Tally Keeping-LTL: An LTL Semantics for Quantitative Evaluation of LTL Specifications

Raphaël Khoury, Sylvain Hallé
Department of Computer Science and Mathematics
Université du Québec à Chicoutimi

Abstract

When monitoring a trace using an LTL specification, the verdict returned by the monitor can often be insufficiently informative to be actionable. In this paper, we propose a generalization of LTL that allows formulae to evaluate to a natural or a real value, thus yielding quantitative information about the underlying trace. We illustrate with examples how this logic can be used to verify meaningful properties of traces. We provide an automata-based representation of this new logic as well as an implementation using the BeepBeep 3 complex event processor.

1 Introduction

There has recently been a growing interest in providing a quantitative, rather than qualitative verdict, on the evaluation of a specification, or security policy on traces. As several authors have observed, a reductionist 3-valued judgement on the evaluation of a formula on a trace is often insufficiently informative. A more practical procedure that provides an indication as to the manner or degree to which a property is satisfied or violated could be much more practical. For instance, if the specification states that any opened file is eventually closed, it is useful to distinguish between those sequences that trivially respect the property since no files are opened during the execution, from those that do so because any opened file is promptly closed. Likewise, it can also be useful to distinguish the number of execution steps that elapse from the moment a file is opened, until it is eventually closed, or the number of completed open-close pairs that are present in otherwise invalid executions.

In this paper, we suggest an extension of LTL, called Tally Keeping-LTL (TK-LTL), with associated semantics, which provides such quantitative judgement. Tally Keeping-LTL allows users to specify formulas that evaluate numerical properties of traces, such as the numbers of occurrences of specific formal patterns, or the distance in the trace between such patterns. We argue that this formalism allows for a more informative judgement.

Consider the following two motivating examples:

- Every *open* action is eventually followed by a corresponding *close* action. This property can be stated in LTL as $\mathbf{G}\, open \rightarrow \mathbf{F}\, close$. Observe that evaluating this property on a trace with the LTL's usual 3-valued semantics will always return the uninformative '?' verdict, for any trace. Using the proposed logic, we may be able to state or verify questions such as :
  - How many completed open/closed pairs occur in the trace?
  - What is the longest, shortest and average time from open to close?
  - How long does it take after a file is closed until it is eventually reopened?
  - During what fraction of the entire execution trace are there open files waiting to be closed?
- A *request* action must occur before any *allocate* action occurs. This property can be stated in LTL as $\neg allocate\, \mathbf{U}\, request$. Once again we can state finer questions such as:
  - How long before *allocate* does *request* occur?
  - How many *request*s occur before *allocate*?

This limitation of formal logic has often been evoked by our industry partners in the context of employing formal logic for trace analysis purposes [6]. In the words of an executive of a software firm who exclaimed when showed how an LTL specification can detect a bug in his company's code: "*All this for a Boolean*". A quantitative verdict would allow the creation of monitoring, testing and diagnostic tools that provide a more informative feedback.

The proposed extension also helps in addressing another difficulty that hampers the acceptance of formal methods in industrial practice, namely the difficulties encountered in writing the desired specification as an LTL formula. Indeed, our field work reveals that industrial partners often complain about the length and complexity of LTL formulas for real properties. An official of a software company even referred to this property as "*the attack of the wall of text*" [6].

In one instance, an industrial partner was seeking to ensure that a certain complex behavior ϕ could never occur more than three times during the same run of a target program. Unfortunately, the most effective way to state this requirement in LTL is to verify $(\mathbf{G}\,\neg\phi) \vee (\mathbf{F}\,\phi \wedge \mathbf{X}\,\neg\mathbf{F}\,\phi) \vee (\mathbf{F}\,\phi \wedge \mathbf{X}\,\mathbf{F}\,\phi \wedge \mathbf{X}\,\neg\mathbf{F}\,\phi) \vee (\mathbf{F}\,\phi \wedge \mathbf{X}\,\mathbf{F}\,\phi \wedge \mathbf{X}\,\mathbf{F}\,\phi \wedge \mathbf{X}\,\neg\mathbf{F}\,\phi)$. This formulation is a highly counterintuitive way to describe the behavior of interest, particularly since ϕ itself can be rather involved. Moreover, a change in ϕ, or otherwise a change in the number of times the occurrence of ϕ can be permitted to occur in the trace, would both necessitate multiple changes in the formula. Nor is it possible for that matter, to state a finer request such as "How many times does pattern ϕ occur?" or to indicate that a trace in which the pattern occurs only once is preferable to one in which the pattern occurs twice, even though both respect the desired property.

The remainder of this paper is organised as follows. In Section 2, we examine related works and in Section 3 we present some preliminary notions related to traces and temporal logic. In Section 4, we sketch out the syntax and semantics of our proposed extension to LTL. Section 5 provides practical examples that illustrate the flexibility and versatility of the proposed logic. Next, in Section 6, we provide an automata-based representation for the proposed logic. Section 7 presents a function implementation, using the BeepBeep complex event processor. Concluding remarks are given in Section 8.

2 Related Work

There has recently been much interest in a finer judgement of property validation than that of an elementary 2 or 3 valued judgement. Bauer et al. suggest a 4-valued interpretation of LTL over finite traces, which distinguishes between a sequence that does not yet respect the property but could do so, if the execution is allowed to proceed with no intervening event, and conversely one that does not yet irremediably violate the property, but likely will if corrective action does not occur. The two added judgements, possibly true possible false, are refinements of the undecided judgement '?', with $\top$ and $\bot$ retaining their usual semantic meaning. In [9], Medhat et al. extend this idea further by proposing a 6-valued semantics for LTL over finite traces. Once again, the additional judgements are refinements of '?' and $\top$ and $\bot$ retaining their usual semantic meaning. As the authors note "*We claim that these truth values provide us with informative verdicts about the status of different components of properties [...] at run time*". The insight that a multi-valued verdict is more informative and actionable is the driving motivation behind the development of TK-LTL.

Our study can be seen as a generalization of theirs. In particular, the logic proposed in this paper provides refinements to the basic judgement $\bot$ and $\top$, which can indicate the degree to which a specification is respected by a trace or the ease at which the specification is met.

Security policies naturally lend themselves to a quantitative judgement, capturing the intuitive notion of expressing the severity of a violation, rather than simply its occurrence. In [10], Ligatti et al. argue that a quantitative judgement indicating the degree to which a security policy is respected would provide multiple benefits. They further generalize the well-established notions of safety and liveness to such a quantitative enforcement context. The extension of LTL proposed in this paper allows users to define such quantitative security properties in a formal manner.

Other researchers focused on enforcement [8, 4, 3], and argued that a quantitative judgement on security policy satisfaction can help in selecting the most adequate reactive measure when faced with a potential violation of the security policy. This paper provides a logical framework to assign a numerical value to each trace, thus allowing the selection of the appropriate corrective action to be based on theoretical footing. Quantitative judgements are also widely used in model checking, (see [2] for ex.), a throughout review of this field, however, would fall outside the scope of this paper.

3 Preliminaries

Let Σ be a finite or countably infinite set of atomic events. A *trace* is finite sequence of events from Σ. We write ϵ for the empty sequence and Σ^* for the set of all finite sequences. We let σ, τ range over sequences. We write $\tau;\sigma$ for the concatenation of τ and σ. We say that τ is a prefix of σ noted $\tau \preceq \sigma$ iff there exists a sequence σ' such that $\tau;\sigma' = \sigma$. The ith event in a sequence σ is given as σ_i, with the initial event being given as σ_1. The length of σ is written $|\sigma|$. We write $\sigma_{i..}$ for the suffix of σ starting on its ith event, and $\sigma_{..i}$ prefix of σ starting at its initial event and ending with event i.

LTL's syntax is based on classical propositional logic, using the connectives $\neg$ ("not"), $\vee$ ("or"), $\wedge$ ("and"), $\rightarrow$ ("implies"), to which five temporal operators have been added. An LTL formula is a well-formed combination of these operators and connectives, according to the usual construction rules:

Definition 1 (LTL Syntax)**.**

1. *If x and y are variables or constants, then $x = y$ is an LTL formula;*

2. *If φ and ψ are LTL formulæ, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\mathbf{G}\,\varphi$, $\mathbf{F}\,\varphi$, $\mathbf{X}\,\varphi$, $\varphi\,\mathbf{U}\,\psi$ are LTL formulæ;*

Boolean connectives carry their usual meaning, and the following identities allow some connectors to be defined from proceeding ones : $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$, $\mathbf{G}\,\varphi \equiv \neg\mathbf{F}\,\neg\varphi$, $\psi\,\mathbf{W}\,\varphi \equiv (\psi\,\mathbf{U}\,\varphi) \vee \mathbf{G}\,\psi$.

We build upon a three-valued semantics for LTL over finite sequences, in which $\sigma \models \varphi$ evaluates to $\top$ iff every possible continuation of σ models φ; $\sigma \models \varphi$ evaluates to $\bot$ iff every possible continuation of σ does not respect φ, and $\sigma \models \varphi$ evaluates to '?' otherwise. We let v range over this set of truth values $V = \{\top, \bot, ?\}$. We write $\sigma \models^v \varphi$ (resp. $\sigma \not\models^v \varphi$) as a shorthand for $\sigma \models \varphi = v$ (resp. $\sigma \models \varphi \neq v$).

Definition 2 (LTL Semantics(from [1]))**.**

$$\sigma \models p \quad = \begin{cases} \top, & \textit{if } p \in \sigma_1; \\ \bot, & \textit{otherwise;} \end{cases}$$

$$\sigma \models \neg\varphi \quad = \begin{cases} \top, & \textit{if } \sigma \not\models \varphi; \\ \bot, & \textit{if } \sigma \models \varphi; \\ ?, & \textit{otherwise;} \end{cases}$$

$$\sigma \models \varphi \vee \psi = \begin{cases} \top, & \textit{if } \sigma \models \varphi = \top \wedge \sigma \models \psi; \\ \bot, & \textit{if } \sigma \models \varphi = \bot \vee \sigma \not\models \psi; \\ ?, & \textit{otherwise;} \end{cases}$$

$$\sigma \models \mathbf{X}\,\varphi \quad = \begin{cases} \sigma_{2..} \models \varphi & \textit{if } |\sigma| \geq 2 \\ ? & \textit{otherwise;} \end{cases}$$

$$\sigma \models \mathbf{F}\,\varphi \quad = \begin{cases} \top & \textit{if } \exists\, 1 \leq j \leq |\sigma| : \sigma_{j..} \models \varphi \\ ? & \textit{otherwise;} \end{cases}$$

$$\sigma \models \mu\,\mathbf{U}\,\eta = \begin{cases} \top & \textit{if } \exists\, 1 \leq j \leq |\sigma| : ((\sigma_{j..} \models \eta) \wedge \\ & \quad (\forall\, 1 \leq k < j \sigma_{k..} \models \mu)) \\ \bot & \textit{if } \exists\, 1 \leq j \leq |\sigma| : ((\sigma_{j..} \not\models \mu) \wedge \\ & \quad (\forall\, 1 \leq k \leq j \sigma_{k..} \not\models \eta)) \\ ? & \textit{otherwise;} \end{cases}$$

4 Syntax and Semantics

TK-LTL extends 3-valued LTL with two sets of semantic structures, namely counters and quantifiers. Counters, which range over $\mathcal{C}, \mathcal{D}$ and $\mathcal{L}$ and serve to count the number of prefix in a sequence that evaluate to a given truth value, usually $\top$, or the index in the trace at which a condition holds. For example, the counter $\mathcal{C}^v_\varphi$, where φ is an LTL formula and v ranges over the truth values in V, returns the number of prefixes of the input traces for which the evaluation of φ evaluates to v. The values returned by counters ranges over $\mathbb{N}$, but arithmetic operators or functions can be freely applied to the outputs of multiples counters over the same sequence to compute information about the trace, yielding a value in $\mathbb{R}$. Quantifiers, such as $\forall_{\sim k}\mathcal{C}$ or $\exists_{\sim k}\mathcal{C}$, verify if the values returned by a counter meet a given condition c, and return a verdict in V. This allows unlimited recursion of alternating LTL formulae, counters and quantifiers.

Definition 3. *(TK-LTL Syntax)*
$\varphi ::= \top \,|\, \bot \,|\, F\varphi \,|\, G\varphi \,|\, \ldots \,|\, \mathcal{Q}$
$\mathcal{C} ::= \mathcal{C}^v_\varphi \,|\, \mathcal{D}^v_\varphi \,|\, {}_\varphi\mathcal{D}^v_\phi \,|\, \mathcal{L}^\top_\varphi \,|\, \mathcal{C} \star \mathcal{C} \,|\, f(\mathcal{C})$
$\mathcal{Q} ::= \mathcal{P}_{\sim k}\mathcal{C} \,|\, \forall_{\sim k}\mathcal{C} \,|\, \exists_{\sim k}\mathcal{C}$
where v ranges over the values of V, f ranges over unary functions in $\mathbb{R} \times \mathbb{R}$, $\star$ ranges over arithmetic operators, $\sim \in \{<, >, \leq, \geq, =, \neq\}$ and $k \in \mathbb{R}$.

Since we expect the evaluation of a formula to yield a result in $\{\top, \bot, ?\} \cup \mathbb{R}$ we redefine the notion of satisfaction so that it yields a value in this range: $\Vdash\; : \Sigma^* \times \{\varphi \cup \mathcal{C} \cup \mathcal{Q}\} \times \{\top, \bot, ?\} \cup \mathbb{R}$. For clarity, we continue to freely use $\sigma \models \varphi$ and its negation $\sigma \not\models \varphi$ when φ is an LTL property with a truth value in $\{\top, \bot, ?\}$.

As stated above, the counter $\mathcal{C}^v_\varphi$, where φ is an LTL formula and v ranges over the truth values of LTL, returns the number of prefixes of the input traces for which the evaluation of φ evaluates to v. We write $\mathcal{C}^{\geq ?}_\varphi$ as a stand-in for $\mathcal{C}^?_\varphi + \mathcal{C}^\top_\varphi$ and $\mathcal{C}^{\leq ?}_\varphi$ for $\mathcal{C}^?_\varphi + \mathcal{C}^\bot_\varphi$.

Definition 4. *(Semantics of $\mathcal{C}$)*
$\sigma \Vdash \mathcal{C}^v_\varphi = |\{\tau | \tau \preceq \sigma \wedge (\tau \models^v \varphi)\}|$.

The token $\mathcal{C}^\top_\top$ naturally returns the length of the trace.

Lemma 1. $\forall \sigma \in \Sigma^* : \sigma \Vdash \mathcal{C}^\top_\top = |\sigma|$.

The values returned by counters can easily be compared using arithmetic operators. For instance $\frac{\mathcal{C}^\top_p}{\mathcal{C}^\top_\top}$ returns the percentage of events in the trace that respect p.

In the course of experimenting with TK-LTL, it was frequently useful to identify the initial point in the input trace where a given property holds. We write $\mathcal{D}^v_\varphi$ for the counter that returns the initial position in σ where φ evaluates to v, or 0 if σ exhibits no prefix for which φ evaluates to v.

Definition 5. *(Semantics of $\mathcal{D}$)*

$$\sigma \Vdash \mathcal{D}^v_\varphi = \begin{cases} i, & \textit{if } \exists i \leq |\sigma| : \sigma_{i..} \models^v \varphi \wedge \forall j < i : \sigma_{j..} \not\models^v \varphi; \\ 0, & \textit{otherwise.} \end{cases}$$

It is also convenient to identify the first position at which a condition holds, starting not from the beginning of the trace, but from the satisfaction of another condition. The binary counter ${}_\phi\mathcal{D}^v_\varphi$ returns this information, or 0 if σ does not exhibit such a prefix.

Definition 6. *(Semantics of $\mathcal{D}$)*

$$\sigma \Vdash {}_\phi\mathcal{D}^v_\varphi = \begin{cases} k, & \textit{if } \exists i \leq |\sigma| : \sigma_i \models^v \phi \wedge \sigma_{i+k..} \models^v \varphi \wedge \\ & \quad \nexists j > i : j < (i+k) : \sigma_{j..} \wedge \not\models^v \varphi; \\ 0, & \textit{otherwise.} \end{cases}$$

Note that, in the above definition, in cases where multiple extensions of σ_i satisfy φ, the second part of the conjunct ensures that k returns the least possible value for which $\sigma_{i+k} \models \varphi$.

Finally, the syntax of TK-LTL also includes the counter *local*, given as $\mathcal{L}_{\varphi}^{v}$, which returns the index of the last occurrence of an event for which the property φ evaluates to v, or 0 if σ has no prefix for which that is the case.

Definition 7. *(Semantics of $\mathcal{L}$)*

$$\sigma \Vdash \mathcal{L}_{\varphi}^{v} = \begin{cases} i, & \textit{if } \exists i < |\sigma| : \sigma_{..i} \models^{v} \varphi \wedge \\ & \quad \neg \exists j > i : \sigma_{..j} \models^{v} \varphi \\ 0, & \textit{otherwise.} \end{cases}$$

In addition to counters, we enrich the semantics of TK-LTL with quantifiers. Quantifiers examine the value returned by a counter for each prefix of the input sequence, and return a value from the same 3-valued truth domain as an LTL-property according to a condition subscripted to the quantifier. TK-LTL defines three quantifiers: the first two are the existential and universal quantifiers, with natural semantics. For instance, the formula $\exists_{=5}\mathcal{C}_{p}^{\top}$ returns $\top$ if the atomic proposition p holds on at least 5 prefixes of the input trace, and returns $'?'$ otherwise. Conversely, the formula $\exists_{<0}\mathcal{C}_{p}^{\top} - \mathcal{C}_{q}^{\top}$ returns $\top$ iff there exists a prefix of the input trace for which the atomic proposition q holds more often than p.

The third quantifier is termed the propositional quantifier, and is written as $\mathcal{P}$. $\sigma \Vdash \mathcal{P}_{\sim k}\mathcal{C}$ evaluates to $\top$ if the comparison $n \sim k$ holds where n is the value returned by $\sigma \Vdash \mathcal{C}$. For example, let $\sigma = a; a; a; b; a;$ be a trace. the formula $\sigma_i \Vdash \mathcal{P}_{=3}\mathcal{C}_{a}^{\top}$ evaluates to $\top$ for $i = 3$ and $i = 4$, and to $\bot$ in all other cases.

Definition 8. *(Semantics of $\forall_{\sim k}\mathcal{C}$)*

$$\sigma \Vdash \forall_{\sim k}\mathcal{C}_{\varphi}^{v} = \begin{cases} \bot, & \textit{if } \exists i \leq |\sigma| : \sigma_{..i} \Vdash \mathcal{C}_{\varphi}^{v} \nsim k; \\ ?, & \textit{otherwise.} \end{cases}$$

Definition 9. *(Semantics of $\exists_{\sim k}\mathcal{C}$)*

$$\sigma \Vdash \exists_{\sim k}\mathcal{C}_{\varphi}^{v} = \begin{cases} \top, & \textit{if } \exists i \leq |\sigma| : \sigma_{..i} \Vdash \mathcal{C}_{\varphi}^{v} \sim k; \\ ?, & \textit{otherwise.} \end{cases}$$

Definition 10. *(Semantics of $\mathcal{P}_{\sim k}\mathcal{C}$)*

$$\sigma \Vdash \mathcal{P}_{\sim k}\mathcal{C}_{\varphi}^{v} = \begin{cases} \top, & \textit{if } \sigma \Vdash \mathcal{C}_{\varphi}^{v} \sim k; \\ \bot, & \textit{otherwise.} \end{cases}$$

Since quantifiers evaluate to a value in V, they can be freely used alongside other constructs in complex TK-LTL formulae. For example, the following formula states that if there are 4 or more requests for a resource waiting for a response then this fact must be logged : $(\exists_{>3}(\mathcal{C}_{\textit{req}}^{\top} - \mathcal{C}_{\textit{resp}}^{\top})) \rightarrow$ $\mathbf{F}$ *log*.

Figure 1 shows several identities over the relationship between LTL formulae, counters and quantifiers. Proofs are provided in Appendix A.

$$\begin{array}{cc} \mathbf{F}\,\exists_{\sim k}\mathcal{C}_{\varphi}^{v} \equiv \exists_{\sim k}\mathcal{C}_{\varphi}^{v} & \text{Ł}\top\mathbf{F}\,\varphi \equiv \mathcal{C}_{\varphi}^{\top} \\ \mathcal{C}_{F\varphi}^{?} \equiv \mathcal{C}_{\top}^{\top} - \mathcal{C}_{F\varphi}^{\top} & \neg\forall_{\sim k}\mathcal{C}_{\varphi}^{v} \equiv \exists_{\nsim k}\mathcal{C}_{\varphi}^{v} \\ \mathbf{G}\,\forall_{\sim k}\mathcal{C}_{\varphi}^{v} \equiv \forall_{\sim k}\mathcal{C}_{\varphi}^{v} & \text{Ł}\bot\mathbf{G}\,\varphi \equiv \mathcal{C}_{\top}^{\top} \\ \mathcal{C}_{\mathbf{G}\,\varphi}^{\bot} \equiv \mathcal{C}_{\top}^{\top} - \mathcal{C}_{G\varphi}^{?} & \neg\exists_{\sim k}\mathcal{C}_{\varphi}^{v} \equiv \forall_{\nsim k}\mathcal{C}_{\varphi}^{v} \end{array}$$

Figure 1: Identities over TK-LTL formulae

5 Examples

We illustrate the application of TK-LTL using sample LTL formulae taken from *Spec Patterns*, an online repository of commonly used LTL patterns[1]. For each property, we will give a few examples of the types of quantitative properties that can be stated using TK-LTL. As these examples illustrate, TK-LTL allows properties that capture useful information about the trace under consideration.

In what follows p,q and r are atomic events.

- Consider the property stating that if p occurs, then q must occur at some point afterwards. This property is given as $\mathbf{G}\,(p \rightarrow \mathbf{F}\,q)$, and is similar to the *open* $\rightarrow$ *close* example mentioned in the introduction. Recall that for this property, the usual 3-valued semantics of LTL always return the uninformative verdict '?', regardless of the input trace. Using the specification language proposed in this paper, we can state and answer questions that provide meaningful information about the underlying execution. Many of these formulas evaluate to a numeric value, and thus provide a much more precise assessment of the trace than can be provided using a 3-values logic.
  - How long after the initial p does q first occur: ${}_{p}\mathcal{D}_{q}^{\top}$.
  - How many qs occur after the initial p, notwithstanding any q that occurs before: $max(\mathcal{C}_{q}^{\top} - \mathcal{C}_{q \wedge \mathbf{G}\,\neg p}^{>?}, 0)$
  - How many qs occur before the first occurrence of p: $\mathcal{C}_{q \wedge \mathbf{G}\,\neg p}^{?}$
  - How many events occur in situations where an event p has occurred, and has not yet been followed by a corresponding q event: $\mathcal{C}_{\mathcal{P}_{>0}(\mathcal{L}_{p}^{\top} - \mathcal{L}_{q}^{\top})}^{\top}$.
  - What percentage of the sequence consists in events that follow the occurrence of a p event, but precedes the occurrence of the corresponding q: $\frac{\mathcal{C}_{\mathcal{P}_{>0}(\mathcal{L}_{p}^{\top} - \mathcal{L}_{q}^{\top})}^{\top}}{\mathcal{C}_{\top}^{\top}}$

[1] http://patterns.projects.cs.ksu.edu/documentation/patterns/ltl.shtml

– Is there an interval of length more than n between the occurrence of a p event and the corresponding q: $(\mathcal{P}_{>n}\mathcal{L}^{\top}_{q} - \mathcal{L}^{\top}_{p}) \wedge \mathbf{F}\, p$. The $\mathbf{F}$ p at the end ensures that a case in which a q occurs without any p preceding it is not counted.
– How many completed pairs of the form (p, q) occur in the sequence: $\mathcal{C}^{\top}_{(\exists_{=1}\mathcal{L}^{\top}_{q} - \mathcal{L}^{\top}_{p}) \wedge \mathbf{F}\, p}$.

- p becomes true before q, if it ever holds. This property is given in LTL as: $\neg q\, \mathbf{U}\, p$.
  – How many ps occur before the first occurrence of q : $\mathcal{C}^{\top}_{p \wedge \mathbf{G}\, \neg q}$.
  – What percentage of the events occurring before the first q are ps : $min(\mathcal{D}^{\top}_{q}, \frac{\mathcal{C}^{\top}_{p \wedge \mathbf{G}\, \neg q}}{|\mathcal{D}^{\top}_{q} - 1|})$. (0 otherwise)
  – Starting from the first p, how many events elapse until the first q is reached (not counting any preceding ones): ${}_{p}\mathcal{D}^{\top}_{q}$.

- p holds at some point between q and r. This property is given in the Spec Patterns library as: $\mathbf{G}\,(q \wedge \neg r \rightarrow (p\, \mathbf{W}\, r))$.
  – How many occurrences of p are there between the first q and the first r: $\mathcal{C}^{\top}_{p \wedge \mathbf{F}\, q} - \mathcal{C}^{\top}_{p \wedge \mathbf{F}\, r}$.
  – What percentage of the events occurring between q and r are ps : (0 if q has not occurred) $min({}_{q}\mathcal{D}^{\top}_{r}, \frac{\mathcal{C}^{\top}_{p \wedge \mathbf{F}\, q} - \mathcal{C}^{\top}_{p \wedge \mathbf{F}\, r}}{|{}_{q}\mathcal{D}^{\top}_{r} - 1|})$.
  – Stating from the first occurrence of q, how many events elapse until the first occurrence of p is reached : $\mathcal{C}^{\top}_{\mathbf{F}\, q \wedge \neg p}$

As can easily be seen, these properties correspond to meaningful program behaviors, and an evaluation of these formulae will provide actionable information about the target program. Consider the first example, $G(p \rightarrow \mathbf{F}\, q)$, an important response property, can capture an essential requirement in resource management, indicating that if a given resource is acquired, that resource must eventually be released. The first TK-LTL property we propose, ${}_{p}\mathcal{D}^{\top}_{q}$ computes the number of steps that occur between acquisition and release, and informative refinement of the previous property. The next two properties capture a mismatch between the number of requests and the number of responses, which may indicate an inefficiency in the allocation process. The fourth property, $\mathcal{C}^{\top}_{\mathcal{P}_{>0}(\mathcal{L}^{\top}_{p} - \mathcal{L}^{\top}_{q})}$, counts the number of events in the sequence that occur while the resource is in use, and the fifth property calculates the portion of the execution during which the resource is allocated. The penultimate property indicates whether or not the resource is ever acquired and held for more than n execution steps. The final property counts the number of times the resource is acquired and then released during the execution.

6 Automata Model

In this section, we propose an automata representation for TK-LTL, which draws upon the edit automaton[7]. Such a model would allow users of TK-LTL to draw upon the numerous tools that already exist to write and manipulate automata. For a given TK-LTL property φ, we build a chain of sequential edit automaton $\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2, ...$ such that the output of one automaton is fed as input in the next. The initial automaton $\mathcal{E}_0$ takes as input σ. The output of the final edit automaton is the result of the evaluation of $\sigma \Vdash \varphi$.

The edit automaton is a deterministic, finite or countably infinite state machine that receives an input sequence, and produces an alternate output sequence.

Definition 11. *Edit Automaton (from [7]) An edit automaton is a tuple $\langle \Sigma, Q, q_0, \delta \rangle$ where:*

- *Σ is the input alphabet of the original input sequence;*
- *Q is a finite set of states;*
- *$q_0 \in Q$ is a distinguished initial state;*
- *$\delta : (Q \times \Sigma) \rightarrow (Q \times \Sigma^*)$ is a transition function. Given the current state and input event, it specifies the automatons output and successor state.*

The conversion from TK-LTL formula to automaton proceeds in three steps. First, any LTL subformula contained in the TK-LTL property is translated to a finite state automaton using an appropriate translation algorithm [11].

Definition 12. *Finite Deterministic Automaton A Finite State Automaton (FSA) $\langle \Sigma, Q, q_0, \delta, F \rangle$ is a finite deterministic state machine where:*

- *Σ is the input alphabet of the original input sequence;*
- *Q is a finite set of states;*
- *$q_0 \in Q$ is a distinguished initial state;*
- *$\delta : Q \times \Sigma \times Q$ is a transition function; and*
- *$F \subseteq Q$ is a (possibly empty) subset of Q that contains accepting states.*

Let $q \in Q$ be a state, we write $reach(q)$ for the set of states that are *reachable* from q, i.e., the set of states that can be reached from q by following any number of transitions. For the sake of simplicity, the elements $\Sigma, Q, q_0, \delta...$ defining a deterministic finite automaton or an edit automaton $\mathcal{A}$ are referred to using the formalism $\mathcal{M}.\Sigma, \mathcal{A}.Q, \mathcal{A}.q_0, \mathcal{A}.\delta...$ or simply $\Sigma, Q, q_0, \delta...$ when $\mathcal{A}$ is clear from the context.

Let $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$ be a finite state automaton that accepts language $\mathcal{L}_{\varphi}$ consisting of exactly those sequences for which φ evaluates to $\top$. We convert $\mathcal{A}$ into an edit automaton $\mathcal{E} = \langle \Sigma', Q', q'_0, \delta' \rangle$ that returns a value from V, in lockstep with each input action as follows:

- $\Sigma' = \Sigma \cup V$;
- $Q' = Q$ This allows a bijection between the states of $\mathcal{A}$ and those of $\mathcal{E}$. For any state $\mathcal{A}.q_i$ of $\mathcal{A}$ there is a corresponding state $\mathcal{E}.q_i$ of $\mathcal{E}$. We write $\mathcal{A}.q_i \leftrightharpoons \mathcal{E}.q_i$ to indicate this relationship.
- $q'_0 = q_0$, where $q'_0 \leftrightharpoons q_0$.
- $\mathcal{E}.\delta'(\mathcal{E}.q, a) =$
$$\begin{cases} (q', \top), & \text{if } \delta(q,a) = \mathcal{A}.q' \wedge \mathcal{A}.q' \leftrightharpoons \mathcal{E}.q' \wedge \\ & \quad \mathcal{A}.q' \in F \\ (q', \bot), & \text{if } \mathcal{A}.\delta(q,a) = \mathcal{A}.q' \wedge \mathcal{A}.q' \leftrightharpoons \mathcal{E}.q' \\ & \wedge reach(\mathcal{A}.q') \cap \mathcal{A}.F = \emptyset \\ (q', ?), & \text{otherwise.} \end{cases}$$

Let $\mathcal{E} = \langle \Sigma, Q, q_0, \delta \rangle$ be an edit automaton as constructed above, and capturing a 3-valued evaluation of a formula φ. We can construct an edit automaton $\mathcal{E}' = \langle \Sigma', Q', q'_0, \delta' \rangle$ capturing the semantics of the counter $\mathcal{C}^v_\varphi$ as follows:

- $\Sigma' = \mathbb{N}$;
- $Q' = \mathbb{N}$;
- $q'_0 = 0$;
- $\mathcal{E}'.\delta(\mathcal{E}'.n, a) = \begin{cases} (n+1, n+1), & \text{if } a = v \\ (n, n), & \text{otherwise.} \end{cases}$

The construction of an automaton for $\mathcal{D}$ is only slightly more involved. That of the automata for $\mathcal{D}$ and $\mathcal{L}$ is also similar, and is omitted here out of space considerations.

- $\Sigma' = \mathbb{N}$;
- $Q' = \langle \mathbb{N} \times \mathbb{B} \rangle$;
- $q'_0 = 0 \times \bot$;
- $\mathcal{E}'.\delta(\langle \mathcal{E}.n, b \rangle, a) =$
$$\begin{cases} (\langle n+1, \top \rangle, n+1), & \text{if } a = v \wedge b = \bot \\ (\langle n, \top \rangle, n), & \text{if } a = v \wedge b = \top \\ (\langle n+1, \bot \rangle, 0), & \text{otherwise.} \end{cases}$$

Finally, an NFA with input alphabet $\mathbb{N}$ or $\mathbb{R}$, can receive as input the output of an edit automaton as described above, (or the results of applying arithmetic operators or functions to the results of multiple such automata in lockstep), and accept a language equivalent to the LTL formula captured by an quantifier. Let $\mathcal{Q}_{\sim k}\mathcal{C}^v_\varphi$ be a quantifier and let σ be the input sequence. We generate a new input sequence σ' over the alphabet $\{\top, \bot\}$ where $\forall i < |\sigma| : \sigma'_i = (\sigma_i \Vdash \mathcal{C}^v_\varphi) \sim k$. The quantifiers $\mathcal{P}_{\sim k}\mathcal{C}, \forall_{\sim k}\mathcal{C}$ and $\exists_{\sim k}\mathcal{C}$ can then be stated as the LTL formulae v, $\mathbf{G}\, v$ and $\mathbf{F}\, v$ respectively.

7 Implementation with BeepBeep

Tally Keeping-LTL was implemented using the event processing tool BeepBeep [5]. BeepBeep is a complex event processing tool that can perform complex manipulations on large data streams efficiently. Internally, BeepBeep decomposes the desired data-processing task into a number of atomic *processors*, each of which takes as input one (or more) event streams, and in turn, outputs one or more event streams. A processor is a short segment of Java code, usually no more than 20 lines, that performs a single manipulation on an element of datum. Processors are chained together to form *processor chains* with the output of one (or more) processor being piped to the input of the next one in such a manner that, feeding BeepBeep's input stream through this chain produces the desired computation. The processor chain itself performs the desired data processing task or algorithm. Processors that simulate LTL's usual operators, as well as a wide variety of arithmetic operations are already included in BeepBeep's palette. We have created a small number of new processors, that simulate the constructs created in this paper namely $\mathcal{C}, \mathcal{D}, \mathcal{L}, \mathcal{P}$ and the particular definitions of $\exists$ and $\forall$.

Each of the counters and quantifiers that form the syntax of TK-LTL was implemented in a single parameterizable processor, while the properties of interest are stated as processor chains. As an example figure 2 gives the processor chain for the property 1 described below. The leftmost elements of the processor chain extract and parse the data from the logfile, creating a data stream of events, which will flow through the processor chain. Each event corresponds to a single line of the log. The first processor checks if this event corresponds to a method call of the method of interest, whose name is a parameter to the processor itself. It outputs a stream of $\top$ or $\bot$ accordingly. The next processor is called the *troolian cast*, and its only purpose is to transform the stream of boolean values into a stream containing the values $\top$ and ? consistent with the semantics of LTL and the property of interest. The stream is then duplicated into two identical copies, running in parallel and passed to two processors that simulate the behavior of the TK-LTL operator $\mathcal{L}$. These processors' output streams of integer values, with the top stream indicating the last index at which the method of interest was observed and the bottom one indicating the last index at which it was not seen. The two streams are merged in the penultimate processor, which performs a substraction between values taken from each input stream, and passes the result along. The final processor simulates the behavior of the $\exists$, as defined above in the context of TK-LTL . It will return $\top$ continuously as soon as it receives an input larger than 10. Until that point, however, it returns $'?'$,

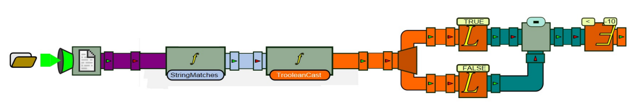


Figure 2: The BeepBeep processor chain for property 1

We tested our implementation using two data traces. The first is a trace of method calls that captures every single method called during the execution of a Java program. The trace has a length 1.2 million lines, with each line corresponding to either a single method call, or a method return. Such traces can be generated using a variety of tools, and serve multiple purposes, notably in development, software maintenance and security enforcement. We wrote and tested three properties on this trace:

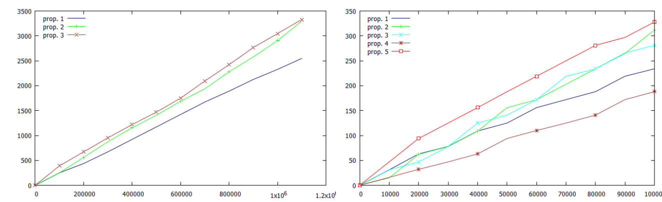
- Property 1: Does there exist in the trace an uninterrupted sequence of calls to a specific method, org/gjt/sp/jedit/buffer/LineManager, of length more than 10? Such a property would be interesting when examining the behavior of a recursive function. This property is given as : $\exists_{>10}\mathcal{L}^{\top}_{\neg p} - \mathcal{L}^{\top}_{p}$, with $p =$ org/gjt/sp/jedit/buffer/LineManager.
- Property 2: A call to the method org/gjt/sp/jedit/buffer/KillRing occurs how many lines after a call to the org/gjt/sp/jedit/buffer/ContentManager. This is a simple instantiation of the $\mathcal{D}$ operator, which provides a baseline for comparing the execution time of properties?
- Property 3: How many calls to the method org/gjt/sp/jedit/buffer/LineManager occur before the first call to org/gjt/sp/jedit/buffer/UndoManager. This is an instantiation of the third property given in the previous section?

The execution times of these properties on the method calls traces described above is given in figure 3a.

As a second example, we used a modified Windows active directory log from the servers of a large local company, from which we deleted part of the information. Each line represents a single active directory event, such as logon, logoff, Credential Validation, Process Creation etc. In total the trace contains 100 000 lines, of about a dozen event types. For the purpose of this example, since our goal is to showcase the features of TK-LTL rather than perform an actual trace analysis, we take into consideration only the types of active directory events, and disregard all other information present in the trace. We tested four properties on this trace:

- Property 1: What percentage of the trace consists in actions taken during a session, meaning that a logon has occurred, but the corresponding logoff has not yet occurred?
- Property 2: Is there a session of length more than 100 (i.e. a logon not followed by the corresponding logoff for 100 lines)?
- Property 3: How many completed pairs of logon and logoff events occur in the trace?
- Property 4: Is there a point on the trace such that at that point, more logoff have occurred than logon ?
- Property 5: How many events of the type Sensitive Privilege Use occur between a logon event and a logoff event?

Properties 1-3 are instantiations of the last three properties of the first example given in the previous section. Property 4 is given as : $\exists_{<0}\mathcal{C}^{\top}_{\text{logon}} - \mathcal{C}^{\top}_{\text{logoff}}$ and property 5 is an instantiation of the first property of the first example of section 5. The execution times of these three properties on the active directory log is given in the figure 3b.



(a) Java Method Names trace (b) Active Directory trace

Figure 3: Experimental results: Execution time

These examples illustrate that TK-LTL can be used to state interesting real-life properties that cannot otherwise be expressed with existing formal logics, and furthermore illustrate that verification can be performed effectively using our implementation through the tool BeepBeep.

8 Conclusion

In this paper we propose a generalization of LTL that allows formulae to evaluate to natural or real values, thus yielding quantitative information about the underlying trace. We additionally provide an automata model as well as an implementation.

In the future work, we seek to use TK-LTL in the context of runtime enforcement. When confronted with a potential violation of the security policy, runtime enforcement requires the selection of the appropriate corrective action, from within a set of possible reactions. By providing a metric for comparing executions, we believe that TK-LTL can form the basis for this selection.

References

[1] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006*, pages 260–272.

[2] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, Sept. 2012.

[3] P. Drábik, F. Martinelli, and C. Morisset. Cost-aware runtime enforcement of security policies. In *Security and Trust Management - 8th International Workshop, STM 2012, Pisa, Italy, Sept. 13-14, 2012*, pages 1–16.

[4] P. Drábik, F. Martinelli, and C. Morisset. *A Quantitative Approach for Inexact Enforcement of Security Policies*, pages 306–321. Springer Berlin Heidelberg.

[5] S. Hallé. When RV meets CEP. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, Sept. 23-30, 2016*, pages 68–91.

[6] S. Hallé, R. Khoury, and S. Gaboury. A few things we heard about rv tools (position paper). In *Inter. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools(RV-CuBES 2017)*, volume 3, pages 89–95.

[7] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.

[8] F. Martinelli, I. Matteucci, and C. Morisset. From qualitative to quantitative enforcement of security policy. In *Proceedings of the 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, MMM-ACNS'12, pages 22–35. Springer-Verlag.

[9] R. Medhat, B. Bonakdarpour, S. Fischmeister, and Y. Joshi. Accelerated runtime verification of LTL specifications with counting semantics. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 251–267, 2016.

[10] D. Ray and J. Ligatti. A theory of gray security policies. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2015.

[11] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

A Proofs of the identities in Figure 1

Theorem 1. $\mathbf{F}\,\exists_{\sim k}\mathcal{C}^v_\varphi \equiv \exists_{\sim k}\mathcal{C}^v_\varphi$

Proof. $\mathbf{F}\,\exists_{\sim k}\mathcal{C}^v_\varphi$

$$\langle def.of\mathbf{F}\rangle$$

$\exists\, 1 \leq j \leq |\sigma| : \exists_{\sim k}\mathcal{C}^v_\varphi$

$$\langle def.9\rangle$$

$\exists\, 1 \leq j \leq |\sigma| : \exists i < |\sigma| : \sigma_i \Vdash \mathcal{C}^v_\varphi \sim k$

$$\langle i = j, def.of\exists\rangle$$

$\exists i < |\sigma| : \sigma_i \Vdash \mathcal{C}^v_\varphi \sim k$

$$\langle def.9\rangle$$

$\exists_{\sim k}\mathcal{C}^v_\varphi$ □

Theorem 2. $\mathbf{G}\,\forall_{\sim k}\mathcal{C}^v_\varphi \equiv \forall_{\sim k}\mathcal{C}^v_\varphi$

Proof. The proof follows exactly as that of Theorem 1 above, and has been elided out of space considerations. □

Theorem 3. $\mathcal{C}^?_{\mathbf{F}\,\varphi} \equiv \mathcal{C}^\top_\top - \mathcal{C}^\top_{F\varphi}$

Proof. Follows immediately from the fact that $\mathcal{C}^\top_\top$ is equal to the length of the input trace, and the fact that $\sigma \Vdash F\varphi$ can only evaluate to ? or to $\top$. □

Theorem 4. $\mathcal{C}^\bot_{\mathbf{G}\,\varphi} \equiv \mathcal{C}^\top_\top - \mathcal{C}^?_{G\varphi}$

Proof. Follows immediately from the fact that $\mathcal{C}^\top_\top$ is equal to the length of the input trace, and the fact that $\sigma \Vdash G\varphi$ can only evaluate to ? or to $\bot$. □

Theorem 5. $\neg\exists_{\sim k}\mathcal{C}^v_\varphi \equiv \forall_{\nsim k}\mathcal{C}^v_\varphi$

Proof. $\neg\exists_{\sim k}\mathcal{C}^v_\varphi$

$$\langle def.9\rangle$$

$\neg\exists i < |\sigma| : \sigma_i \Vdash \mathcal{C}^v_\varphi \sim k$

$$\langle def.of\exists\rangle$$

$\forall i < |\sigma| : \sigma_i \Vdash \mathcal{C}^v_\varphi \nsim k$

$$\langle def.8\rangle$$

$\forall_{\nsim k}\mathcal{C}^v_\varphi$ □

Theorem 6. $\neg\forall_{\sim k}\mathcal{C}^v_\varphi \equiv \exists_{\nsim k}\mathcal{C}^v_\varphi$

Proof. The proof follows exactly as that of Theorem 5 above, and has been elided out of space considerations. □

Theorem 7. $\mathcal{L}^\top_{\mathbf{F}\,\varphi} \equiv \mathcal{C}^\top_\varphi$

Proof. Case where $\mathcal{L}^\top_{\mathbf{F}\,\varphi} \neq 0$:

$\mathcal{L}^\top_{\mathbf{F}\,\varphi}$ $\langle def.7\rangle$

$\{i | i < |\sigma| : \sigma_{i..} \models \mathbf{F}\,\varphi \wedge \neg\exists j > i : \sigma_{j..} \models \mathbf{F}\,\varphi\}$

$$\langle \sigma \models \mathbf{F}\,\varphi \Rightarrow \forall\tau \preceq \sigma : \tau \models \mathbf{F}\,\varphi\rangle$$

$\{i | i < |\sigma| : \sigma_{i..} \models \mathbf{F}\,\varphi \wedge \forall k \leq i\sigma_{k..} \models \mathbf{F}\,\varphi \wedge \neg\exists j > i : \sigma_{j..} \models \mathbf{F}\,\varphi\}$

$$\langle def.4\rangle$$

$\mathcal{C}^\top_\varphi$

Case where $\mathcal{L}^\top_{\mathbf{F}\,\varphi} = 0$: Follows trivially from the fact that in this case, the input sequence has no prefix τ such that $\tau \models \mathbf{F}\,\varphi$. □

Theorem 8. $\mathcal{L}^\bot_{\mathbf{F}\,\varphi} \equiv \mathcal{C}^\top_\top$

Proof. Case where $\mathcal{L}^\bot_{\mathbf{G}\,\varphi} \neq 0$:

$\mathcal{L}^\top_{\mathbf{G}\,\varphi}$ $\langle def.7\rangle$

$\{i | i < |\sigma| : \sigma_{i..} \not\models \mathbf{G}\,\varphi \wedge \neg\exists j > i : \sigma_{j..} \not\models \mathbf{G}\,\varphi\}$

$$\langle \sigma \models \mathbf{G}\,\varphi = \bot \Rightarrow \forall\tau : \sigma \preceq \tau : \tau \not\models \mathbf{G}\,\varphi\rangle$$

$\{i | i < |\sigma| : \sigma_{i..} \not\models \mathbf{G}\,\varphi \wedge \nexists k > i : \sigma_{i..} \not\models \mathbf{G}\,\varphi \wedge \neg\exists j > i : \sigma_{j..} \not\models \mathbf{G}\,\varphi\}$

$$\langle |i = |\sigma|| \rangle$$

$\mathcal{C}^\top_\top$

Case where $\mathcal{L}^\bot_{\mathbf{G}\,\varphi} = 0$: Follows trivially from the fact that in this case, the input sequence has no prefix τ such that $\tau \models \mathbf{G}\,\varphi$. □