Parallelized Runtime Verification of First-order LTL Specifications

Ramy Medhat, Yogi Joshi, Borzoo Bonakdarpour, and Sebastian Fischmeister
University of Waterloo, Canada
{rmedhat, y2joshi, sfischme}@uwaterloo.ca, borzoo@cs.uwaterloo.ca

Technical report: CS-2014-11
April 2014

ABSTRACT

Runtime verification is an effective automated method for specification-based offline testing and analysis as well as online monitoring of complex systems. The specification language is often a variant of regular expressions or a popular temporal logic, such as LTL. This paper presents a novel and efficient parallel algorithm for verifying a highly expressive fragment of first-order LTL specifications, where nested quantifiers can be subject to second-order numerical constraints. Such constraints are useful in evaluating thresholds (e.g., expected uptime of a web server). The significance of this extension is that it enables us to reason about the correctness of a large class of systems, such as web servers, OS kernels, and network behavior, where properties are required to be instantiated for parameterized requests, kernel objects, network nodes, etc. Our algorithm uses the popular *MapReduce* architecture to split a program trace into variable-based clusters at run time. Each cluster is then mapped to its respective monitor instances, verified, and reduced collectively on a multi-core CPU or the GPU. Our algorithm is fully implemented and we report very encouraging experimental results, where the monitoring overhead is negligible on real-world data sets.

1. INTRODUCTION

In this paper, we study runtime verification of properties specified in a fragment first-order linear temporal logic (LTL) with second-order numerical constraints. Runtime verification (RV) is an automated specification-based technique, where a *monitor* evaluates the correctness of a set of logical properties on a particular execution either on the fly (i.e., at run time) or based on log files. Runtime verification complements exhaustive approaches such as model checking and theorem proving and under-approximated methods such as testing. First-order properties are of particular interest, as they can express parametric requirements on types of execution entities (e.g., processes and threads), user- and kernel-level events and objects (e.g., locks, files, sockets), web services (e.g., requests and responses), and network traffic. For example, the requirement 'every open file should eventually be closed' specifies a rule for causal and temporal order of opening and closing individual objects which generalizes to *all* files. Such properties cannot be expressed using traditional RV frameworks, where the specification language is propositional LTL or regular expressions.

In this paper, we extend the 4-valued semantics of LTL, designed for runtime verification (LTL_4) [1], to first-order LTL_4 with second-order numerical constraints and propose an efficient parallel algorithm for their verification at run time. The syntax of our language $\text{LTL}_4{-}\text{FO}_c$ allows formulas that include nested universal and existential quantifiers over data variables followed by an LTL subformula in terms of n-ary predicates. Each quantifier may be subject to a numerical constraint. For example, the following $\text{LTL}_4{-}\text{FO}_c$ formula:

$$\forall x : \mathsf{user}(x) \Rightarrow (\exists_{\leq 3}\, r : \mathsf{rid}(r) \Rightarrow (\mathsf{login} \wedge \mathsf{unauthorized}))$$

intends to capture the requirement that 'for all users, there exist at most 3 requests of type login that end with an unauthorized status'. Also, the formula:

$$\forall_{\geq 0.95}\, s : \mathsf{socket}(s) \Rightarrow (\mathbf{G}\, \mathsf{receive}\,(s) \implies \mathbf{F}\, \mathsf{respond}\,(s))$$

intends to express the property that 'at least 95% of open TCP/UDP sockets must eventually be closed'. The semantics of $\text{LTL}_4{-}\text{FO}_c$ is defined over six truth values:

- **True** ($\top$) denotes that the property is already permanently satisfied.
- **False** ($\bot$) denotes that the property is already permanently violated.
- **Currently true** ($\top_c$) denotes that the current execution satisfies the quantifier constraint of the property, yet it is possible that an extension violates the constraint.
- **Currently false** ($\bot_c$) denotes that the current execution violates the quantifier constraint of the property, yet it is possible that an extension satisfies it.
- **Presumably true** ($\top_p$) denotes that the current execution satisfies the inner LTL property and the quantifier constraint of the property.
- **Presumably false** ($\bot_p$) denotes that the current execution violates the inner LTL property and the quantifier constraint of the property.

We claim that these truth values provide us with informative verdicts about the status of different components of properties (i.e., quantifiers and their numerical constraints as well as the inner LTL formula) at run time.

The second contribution of this paper is a divide-and-conquer-based online monitor generation technique for an $\text{LTL}_4{-}\text{FO}_c$ specification. In fact, $\text{LTL}_4{-}\text{FO}_c$ monitors have to

be generated at run time, otherwise, an enormous number of monitors (i.e., in the size of cross-product of domains of all variables), which is clearly impractical. Our technique first synthesizes an LTL$_4$ monitor for the inner LTL property of LTL$_4$−FO$_c$ properties pre-compile time using the technique in [1]. Then, based upon the values of variables observed at run time, submonitors are generated and merged to compute the current truth value of a property for the current program trace.

Our third contribution is an algorithm that implements the above approach for verification of LTL$_4$−FO$_c$ properties at run time. This algorithm enjoys two levels of parallelism: the monitor (1) works in parallel with the program under inspection, and (2) evaluates properties in a parallel fashion as well. While the former ensures that the runtime monitor does not intervene with the normal operation of the program under inspection, the latter attempts to maximize the throughput of the monitor. The algorithm utilizes the popular *MapReduce* technique to (1) spawn submonitors that aim at evaluating subformulas using partial quantifier elimination, and (2) merge partial evaluations to compute the current truth value of properties.

Our parallel algorithm for verification of LTL$_4$−FO$_c$ properties is fully implemented on multi-core CPU and GPU technologies. We report rigorous experimental results by conducting three real-world independent case studies. The first case study is concerned with monitoring HTTP requests and responses on an Apache Web Server. The second case study attempts to monitor users uploading maximum chunk packets repeatedly to a personal cloud storage service based on a dataset for profiling DropBox traffic. The third case study monitors a network proxy cache to reduce the bandwidth usage of online video services, based on a YouTube request dataset. We present performance results comparing single-core CPU, multi-core CPU, and GPU implementations. Our results show that our GPU-based implementation provides an average speed up of 7x when compared to single-core CPU, and 1.75x when compared to multi-core CPU. The CPU utilization of the GPU-based implementation is negligible compared to multi-core CPU, freeing up the system to perform more computation. Thus, the GPU-based implementation manages to provide competitive speedup while maintaining a low CPU utilization, which are two goals that the CPU cannot achieve at the same time. Put it another way, the GPU-based implementation incurs minimal monitoring costs while maintaining a high throughput.

The rest of the paper is organized as follows. Section 2 describes the syntax and semantics of LTL$_4$−FO$_c$. In Section 3, we explain our online monitoring approach, while Section 4 presents our parallelization technique based on MapReduce. Experimental results are presented in Section 5. Related work is discussed in Section 6. Finally, we make concluding remarks and discuss future work in Section 7.

2. FIRST-ORDER LTL WITH NUMERICAL CONSTRAINTS

To introduce our logic, we first define a set of basic concepts.

DEFINITION 1 (PREDICATE). *Let $V = \{x_1, x_2, \ldots, x_n\}$ be a set of variables with (possibly infinite) domains $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$, respectively. A* predicate *p is a binary-valued function on the domains of variables in V such that*

$$p : \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n \rightarrow \{true, false\}$$ ■

The arity of a predicate is the number of variables it accepts. A predicate is *uninterpreted* if the domain of variables are not known concrete sets. For instance, $p(x_1, x_2)$ is an uninterpreted predicate, yet we can interpret it as (for instance) a binary function that checks whether or not x_1 is less than x_2 over natural numbers.

Let UP be a finite set of uninterpreted predicates, and let $\Sigma = 2^{UP}$ be the power set of UP. We call each element of Σ an *event*.

DEFINITION 2 (TRACE). *A* trace *$w = w_0 w_1 \cdots$ is a finite or infinite sequence of events; i.e, $w_i \in \Sigma$, for all $i \geq 0$.* ■

We denote the set of all infinite traces by Σ^ω and the set of all finite traces by Σ^*. A *program trace* is a sequence of events, where each event consists of *interpreted* predicates only. For instance, the following trace is a program trace:

$$w = \{\mathsf{open}(1), \mathsf{r}, \mathsf{anony})\}\, \{\mathsf{open}(2), \mathsf{rw}, \mathsf{user}(5)\}\, \cdots$$

where open and user are unary predicates and r, anony, and rw are 0-arity predicates. Predicate open is interpreted as opening a file, r is interpreted as read-only permissions, anony is interpreted as an anonymous user, and so on.

2.1 Syntax of LTL4-FOC

DEFINITION 3 (LTL$_4$−FO$_c$ SYNTAX). LTL$_4$−FO$_c$ *formulas are defined using the following grammar:*

$$\varphi ::= \forall_{\sim k}\, x : p(x) \Rightarrow \varphi \mid \exists_{\sim l}\, x : p(x) \Rightarrow \varphi \mid \psi$$
$$\psi ::= \top \mid p\,(x_1 \cdots x_n) \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \mathbf{X}\,\psi \mid \psi_1\, \mathbf{U}\, \psi_2$$

where x, $x_1 \cdots x_n$ are variables with possibly infinite domains $\mathcal{D}, \mathcal{D}_1, \cdots \mathcal{D}_n$, $\sim \in \{<, \leq, >, \geq, =\}$, $k : \mathbb{R} \in [0, 1]$, $l \in \mathbb{Z}^+$, $\mathbf{X}$ is the next, and $\mathbf{U}$ is the until temporal operators. ■

If we omit the numerical constraint in $\forall_{\sim k}$ (respectively, $\exists_{\sim l}$), we mean $\forall_{=1}$ (respectively, $\exists_{\geq 1}$). The syntax of LTL$_4$−FO$_c$ forces constructing formulas, where a string of quantifiers is followed by a quantifier-free formula.

Consider LTL$_4$−FO$_c$ property $\varphi = \forall x : p(x) \Rightarrow \psi$, where the domain of x is $\mathcal{D}$. This property denotes that for any possible valuation of the variable x ($[x := v]$), if $p(v)$ holds, then ψ should hold. If $p(v)$ does not hold, then $p(v) \Rightarrow \psi$ trivially evaluates to true. This effectively means that the quantifier $\forall x$ is in fact applied only over the following subdomain:

$$\{v \in \mathcal{D} \mid p(v)\} \subseteq \mathcal{D}$$

To give an intuition, consider the scenarios where file management anomalies can cause serious problems at run time (e.g., in NASA's Spirit Rover on Mars in 2004). For example, the following LTL$_4$−FO$_c$ property expresses "if a process wants to open a new file, then at least half of the files that it has previously opened must be closed":

$$\varphi_1 = \forall_{\geq 50\%}\, f : \mathsf{intrace}(f) \Rightarrow (\mathsf{opened}(f)\, \mathbf{U}\, \mathsf{close}(f)) \qquad (1)$$

where intrace denotes the fact that the concrete file appeared in any event in the trace.

2.2 4-Valued LTL [1]

First, we note that the syntax of LTL$_4$ can be easily obtained from Definition 3 by (1) removing the quantifier rules and (2) reducing the arity of predicates to 0 (i.e., predicates become atomic propositions).

2.2.1 FLTL

To introduce LTL$_4$ semantics, we first introduce Finite LTL. Finite LTL (FLTL) [11] allows us to reason about finite traces for verifying properties at run time. The semantics of FLTL is based on the truth values $\mathbb{B}_2 = \{\top, \bot\}$.

DEFINITION 4 (FLTL SEMANTICS). *Let φ and ψ be* LTL *properties, and $u = u_0 u_1 \cdots u_{n-1}$ be a finite trace.*

$$[u \models_{\text{F}} \mathbf{X}\,\varphi] = \begin{cases} [u_1 \models_{\text{F}} \varphi] & \text{if } u_1 \neq \epsilon \\ \bot & \text{otherwise} \end{cases}$$

$$[u \models_{\text{F}} \varphi\,\mathbf{U}\,\psi] = \begin{cases} \top & \exists k \in [0, n-1] : [u_k \models_{\text{F}} \psi] = \top \ \wedge \\ & \forall l \in [0, k) : [u^l \models_{\text{F}} \varphi] = \top \\ \bot & \text{otherwise} \end{cases}$$

where ϵ is the empty trace. The semantics of FLTL *for atomic propositions and Boolean combinations are identical to that of* LTL. ■

Similar to standard LTL, $\mathbf{F}p \equiv \top\,\mathbf{U}\,p$ and $\mathbf{G}p \equiv \neg\mathbf{F}\neg p$.

2.2.2 LTL4 Semantics

LTL$_4$ is designed for runtime verification by producing more informative verdicts than FLTL. The semantics of LTL$_4$ is defined based on values $\mathbb{B}_4 = \{\top, \top_p, \bot_p, \bot\}$ (*true*, *presumably true*, *presumably false*, and *false* respectively). The semantics of LTL$_4$ is defined based on the semantics LTL and FLTL.

DEFINITION 5 (LTL$_4$ SEMANTICS). *Let φ be an* LTL$_4$ *property and u be a finite prefix of a trace.*

$$[u \models_4 \varphi] = \begin{cases} \top & \forall v \in \Sigma^\omega : uv \models \varphi \\ \bot & \forall v \in \Sigma^\omega : uv \not\models \varphi \\ \top_p & [u \models_F \varphi] \wedge \exists v \in \Sigma^\omega : uv \not\models \varphi \\ \bot_p & [u \not\models_F \varphi] \wedge \exists v \in \Sigma^\omega : uv \models \varphi \ \blacksquare \end{cases}$$

In this definition, $\models$ denotes the satisfaction relation defined by standard LTL semantics over infinite traces. Thus, an LTL$_4$ property evaluates to $\top$ with respect to a finite trace u, if the property remains *permanently satisfied*, meaning that for all possible infinite continuations of the trace, the property will always be satisfied in LTL. Likewise, a valuation of $\bot$ means that the property will be *permanently violated*. If the property evaluates to $\top_p$, this denotes that currently the property is satisfied yet there exists a continuation that could violate it. Finally, value $\bot_p$ denotes that currently the property is violated yet there exists a continuation that could satisfy it.

2.2.3 LTL4 Monitors

In [1], the authors introduce a method of synthesizing a *monitor*, as a deterministic finite state machine (FSM), for an LTL$_4$ property.

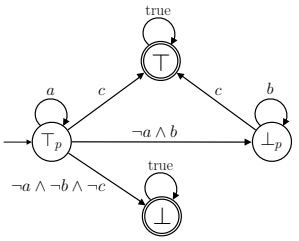

Figure 1: LTL$_4$ monitor for property $\varphi = \mathbf{G}a \ \vee \ (b\,\mathbf{U}\,c)$.

DEFINITION 6 (LTL$_4$ MONITOR). *Let φ be an* LTL$_4$ *formula over Σ. The* monitor *$\mathcal{M}_\varphi$ of φ is the unique FSM $(\Sigma, Q, q_0, \delta, \lambda)$, where Q is a set of states, q_0 is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $\lambda : Q \to \mathbb{B}_4$ is a function such that:*

$$[u \models_4 \varphi] = \lambda(\delta(q_0, u)). \ \blacksquare$$

Thus, given an LTL$_4$ property φ and a finite trace u, monitor $\mathcal{M}_\varphi$ is capable of producing a truth value in $\mathbb{B}_4$, which is equal to $[u \models_4 \varphi]$. For example, Figure 1 shows the monitor for property $\varphi = \mathbf{G}a \ \vee \ (b\,\mathbf{U}\,c)$. Observe that a monitor has two *trap* states (only an outgoing self loop), which map to truth values $\top$ and $\bot$. They are trap states since these truth values imply permanent satisfaction (respectively, violation). Otherwise, states labeled by $\top_p$ and $\bot_p$ can have outgoing transitions to other states.

2.3 Truth Values of LTL4-FOC

The objective of LTL$_4$−FO$_c$ is to verify the correctness of quantified properties at run time with respect to finite program traces. Such verification attempts to produce a sound verdict regardless of future continuations.

We incorporate six truth values to define the semantics of LTL$_4$−FO$_c$: $\mathbb{B}_6 = \{\top, \bot, \top_c, \bot_c, \top_p, \bot_p\}$; *true*, *false*, *currently true*, *currently false*, *presumably true*, *presumably false*, respectively. The values in $\mathbb{B}_6$ form a lattice ordered as follows: $\bot < \bot_c < \bot_p < \top_p < \top_c < \top$. Given a finite trace u and an LTL$_4$−FO$_c$ property φ, the informal description of evaluation of u with respect to φ is as follows:

- **True** ($\top$) denotes that any infinite extension of u satisfies φ.
- **False** ($\bot$) denotes that any infinite extension of u violates φ.
- **Currently true** ($\top_c$) denotes that currently u satisfies the quantifier constraint of φ, yet it is possible that a suffix of u violates the constraint. For instance, the valuation of Property 1 (i.e., φ_1) is $\top_c$, if in a trace u, currently 50% of files previously opened are closed. This is because (1) the inner LTL property is permanently satisfied for at least 50% of files previously opened, and (2) it is possible for a trace continuation to change this percentage to less than 50% in the future (a trace in which enough new files are opened and not closed).
- **Currently false** ($\bot_c$) denotes that currently u violates the quantifier constraint of φ, yet it is possible that a suffix of u satisfies the constraint. For instance, the valuation of Property 1 (i.e., φ_1) in a finite trace u is $\bot_c$, if the number of files that were not successfully opened is currently greater than 50%. This could

happen in the scenario where opening a file fails, possibly due to lack of permissions. Analogous to $\top_c$, the property is valuated to $\bot_c$ because (1) the inner LTL property is permanently satisfied for less than 50% of files in the program trace, and (2) it is possible for a trace continuation to change this percentage to at least 50% in the future.

Now let us consider modifying the property to support multiple open and close operations on the same file. For this purpose, we reformulate the property as follows:

$$\varphi_2 = \forall_{\geq 50\%}\, f : \mathsf{intrace}(f) \Rightarrow (\mathbf{G}\,(\mathsf{opened}(f)\,\mathbf{U}\,\mathsf{close}(f))) \quad (2)$$

- **Presumably true** ($\top_p$) extends the definition of *presumably true* in LTL$_4$ [2], which $\top_p$ denotes that u satisfies the inner LTL property and the quantifier constraint in φ, if the program terminates after execution of u. For example, Property 2 (i.e., φ_2) evaluates to $\top_p$, if at least 50% of the files in the program trace are closed. Closed files presumably satisfy the property, since they satisfy the **G** operator thus far, yet can potentially violate it if the file is opened a subsequent time without being closed. Note that this property can never evaluate to $\top_c$, since no finite trace prefix can permanently satisfy the inner LTL property. However, if the inner property can be permanently satisfied ($\top$) and presumably satisfied ($\top_p$), then the entire LTL$_4$−FO$_c$ property can potentially evaluate to $\top_c$ if the numerical condition of the quantifier is satisfied. A property can evaluate to $\top_p$ only if the conditions for $\top_c$ are not met, since $\top_c$ is higher up the partial order of $\mathbb{B}_6$.

- **Presumably false** ($\bot_p$) extends the definition of *presumably false* in LTL$_4$ [2], which denotes that u presumably violates the quantifier constraint in φ. According to the Property 2, this scenario will occur when the number of files that are either closed or opened and not yet closed is at least 50% of all files in the trace. Opened files presumably violate the inner property, since closing the file is required but has not yet occurred. This condition should not conflict with $\top_p$ or $\top_c$, since they precede $\bot_p$ in the partial order of $\mathbb{B}_6$ and thus $\bot_p$ only occurs if the conditions for $\top_p$ and $\top_c$ do not hold.

2.4 Semantics of LTL4-FOC

An LTL$_4$−FO$_c$ property essentially defines a set of traces, where each traces is a sequences of events (i.e., sets of uninterpreted predicates). We define the semantics of LTL$_4$−FO$_c$ with respect to finite traces and present a method of utilizing these semantics for runtime verification. In the context of runtime verification, the objective is to ensure that a program trace (i.e., a sequence of sets of *interpreted* predicates) is in the set of traces that the property defines, given the interpretations of the property predicates within the program trace.

To introduce the semantics of LTL$_4$−FO$_c$, we examine quantifiers further. Since the syntax of LTL$_4$−FO$_c$ allows nesting of quantifiers, a canonical form of properties is as follows:

$$\varphi = \mathbb{Q}_\varphi\ \psi \quad (3)$$

where ψ is a propositional LTL property and $\mathbb{Q}_\varphi$ is a string of quantifiers

$$\mathbb{Q}_\varphi = \mathcal{Q}_0\mathcal{Q}_1\cdots\mathcal{Q}_{n-1} \quad (4)$$

such that each $\mathcal{Q}_i = \langle Q_i, \sim_i, c_i, x_i, p_i\rangle$, $0 \leq i \leq n-1$, is a tuple encapsulating the quantifier information. That is, $Q_i \in \{\forall, \exists\}$, $\sim_i \in \{<, \leq, >, \geq, =\}$, c_i is the constraint constant, x_i is the bound variable, and p_i is the predicate within the quantifier (see Definition 3).

We presents semantics of LTL$_4$−FO$_c$ in a stepwise manner:

1. **Variable valuation.** First, we demonstrate how variable valuations are extracted from the trace and used to substitute variables in the formula.
2. **Canonical variable valuations.** Next, we demonstrate how to build a canonical structure of the variable valuations provided in Step 1. This canonical structure mirrors the canonical structure of LTL$_4$−FO$_c$ properties.
3. **Valuation of property instances.** A *property instance* is a unique substitution of variables in the property with values from their domains. This step demonstrate how to evaluate property instances.
4. **Applying quantifier numerical constraints.** This step demonstrates how to evaluate quantifiers by applying their numerical constraints on the valuation of a set of property instances from Step 3. The set of property instances is retrieved with respect to the canonical structure defined in Step 2.
5. **Inductive semantics.** Using the canonical structure in Step 2, and valuation of quantifiers in Step 4, we define semantics that begin at the outermost quantifier of an LTL$_4$−FO$_c$ property and evaluate quantifiers recursively inwards.

2.4.1 Variable Valuation

We define a vector D_φ with respect to a property φ as follows:

$$D_\varphi = \langle d_0, d_1, \cdots, d_{n-1}\rangle$$

where $n = |\mathbb{Q}_\varphi|$ and d_i, $0 \leq i \leq n-1$, is a value for variable x_i. We denote the first m components of the vector D_φ (i.e., $\langle d_0, d_1, \cdots, d_{m-1}\rangle$) by $D_\varphi|^m$. We refer to D_φ as a *value vector* and to $D_\varphi|^m$ as a *partial value vector*.

A *property instances* $\hat{\varphi}(D_\varphi|^m)$ is obtained by replacing every occurrence of the variables $x_0 \cdots x_{m-1}$ in φ with the values $d_0 \cdots d_{m-1}$, respectively. Thus, $\hat{\varphi}(D_\varphi|^m)$ is free of quantifiers of index less than m, yet remains quantified over variables $x_m \cdots x_{n-1}$. For instance, for the following property

$$\varphi = \forall_{>c_1}\, x : p_x(x) \Rightarrow (\forall_{<c_2}\, y : p_y(y) \Rightarrow \mathbf{G}\, q(x, y))$$

and value vector $D_\varphi = \langle 1, 2\rangle$ (i.e., the vector of values for variables x and y, respectively), $\hat{\varphi}(D_\varphi)$ will be

$$\hat{\varphi}(\langle 1, 2\rangle) = p_x(1) \Rightarrow (p_y(2) \Rightarrow \mathbf{G}\, q(1, 2))$$

We now define the set $\mathbb{D}_{\varphi,u}$ as the set of all value vectors with respect to a property $\varphi = \mathbb{Q}_\varphi\ \psi$ and a finite trace $u = u_0u_1\cdots u_k$:

$$\mathbb{D}_{\varphi,u} = \{D_\varphi \mid \exists j \in [0, k] : \forall i \in [0, n-1] : p_i(d_i) \in u_j\} \quad (5)$$

where $n = |\mathbb{Q}_\varphi|$.

2.4.2 Canonical Variable Valuations

An LTL4−FO$_c$ property follows a canonical structure, in which every quantifier $\mathcal{Q}_i$ has a *parent* quantifier $\mathcal{Q}_{i-1}$, except for $\mathcal{Q}_0$ which is the *root* quantifier. A quantifier $\mathcal{Q}_i$ is applied over all valuations of its variable x_i given a unique valuation of its predecessor variables $x_0, \cdots, x_{i-1}$. Hence, we define function $\mathcal{P}$ which takes as input a partial value vector $D_\varphi|^m$, and returns all partial value vectors in $\mathbb{D}_{\varphi,u}$ of length $m+1$, such that the first m elements of these vectors is the same as $D_\varphi|^m$. In this context, we refer to $D_\varphi|^m$ as a *parent* vector and all the returned vectors as *child* vectors. Similarily, a property instance can have a parent; for instance, $\hat{\varphi}(D_\varphi|^m)$ is the parent of $\hat{\varphi}(D_\varphi|^{m+1})$.

$$\mathcal{P}(\varphi, u, D_\varphi|^m) = \left\{ D'_\varphi|^{m+1} \;\middle|\; D'_\varphi \in \mathbb{D}_{\varphi,u} \wedge D'_\varphi|^m = D_\varphi|^m \right\}$$

Following the example above, assume there are two value vectors: $\langle 1,2\rangle$ and $\langle 1,3\rangle$. In this case,

$$\mathcal{P}(\varphi, u, \langle 1\rangle) = \{\langle 1,2\rangle, \langle 1,3\rangle\}$$

2.4.3 Valuation of Property Instances

As per the definition of $\mathbb{D}_{\varphi,u}$, every value vector $D_\varphi = \langle d_0 \cdots d_{n-1}\rangle$ in $\mathbb{D}_{\varphi,u}$ contains values for which the predicates $p_i(d_i)$ hold in some trace event u_j. For simplicity, we denote this as a value vector *in* a trace event u_j. These value vectors can possibly be in multiple and interleaved events in the trace. Thus, we define a trace $u^{D_\varphi} = u_0^{D_\varphi} u_1^{D_\varphi} \cdots u_l^{D_\varphi}$ as a subsequence of the trace u such that the value vector D_φ is in every event:

$$\forall\, j \in [0,l] : \forall\, i \in [0, n-1] : p_i(d_i) \in u_j^{D_\varphi}$$

For any property instance $\hat{\varphi}(D_\varphi)$, we wish to evaluate $[u^{D_\varphi} \models_6 \hat{\varphi}(D_\varphi)]$ (read as valuation of $\hat{\varphi}(D_\varphi)$ with respect to u^{D_φ} for LTL4−FO$_c$), since any other event in trace u is not of interest to $\hat{\varphi}(D_\varphi)$.

By leveraging u^{D_φ}, we define function $\mathcal{B}$ as follows:

$$\mathcal{B}(\varphi, u, D_\varphi|^m, b) =$$

$$\begin{cases} D'_\varphi|^{m+1} \in \mathcal{P}(\varphi, u, D_\varphi|^m) \mid \\ \quad [u^{D'_\varphi|^{m+1}} \models_6 \hat{\varphi}(D'_\varphi|^{m+1})] = b & \text{iff } m < |\mathbb{Q}_\varphi| - 1 \\ D'_\varphi|^{m+1} \in \mathcal{P}(\varphi, u, D_\varphi|^m) \mid \\ \quad [u^{D'_\varphi|^{m+1}} \models_4 \hat{\varphi}(D'_\varphi|^{m+1})] = b & \text{iff } m = |\mathbb{Q}_\varphi| - 1 \end{cases}$$

where b is a truth value in $\mathbb{B}_6$. Function $\mathcal{B}$ can be implemented in a straightforward manner, where it iterates over all its children value vectors $D'_\varphi|^{m+1}$ which are retrieved using $\mathcal{P}$. For every child vector, the function checks whether $\hat{\varphi}(D'_\varphi|^{m+1})$ evaluates to b with respect to the trace subsequence $u^{D'_\varphi|^{m+1}}$.

To clarify $\mathcal{B}$, let us refer to our example earlier. Let a program trace u be as follows:

$$u = \{p_x(1), p_y(2), \cdots\}, \{p_x(1), p_y(3), \cdots\}, \{p_x(1), p_y(2), \cdots\}$$

With respect to this trace, $\mathcal{P}(\varphi, u, \langle 1\rangle) = \{\langle 1,2\rangle, \langle 1,3\rangle\}$. As per the definition of u^{D_φ}, $u^{\langle 1,2\rangle} = u_0 u_2$, and $u^{\langle 1,3\rangle} = u_1$. Thus, $\mathcal{B}(\varphi, u, \langle 1\rangle, b)$ checks the following:

$$[u^{\langle 1,2\rangle} \models_4 p_x(1) \Rightarrow (p_y(2) \Rightarrow \mathbf{G}\, q(1,2))] = b$$
$$[u^{\langle 1,3\rangle} \models_4 p_x(1) \Rightarrow (p_y(3) \Rightarrow \mathbf{G}\, q(1,3))] = b$$

The definition of u^{D_φ} implies that $p_i(d_i) \in u_j^{D_\varphi}$ for all j. Thus, we can simplify the property by omitting the p predicates since they hold by definition:

$$[u^{\langle 1,2\rangle} \models_4 \mathbf{G}\, q(1,2)] = b$$
$$[u^{\langle 1,3\rangle} \models_4 \mathbf{G}\, q(1,3)] = b$$

For instance, if only $[u^{\langle 1,2\rangle} \models_4 \mathbf{G}\, q(1,2)] = b$ holds, then

$$\mathcal{B}(\varphi, u, \langle 1\rangle, b) = \{\langle 1,2\rangle\}$$

As can be seen in the example, the property instances that are evaluated are LTL4 properties. This is because the input to $\mathcal{B}$ is $D_\varphi|^1 = D_\varphi|^{|\mathbb{Q}_\varphi|-1}$, which represents the inner most quantifier.

2.4.4 Applying Quantifier Numerical Constraints

Finally, numerical constraints should be incorporated in the semantics. We define function $\mathcal{S}$ as follows:

$$\mathcal{S}(\varphi, u, D_\varphi|^m, B) = \begin{cases} \left| \bigcup_{b\in B} \mathcal{B}(\varphi, u, D_\varphi|^m, b) \right| \sim_i \\ \quad c_i \times |\{\mathcal{P}(\varphi, u, D_\varphi|^m)\}| & \text{iff } Q_m = \forall \\ \left| \bigcup_{b\in B} \mathcal{B}(\varphi, u, D_\varphi|^m, b) \right| \sim_i c_i & \text{iff } Q_m = \exists \end{cases} \tag{6}$$

where $B \subseteq \mathbb{B}_6$ is a set of truth values. This function returns whether a quantifier constraint is satisfied or not based on any of the truth values $b \in B$. Observe that, for universal quantifiers, the constraint value denotes the percentage of property instances that evaluate to b. For existential quantifiers, the constraint value denotes the number of property instances that evaluate to b. For instance, consider Property 7 which is read as: for all users, there exists at most 3 requests of type login that end with an unauthorized status. For such a property, if 4 or more unauthorized login attempts are detected for the same user, the property is permanently violated.

2.4.5 Inductive Semantics

Using the previously defined set of of functions, we now formalize LTL4−FO$_c$ semantics.

DEFINITION 7 (LTL4−FO$_c$ SEMANTICS). *LTL4−FO$_c$ semantics for properties with quantifiers are defined as follows:*

$$[u \models_6 \varphi] = \begin{cases} \top & \textit{iff } \mathcal{S}(\varphi, u, \langle\rangle, \{\top\}) = 1 \wedge \\ & \forall v \in \Sigma^\omega : [uv \models_6 \varphi] = \top \\ \bot & \textit{iff } \mathcal{S}(\varphi, u, \langle\rangle, \mathbb{B}_6 - \{\bot\}) = 0 \wedge \\ & \forall v \in \Sigma^\omega : [uv \models_6 \varphi] = \bot \\ \top_c & \textit{iff } \mathcal{S}(\varphi, u, \langle\rangle, \{\top, \top_c\}) = 1 \wedge \\ & \exists v \in \Sigma^\omega : [uv \models_6 \varphi] \neq \top_c \\ \bot_c & \textit{iff } \mathcal{S}(\varphi, u, \langle\rangle, \mathbb{B}_6 - \{\bot, \bot_c\}) = 0 \wedge \\ & \exists v \in \Sigma^\omega : [uv \models_6 \varphi] \neq \bot_c \\ \top_p & \textit{iff } \mathcal{S}(\varphi, u, \langle\rangle, \{\top, \top_c, \top_p\}) = 1 \wedge \\ & \mathcal{S}(\varphi, u, \langle\rangle, \{\top, \top_c\}) = 0 \\ \bot_p & \textit{iff } \mathcal{S}(\varphi, u, \langle\rangle, \{\top, \top_c, \top_p\}) = 0 \wedge \\ & \mathcal{S}(\varphi, u, \langle\rangle, \mathbb{B}_6 - \{\bot, \bot_c\}) = 0 \;\blacksquare \end{cases}$$

Note that these semantics are applied recursively until there is only one quantifier left in the formula, at which point $\mathcal{B}$ checks the valuation based on LTL4 semantics ($[u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] = b$). When checking the valuation of these LTL4

properties, $\mathcal{B}$ will always return an empty set in case the input b is $\top_c$ or $\bot_c$, since these truth values are inapplicable to LTL$_4$ properties. As mentioned earlier, truth values in $\mathbb{B}_6$ form a lattice. Standard lattice operators $\sqcap$ and $\sqcup$ are defined as expected based on the lattice's partial order. Permanent satisfaction ($\top$) or violation ($\bot$) is applicable to $\exists$ quantifiers regardless of the comparison operator, as well as a special case of $\forall$ quantifiers:

- $\forall$ **quantifier.** As mentioned earlier, if the $\forall$ quantifier is not subscripted, it is assumed to denote $\forall_{=1}$. In this case, a single violation in its child property instances causes a permanent violation of the quantified property.
- $\exists$ **quantifier.** Permanent violation is possible for any numerical constraint attached to an $\exists$ quantifier, since it is a condition on the *number* of satisfied property instances.

Property 7 illustrates an example of an $\exists$ quantifier that can be permanently violated. Also, since the $\forall$ quantifier in Property 7 defaults to $\forall_{=1}$, it will be violated if a single user makes more than three unauthorized login attempts. In such a case, the entire property evaluates to $\bot$. Table 1 illustrates how permanent satisfaction or violation apply to the different numerical constraints of $\exists$ quantifiers.

$$\forall x : \mathsf{user}(x) \Rightarrow (\exists_{\leq 3}\, r : \mathsf{rid}(r) \Rightarrow (\mathsf{login} \wedge \mathsf{unauthorized})) \quad (7)$$

Table 1: Rules of permanent satisfaction or violation of $\exists$ constraints

Operator	Verdict
$> c$	Permanent satisfaction if $> c$
$\geq c$	Permanent satisfaction if $\geq c$
$= c$	Permanent violation if $> c$
$< c$	Permanent violation if $\geq c$
$\leq c$	Permanent violation if $> c$

To clarify the semantics, consider Property 7 and the following program trace:

$$\{\mathsf{rid}(12), \mathsf{user}(Adam), \mathsf{login}, \mathsf{unauthorized}\}$$
$$\{\mathsf{rid}(13), \mathsf{user}(Adam), \mathsf{login}, \mathsf{unauthorized}\}$$
$$\{\mathsf{rid}(14), \mathsf{user}(Jack), \mathsf{login}, \mathsf{authorized}\}$$
$$\{\mathsf{rid}(15), \mathsf{user}(Adam), \mathsf{login}, \mathsf{authorized}\}$$
$$\{\mathsf{rid}(16), \mathsf{user}(Adam), \mathsf{login}, \mathsf{authorized}\}$$

where each line represents an event: a set of interpreted predicates. Each event contains a request identifier (rid), a username, a request type (login), and response status (authorized or unauthorized). As seen in the trace, there are 5 distinct value vectors: $\langle Adam, 12\rangle$, $\langle Adam, 13\rangle$, $\langle Jack, 14\rangle$, $\langle Adam, 15\rangle$, and $\langle Adam, 16\rangle$. Now, let us apply the inductive semantics on the property.

Step 1. We begin by checking the truth value of $[u \models_6 \varphi]$, which requires determining which condition in Definition 7 applies. This requires the evaluation of function $\mathcal{S}$ for the different truth values shown. Since we are verifying φ, we begin with the outermost quantifier, which is a $\forall$ quantifier. Thus, $\mathcal{S}$ will require calculating the cardinality of the set $\mathcal{P}(\varphi, u, D_\varphi|^0)$, which in case of the trace should be $|\{Adam, Jack\}| = 2$. Now, in order to evaluate $\mathcal{S}$, one has to evaluate $\mathcal{B}$ to determine whether each property instance evaluates to a certain truth value or not. The two property instances thus far are:

$$\hat{\varphi}(D_\varphi|^1) = \hat{\varphi}(Adam) = \exists_{\leq 3}\, r : \mathsf{rid}(r) \Rightarrow (\mathsf{login} \wedge \mathsf{unauthorized})$$
$$\hat{\varphi}(D'_\varphi|^1) = \hat{\varphi}(Jack) = \exists_{\leq 3}\, r : \mathsf{rid}(r) \Rightarrow (\mathsf{login} \wedge \mathsf{unauthorized})$$

And the trace subsequences for these property instances respectively are:

$$u^{D_\varphi|^1} = \{\mathsf{rid}(12), \cdots\}\{\mathsf{rid}(13), \cdots\}\{\mathsf{rid}(15), \cdots\}\{\mathsf{rid}(16), \cdots\}$$
$$u^{D'_\varphi|^1} = \{\mathsf{rid}(14), \cdots\}$$

Note that $\mathsf{user}(Adam) \Rightarrow \cdots$ is ommitted from $\hat{\varphi}(D_\varphi|^1)$ since $\mathsf{user}(Adam)$ holds according to the trace subsequence. The same applies to $\mathsf{user}(Jack)$. Evaluating these property instances with respect to the trace subsequences requires referring to Definition 7 again, which marks the second level of recursion.

Step 2. Let us consider the property instance $\hat{\varphi}(D_\varphi|^1)$, which begins with an $\exists$ quantifier and has LTL$_4$ properties as child instances (refer to $\mathcal{P}$). These properties are in the form of $\mathsf{login} \wedge \mathsf{unauthorized}$, where there is one instance for each distinct request identifier. We can deduce that the property holds for all 4 requests: 12, 13, 15, and 16, thus evaluating to $\top$. Therefore, the following holds:

$$\mathcal{B}(\hat{\varphi}(D_\varphi|^1), u^{D_\varphi|^1}, D_\varphi|^1, \top) = 4$$

This value, when used in $\mathcal{S}(\hat{\varphi}(D_\varphi|^1), u^{D_\varphi|^1}, D_\varphi|^1, \{\top\})$ will violate the numerical condition: $4 \not\leq 3$, resulting in $\mathcal{S}$ returning 0 (false). Based on the conditions in Definition 7 and the rules of permanent violation, this property instance becomes permanently violated and thus returns $\bot$.

The other property instance $\hat{\varphi}(D'_\varphi|^1)$ will however evaluate to $\top$ since its child property instance

$$\hat{\varphi}(D'_\varphi|^2) = \hat{\varphi}(\langle Jack, 14\rangle) = \mathsf{login} \wedge \mathsf{unauthorized}$$

is violated, and thus the number of satisfied instances is still less than 3.

Step 3. In this step we use the valuations determined in Step 2 to produce verdicts for the property instances in Step 1. Based on $\mathcal{S}$, the $\forall$ quantifier's numerical condition is violated, since not *all* instances are satisfied. The final verdict should thus be $[u \models_6 \varphi] = \bot$, which denotes a permanent violation of the property.

3. DIVIDE-AND-CONQUER-BASED MONITORING OF LTL4-FOC

In this section, we describe our technique inspired by divide-and-conquer for evaluating LTL$_4$−FO$_c$ properties at run time. This approach forms the basis of our parallel verification algorithm in Section 4.

Unlike runtime verification of propositional LTL$_4$ properties, where the structure of a monitor is determined solely based on the property itself, a monitor for an LTL$_4$−FO$_c$ needs to evolve at run time, since the valuation of quantified variables change over time. More specifically, the monitor $\mathcal{M}_\varphi$ for an LTL$_4$−FO$_c$ property $\varphi = \mathbb{Q}_\varphi \psi$ relies on instantiating a *submonitor* for each property instance $\hat{\varphi}$ obtained at run time. We incorporate two type of submonitors: (1) LTL$_4$ *submonitors* evaluate the inner LTL property ψ, and (2) *quantifier submonitors* deal with quantifiers in $\mathbb{Q}_\varphi$, described in Subsections 3.1 and 3.2. In Subsection 3.3, we

explain the conditions under which a submonitor is instantiated at run time. Finally, in Subsection 3.4, we elaborate on how submonitors evaluate an LTL$_4$−FO$_c$ property.

3.1 LTL4 Submonitors

Let $\varphi = \mathbb{Q}_\varphi \psi$ be an LTL$_4$−FO$_c$ property. If $|\mathbb{Q}_\varphi| = 0$ (respectively, one wants to evaluate $\hat{\varphi}(D_\varphi|^i)$, where $i = |\mathbb{Q}_\varphi|$), then φ (respectively, $\hat{\varphi}(D_\varphi|^i)$) is free of quantifiers and, thus, the monitor (respectively, submonitor) of such a property is a standard LTL$_4$ monitor (see Definition 6). We denote LTL$_4$ submonitors as $\mathcal{M}^*_{D_\varphi}$, where D_φ is the value vector with which the monitor is initialized.

3.2 Quantifier Submonitors

Given a finite trace u and an LTL$_4$−FO$_c$ property $\varphi = \mathbb{Q}_\varphi \psi$, a *quantifier submonitor* ($\mathcal{M}^{\mathcal{Q}}$) is a monitor responsible for determining the valuation of a property instance $\hat{\varphi}(D_\varphi|^i)$ with respect to a trace subsequence $u^{D_\varphi|^i}$, if $i < |\mathbb{Q}_\varphi|$. Obviously, such a valuation is in $\mathbb{B}_6$. Let $\mathbb{V}$ be a six-dimensional vector space, where each dimension represents a truth value in $\mathbb{B}_6$.

Definition 8 (Quantifier Submonitor). *Let $\varphi = \mathbb{Q}_\varphi \psi$ be an* LTL$_4$−FO$_c$ *property and $\hat{\varphi}(D_\varphi|^i)$ be a property instance, with $i \in [0, |\mathbb{Q}_\varphi| - 1]$. The* quantifier submonitor *for $\hat{\varphi}(D_\varphi|^i)$ is the tuple $\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^i} = \langle \mathcal{Q}_i, \mathbb{M}_{D_\varphi|^i}, v, b \rangle$, where*

- *$\mathcal{Q}_i$ encapsulates the quantifier information (see Equation 4)*
- *$v \in \mathbb{V}$ represents the current number of child property instances that evaluate to each truth value in $\mathbb{B}_6$ with respect to their trace subsequences,*
- *$b \in \mathbb{B}_6$ is the current value of $[u^{D_\varphi|^i} \models_6 \hat{\varphi}(D_\varphi|^i)]$,*
- *$\mathbb{M}_{D_\varphi|^i}$ is the set of child submonitors (submonitors of child property instances) defined as follows:*

$$\mathbb{M}_{D_\varphi|^i} = \begin{cases} \{\mathcal{M}^*_{D'_\varphi} \mid D'_\varphi|^i = D_\varphi|^i\} & \text{if } i = |\mathbb{Q}_\varphi| - 1 \\ \{\mathcal{M}^{\mathcal{Q}}_{D'_\varphi|^{i+1}} \mid D'_\varphi|^i = D_\varphi|^i\} & \text{if } i < |\mathbb{Q}_\varphi| - 1 \end{cases}$$

Thus, if $i = |\mathbb{Q}_\varphi| - 1$, all child submonitors are LTL$_4$ *submonitors. Otherwise, they are quantifier submonitors of the respective child property instances.* ■

Based on the definition, every quantifier submonitor references a set of child monitors. We use the following notation to denote a hierarchy of a submonitor and its children:

$$\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^i}\{\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^{i+1}}, \mathcal{M}^{\mathcal{Q}}_{D'_\varphi|^{i+1}}, \mathcal{M}^{\mathcal{Q}}_{D''_\varphi|^{i+1}}, \cdots\}$$

such that $D_\varphi|^i = D'_\varphi|^i = D''_\varphi|^i \cdots$ and $i < |\mathbb{Q}_\varphi| - 1$ which is why the child monitors are quantifier submonitors.

3.3 Instantiating Submonitors

Let an LTL$_4$−FO$_c$ monitor $\mathcal{M}_\varphi$ for property φ evaluate the property with respect to a finite trace $u = u_0 u_1 \cdots$. Let $D_\varphi = \langle d_0, d_1, \cdots \rangle$ be a value vector and u_k the first trace event such that $\forall d_i : p_i(d_i) \in u_k$, where p_i is the predicate within each quantifier (i.e. $\forall x_i : p_i(x_i) \Rightarrow \cdots$). In this case, the LTL$_4$−FO$_c$ monitor instantiates submonitors for every property instance resulting from that value vector. A value vector of length $|\mathbb{Q}_\varphi|$ results in $|\mathbb{Q}_\varphi| + 1$ property instances:

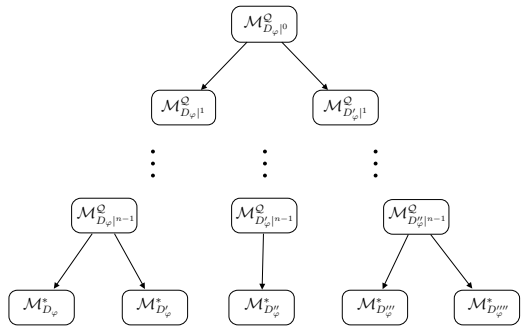

Figure 2: Tree structure of an LTL$_4$−FO$_c$ monitor.

one for each quantifier in addition to an LTL$_4$ inner property. The hierarchy of the instantiated submonitors is as follows:

$$\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^0}\Big\{\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^1}\Big\{\cdots\{\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^{|\mathbb{Q}_\varphi-1|}}\{\mathcal{M}^*_{D_\varphi}\}\}\Big\}\Big\}$$

If another value vector D'_φ is subsequently encountered for the first time, the hierarchy of submonitors becomes as follows:

$$\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^0}\Big\{\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^1}\big\{\cdots\{\mathcal{M}^*_{D_\varphi}\}\big\}, \mathcal{M}^{\mathcal{Q}}_{D'_\varphi|^1}\big\{\cdots\{\mathcal{M}^*_{D'_\varphi}\}\big\}\Big\}$$

Since the hierarchy is formulated as a recursive set, no duplicate submonitors are allowed. Two submonitors are duplicates, if they represent identical value vectors. If $D_\varphi|^1 = D'_\varphi|^1$, the respective monitors are merged. Such merging is explained in detail in Section 4.

3.4 Evaluating LTL4-FOC Properties

Once the LTL$_4$−FO$_c$ monitor instantiates its submonitors, every submonitor is responsible for updating its truth value. The truth value of LTL$_4$ submonitors ($\mathcal{M}^*$) maps to the current state of the submonitor's automaton as described in Definition 6. Quantifier submonitors update their truth value based on the truth values of all child submonitors. The number of child submonitors whose truth value is $\top$ is stored in $v_\top$ (i.e., the $\top$ dimension of vector v) and so on for all truth values in $\mathbb{B}_6$. Then, LTL$_4$−FO$_c$ semantics are applied, beginning with function $\mathcal{S}$ (see Equation 6), which in turn relies on the cardinality of function $\mathcal{B}(\varphi, u, D_\varphi|^i, b)$ where b is a truth value. This cardinality is readily provided by the vector v, such that for instance $\mathcal{B}(\varphi, u, D_\varphi|^i, \top) = v_\top$ and so on.

Since each submonitor depends on its child submonitors, updating truth values proceeds outwards, starting at LTL$_4$ submonitors, then recursively parent submonitors update their truth values until the submonitor $\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^0}$, which is the root submonitor. The truth value of the root submonitor is the truth value of property φ with respect to trace u. This is visualized as the tree shown in Figure 2.

4. PARALLEL ALGORITHM DESIGN

The main challenge in designing a runtime monitor is to ensure that its behavior does not intervene with functional and extra-functional (e.g., timing constraints) behavior of the program under scrutiny. This section presents a parallel algorithm for verification of LTL$_4$−FO$_c$ properties. Our idea is that such a parallel algorithm enables us to offload the monitoring tasks into a different computing unit (e.g., the GPU). The algorithm utilizes the popular *MapReduce* technique to spawn and merge submonitors to determine the

final verdict. This section is organized as follows: Subsection 4.1 describes how valuations are extracted from a trace in run time, and Subsection 4.2 describes the steps of the algorithm in detail.

4.1 Valuation Extraction

Valuation extraction refers to obtaining a valuation of quantified variables from the trace. As described in LTL$_4$−FO$_c$ semantics, the predicate $p_i(x_i)$ identifies the subset of the domain of x_i over which the quantifier is applied: namely the subset that exists in the trace. From a theoretical perspective, we check whether the predicate is a member of some trace event, which is a set of predicates. From an implementation perspective, the trace event is a key-value structure, where the key is for instance a string identifying the quantified variable, and the value is the concrete value of the quantified variable in that trace event. Consider the following property:

$$\varphi = \forall_{\geq 0.95}\, s : \mathsf{socket}(s) \Rightarrow (\mathbf{G}\ \mathsf{receive}\,(s) \Rightarrow \mathbf{F}\ \mathsf{respond}\,(s)) \tag{8}$$

Predicate p in this case is $\mathsf{socket}(s)$, and a trace event should contain a key socket and a value $\in [0, 65535]$ representing the socket file descriptor in the system. Thus, the valuation extraction function $\varepsilon(u_i, K) = D_\varphi$ returns a map where keys are in K, and the value of each key is the value of the quantified variable corresponding to this key. These keys are defined by the user.

4.2 Algorithm Steps

Algorithm 1 presents the pseudocode of the parallel monitoring algorithm. Given an LTL$_4$−FO$_c$ property $\varphi = \mathbb{Q}_\varphi\, \psi$, the input to the algorithm is the LTL$_4$ monitor $\mathcal{M}^*$ of LTL$_4$ property ψ, a finite trace u, the set of quantifiers $\mathbb{Q}_\varphi$, and the vector of keys K used to extract valuations. The entry point to the algorithm is at Line 5 which is invoked when the monitor receives a trace to process. The algorithm returns a truth value of the property at Line 8. Subsections 4.2.1 – 4.2.4 describe the functional calls between Lines 5 – 8. The MapReduce operations are visible in functions *SortTrace* and *ApplyQuantifiers*, which perform a *map* ($\rightrightarrows$) in Lines 10 and 51 respectively. *ApplyQuantifiers* also performs a reduction ($\rightarrowtail$) in Line 52.

4.2.1 Trace Sorting

As shown in Algorithm 1, the first step in the algorithm is to sort the input trace u (Line 5). The function *SortTrace* performs this functionality as follows:

1. The function performs a parallel map of every trace event to the value vector that it holds using ε (Line 10).

2. The mapped trace is sorted in parallel using the quantifier variable keys (Line 11). For instance, according to Property 8, the key used for sorting will be socket, effectively sorting the trace by socket identifier.

3. The sorted trace is then compacted based on valuations, and the function returns a map μ where keys are value vectors and values are the ranges of where these value vectors exist in trace u (Line 12). A range contains the start and end index. This essentially defines the subsequences u^{D_φ} for each property instance $\hat{\varphi}(D_\varphi)$ (refer to Subsection 2.4).

4.2.2 Monitor Spawning

Monitor spawning is the second step of the algorithm (Line 6). The function *SpawnMonitors* receives a map μ and searches the cached collection of previously encountered value vectors $\mathbb{D}$ for duplicates. If a value vector in μ is new, it creates submonitors and inserts them in the tree of submonitors T (Line 19). The function *AddToTree* attempts to generate $|\mathbb{Q}_\varphi| - 1$ quantifier submonitors $\mathcal{M}^{\mathcal{Q}}$ (Line 26) ensuring there are no duplicate monitors in the tree (Line 27). After all quantifier submonitors are created, *SpawnMonitors* creates an LTL$_4$ submonitor $\mathcal{M}^*$ and adds it as a child to the leaf quantifier submonitor in the tree representing the value vector (Line 20). This resembles the structure in Figure 2. Creation of submonitors is performed in parallel for all value vectors in trace u.

4.2.3 Distributing the Trace

The next step in the algorithm is to distribute the sorted trace to all LTL$_4$ submonitors (Line 7). The function *Distribute* instructs every LTL$_4$ submonitor to process its respective trace by passing the full trace and the range of its respective subsequence, which is provided by the map μ (Line 42). The LTL$_4$ monitor updates its state according to the trace subsequence and stores its truth value b.

4.2.4 Applying Quantifiers

Applying quantifiers is a recursive process, beginning with the leaf quantifier submonitors and proceeding upwards towards the root of the tree (Line 8). Function *ApplyQuantifiers* operates in the following steps:

1. The function retrieves all quantifier submonitors at the i^{th} level in the tree T (Line 50).

2. In parallel, for each quantifier submonitor, all child submonitor truth values are reduced into a single truth value of that quantifier submonitor (Lines 51-53). This step essentially *reduces* all child truth vectors into a single vector and then applies LTL$_4$−FO$_c$ semantics to determine the truth value of the current submonitor.

3. The function proceeds recursively calling itself on submonitors that are one level higher. It terminates when the root of the tree is reached, where the truth value is the final verdict of the property with respect to the trace.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented Algorithm 1 for two computing technologies: Multi-core CPUs and GPUs. We applied three optimizations in our GPU-based implementation: (1) we use *CUDA Thrust API* to implement parallel sort, (2) we using *Zero-Copy Memory* which parallelizes data transfer with kernel operation without caching, and (3) we enforced alignment, which enables coalesced read of trace events into monitor instances. In order to inercept systems calls, we have integrated our algorithm with the Linux `strace` application, which logs all system calls made by a process, including the parameters passed, the return value, the time the call was made, etc. Notice that using `strace` has the benefit of eliminating static analysis for instrumentation. The work in [5, 14, 15] also use `strace` to debug the behavior of applications.

Algorithm 1 LTL$_4$−FO$_c$ monitoring algorithm

```
1: INPUT: An LTL4 monitor M* of LTL property ψ, a finite trace u, a
   set of quantifiers Q_φ, and a vector of keys K to extract valuations
   of quantified variables.
2: declare T = {M^Q_{D|0}}                          ▷ Tree of quantifier submonitors
3: declare 𝔻 = {}                                   ▷ Value vector set
4: declare 𝕄* = {}                                  ▷ LTL4 submonitor set
5: μ ← SortTrace(u)                                 ▷ The entry point
6: SpawnMonitors(μ)
7: Distribute(u,μ)
8: return ApplyQuantifiers(|Q_φ − 1|)

9: function SortTrace(u)                            ▷ Trace sorting and compaction
10:     u_i ⇉ u'_i := ε(u_i, K)                     ▷ ‖ map to value vectors
11:     ParallelSort(u',K)
12:     μ⟨D, r⟩ ← ParallelCompact(u')
13:     return μ
14: end function

15: function SpawnMonitors(μ)                       ▷ Monitor spawning
16:     for D ∈ μ do in parallel
17:         if D ∉ 𝔻 then
18:             Add(𝔻,D)
19:             t ← AddToTree(D)
20:             t.addMonitor(CreateMonitor(D))
21:         end if
22:     end for
23: end function

24: function AddToTree(D)
25:     t = T.root
26:     for i ∈ [1, |Q_φ| − 1] do
27:         if M^Q_{D|^i} ∉ t.children then
28:             t.addchild(M^Q_{D|^i})
29:         end if
30:         t ← t.children[M^Q_{D|^i}]
31:     end for
32:     return t
33: end function

34: function CreateMonitor(D)                       ▷ Monitor creation
35:     M*_D ← LaunchMonitorThread(D)
36:     M*_D.D ← D
37:     add(𝕄*,M*_D)
38:     return M*_D
39: end function

40: function Distribute(u,μ)                        ▷ Distribute trace to monitors
41:     for M*_D ∈ 𝕄* do in parallel
42:         ProcessBuffer(M*_D,u,μ[M*_D.D])
43:     end for
44: end function

45: function ProcessBuffer(M*_D,u,r)                ▷ Process trace subsequence
46:     filter include u ⇉ u' := u[r.start, r.end]  ▷ ‖ filter
47:     M*_D.b ←UpdateMonitor(M*_D, u')
48: end function

49: function ApplyQuantifiers(i)                    ▷ Apply quantifiers
50:     for t ∈ T.nodesAtDepth(i) do in parallel
51:         t.children ⇉ {s := [v, v', ···]}        ▷ ‖ map to truth vectors
52:         s ⇀ t.v                                 ▷ ‖ reduction to truth vector
53:         t.b ← Valuation(t)                      ▷ LTL4−FOc semantics
54:     end for
55:     if i = 0 then
56:         return t.b
57:     end if
58:     return ApplyQuantifiers(i − 1)
59: end function
```

Subsection 5.1 presents the case studies implemented to study the effectiveness of the GPU implementation in online and offline monitoring. Subsection 5.2 discusses the experimental setup, while Subsection 5.3 analyzes the results.

5.1 Case studies

We have conducted the following three case studies:

1. **Ensuring every request on a socket is responded to.** This case study monitors the responsiveness of a web server. Web servers under heavy load may experience some timeouts, which results in requests that are not responded to. This is a factor contributing to the uptime of the server, along with other factors like power failure, or system failure. Thus, we monitor that at least 95% of requests are indeed responded:

$$\forall_{\geq 0.95}\, s:\ \mathsf{socket}(s)\mathbf{G}\ \mathsf{receive}\,(s) \Rightarrow \mathbf{F}\ \mathsf{respond}\,(s)$$

   We utilize the Apache Benchmarking tool to generate different load levels on the Apache Web Server.

2. **Ensuring fairness in utilization of personal cloud storage services.** This case study is based on the work in [8], which discusses how profiling DropBox traffic can identify the bottlenecks and improve the performance. Among the issues detected during this analysis, is a user repeatedly uploading chunks of maximum size to DropBox servers. This is possibly attributed to failure in the client or misuse of the service, or even some legitimate use that is not yet explained. Thus, it is beneficial for a runtime verification system to ensure that the average chunk size of all clients falls below a predefined maximum threshold, effectively ensuring fairness of service use. The corresponding LTL$_4$−FO$_c$ property is as follows:

$$\forall u:\ \mathsf{user}(u) \Rightarrow \mathbf{F}\,(\mathsf{avg\_chunksize}\,(u) \leq \mathsf{maximum})$$

3. **Ensuring proxy cache is functioning correctly.** This experiment is based on a study that shows the effectiveness of utilizing proxy cache in decreasing YouTube videos requests in a large university campus [16]. Thus, we monitor that no video is requested externally while existing in the cache:

$$\forall v : \mathsf{vid}(v) \Rightarrow \exists_{=0}\, r : \mathsf{req}(r) \Rightarrow (\mathsf{cached}(v)\ \wedge\ \mathsf{external}(r))$$

5.2 Experimental Setup

Experiment Hardware and Software. The machine we use to run experiments comprises of a 12-core Intel Xeon E5-1650 CPU, an Nvidia Tesla K20c GPU, and 32GB of RAM, running Ubuntu 12.04.

Experimental Factors. The experiments involve comparing the following factors:

- *Implementation.* We compare three implementations of the LTL$_4$−FO$_c$ monitoring algorithm:
  - *Single Core CPU.* A CPU implementation running on a single core. The justification for using a single core is to allow the remaining cores to perform the main functionality of the system without causing contention from the monitoring process.
  - *Parallel CPU.* A CPU implementation running on all 12 cores of the system. The implementation uses OpenMP.
  - *GPU.* A parallel GPU-based implementation.
- *Trace size.* We also experiment with different trace sizes to study the scalability of the monitoring solution, increasing exponentially from 16,384 to 8,388,608.

Experimental Metrics. Each experiment results in values for the following metrics:

- *Total execution time.* The total execution time of the monitor.
- *Monitor CPU utilization.* The CPU utilization of the monitor process.

In addition, we measure the following metrics for Case Study 1, since it utilizes an online monitor:

- *Monitored program CPU utilization.* The CPU utilization of the monitored program. This is to demonstrate the impact of monitoring on overall CPU utilization.
- `strace` *parsing CPU utilization.* The CPU utilization of the strace parsing module. This module translates strace strings a numerical table.

We perform 20 replicates of each experiment and present error bars of a 95% confidence interval.

5.3 Results

Table 2 shows the impact of online monitoring on CPU overhead. The table demonstrates the average CPU time consumed by the Apache Web Server, the average CPU time consumed by strace parsing, and the CPU time consumed by monitoring for each implementation. The results in the table are only for a trace size of 262144, yet different trace sizes show the same trends. As seen in the table, the monitoring overhead of GPU is almost negligable compared to the CPU time of Apache. Single core CPU also imposes low monitoring overhead, while parallel CPU imposes large monitoring overhead distributed over all cores.

	Apache	strace	Monitor
Single Code CPU			15
Parallel CPU	10000	4099	2142
GPU			7

Table 2: Processing time (msec) of Apache, strace, and monitoring for all 3 implementations. Trace size is 262144.

The results of Case Study 1 are shown in Figure 3. As seen in the figure, the GPU implementation scales efficiently with increasing trace size, resulting in the lowest monitoring time of all three implementations. The GPU versus single core CPU speedup ranges from 0.8 to 1.6, increasing with the increasing trace size. When compared to parallel CPU (CPU ||), the speedup ranges from 0.78 to 1.59. This indicates that parallel CPU outperforms GPU for smaller traces (32768), yet does not scale as well as GPU. CPU utilization results in Figure 3 show a common trend with the increase of trace size. When the trace size is small, parallel implementations incur high CPU utilization as opposed to a single core implementation, which could be attributed to the overhead of parallelization relative to the small trace size. On the other hand, GPU shows a stable utilization percentage, with a 78% average utilization. The single core CPU implementation shows a similar trend, yet slightly elevated average utilization (average 86%). The parallel CPU implementation imposes a higher CPU utilization (average 1.15%), since more cores are being used to process the trace. This result indicates that shipping the monitoring workload to GPU consistently provides more time for CPU to execute other processes including the monitored process. The results

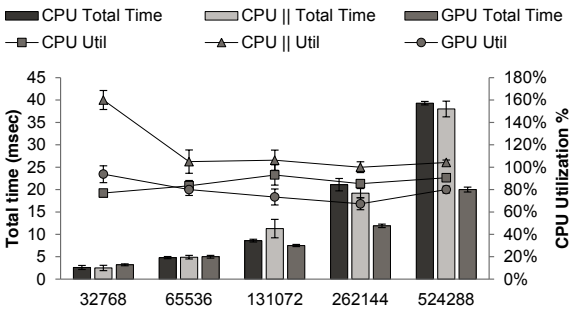

Figure 3: Results of Case Study 1.

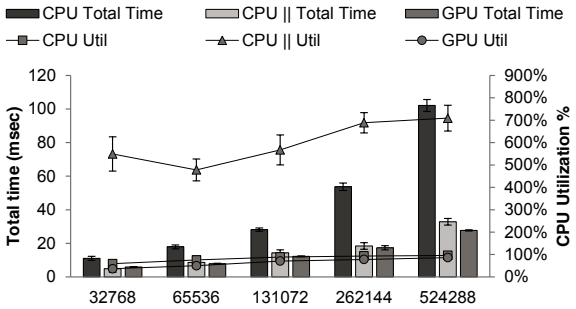

Figure 4: Results of Case Study 2.

of Case Study 2 and Case Study 3 in Figures 4 and 5 respectively show similar trends. For Case Study 2, the speedup of the GPU implementation over single core CPU ranges from 1.8 to 3.6, and 0.83 to 1.18 over parallel CPU. The average CPU utilization of GPU, single core CPU, and parallel CPU is 64%, 82%, and 598% respectively. For Case Study 3, speedup is more significant, with 6.3 average speedup of GPU over single core CPU, and 1.75 over parallel CPU. The average CPU utilization of GPU, single core CPU, and parallel CPU is 73%, 95%, and 680% respectively.

> *Although the parallel CPU implementation provides reasonable speedup, and the single-core CPU implementation imposes low CPU utilization overhead, the GPU implementation manages to achieve both simultaneously.*

6. RELATED WORK

Runtime verification of parametric properties has been studied by Rosu et al [9,10,12]. In this line of work, it is possible to build a runtime monitor parameterized by objects in a Java program. The work by Chen and Rosu [6] presents a method of monitoring parametric properties in which a trace is divided into slices, such that each monitor operates on its slice. This resembles our method of identifying trace subsequences and how they are processed by submonitors. However, parametric monitoring does not provide a formalization of applying existential and numerically constrained quantifiers over objects.

Bauer et al. [3] present a formalization of a variant of first order logic combined with LTL. This work is related to our work in that it instantiates monitors at run time according to valuations, and defines quantification over a finite subset of the quantified domain, normally with that subset being defined by the trace. Our work extends this notion with numerical constraints over quantifiers, as well as a parallel algorithm for monitoring such properties.

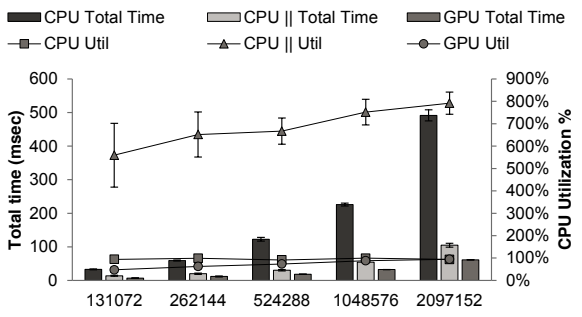

Figure 5: Results of Case Study 3.

The work by Leucker et al. presents a generic approach for monitoring modulo theories [7]. This work provides a more expressive specification language. Our work enforces a canonical syntax which is not required in [7], resulting in more expressiveness. However, the monitoring solution provided requires SMT solving at run time. This may induce substantial overhead as opposed to the lightweight parallel algorithm presented in this paper, especially since it is designed to allow offloading the workload on GPU. SMT solving also runs the risk of undecidability, which is not clear whether it is accounted for or not. LTL_4-FO_c is based on six-valued semantics, extending LTL_4 by two truth values: $\top_c$ and $\bot_c$. These truth values are added to support quantifiers and their numerical constraints. This six-valued semantics provides a more accurate assessment of the satisfaction of the property based on finite traces as opposed to the three-valued semantics in [7]. Finally, although LTL_4-FO_c does not support the expressiveness of full first-order logic, numerical constraints add a flavor of second-order logic increasing its expressiveness in the domain of properties where some percentage or count of satisfactied instances needs to be enforced.

Finally, the work in [4] presents two parallel algorithms for verification of propositional LTL specifications at run time. These algorithms are implemented in the tool RiTHM [13]. This paper enhances the framework in [4,13] by introducing a significantly more expressive formal specification language along with a parallel runtime verification system.

7. CONCLUSION

In this paper, we proposed a specification language (LTL_4-FO_c) for runtime verification of properties of types of objects in software and networked systems. Our language is a highly expressive fragment of first-order LTL with second-order numerical constraints. The six truth values of the semantics of LTL_4-FO_c allows system designers to obtain informative verdicts about the status of system properties at runt time. We also introduced an efficient and effective parallel algorithm with two implementations on multi-core CPU and GPU technologies. The results of our experiments on three real-world case studies show that runtime monitoring using GPU provides us with the best throughput and CPU utilization, resulting in minimal intervention in the normal operation of the system under inspection.

For future work, we are planning to design a framework for monitoring LTL_4-FO_c properties in distributed systems and cloud services. Another direction is to extend LTL_4-FO_c such that it allows non-canonical strings of quantifiers. Finally, we are currently integrating LTL_4-FO_c in our tool RiTHM [13].

8. REFERENCES

[1] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[2] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[3] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In *Runtime Verification*, pages 59–75. Springer Berlin Heidelberg, 2013.

[4] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. GPU-based runtime verification. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1025–1036, 2013.

[5] Maximiliano Caceres. Syscall proxying-simulating remote execution. *Core Security Technologies*, 2002.

[6] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 246–261. Springer, 2009.

[7] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories.

[8] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[9] Soha Hussein, Patrick Meredith, and Grigore Roşlu. Security-policy monitoring and enforcement with javamop. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, pages 3:1–3:11, New York, NY, USA, 2012. ACM.

[10] Dongyun Jin, P.O. Meredith, Choonghwan Lee, and G. Rosu. Javamop: Efficient parametric runtime monitoring framework. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1427–1430, June 2012.

[11] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[12] Patrick Meredith and Grigore Rosu. Efficient parametric runtime verification with deterministic string rewriting. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 70–80. IEEE, 2013.

[13] S. Navabpour, Y. Joshi, C. W. W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. Rithm: a tool for enabling time-triggered runtime verification for c programs. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 603–606, 2013.

[14] Daniel Ramsbrock, Robin Berthier, and Michel Cukier. Profiling attacker behavior following ssh compromises. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 119–124. IEEE, 2007.

[15] Feng Wang, Qin Xin, Bo Hong, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Tyce T McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the*

21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies, pages 139–152, 2004.

[16] Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. Watch global, cache local: Youtube network traffic at a campus network: measurements and implications. In *Electronic Imaging 2008*, pages 681805–681805. International Society for Optics and Photonics, 2008.