# An Algorithmic Approach to Bounding the Time Complexity of Stream Specifications*

David M. Cerna and Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
{David.Cerna,Wolfgang.Schreiner}@risc.jku.at

**Abstract**

In previous work an algorithmic procedure for analysing the space complexity of monitor specifications, written in a fragment of predicate logic, was presented. These monitor specifications were developed for the runtime monitoring of event streams. The algorithm provides accurate results for a large fragment of the possible specifications and can be easily extended to all specifications. In this work we follow the same approach, using the same theoretical foundations, to develop a algorithm computing the time complexity of our monitor specifications. In our previous work, the measure for space complexity was the monitor's *runtime representation size*, i.e. the number of instances in memory during checking of the specification. In this work we consider the number of *quantifier variable value assignments* as the measurement of time complexity. The result is an algorithm which gives precise results for the entire core specification language.

## 1 Introduction

In this work we continue previous investigations into the runtime properties of the LogicGuard stream monitor specification language [17]. This language and its implementation were developed in an industrial collaboration for the runtime monitoring of networks for security violations [14]. LogicGuard was developed as an alternative to linear temporal logic (LTL) [15]. Many LTL based specification languages trade expressiveness for efficiency, thus making it difficult to formulate more complex properties of interest. The expressive power of the LogicGuard language is a result of it encompassing a larger fragment of predicate logic than LTL, in particular, it supports value computation and the construction of internal streams. Of course, inclusion of these structures may lead specifications that may be costly to monitor. However, a better understanding of the dynamics of runtime monitoring can help identify expressive fragments of the language which can be monitored effectively. While in previous work we have analysed the space complexity of the monitoring specification in a core of the LogicGuard language, in this paper we investigate its time complexity.

*The project "LogicGuard II: The Optimized Checking of Time-Quantified Logic Formulas with Applications in Computer Security" is sponsored by the FFG BRIDGE program, project No. 846003.

The LogicGuard core language has much in common with Monadic First-Order Logic (MFO) [16]. LTL captures the class of star-free languages; its formulas can be translated into MFO formulas. The full language on the other hand is more closely related to Monadic Second-Order Logic (MSO) [1] which captures the class of omega-regular languages. Most space complexity results with respect to MFO and MSO use as a measure the size of the non-deterministic Büchi automaton that accepts the language of a formula. For MFO this size is in the worst case exponential in the formula size [18]; for MSO, it is in general even non-elementary [10]. Since such an automaton processes each input symbol in a single step (state transition), time complexity is always constant. However, in our work we do not consider the translation of formulas to automata, nor use automata sizes or paths as a space/time complexity measures.

So far investigations into the runtime properties of the LogicGuard language have focused on monitors which "look into the past" and monitors which "look into the future". Monitors which look into the past require the stream history to be preserved. In [13], A static analysis was developed to determine whether a given monitor can be evaluated using a finite portion of the stream history. This lead to the "history pruning" optimization that enabled effective monitoring. Concerning monitors which "look into the future", we investigated the number of formula instances which have to be preserved in memory because their truth values cannot be determined from the available portion of the stream. Previously published results provided both analytic formulas and algorithms for core language specifications [4, 5]. It is possible to use these results for the full specification language as well. A syntactic transformation simplifying a given specification, the *dominating monitor transformation*, was used to obtain these results. In [2], we analyse the effect of this transformation on the space complexity bounds provided by our algorithm. All our space complexity results are based on the interpretation of space complexity as *runtime representation size*.

In this work we investigate the time complexity of the core specification language where we define time complexity as the number of values assigned to quantified variables per step. This choice was made because it is fundamental and also most time consuming during the evaluation of a monitor specification resulting in new instances of the formulas being monitored. It is essentially the number of instances produced per step. At first glance, it seems to be directly related to runtime representation size, but in fact it only counts newly produced instances and not instances previously generated and kept in memory. In our investigations, we follow the same methodology as our work concerning space complexity [5], that is we develop an algorithm that computes the maximum number of assignments performed per evaluation step. However, unlike our space complexity results we use fewer transformations and our algorithm is precise for every core language specification.

LogicGuard shares some characteristics with two other systems, LOLA [8], for monitoring of synchronous systems, and LARVA [7] for real-time monitoring of Java programs. One of the main advantages of LOLA is independence of stream history, which is exactly the property shown in the history analysis of LogicGuard; however, LogicGuard allows the users to run monitors without history bounds. Time analysis within these systems is straight forward. For example, time complexity of LOLA is a function of the space complexity. Thus, not much time has been put into this type of analysis. However, for LogicGuard the relationship between time and space is less straight forward.

The rest of this paper is structured as follows: in Section 2, we present the core of the LogicGuard specification language, sketch its operational semantics, and provide the abstraction introduced in [5]. In Section 3, we define the time complexity of monitors and describe the algorithm that represents the core of the analysis. In Section 4, We discuss the relationship

between the results of the algorithm and the operational semantics as well as the relationship between space and time complexity. In Section 5, we conclude by outlining a few open problems which we would like to address in future work.

# 2 Basic Notions

The LogicGuard language [17] for monitoring event streams allows the derivation of a higher level stream (representing e.g. a sequence of messages transmitted by the datagrams) from a lower level input stream (representing e.g. a sequence of TCP/IP datagrams). These streams are processed by a monitor for specific properties (e.g. that every message is within a certain time bound followed by another message whose value is related in a particular way to the value of the first one). The following is an example specification:

```
type tcp; type message; ...
stream<tcp> IP;
stream<message> S = stream<IP> x satisfying start(@x) :
  value[seq,@x,combine]<IP> y
      with x < _ satisfying same(@x,@y) until end(@y) :
    @y ;
monitor<S> M = monitor<S> x satisfying trigger(@x) :
  exists<S> y with x < _ <=# x+T:
    match(@x,@y);
```

After the declaration of types `tcp` and `message` and external functions and predicates operating on objects of these types, a stream `IP` of TCP/IP datagrams is declared that is connected by the runtime system to the network interface. From this stream, a "virtual" stream `S` of "messages" is derived; each message is created by sequentially combining every datagram at position `x` on `IP` (whose value is denoted by `@x`) that satisfies a predicate `start` by application of a function `combine` with every subsequent datagram at position `y` that is related to the first one by a predicate `same` until a termination condition `end` is satisfied. The stream `S` is monitored by a monitor `M` that checks whether for every message on `S` that satisfies a `trigger` predicate within `T` time, a partner message appears that fits with the first message according to some `match` predicate.

## 2.1 Core Language

A Formal analysis is difficult to carry out for the full language. A core version of the LogicGuard language [13] was defined and given a formal operational semantics; it is this core version of the language that is addressed in this paper.

In the remainder of this section, we introduce this core language, partially relying on material from [6]. Its syntax is depicted to the left of Fig. 1 where the typed variables $M, F, \ldots$ denote elements of the syntactic domains $\mathbb{M}, \mathbb{F}, \ldots$ of monitors, formulas, etc. A monitor $M$ has form $\forall_{0 \leq V} : F$ for some variable $V$ and formula $F$; it processes an infinite stream of truth values $\top$ (true) or $\bot$ (false) by evaluating $F$ for $V = 0, V = 1, \ldots$ The predicate $@V$ denotes the value in the stream at position $V$, $\neg F$ denotes the negation of $F$, $F_1 \ \wedge \ F_2$ denotes parallel conjunction (both $F_1$ and $F_2$ are evaluated simultaneously), $F_1$ & $F_2$ denotes sequential conjunction (evaluation of $F_2$ is delayed until the value of $F_1$ becomes available), $\forall_{V \in [B_1, B_2]} : F$ denotes quantification over the interval $[B_1, B_2]$.

$$
\begin{array}{rcl}
M ::= & \forall_{0\leq V} : F. \\
F ::= & @V \mid \neg F \mid F \ \wedge \ F \mid F \ \& \ F \\
\mid & \forall_{V\in[B,B]} : F. \\
B ::= & 0 \mid \infty \mid V \mid B \pm N. \\
V ::= & x \mid y \mid z \mid \ \ldots \\
N ::= & 0 \mid 1 \mid 2 \mid \ \ldots
\end{array}
\qquad
\begin{array}{rcl}
m ::= & \forall^{\mathbb{P}(\mathbb{N}\times f\times c)}_{0\leq V} : f \\
f ::= & \mathsf{d}(\top) \mid \mathsf{d}(\bot) \mid \mathsf{n}(g) \\
g ::= & @V \mid \neg f \mid f \ \wedge \ f \mid f \ \& \ f \\
\mid & \forall_{V\in[b,b]} : f \mid \forall_{V\in[\mathbb{N}^\infty,\mathbb{N}^\infty]} : f \\
\mid & \forall^{\mathbb{P}(\mathbb{N}\times f\times c)}_{V\leq\mathbb{N}^\infty} : f \\
b ::= & c \to \mathbb{N}^\infty \\
c ::= & (V \to^{\text{part.}} \mathbb{N}) \times (V \to^{\text{part.}} \{\top, \bot\})
\end{array}
$$

$$
\begin{array}{rcl}
T(\forall_{0\leq V} : F) & := & \forall^{\emptyset}_{0\leq V} : T^{\mathrm{F}}(F) \\
T^{\mathrm{F}}(@V) & := & \mathbf{n}(@V) \\
T^{\mathrm{F}}(\neg F) & := & \mathbf{n}(\neg T^{\mathrm{F}}(F)) \\
T^{\mathrm{F}}(F_1 \ \wedge \ F_2) & := & \mathbf{n}(T^{\mathrm{F}}(F_1) \ \wedge \ T^{\mathrm{F}}(F_2)) \\
T^{\mathrm{F}}(F_1 \ \& \ F_2) & := & \mathbf{n}(T^{\mathrm{F}}(F_1) \ \& \ T^{\mathrm{F}}(F_2)) \\
T^{\mathrm{F}}(\forall_{V\in[B_1,B_2]} : F) & := & \forall_{V\in[T^B(B_1),T^B(B_2)]} : T^{\mathrm{F}}(F)
\end{array}
\qquad
\begin{array}{rcl}
T^{\mathrm{B}}(0) & := & \lambda c.\ 0 \\
T^{\mathrm{B}}(\infty) & := & \lambda c.\ \infty \\
T^{\mathrm{B}}(V) & := & \lambda c.\ c.1(V) \\
T^{\mathrm{B}}(B \pm N) & := & \lambda c.\ T^{\mathrm{B}}(B)(c) \pm N
\end{array}
$$

Figure 1: The core language: syntax, runtime representation, translation.

**Example 1.** *The following monitor $M$ is an example of a core language specification:*

$$\forall_{0\leq x} : \forall_{y\in[x+1,x+5]} : ((\forall_{z\in[y,x+3]} : \neg @z \ \& \ @z) \ \& \ G(x,y))$$

$$G(x,y) = \forall_{w\in[x+2,y+2]} : (\neg @y \ \& \ (\forall_{m\in[y,w]} : \neg @x \ \& \ @m))$$

Monitor $M \in \mathbb{M}$ is translated by the function $T : \mathbb{M} \to \mathcal{M}$ defined at the bottom of Fig. 1 into its runtime representation $m = T(M) \in \mathcal{M}$ whose structure is depicted to the right; here the typed variables $m, f, \ldots$ denote elements of the runtime domains $\mathcal{M}, \mathcal{F}, \ldots$, i.e., the runtime representations of $M, F, \ldots$ Over the domain $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ arithmetic operations are interpreted in the usual way, i.e., the operator $-$ is interpreted as truncated subtraction and for every $n \in \mathbb{N}$ we have $\infty \pm n = \infty$. The notions $\mathbb{P}(S)$, for some set $S$, and $A \to^{\text{part.}} B$ denote the powerset of $S$ and the set of partial mappings from $A$ to $B$, respectively. A context $c$ consists of a pair of partial functions that assign to every variable its position and the truth value that the stream holds at that position, respectively.

## 2.2 Operational Semantics

During the execution of monitor $M$, its runtime representation $m = \forall^I_{0\leq V} : f$ holds in set $I$ those instances of its body $F$ which could not yet be evaluated to $\top$ or $\bot$; each such instance is represented by a tuple $\langle p, f, c\rangle$ where $p$ is the position assigned to $V$, $f$ is the (current) runtime representation of $F$, and $c$ represents the context to be used for the evaluation of $f$. A runtime representation $f$ can be a tagged value $\mathbf{n}(g)$ where $g$ represents the runtime representation of the formula to be evaluated in the **n**ext step; when the evaluation has completed, its value becomes $\mathbf{d}(t)$ where the truth value $t$ represents the result when evaluation is **d**one.

The evaluation of a monitor's runtime representation is formally defined by a small-step operational semantics with a 7-ary transition relation $m \to_{p,s,v,R,t} m'$ where $m$ is the runtime representation of the monitor prior to the transition, $m'$ is its representation after the transition, $p$ is the stream position of the next message value $v$ to be processed, $s$ denotes the sequence

of $p$ messages that have previously been processed, $R$ denotes the set of those positions which are reported by the transition to make the monitor body false, and $t$ is the count of variable assignments for the current transition. The monitor thus processes a stream $\langle v_0, v_1, \ldots \rangle$ by a sequence of transitions

$$\left(\forall^{I_0}_{0\leq x} : f\right) \to_{0,s_0,v_0,R_0,t_0} \left(\forall^{I_1}_{0\leq x} : f\right) \to_{1,s_1,v_1,R_1,t_1} \left(\forall^{I_2}_{0\leq x} : f\right) \to \cdots$$

where $s_p = \langle v_0, \ldots, v_{p-1} \rangle$. Each set $I_p$ contains those instances of the monitor which, by the $p$ messages processed so far, could not be evaluated to a truth value yet and each set $R_p$ contains the positions of those instances that were reported to become false by transition $p$. The value of $t_p$ is determined by computing the number of instances created by each member of $I_p$. In particular, we have

$$\begin{aligned} I_{p+1} &= \{(t, \mathbf{n}(g), c) \in \mathcal{I} \mid \exists f \in \mathcal{F} : (t, f, c) \in I' \wedge \vdash f \to_{p,s_p,v_p,c,t} \mathbf{n}(g)\} \\ R_{p+1} &= \{t \in \mathbb{N} \mid \exists f \in \mathcal{F}, c \in \mathcal{C} : (t, f, c) \in I' \wedge \vdash f \to_{p,s_p,v_p,c,t} \mathbf{d}(\bot)\} \\ t_p &= \sum_{\substack{(t,f,c) \in I' \\ f \to_{p,s_p,v_p,c,i} \mathbf{n}(g)}} i \end{aligned}$$

where $I' = I_p \cup \{(p, f, ((V, p), (V, v_p)))\}$. The transition relation on monitors depends on a corresponding transition relation $f \to_{p,s,v,c,t} f'$ on formulas where $c$ represents the context for the evaluation of $f$ and $t$ is a counter for the number of instances generated this monitor step by the formula $f$. In each step $p$ of the monitor transition, a new instance of the monitor body $F$ is added to set $I_p$, and all instances in that set are evaluated according to the formula transition relation. Note that each formula instance in that set contains the runtime representation of a quantified formula (otherwise, it could have been immediately evaluated) which in turn contains its own instance set; thus instance sets are nested up to a depth that corresponds to the quantification depth of the monitor.

Fig. 2 shows an excerpt of the operational semantics of formula evaluation (the full semantics is given in [13]) where the transition arrow $\to_t$ is to be read as $\to_{p,s,v,c,t}$ and rules Q4–Q7 are based on the following definitions.

$$\begin{aligned} I_0 &= \{(i, f, (c.1[V \mapsto i], c.2[V \mapsto s(i + p - |s|)])) \mid p_1 \leq i \leq \min\{p_2 + 1, p\}\} \\ I' &= \begin{cases} I & \text{if } p_2 < p \\ I \cup (p, f, (c.1[V \mapsto p], c.2[V \mapsto v])) & \text{otherwise} \end{cases} \\ I'' &= \{(t, \mathbf{n}(g), c) \in \mathcal{I}' \mid (t, f, c) \in I' \wedge \vdash f \to \mathbf{n}(g)\} \\ DF &\equiv \exists t \in \mathbb{N}, f \in \mathcal{F}, c \in \mathcal{C} : (t, f, c) \in I' \wedge \vdash f \to \mathbf{d}(\bot) \\ t'(X) &= \sum_{\substack{(t,f,c) \in X \\ f \to_i \mathbf{n}(g)}} i \end{aligned}$$

We provide an example adapted from [6] concerning the application of these rules.

**Example 2.** *We take the monitor* $M = \forall_{0\leq x} : \forall_{y\in[x+1,x+2]} : \mathbf{@}x \;\&\; \mathbf{@}y$*, which states that the current position of the stream is true as well as the next two future positions. We determine its runtime representation* $m = T(M)$ *as* $m = \forall^{\emptyset}_{0\leq x} : f$ *with* $f = \forall_{y\in[b_1,b_2]} : g$ *for some* $b_1$ *and* $b_2$

| Atomic Formulas | | |
|---|---|---|
| # | Transition | Constraints |
| A1 | $\mathbf{n}(@V) \rightarrow_0 \mathbf{d}(c.2(V)))$ | $V \in dom(c.2)$ |
| ... | | |
| Sequential conjunction | | |
| C1 | $\mathbf{n}(f_1 \ \& \ f_2) \rightarrow_t \mathbf{n}(\mathbf{n}(f_1') \ \& \ f_2)$ | $f_1 \rightarrow_t \mathbf{n}(f_1')$ |
| C2 | $\mathbf{n}(f_1 \ \& \ f_2) \rightarrow_t \mathbf{d}(\bot)$ | $f_1 \rightarrow_t \mathbf{d}(\bot)$ |
| C3 | $\mathbf{n}(f_1 \ \& \ f_2) \rightarrow_{t_1+t_2} \mathbf{n}(f_2')$ | $f_1 \rightarrow_{t_1} \mathbf{d}(\top)$, $f_2 \rightarrow_{t_2} \mathbf{n}(f_2')$ |
| Quantification | | |
| Q1 | $\forall_{V\in[b_1,b_2]} : f \rightarrow_0 \mathbf{d}(\top)$ | $p_1 = b_1(c)$ , $p_2 = b_2(c)$ , $p_1 = \infty \vee p_1 > p_2$ |
| Q2 | $\forall_{V\in[b_1,b_2]} : f \rightarrow_t f'$ | $p_1 = b_1(c)$ , $p_2 = b_2(c), p_1 \neq \infty$ , $p_1 \leq p_2$, $\mathbf{n}(\forall_{V\in[p_1,p_2]} : f) \rightarrow_t f'$ |
| Q3 | $\mathbf{n}(\forall_{V\in[p_1,p_2]} : f) \rightarrow_0 \mathbf{n}(\forall_{V\in[p_1,p_2]} : f)$ | $p < p_1$ |
| Q4 | $\mathbf{n}(\forall_{V\in[p_1,p_2]} : f) \rightarrow_{\lvert I_0\rvert+t'(I_0)} f'$ | $p_1 \leq p$, $\mathbf{n}(\forall^{I_0}_{V\leq p_2} : f) \rightarrow_{\lvert I_0\rvert+t'(I_0)} f'$ |
| Q5 | $\mathbf{n}(\forall^{I}_{V\leq p_2} : f) \rightarrow_{\lvert I'\setminus I\rvert+t'(I')} \mathbf{d}(\bot)$ | $DF$ |
| Q6 | $\mathbf{n}(\forall^{I}_{V\leq p_2} : f) \rightarrow_{\lvert I'\setminus I\rvert+t'(I')} \mathbf{d}(\top)$ | $\neg DF, I'' = \emptyset, p_2 < p$ |
| Q7 | $\mathbf{n}(\forall^{I}_{V\leq p_2} : f) \rightarrow_{\lvert I'\setminus I\rvert+t'(I')} \mathbf{n}(\forall^{I''}_{V\leq p_2} : f)$ | $\neg DF, (I'' \neq \emptyset \vee p \leq p_2)$ |

Figure 2: The operational semantics of formula evaluation.

*and* $g = @x \ \& \ @y$. *We evaluate* $m$ *over the stream* $\langle \top, \top, \bot, \ldots \rangle$. *First consider the transition* $(\forall^{\emptyset}_{0\leq x} : f) \rightarrow_{0,\langle\rangle,\top,\emptyset,1} (\forall^{I_0}_{0\leq x} : f)$. *which generates the instance set*

$$I^0 = \left\{(0, \mathbf{n}(\forall_{y\in[1,2]} : g), (\{(x,0)\}, \{(x,\top)\}))\right\}.$$

*Performing another step* $(\forall^{I_0}_{0\leq x} : f) \rightarrow_{1,\langle\top\rangle,\top,\emptyset,1} (\forall^{I_1}_{0\leq x} : f)$ *we get*

$$I^1 = \begin{array}{l}\big\{(1, \mathbf{n}(\forall_{y\in[2,3]} : g), (\{(x,1)\}, \{(x,\top)\})),\\ \ (0, \mathbf{n}(\forall^{\emptyset}_{y\leq 2} : g), (\{(x,0)\}, \{(x,\top)\}))\big\}\end{array}.$$

*The instance set* $\emptyset$ *in the runtime representation of the formula is empty, because the body of the quantified formula is propositional and evaluates instantly. Notice that the new instance is the same as the instance in* $I^0$ *but the positions are shifted by* 1. *The next step is* $(\forall^{I_1}_{0\leq x} : f) \rightarrow_{2,\langle\top,\top\rangle,\bot,\{0,1\},1} (\forall^{I_2}_{0\leq x} : f)$ *where*

$$I^2 = \left\{(2, \mathbf{n}(\forall_{y\in[3,4]} : g), (\{(x,2)\}, \{(x,\bot)\}))\right\}.$$

*The first two instances evaluate at this point and both violate the specification, thus yielding the set* $\{0, 1\}$ *of violating positions of the monitor. Again, the remaining instance is shifted by one position.*

## 2.3 Quantifier Trees

The algorithm introduced in [5] as well as the algorithm presented in the next section, and the analytic expressions of [3, 4] are based on a translation of monitor formulas into an abstract representation that we call *quantifier trees*. In the following, we describe the core of this concept relevant for this paper; we will focus on the important concepts which directly apply to this paper and we refer the reader to [3, 4] for further details.

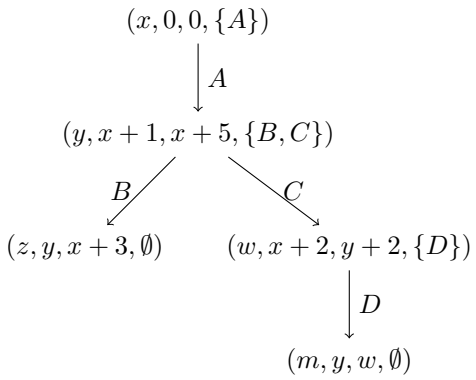


Figure 3: Quantifier trees for the monitor of Ex. 1.

**Definition 1** (Quantifier Trees)**.** *A* quantifier tree *is inductively defined to be either* $\emptyset$ *or a tuple of the form* $(y, b_1, b_2, Q)$ *where* $y \in V$*,* $b_1, b_2 \in B$*, and* $Q$ *is a set of quantifier trees. Let* $\mathbb{QT}$ *be the set of all quantifier trees. We define* $(y, b_1, b_2, Q)_1 = y$*,* $(y, b_1, b_2, Q)_2 = b_1$*,* $(y, b_1, b_2, Q)_3 = b_2$*, and* $(y, b_1, b_2, Q)_4 = Q$*.*

**Definition 2** (Quantifier Tree Transformation)**.** *We define the* quantifier tree transformation $QT : \mathbb{M} \to \mathbb{QT}$*, respectively* $QT : \mathbb{F} \to \mathbb{QT}$*, recursively as follows:*

$$\begin{array}{llll} QT(\forall_{0\leq V} : F) & = (V, 0, 0, QT(F)) & QT(F \& G) & = QT(F) \cup QT(G) \\ QT(F \wedge G) & = QT(F) \cup QT(G) & QT(\neg F) & = QT(F) \\ QT(\forall_{V\in[B_1,B_2]} : F) & = (V, B_1, B_2, QT(F)) & QT(@V) & = \emptyset \end{array}$$

We do not distinguish between sequential and parallel conjunction in quantifier trees because their effect on memory is dependent on the stream they are evaluated over, thus, we just take the worst case scenario of parallel conjunction. Fig. 3 depicts the quantifier tree $QT(M)$ where $M$ is the monitor from Ex. 1.

**Definition 3** (Sub-quantifier Tree)**.** *Let* $q, q' \in \mathbb{QT}$*. We call* $q'$ *a* sub-quantifier tree *of* $q$ *if* $q = (y, b_1, b_2, Q)$ *and one of the following holds:* $q' = q$*, or* $q' \in Q$*, or there exists* $q'' \in Q$ *such that* $q'$ *is a* sub-quantifier tree *of* $q''$*. We call a sub-quantifier tree* $q'$ *of* $q$ proper *if* $q' \neq q$*. We will use the notation* $q.\nu$ *to denote a sub-quantifier tree* $\nu$ *of* $q$*.*

**Definition 4** (Maximum Depth)**.** *Let* $q \in \mathbb{QT}$*. We define* $d(q)$ *as follows:*

$$d(q) = \begin{cases} 1 + max_{q'\in(q)_4}\{d(q')\} & (q)_4 \neq \emptyset \\ 0 & otherwise \end{cases}$$

For dealing with nested variables, i.e. variables whose values depend on the value of another variable, we introduce substitution of values into variable bounds.

**Definition 5.** *A term* $t$ *is an element of* $B \setminus \{\infty\}$ *(see Fig. 1). We define* $T = B \setminus \{\infty\}$ *as the set of all terms.*

Sometimes we will write a term as $t(y)$, meaning that $y$ may be free in $t$.

**Definition 6** (Quantifier Tree substitution)**.** *Let* $q \in \mathbb{QT}$, $a, c_1, c_2 \in \mathbb{Z}$, $x_1, \cdots x_n, y_1, y_2, y \in \mathbb{V}$, *and* $b_1, b_2, t_1, \cdots t_n \in T$, *where* $q = (y, b_1(y_1), b_2(y_2), Q)$. *We define* $q\sigma$, *where* $\sigma = \{x_1 \leftarrow t_1, \cdots, x_n \leftarrow t_n\}$, *as follows:*

$$(y, b_1(y_1), b_2(y_2), Q)\sigma = (y, b_1(y_1)\sigma, b_2(y_2)\sigma, \bigcup_{q \in Q'} q\sigma),$$

*where for* $j = 1, 2$ *we have:*

$$b_j(y_j)\sigma = \begin{cases} b_j(t_i) & \text{if } x_i = y_j, \text{for some } 1 \leq i \leq n \\ b_j(y_j) & \text{otherwise} \end{cases}$$

Essentially, we are using a standard variable substitution method. A substitution $\sigma = \{x_1 \leftarrow t_1, \cdots, x_n \leftarrow t_n\}$ can also be written as $\sigma'\{x_n \leftarrow t_n\}$ where $\sigma' = \{x_1 \leftarrow t_1, \cdots, x_{n-1} \leftarrow t_{n-1}\}$. We assume all quantifier trees we will be dealing with result from the translation of closed formula with a single free variable $x$ (the monitor variable); furthermore we assume they are *proper*, that is they have a root node $Q = (x, 0, 0, Q')$, and for every $Q.\nu$ and $Q.\mu$ it is the case that $Q.\nu \neq Q.\mu$ $(Q.\nu)_1 \neq (Q.\mu)_1$. These assumptions allow us to define an initial set for our evaluation method.

**Definition 7.** *Let* $Q \in \mathbb{QT}$ *be proper and let* $a \in \mathbb{N} \cup \{0\}$. *We define the* initial instance set $I(a, Q)$ *of* $Q$ *at* $a$, *as* $I(a, Q) = \bigcup_{q \in Q'} q\{x \leftarrow a\}$.

# 3 Analysis

In previous work [5] we developed an algorithmic method for computing the space complexity of monitoring specifications. This method computes the size of the runtime representation of a monitor specification which is roughly the number of unresolved quantifier instances in memory. This measurement only accounts for part of the overhead of monitor evaluation. As messages arrive from the external streams there are a number of computation steps which need to be performed in order to compute the next instance set. The most expensive of these operations is the assignment of values to quantified variables because these operations result in the new instances.

Note that there isn't a one to one correspondence between runtime representation size and quantified variable assignment even though they both, theoretically, count instances. The runtime representation size only counts unresolved instances and quantified variable assignment counts all generated instances. To capture this difference we added a counter to the the monitor transition relation of the operational semantics (see Subsection 2.2). The value of this counter is computed fresh at each step. Using this counter we can define a relation similar to Definition 2 of [5] computing the maximum number of quantified variable assignments up to position $n$.

**Definition 8.** *We define the relation* $\twoheadrightarrow \subseteq \mathcal{M} \times \mathbb{N} \times \{\top, \bot\}^* \times \mathbb{N} \times \mathbb{N}$ *inductively as follows:*

$$\begin{aligned} &M \twoheadrightarrow_{p,s,0} t \leftrightarrow \exists R.\ (M \rightarrow_{p,s,s(p),R,t} M') \\ &M \twoheadrightarrow_{p,s,(n+1)} t'' \leftrightarrow \\ &\quad \exists R.\ (M \rightarrow_{p,s,s(p),R,t} M') \wedge (M' \twoheadrightarrow_{p+1,s,n} t') \wedge t'' = \max\{t, t'\} \end{aligned}$$

Since $\rightarrow_t$ is deterministic $M \twoheadrightarrow_{p,s,n} t$ uniquely determines $t$ from $M, p, s$, and $n$. Essentially, $M \twoheadrightarrow_{p,s,n} t$ states that $t$ is the maximum number of value assignments to the quantified variables of monitor $m$ during the execution of $n$ transitions over the stream $s$ starting at position $p$.

**Algorithm 1** Time complexity of a Quantifier Tree

```
1:
2: function TC(qt,p)                                   ▷ qt is a quantifier tree (y, a, b, Q)
3:     ret ← 0
4:     if a > b then
5:         return 0
6:     else if p < a then
7:         return TC(qt, a)
8:     else
9:         if Q ≠ ∅ then
10:            for all qt' ∈ Q do
11:                for all a ≤ i ≤ min{b, p} do
12:                    ret ← ret + TC(qt'{y ← i}, p) + 1
13:                end for
14:            end for
15:        else
16:            ret ← |1 + min{p, b} − a|
17:        end if
18:        if b ≤ p then
19:            return ret
20:        else
21:            return ret + TC((y, p + 1, b, Q), p + 1)
22:        end if
23:    end if
24: end function
```

We algorithmically compute the value of $t$ without having to actually perform the transitions. The procedure is similar to the one presented in [5]. Later we provide Theorem 1 formalizing the relationship between our analysis and Definition 8. In a nutshell, this analysis proceeds as follows:

1. We translate $M \in \mathbb{M}$ into a *quantifier tree* $qt = QT(M)$ which contains the essential information required for the analysis.
2. We compute $I(0, qt)$.
3. We compute $\sum_{q \in I(0,qt)} TC(q, 0)$ which is abbreviated as $|M|_{tc}$ by application of Algorithm 1.

In the following section we state a proof of correctness, but for now we give a short explanation of the rationale behind the algorithm. The heart of the algorithm is found in lines 9-16 of Algorithm 1. If the current formula has quantified subformulas then we proceed to the loop. Otherwise we output the number of instances in the quantifiers instance set. This comes directly, from Definition **??**. If the current formula has quantified subformulas then we call the function TC on the quantified subformulas after making the variable assignment. Line 21 goes to the next time step. On Line 21, we add the current total to the next step because we are not just computing the function of Definition **??** but the maximum number of assignments. This maximum occurs when instances at all levels of evaluation are in memory at the same time. This assumption is built into the algorithm. Of course in practice this would most likely never happen, but in the worst case it can.

Now that we have introduced a concept of time and an algorithm computing it, we can present our definition of time complexity for monitor specifications.

**Definition 9** (Time Complexity)**.** *Let $m \in \mathbb{M}$. Then the* time complexity *of $m$, i.e. $|m|_{tc}$ is defined as follows:*

$$|m|_{tc} = \sum_{q \in I(0,QT(m))} TC(q,0)$$

*where $TC(\cdot,\cdot)$ is defined in Algorithm 1.*

# 4 Correctness of Analysis

We now present a theorem formalizing the relationship between the relation of Definition 8 and Algorithm 1.

**Theorem 1.** *Let $M \in \mathbb{M}$. Then for all $n, p, S \in \mathbb{N}$ and $s \in \{\top, \bot\}^\omega$ such that $T(M) \twoheadrightarrow_{p,s,n} S$, we have $S \leq |M|_{tc}$.*

*Proof(sketch).* Let us assume for our basecase that $M \in \mathbb{M}$ is such that $|I(0, QT(m))| = 1$ and $(QT(m))_4 = \emptyset$. This would imply that $c_{\mathrm{m}}(m) = \sum_{g \in I} |I_g|$ where $|I_g|$ is the size of the instance set in a formula of the form $\forall^I_{V \leq p} : f$. Given the above constraints, These are the only type of instance in the runtime representation. Obviously, this sum is equivalent to $\sum_{i=1}^{n} i$, where $n = ((QT(m)\{x \leftarrow 0\})_3 - (QT(m)\{x \leftarrow 0\})_2)$, which is exactly the value computed by $|m|_{tc}$. For the first inductive step case we assume that the theorem holds for all monitors $M \in \mathbb{M}$ such that $|I(0, QT(m))| = n$ and for each $q \in I(0, QT(m))$, $(q)_4 = \emptyset$. We show that the theorem holds for a monitor $M \in \mathbb{M}$ such that $|I(0, QT(m))| = n + 1$ and for each $q \in I(0, QT(m))$, $(q)_4 = \emptyset$. This is essentially adding the basecase to the induction hypothesis.

To finish the proof we now assume the theorem holds for all monitors $M \in \mathbb{M}$ such that $d(QT(M)) = n$ and show that the theorem holds for a monitor $M' \in \mathbb{M}$ where $d(QT(M')) = n + 1$. Let us take two monitors $M, N \in \mathbb{M}$ such that $d(QT(M)) = n$ and $d(QT(N)) = 1$. We assume that the intersection of the quantified variable names of $M$ and $N$ is empty. We will allow $M$ to have one free variable $y$ which appears quantified in $N$. We construct $QT(M')$ by taking $q \in (QT(N))_4$ such that $q$ quantifies the variable $y$. We then construct the quantifier tree $q'$ where $(q)_i = (q')_i$ for $1 \leq i \leq 3$ and $(q)_4 = (QT(M))_4$. Thus, $(QT(M'))_i = (QT(N))_i$ for $1 \leq i \leq 3$ and $(QT(M'))_4 = (QT(N))_4 - \{q\} \cup \{q'\}$. Obviously computing $|M'|_{tc}$ requires computing $|M|_{tc}$ for every instance counted for $q$ and thus is just an extension of the basecase computing $|M|_{tc}$ at each step. This proofs the theorem. □

A theorem very similar to Theorem 1 was presented in [5] concerning space complexity. Given the discussion of Section 3 concerning the relationship between our notions of space and time complexity one would expect a precise relationship between the two methods. Sadly, the subtle differences result in very different behaviour. Applying Algorithm 1 to a typical monitor specification gives the impression that our notion of time complexity always upper bounds space complexity. However, choosing more pathological cases highlights the complexity of the relationship. For example, for the following monitor specification

$$M = \forall_{0 \leq x} : \forall_{y \in [x+5, x+5]} : \forall_{y \in [x+8, x+8]} : @x$$

the time complexity of $M$ is $|M|_{tc} = 2$, but the space complexity $SR(0, AQT(D(M))) = 8$ (Using the definitions and algorithm of [5]). These cases make it quite difficult to discuss the relationship in more concrete terms.

# 5 Conclusion

In this paper we followed the same methodology as in previous work [5] in an attempt to develop an algorithmic procedure analysing the time complexity of monitor evaluation. For space complexity [5], the size of the runtime representation of a monitor specification was considered as the measurement of space. For time complexity, we chose the number of quantified variable value assignments because it is the most expensive operation carried out during evaluation. This measure is closely related to the runtime representation size because value assignments generate instances, though, not all of those instances stay in memory. In the end, we were able to develop an algorithmic procedure, that differs from the procedure for space complexity and is in certain sense simpler, because we do not need to carry out as many transformations for time complexity as for space complexity. Even though the two procedures are related, these minor differences make it difficult to discuss their relationship. It is quite obvious that these are not the only notions of time and space we can use to define complexity; thus, in future work we plan to investigate these notions further to deepen our understanding of their relationship and potentially develop alternative formulations.

# References

[1] Julius Richard Büchi. Weak Second-Order Arithmetic and Finite Automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[2] David M. Cerna and Wolfgang Schreiner. Measuring the gap: Algorithmic approximation bounds for the space complexity of stream specifications. Sent to SCSS 2017.

[3] David M. Cerna, Wolfgang Schreiner, and Temur Kutsia. Analytic and Algorithmic Methods for Computing the Space Requirements of Stream Monitor Specifications. 2016. Submitted for publication.

[4] David M. Cerna, Wolfgang Schreiner, and Temur Kutsia. Better Space Bounds for Future-Looking Stream Monitors. 2016. Submitted for publication.

[5] David M. Cerna, Wolfgang Schreiner, and Temur Kutsia. Predicting Space Requirements for a Stream Monitor Specification Language. In *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 28-30, 2016. Proceedings*, pages 135–151. Springer International Publishing, 2016.

[6] David M. Cerna, Wolfgang Schreiner, and Temur Kutsia. Space Analysis of a Predicate Logic Fragment for the Specification of Stream Monitors. In James H. Davenport and Fadoua Ghourabi (ed.), editors, *Proceedings of The 7th International Symposium on Symbolic Computation in Software Science*, volume 39 of *EPiC Series in Computing*, pages 29–41, 2016.

[7] C. Colombo, G. J. Pace, and G. Schneider. LARVA – Safer Monitoring of Real–Time Java Programs (Tool Paper). In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 33–37, Nov 2009.

[8] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime Monitoring of Synchronous Systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, June 2005.

[9] Bernd Finkbeiner and Lars Kuhtz. Monitor Circuits for LTL with Bounded and Unbounded Future. In *Runtime Verification, 9th International Workshop, RV 2009*, volume 5779 of *Lecture Notes in Computer Science*, pages 60–75, Grenoble, France, June 26–28, 2009. Springer, Berlin.

[10] Markus Frick and Martin Grohe. The Complexity of First-Order and Monadic Second-Order Logic Revisited. *Annals of Pure and Applied Logic*, 130(1–3):3–31, 2004.

[11] IEEE Std 1850-2007: Standard for Property Specification Language (PSL)., 2007.

[12] Orna Kupferman, Yoad Lustig, and Moshe Y. Vardi. On Locally Checkable Properties. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006*, volume 5779 of *Lecture Notes in Artificial Intelligence*, pages 302–316, Phnom Penh, Cambodia, November 13–17, 2006. Springer, Berlin, Germany.

[13] Temur Kutsia and Wolfgang Schreiner. Verifying the Soundness of Resource Analysis for LogicGuard Monitors (Revised Version). Technical Report 14-08, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2014.

[14] LogicGuard II, November 2015. http://www.risc.jku.at/projects/LogicGuard2/.

[15] Oded Maler, Dejan Nickovic, and Amir Pnueli. Real Time Temporal Logic: Past, Present, Future. In Paul Pettersson and Wang Yi, editors, *Formal Modeling and Analysis of Timed Systems, Third International Conference (FORMATS)*, volume 3829 of *Lecture Notes in Computer Science*, pages 2–16, Uppsala, Sweden, September 26–28, 2005. Springer, Berlin, Germany.

[16] Robert McNaughton and Seymour Papert. *Counter-Free Automata*, volume 65 of *Research Monograph*. MIT Press, Cambridge, MA, USA, 1971.

[17] Wolfgang Schreiner, Temur Kutsia, David Cerna, Michael Krieger, Bashar Ahmad, Helmut Otto, Martin Rummerstorfer, and Thomas Gössl. The LogicGuard Stream Monitor Specification Language (Version 1.01). Tutorial and reference manual, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, November 2015.

[18] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18*, pages 332–344. IEEE Computer Society, 1986.