made is that this execution environment is the simplest of all we tested: excluding the code for coordinating Mappers and Reducers (itself made of only 250 lines of code), the total implementation of the validator amounts to 1,000 lines of Java code. This should be put in contrast with BeepBeep, which is also implemented in Java and rather uses the classical, on-the-fly algorithm for the evaluation of LTL formulæ on traces [13]; BeepBeep is is made up of twice as many lines of Java code.

Hadoop

The third environment used in our experiments is Apache Hadoop version 1.2.1, already introduced earlier.

While Hadoop is regularly presented as the canonical example of a MapReduce implementation, it turns out that there are significant differences between the *theoretical* principle of MapReduce described in Section 'An overview of MapReduce', and the *real-life* implementation of MapReduce in Apache Hadoop.

Chaining MapReduce jobs In the theoretical MapReduce framework (and in MrSim), tuples output by reducers can be sent directly as input to mappers, making multiple iterations of MapReduce cycles possible. Hadoop does not support this: tuples produced by reducers must be sent serialized to an output collector, saved to a file, and then be re-read from an input collector and converted back into tuples. This makes the chaining of cycles of MapReduce jobs, necessary in our context for but the simplest LTL formulæ, very cumbersome, inefficient, and ultimately uncalled for, as the MrSim environment does not require such a mandatory serialization to chain cycles of MapReduce jobs.

Line input format A second limitation of the Hadoop implementation is the fact that input readers are line-based —that is, an input reader is fed one line at a time from the input source, and elements from which tuples are created cannot span multiple lines. While this behaviour is appropriate for simple log formats, it makes it hard to support rich data models such as, in our case, XML. We had to preprocess our input traces to remove line endings inside all events, so that each event occupies exactly one line. Again, input readers in MrSim do not present this limitation and can be fed arbitrary chunks of an input source.

Object inheritance An LTL expression is represented as a top-level LTL operator, which in turn may contain a number of children operators, hence creating a nested structure representing the contents of the formula. Every operator is a subclass of the general class LTLOperator; it is therefore natural to declare the key of tuples as an object of type LTLOperator, and this is precisely what is done in MrSim.

Yet, Hadoop does not support inheritance when manipulating tuples. This means that if a map or a reduce job is declared to accept tuples of type (K, V), Hadoop throws a runtime error when trying to process a tuple of type (K', V'), where K' and V' are descendants (in the object-oriented sense of the term) of K and V respectively. This poses a serious problem in our context, which necessitated rewriting our representation of LTL formulæ as a single object, using a member field whose value makes it behave like a conjunction, an operator **G**, etc. This unexplained limitation belongs to Hadoop, and is not an inherent limitation of the MapReduce principle, as MrSim does not present this problem. This arguably renders the validation code inelegant, and again, most assuredly affects its performance.

Results

To test and compare each validation solution, we randomly generated traces of XML events with abstract names `p0`, `p1`, etc., each containing a random integer value. The length of each trace varied from less than 10 events to more than 9 million; in this last case, the XML file weighed just short of 1 gigabyte. Each property P1–P4

was validated on each trace, and various statistics on the process were computed.

All tools operate in the same way: they accept as input a character string (the same for every tool) representing an LTL formula, and a filename pointing to an XML trace saved locally. Each tool then processes the formula and the trace using its own algorithm (the classical LTL evaluation algorithm for BeepBeep, and the MapReduce implementation described in this paper for the others).

It should be noted at the onset that all three MapReduce solutions (Hadoop and the two versions of MrSim) crashed when evaluating Property 4 on the two largest traces we generated (respectively 3.5 and 9 million events). Apart from these two events, every tool successfully completed each verification task and returned the same (correct) verdict.

Number of tuples

The first measurement is the number of tuples produced by the algorithm. This value is taken as the sum of T_i, the total number of tuples processed at the Map phase of each MapReduce cycle number i, for all cycles $i \in [1, \delta(\varphi)]$, as shown in Table 1. One can see that the number of tuples increases with the complexity of the formula: while Property #1 produces 180,000 tuples for a trace, the validation of Property #4 on the same trace generates more than 8 million such units.

For example, validating property P2 produces roughly twice as many tuples, in total, than there are events in the trace to analyze. From this ratio, it is possible to guess that the total number of tuples that would have been produced when evaluating property 4 on the largest trace is around 400 million.

The distribution of tuples across iterations is far from uniform, however. Table 2 shows the number of tuples produced by the reduce phase of each cycle for each property and two different traces.

We also computed the "sequential ratio" of the validation process. At each MapReduce cycle, we keep the largest number of tuples processed by a single instance of a reducer. This value, noted t_i, represents the minimum number of tuples that must be processed sequentially in that particular cycle. If all reducers for that cycle were allowed to run in parallel, and assuming similar processing time for each tuple, the ratio t_i/T_i is an indicator of the time the "parallel" cycle requires with respect to the "sequential" version. The global sequential ratio shown in Table 3 is taken as

$$s = \frac{\sum_{i=1}^{\delta(\varphi)} t_i}{\sum_{i=1}^{\delta(\varphi)} T_i}$$

This sequential ratio shows one of the limits of the validation algorithm in its present incarnation: the potential for parallelism is bounded by the structure of the formula to validate, as there can be at most one instance of reducer for every possible subformula of the property to verify.

Table 1 The total number of tuples produced for the validation of each properties on traces of increasing sizes

Trace size	P1	P2	P3	P4
11	14	17	20	221
180,035	180,062	346,642	815,449	8,194,477
1,770,922	1,770,938	3,409,468	8,010,419	80,552,405
3,523,211	3,523,218	6,782,937	15,938,586	> 54 M
9,025,596	9,025,603	17,375,057	40,830,370	—
Ratio	1	1.9	4.5	45

Table 2 Number of tuples produced at each MapReduce iteration for the validation of properties P1–P4, for a trace of a few kilobytes (a) and a trace of around 100 megabytes (b)

(a)

Iteration	P1	P2	P3	P4
1	8	8	9	11
2	3	3	4	43
3	3	3	4	42
4		3	3	59
5				34
6				24
7				8

(b)

Iteration	P1	P2	P3	P4
1	132,392	132,392	662,013	856,275
2	1,638,530	1,638,530	3,674,133	26,563,830
3	16	1,638,530	3,674,133	26,563,830
4		16	47	26,564,584
5			31	1,508
6			31	1,022
7			31	750
8				336
9				28
10				28
11				54
12				80
13				80

Table 3 Sequential ratio each of the four properties P1–P4

	Property #1	Property #2	Property #3	Property #4
Sequential ratio	100 %	92 %	19 %	3 %

Therefore, for simple formulæ such as Property #1 and #2, which have very few different subformulæ at each MapReduce cycle, almost all the work must be done sequentially (100% in the case of Property #1, and 92% in the case of Property #2). However, as soon as the property becomes more complex, as is the case for Properties #3 and #4, the situation is reversed, and each reducer handles a small fraction of the total number of tuples. Property #4 is most dramatic in that respect, since 97% of all tuples involved can be processed in parallel. The presence of quantifiers accounts for a large part of this phenomenon, as it rapidly blows up the size of the actual LTL formula passed to the trace validator: 50 copies of the same template are validated, with various combinations of values for m and x.

Running time

The second measurement is the total running time required to vaildate each property, on traces of increasing size. A summary of these results is given in Table 4, which shows, for each property, the average running time per event, which is the total processing time divided by the number of events in the trace; the value shown in the table is the average of these values over all traces.

From the sequential ratio s and the average sequential running time per event r obtained for each property, it is also possible to infer the theoretical validation time in the maximally-parallel case by computing $r \times s$; this inferred running time is also shown in Table 4, using the sequential run of MrSim as the baseline.

A first observation that can be made is that, for most properties, the BeepBeep runtime monitor is *many* orders of magnitude slower than any MapReduce implementation. Even the sequential MrSim setup provides runtime performance superior to BeepBeep, indicating that the use of a MapReduce algorithm in itself presents an intrinsic performance advantage.

Detailed results on the running time for every property, and trace are plotted, for each validation environment, in Figure 7. The figure reveals a running time that is roughly linear in terms of the length of the trace for all properties and all tools. The rate of growth, however, varies widely from one tool to the next, with BeepBeep faring the worst. Surprisingly, the multi-thread versions of MrSim and the Hadoop cluster exhibit slower validation times than the sequential version of MrSim; depending on the properties, these tools are sometimes 40 times slower than the sequential version of MrSim. The running times estimated from the sequential ratio (last line of Table 4) are also sometimes orders of magnitude lower than the actual running times observed with the multi-thread version of MrSim. Since the sequential MrSim uses the same Map and Reduce jobs, the culprit cannot lie in the algorithm itself, but rather in the introduction of parallelism.

Table 4 Average running time in microseconds per event for each property and each tool

Tool	P1	P2	P3	P4
BeepBeep	15,788	6,332	0.2	31
MrSim (sequential)	1.9	3.6	10.4	335
Hadoop cluster	44	96	243	4,694
MrSim (multi-thread)	73	138	433	5,734
MrSim (predicted)	1.9	3.3	2.0	10

Worst-case bandwidth

Given that all three MapReduce approaches crashed for traces and formulæ producing the most tuples, we performed a theoretical analysis of the bandwidth required to evaluate a formula in the worst csae.

First, one can realize that a tuple can be reduced to at most four integers. Assuming 32-bit integers, a tuple can hence accommodate roughly 4 billion events (2^{32}), as many subformulæ and iteration cycles, and be serialized as a 12-byte string.

We can then estimate the maximum number of tuples that can be generated during the evaluation of a formula. There can be as many tuples as there are events in the trace to process, and one such tuple can be produced for each subformula of the formula to verify. If we let $|\varphi|$ be the length (i.e. number of symbols) of φ, one can conclude that there are at most $2|\varphi|$ subformulæ; indeed, each logical connective occurring in φ brings at most two proper subformulæ. Hence the cumulative tuple bandwidth B to be exchanged can be given by:

$$B = T \times \delta(\varphi) \times \ell \times 2|\sigma(\varphi)|$$

where T is the tuple size, $\delta(\varphi)$ is the depth of the formula to verify (and hence the number of MapReduce cycles), ℓ is the event trace length, and $2|\sigma(\varphi)|$ denotes the total number of subformulæ. The $\ell \times 2|\sigma(\varphi)|$ term is indeed a crude worst case bound, as it assumes that each event of the input trace generates one tuple for each subformula and at each cycle, that all subformulæ are present at each cycle (they obviously aren't), and that the formula is only composed of binary connectives (it generally isn't). It also assumes that every produced tuple must be transmitted to another site to be processed in the next step in the algorithm. For the properties given as an example in this paper, $|\sigma(\varphi)|$ is effectively bounded by 3.

One can see that, while this bound can amount to a large number of tuples, it nevertheless remains linear both in the length of the trace and the size of the LTL formula to evaluate. Therefore, the cheerless results we obtained with the parallel versions of the algorithm seem to be related to their implementation, and not to asymptotic complexity.

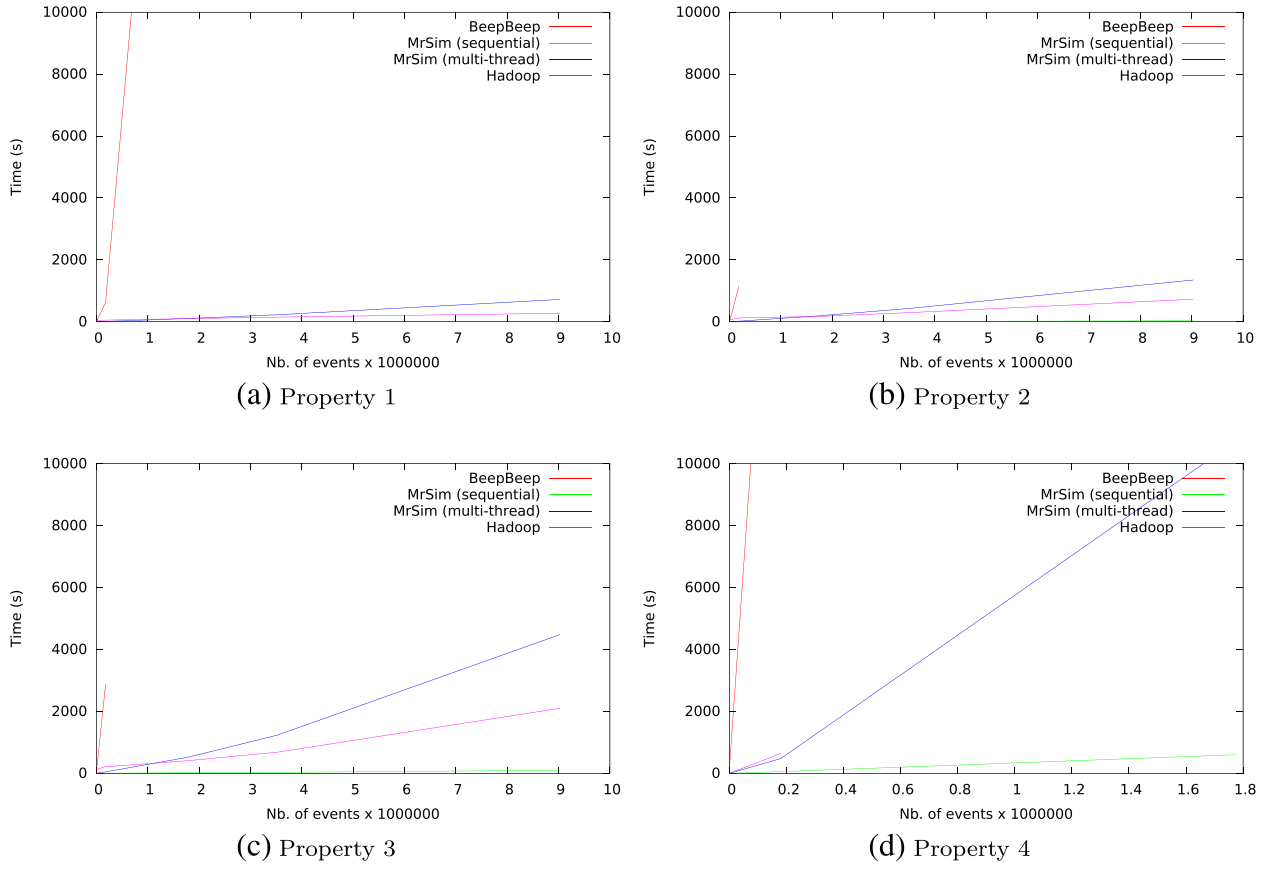

Figure 7 Running time for the validation of properties P1 **(a)** to P4 **(d)**, using the four validtion methods presented in this paper.

This is confirmed by the much better performance of the single-threaded MapReduce implementation.

To simplify the notation and reduce bandwidth, a simple modification can be made to the current method by computing all subformulæ of φ in advance and later on designate them with a single digit, as shown in Table 5 for the formula used as an example throughout the paper. This modification effectively reduces the amount of data that needs to be exchanged between mappers and reducers, and emphasizes the fact that no manipulation of the formulæ is necessary during the validation process (apart from fetching formulæ from the table to determine which action to follow).

Table 5 Shorthand symbols can be assigned to each subformula of the property to verify

Formula	Symbol
a	0
b	1
$\neg c$	2
$a \vee b$	3
$\mathbf{F}\ (a \vee b)$	4
$\neg c \vee (\mathbf{F}\ (a \vee b))$	5
$\mathbf{G}\ (\neg c \vee (\mathbf{F}\ (a \vee b)))$	6

Discussion and conclusion

In this paper, we have presented an algorithm for the automated validation of Linear Temporal Logic properties on large traces of events using the MapReduce development framework. As far as we know, this work is the first published algorithm that leverages the MapReduce framework for the validation of temporal logic properties on large event traces. It opens the way to the use of cloud computing services for the efficient compliance checking of program traces and event logs of various kinds.

Summary

We have shown experimentally on a sample dataset how the algorithm presents reasonable running times when the MapReduce environment is restricted to a single thread. The breaking up of the algorithm into several phases of

independent mappers and reducers presents the potential of reducing the number of operations that must be performed linearly by executing these processes in parallel, yielding a *potential* speedup of 90% in some cases.

We were surprised to discover that among the three MapReduce implementations, the two that use parallelism (Hadoop and the multi-threaded MrSim) are *vastly* outperformed by the sequential, single-threaded implementation of MrSim –by more than two orders of magnitude in most cases. Since all three versions were given the same map and reduce jobs, the culprit cannot be put on the algorithm we propose and is therefore inherent in the actual environment used. We could not witness any of the potential execution time savings brought on by the use of parallel processing, as the sequential ratio of Table 3 led us to believe.

Strata and monotonic logic

This tends to indicate that verifying temporal properties on a trace of events is an essentially linear process, and that its breaking up into parallel steps requires a communication overhead that largely offsets the presence of parallelism. In some cases, we observed that the number of tuples produced (and hence the amount of data exchanged) amounts to more than 40 times the number of events in the trace to analyze. It might be tempting to explain the relatively poor runtime performance of the distributed tools by the volume of tuples that need to be produced and exchanged between nodes or threads; yet the reader shall be reminded that the sequential version of MrSim produces and manages the same tuples as well.

This sequential nature of the LTL validation process bears close resemblance to the concept of *stratification* in distributed Datalog [36]. In this context, the execution of a distributed database query is divided into "non-monotonic stratification boundaries": the evaluation of each strata can be split into multiple independent and distributed processes, but a global coordination of all process at strata N must be done before any process of strata $N+1$ can start. In the case of LTL property evaluation, the boundaries can clearly be equated to the levels of nesting of each subformula: evaluating subformulaæ of a level of nesting $N-1$ can be done independently of each other, but the evaluation of a subformula of level of nesting N may require using the result of any subformula of level N, and hence a global coordination —materialized by the shuffling of tuples between reducers and mappers of the next iteration.

Potential uses

The results described in this paper are promising. They show how the use of a MapReduce framework can provide much better runtime performance than the classical LTL algorithm (sometimes by many orders of magnitude), especially for large traces. Despite the mild disappointment at the runtime performance of the Hadoop cluster, we conclude that the use of MapReduce to perform log analysis still is a viable solution that can analyze, in the worst case, hundreds if not thousands of events per second on very complex LTL formulæ. In addition to the performance argument, there also exist situations where the use of MapReduce is the *only* option; we list a few cases below.

- The property contains LTL past operators. Existing solutions (including BeepBeep) cannot handle LTL with past, while we have seen in Section 'LTL with past' that the MapReduce implementation provides these operators for free.
- The trace to analyze is fragmented across multiple locations. Existing trace validation tools all require the trace to be accessible sequentially from start to finish, which in general entails that the trace must be reconstructed and saved in a single location prior to analysis. In contrast, in the MapReduce implementation each single event can be located arbitrarily in any node, and each node needs not even to store contiguous sets of events. As a matter of fact, as long as the trace can be unambiguously ordered, events can be produced and recorded in multiple locations. This makes it particularly suited to verify, e.g. transaction processing systems [37].
- Using a professional MapReduce environment such as Hadoop provides side benefits, such as failure protection, generally absent from trace validators like BeepBeep or ProM.

Extensions and future work

The results obtained on the implementation discussed in this paper lead to a number of extensions and improvements over the current method. First, the algorithm presents an interest in that it can be reused as a basis for other temporal languages that intersect with LTL. This is the case, for example, of specifications written as finite-state machines, PSL [38] or DecSerFlow [12]. It is expected that similar techniques could also apply to to other logical formalisms, such as deontic logic [39]. Second, the technique itself could be expanded to take into account data parameters and quantification; the formulæ described in Section 'Sample properties' gave a foretaste of such quantification and initial results indicate that quantification is a fertile ground for parallelism. The proposed implementation is currently being ported as a free software suite for Apache Hadoop.

Finally, we have seen that the potential for parallelism is bounded by the structure of the formula to validate, as there can be at most one instance of reducer for every

possible subformula of the property to verify. This entails that one cannot freely distribute the processing of the trace to an arbitrary number of parallel processes: for a simple formula, or one that contains few nested expressions, few reducers can be started in parallel. Therefore, a sought after refinement of the current method is currently being worked on, which will allow multiple Reducer instances for the same key to be merged in a later step.

Endnotes

[a]We implicitly assume a finite-trace semantics where $\epsilon \not\models \mathbf{X}\,\varphi$, $\epsilon \not\models \mathbf{F}\,\varphi$, and $\epsilon \models \mathbf{G}\,\varphi$, where ϵ represents the empty trace.

[b]http://beepbeep.sourceforge.net. The analysis in this paper has been done on version 1.7.6.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
SH was responsible for designing the MapReduce pseudo-code and providing the initial implementation of MrSim. MSB ported this work to the Hadoop framework and performed all the experiments and data analysis. Both authors read and approved the final manuscript.

Received: 3 September 2014 Accepted: 16 March 2015
Published online: 14 April 2015

References

1. Böhlen MH, Chomicki J, Snodgrass RT, Toman D (1996) Querying TSQL2 Databases with Temporal Logic. In: Apers PMG, Bouzeghoub M, Gardarin G (eds). EDBT Volume 1057 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 325–341
2. Hallé S, Villemaire R (2008) XML Methods for Validation of Temporal Properties on Message Traces With Data. In: Meersman R, Tari Z (eds). CoopIS/DOA/ODBASE, Volume 5331 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 337–353
3. Basin DA, Klaedtke F, Müller S (2010) Policy Monitoring in First-Order Temporal Logic. In: Touili T, Cook B, Jackson P (eds). CAV, Volume 6174 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 1–18
4. Verbeek HMW, Buijs JCAM, van Dongen BF, van der Aalst WMP XES (2010) ESame, and ProM 6. In: Soffer P, Proper E (eds). CAiSE Forum, Volume 72 of Lecture Notes in Business Information Processing. Springer, Heidelberg. pp 60–75
5. Barringer H, Groce A, Havelund K, Smith M (2010) Formal Analysis of Log Files. J Aerospace Comput Inf Commun 7(11):365–390
6. Barringer H, Rydeheard D, Havelund K (2010) Rule Systems for Run-Time Monitoring: From Eagle to RuleR. J Logic Comput 20(3):675–706
7. van der Aalst WMP (2012) Distributed Process Discovery and Conformance Checking. In: de Lara J, Zisman A (eds). FASE, Volume 7212 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 1–25
8. Dean J, Ghemawat S (2004) MapReduce: Simplified Data Processing on Large Clusters. In: OSDI. USENIX, Berkeley, CA. pp 137–150
9. Page L, Brin S, Motwani R, Winograd T (1999) The PageRank Citation Ranking Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab 1999, [http://ilpubs.stanford.edu:8090/422/]
10. An act to protect investors by improving the accuracy and reliability of corporate disclosures made pursuant to the securities laws, and for other purposes (2002). [U.S. Pub.L. 107-204, 116 Stat. 745]
11. Payment Card Industry Data Security Standard, version 2.0(2010). [https://www.pcisecuritystandards.org/security_standards/pci_dss.shtml]
12. van der Aalst WMP, Pesic M (2007) Specifying and Monitoring Service Flows: Making Web Services Process-Aware. In: Baresi L, Nitto ED (eds). Test and Analysis of Web Services. Springer, Heidelberg. pp 11–55
13. Hallé S, Villemaire R (2011) Runtime Enforcement of Web Service Message Contracts with Data. IEEE Trans Serv Comput 5(2):192–206. [doi:10.1109/TSC.2011.10]
14. Naldurg P, Sen K, Thati P (2004) A Temporal Logic Based Framework for Intrusion Detection. In: de Frutos-Escrig D, Núñez M (eds). FORTE, Volume 3235 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 359–376
15. Rosu G, Havelund K (2005) Rewriting-Based Techniques for Runtime Verification. Autom Softw Eng 12(2):151–197
16. Reinbacher T, Geist J, Moosbrugger P, Horauer M, Steininger A (2012) Parallel runtime verification of temporal properties for embedded software. In: MESA. IEEE, New York, NY. pp 224–231
17. Medhat R, Joshi Y, Bonakdarpour B, Fischmeister S (2014) Parallelized Runtime Verification of First-order LTL Specifications. Tech. Rep. CS-2014-11, University of Waterloo
18. Berkovich S (2012) Parallel Run-Time Verification. Master's thesis, [https://uwspace.uwaterloo.ca/bitstream/handle/10012/7252/Berkovich_Shay.pdf]
19. Elmas T, Okur S, Tasiran S (2011) Rethinking Runtime Verification on Hundreds of Cores: Challenges and Opportunities. Tech. Rep. UCB/EECS-2011-74, EECS Department, University of California, Berkeley [http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-74.html]
20. Snare: gathering and filtering IT-event data (2014). http://www.intersectalliance.com/projects/index.html
21. ManageEngine: Network Management Software (2014). http://www.manageengine.com
22. Splunk: Operational Intelligence, Log Management, Application Management, Enterprise Security and Compliance (2014). http://www.splunk.com
23. Garavel H, Mateescu R (2004) SEQ.OPEN: A Tool for Efficient Trace-Based Verification. In: Graf S, Mounier L (eds). SPIN, Volume 2989 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 151–157
24. Amazon Web Services (2008) Building GrepTheWeb in the Cloud, Part 1: Cloud Architectures. Tech. rep [http://aws.amazon.com/articles/1632]
25. Lee Y, Kang W, Lee Y (2011) A Hadoop-Based Packet Trace Processing Tool. In: Domingo-Pascual J, Shavitt Y, Uhlig S (eds). TMA, Volume 6613 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 51–63
26. Bauer A, Falcone Y Decentralized LTL Monitoring. Tech. Rep arXiv:1111.5133v3 2011
27. Hadoop web site (2014). http://hadoop.apache.org
28. Lee KH, Lee YJ, Choi H, Chung YD, Moon B (2011) Parallel data processing with MapReduce: a survey. SIGMOD Record 40(4):11–20
29. Kuhtz L, Finkbeiner B (2009) LTL Path Checking Is Efficiently Parallelizable. In: Albers S, Marchetti-Spaccamela A, Matias Y, Nikoletseas SE, Thomas W (eds). ICALP (2), Volume 5556 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 235–246
30. Pnueli A, Zaks A (2008) On the Merits of Temporal Testers. In: Grumberg O, Veith H (eds). 25 Years of Model Checking, Volume 5000 of Lecture Notes in Computer Science. Springer, Heidelberg. pp 172–195
31. Harel D, Kozen D, Parikh R (1980) Process Logic: Expressiveness, Decidability, Completeness. In: FOCS, IEEE Computer Society, Los Alamitos, CA. pp 129–142
32. Kimchi O (2013) How to Set Up a Hadoop Cluster Using Oracle Solaris Zones. [http://www.oracle.com/technetwork/articles/servers-storage-admin/howto-setup-hadoop-zones-1899993.html]
33. Hallé S, Villemaire R (2012) Runtime Enforcement of Web Service Message Contracts with Data. IEEE T Serv Comput 5(2):192–206
34. Vallet J, Mrad A, Hallé S (2013) The Relational Database Engine: an Efficient Validator of Temporal Properties on Event Traces. In: EDOCW. IEEE Computer Society, Los Alamitos, CA. pp 285–294
35. MrSim project page. (2014). http://github.com/sylvainhalle/MrSim
36. Hellerstein JM (2010) The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Record 39:5–19
37. Su G, Iyengar A (2012) A highly available transaction processing system with non-disruptive failure handling. In: NOMS. IEEE, New York, NY. pp 409–416
38. Eisner C, Fisman D (2006) A Practical Introduction to PSL. Springer, Heidelberg
39. Åqvist L (1994) Deontic Logic, Kluwer, Alphen aan den Rijn