Predicting Space Requirements for a Stream Monitor Specification Language*

David M. Cerna, Wolfgang Schreiner, Temur Kutsia

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University (JKU), Linz, Austria

Abstract. The LogicGuard specification language for the runtime monitoring of message/event streams specifies monitors by predicate logic formulas of a certain kind. In this paper we present an algorithm that gives upper bounds for the space requirements of monitors specified in a formally elaborated core of this language. This algorithm has been implemented in the LogicGuard software and experiments have been carried out to demonstrate the accuracy of its predictions.

1 Introduction

We investigate the space complexity of the LogicGuard stream monitor specification language [14], which was developed in an industrial collaboration for the runtime monitoring of networks for security violations [11]. LogicGuard represents an alternative to the commonly used language of linear temporal logic (LTL) [12], from which efficient stream monitors can be generated but in which, due to its limited expressiveness, it can be difficult to formulate more complex properties of interest. The LogicGuard language is more expressive than LTL, because it encompasses a large fragment of predicate logic, in particular, it supports value computation and the construction of virtual streams by a form of set builder notation. However, the inclusion of such elements can make monitoring of specifications inefficient. We thus aim to identify such specifications expressing properties of interest for which monitoring is effectively possible.

First, since a LogicGuard monitor is able to "look into the past", it has in general to preserve the complete history of the stream in memory. Thus a static analysis was developed to determine whether a specification gives rise to a monitor that is able to operate with only a finite number of past messages in memory. This analysis was shown to be sound [10] and resulted in a "history pruning" optimization that enabled effective monitoring. For the soundness proof, a simplified core language with a formal operational semantics was devised.

Second, we investigated for the same core language a complementary analysis to determine the space requirements of monitors which "look into the future". In particular, we are interested in the number of formula instances which have to be preserved in memory, because their truth values cannot be determined

* Supported by the Austrian Research Promotion Agency (FFG) in the frame of the BRIDGE program 846003 "LogicGuard II".

from the observations made so far. Based on preliminary investigations we have in [3] provided upper bounds for the space complexity of monitors. The present paper improves this work by capturing these bounds more precisely, in some cases even optimally. Together with the history size, the result of this analysis bounds the memory requirements of the monitor and also the time required for processing every message/event. Also, these results seem to pinpoint a possible optimization by exchanging the order of logically independent nested quantifiers. This direction of research is based on the presented algorithm and the technical report [2].

The LogicGuard core language has much in common with Monadic First-Order Logic (MFO) [13]. 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 [15]; for MSO, it is in general even non-elementary [7]. These measures are relevant for model checkers which operate on non-deterministic automata; for runtime monitoring, the automata have to be determinized which results in another exponential blowup.

Because of this state space explosion, more restricted logics have been investigated. The hardware design language PSL [8] which is based on LTL defines a "simple subset" that restricts the use of disjunction to avoid exponential blow up. In [9], the class of "locally checkable" properties (a subclass of the "locally testable" properties introduced in [13]) is defined, where a word satisfies a property, if every k-length subword of the word does; such properties can be recognized by deterministic automata whose size is exponential in k but independent of the formula size. In [6] a procedure for synthesizing monitor circuits from LTL specifications is defined that restricts the exponential blow-up to those parts of a formula that involve unbounded-future operators.

LogicGuard shares some characteristics with two other systems, LOLA [5], for monitoring of synchronous systems, and LARVA [4] 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. Our work in this paper is tangent to this analysis, we focus on the memory usage of evaluation given independence of the monitor specification from stream history size. Concerning LARVA, which gives a more complete picture of resource requirements, our work in this paper plus the history analysis aims at providing a similar complete picture of monitor execution. Though, we do not provide time bounds, such bounds are closely related to the runtime representation size. More concerning this issue is discussed in the conclusion. LogicGuard can be thought of as similar to the above software, but more of an application specific monitoring specification language, that is a language for monitoring complex safety and security properties.

In our work we do not consider the translation of formulas to automata, nor use automata sizes as a space complexity measure. The operational form of a LogicGuard monitor is not an automaton but a structure that keeps in memory a (nested) set of formula instances that dynamically grows and shrinks during by the evaluation of the monitor on the stream. Thus we have investigated in [3] the number of instances kept in memory by abstracting the operational semantics into a rewriting system that is applied recursively to the formula structure. This allowed for complexity results not only in terms of asymptotic bounds but also in terms of concrete complexity functions. However, the method suffered from severe overestimation; in the present paper, we devise an analysis that provides much more accurate results. This analysis was also implemented in the LogicGuard software to estimate the space requirements of real monitors.

The rest of this paper is structured as follows: in Section 2, we present the core of the LogicGuard specification language and sketch its operational semantics. In Section 3, we define the space complexity of monitors and describe the algorithm that represents the core of the analysis. In Sections 4 and 5, we present the formal details; in Section 6, we discuss experimental results. In Section 7, we conclude by outlining a few open problems which we would like to address in future work.

2 Core Language

The LogicGuard language [14] for monitoring event streams allows, for example, 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). Such a stream is processed by a monitor for a particular property (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). A specification of this kind has the following form:

```
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

$$
\begin{aligned}
M &::= \forall_{0 \le V} \colon F. \\
F &::= @V \mid \neg F \mid F \wedge F \mid F \,\&\, F \\
&\quad\mid \forall_{V \in [B,B]} \colon F. \\
B &::= 0 \mid \infty \mid V \mid \mid B \pm N. \\
V &::= x \mid y \mid z \mid \ \ldots \\
N &::= 0 \mid 1 \mid 2 \mid \ \ldots
\end{aligned}
$$

$$
\begin{aligned}
m &::= \forall_{0 \le V}^{\mathbb{P}(\mathbb{N} \times f \times c)} \colon f \\
f &::= \mathsf{d}(\top) \mid \mathsf{d}(\bot) \mid \mathsf{n}(g) \\
g &::= @V \mid \neg f \mid f \wedge f \mid f \,\&\, f \\
&\quad\mid \forall_{V \in [b,b]} \colon f \mid \forall_{V \in [\mathbb{N}^\infty, \mathbb{N}^\infty]} \colon f \\
&\quad\mid \forall_{V \le \mathbb{N}^\infty}^{\mathbb{P}(\mathbb{N} \times f \times c)} \colon f \\
b &::= c \to \mathbb{N}^\infty \\
c &::= (V \to^{\text{part.}} \mathbb{N}) \times (V \to^{\text{part.}} \{\top, \bot\})
\end{aligned}
$$

$$
\begin{aligned}
T(\forall_{0 \le V} \colon F) &:= \forall_{0 \le V}^{\emptyset} \colon T^{\mathrm{F}}(F) \\
T^{\mathrm{F}}(@V) &:= \mathsf{n}(@V) \\
T^{\mathrm{F}}(\neg F) &:= \mathsf{n}(\neg T^{\mathrm{F}}(F)) \\
T^{\mathrm{F}}(F_1 \wedge F_2) &:= \mathsf{n}(T^{\mathrm{F}}(F_1) \wedge T^{\mathrm{F}}(F_2)) \\
T^{\mathrm{F}}(F_1 \,\&\, F_2) &:= \mathsf{n}(T^{\mathrm{F}}(F_1) \,\&\, T^{\mathrm{F}}(F_2)) \\
T^{\mathrm{F}}(\forall_{V \in [B_1, B_2]} \colon F) &:= \forall_{V \in \left[T^B(B_1), T^B(B_2)\right]} \colon T^{\mathrm{F}}(F)
\end{aligned}
$$

$$
\begin{aligned}
T^{\mathrm{B}}(0) &:= \lambda c.\ 0 \\
T^{\mathrm{B}}(\infty) &:= \lambda c.\ \infty \\
T^{\mathrm{B}}(V) &:= \lambda c.\ c.1(V) \\
T^{\mathrm{B}}(B \pm N) &:= \lambda c.\ T^{\mathrm{B}}(B)(c) \pm N
\end{aligned}
$$

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

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.

To support a formal analysis, in [10] a core version of the LogicGuard language was defined and given a formal operational semantics. This core language has been subsequently used to analyze the complexity of monitoring and to derive the results presented in this paper. The analysis was also implemented in the LogicGuard system by translating specifications from the full language to the core language such that the analysis of the translated specification also predicts the complexity of monitoring the original specification (the translation is not semantics-preserving but generates a specification for which monitoring is at least as complex as the monitoring of the original one).

In the remainder of this section, we introduce this core language, partially relying on material from [3]. 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 $\forall_{0 \le V} \colon F$ 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), $\forall_{V \in [B_1, B_2]} \colon F$ denotes quantification over the interval $[B_1, B_2]$.

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.

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 $\mathsf{n}(g)$ where g represents the runtime representation of the formula to be evaluated in the next step; when the evaluation has completed, its value becomes $\mathsf{d}(t)$ where the truth value t represents the evaluation result.

The evaluation of a monitor's runtime representation is formally defined by a small-step operational semantics with a 6-ary transition relation $m \to_{p,s,v,R} m'$ 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, and R denotes the set of those positions which are reported by the transition to make the monitor body false. The monitor thus processes a stream $\langle v_0, v_1, \ldots \rangle$ by a sequence of transitions

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

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. In particular, we have

$$I_{p+1} = \{(t, \mathsf{n}(g), c) \in \mathcal{I} \mid \exists f \in \mathcal{F} : (t, f, c) \in I' \wedge \vdash f \to_{p,s_p,v_p,c} \mathsf{n}(g)\}$$
$$R_{p+1} = \{t \in \mathbb{N} \mid \exists f \in \mathcal{F}, c \in \mathcal{C} : (t, f, c) \in I' \wedge \vdash f \to_{p,s_p,v_p,c} \mathsf{d}(\bot)\}$$

where $I' = I_p \cup \{(p, f, ((V, p), (V, v_p)))\}$. The transition relation on monitors depends on a corresponding transition relation $f \to_{p,s,v,c} f'$ on formulas where c represents the context for the evaluation of 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 [10]) where the transition arrow $\to$ is to be read as $\to_{p,s,v,c}$ and rules Q4–Q7 are based on the following definitions.

$$I_0 = \{(i, f, (c.1[V \mapsto i], c.2[V \mapsto s(i + p - |s|)])) \mid p_1 \leq i \leq \min\{p_2 + 1, p\}\}$$

Atomic Formulas		
#	Transition	Constraints
A1	$\mathsf{n}(@V) \to \mathsf{d}(c.2(V)))$	$V \in dom(c.2)$
...		
Sequential conjunction		
C1	$\mathsf{n}(f_1 \,\&\, f_2) \to \mathsf{n}(\mathsf{n}(f_1') \,\&\, f_2)$	$f_1 \to \mathsf{n}(f_1')$
C2	$\mathsf{n}(f_1 \,\&\, f_2) \to \mathsf{d}(\bot)$	$f_1 \to \mathsf{d}(\bot)$
C3	$\mathsf{n}(f_1 \,\&\, f_2) \to \mathsf{n}(f_2')$	$f_1 \to \mathsf{d}(\top),\ f_2 \to \mathsf{n}(f_2')$
Quantification		
Q1	$\forall_{V\in[b_1,b_2]}\colon f \to \mathsf{d}(\top)$	$p_1 = b_1(c)\ ,\ p_2 = b_2(c)\ ,\ p_1 = \infty \vee p_1 > p_2$
Q2	$\forall_{V\in[b_1,b_2]}\colon f \to f'$	$p_1 = b_1(c)\ ,\ p_2 = b_2(c), p_1 \neq \infty\ ,\ p_1 \leq p_2,$ $\mathsf{n}(\forall_{V\in[p_1,p_2]}\colon f) \to f'$
Q3	$\mathsf{n}(\forall_{V\in[p_1,p_2]}\colon f) \to \mathsf{n}(\forall_{V\in[p_1,p_2]}\colon f)$	$p < p_1$
Q4	$\mathsf{n}(\forall_{V\in[p_1,p_2]}\colon f) \to f'$	$p_1 \leq p,\ \mathsf{n}(\forall^{I_0}_{V\leq p_2}\colon f) \to f'$
Q5	$\mathsf{n}(\forall^{I}_{V\leq p_2}\colon f) \to \mathsf{d}(\bot)$	DF
Q6	$\mathsf{n}(\forall^{I}_{V\leq p_2}\colon f) \to \mathsf{d}(\top)$	$\neg DF, I'' = \emptyset, p_2 < p$
Q7	$\mathsf{n}(\forall^{I}_{V\leq p_2}\colon f) \to \mathsf{n}(\forall^{I''}_{V\leq p_2}\colon f)$	$\neg DF, (I'' \neq \emptyset \vee p \leq p_2)$

Fig. 2. The operational semantics of formula evaluation.

$$I' = \begin{cases} I & \text{if } p_2 < p \\ I \cup (p, f, (c.1[V \mapsto p], c.2[V \mapsto v])) & \text{otherwise} \end{cases}$$
$$I'' = \{(t, \mathsf{n}(g), c) \in \mathcal{I}' \mid (t, f, c) \in I' \wedge\ \vdash f \to \mathsf{n}(g)\}$$
$$DF \equiv \exists t \in \mathbb{N}, f \in \mathcal{F}, c \in \mathcal{C} : (t, f, c) \in I' \wedge\ \vdash f \to \mathsf{d}(\bot)$$

We provide an example adapted from [3] on the application of these rules.

Example 1. We take the monitor $M = \forall_{0\leq x}\colon \forall_{y\in[x+1,x+2]}\colon @x \,\&\, @y$, 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 $m = \forall^{\emptyset}_{0\leq x}\colon f$ with $f = \forall_{y\in[b_1,b_2]}\colon g$ for some b_1 and b_2 and $g = @x \,\&\, @y$. We evaluate m over the stream $\langle\top,\top,\bot,\ldots\rangle$. First consider the transition $(\forall^{\emptyset}_{0\leq x}\colon f) \to_{0,\langle\rangle,\top,\emptyset} (\forall^{I_0}_{0\leq x}\colon f)$. which generates the instance set

$$I^0 = \big\{(0, \mathsf{n}(\forall_{y\in[1,2]}\colon g), (\{(x,0)\}, \{(x,\top)\}))\big\}\,.$$

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

$$I^1 = \big\{(1, \mathsf{n}(\forall_{y\in[2,3]}\colon g), (\{(x,1)\}, \{(x,\top)\})),\\ (0, \mathsf{n}(\forall^{\emptyset}_{y\leq 2}\colon g), (\{(x,0)\}, \{(x,\top)\}))\big\}\,.$$

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 $(\forall^{I_1}_{0\leq x}\colon f) \to_{2,\langle\top,\top\rangle,\bot,\{0,1\}} (\forall^{I_2}_{0\leq x}\colon f)$ where

$$I^2 = \big\{(2, \mathsf{n}(\forall_{y\in[3,4]}\colon g), (\{(x,2)\}, \{(x,\bot)\}))\big\}\,.$$

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.

3 Space Complexity

Our goal is to determine the maximum size of the runtime representation of a monitor during its execution. For doing this we have to define the size of the runtime representation of monitors, formulas and formula instances.

Definition 1. *We define the functions $c_m : \mathcal{M} \to \mathbb{N}$, $c_f : \mathcal{F} \to^{part.} \mathbb{N}$, $c_g : \mathcal{G} \to \mathbb{N}$, and $c_i : \mathcal{I} \to \mathbb{N}$ which denote the size of the runtime representation of a monitor respectively unevaluated formula (with and without tag) respectively formula instance:*

$$\begin{array}{rlrl} c_m(\forall^I_{0\leq V}\colon f) &= \sum_{g\in I} c_i(g) & c_f(\mathsf{n}(g)) &= c_g(g) \\ c_g(@V) &= 0 & c_g(f_1 \wedge f_2) &= c_f(f_1) + c_f(f_2) \\ c_g(\neg f) &= c_f(f) & c_g(f_1 \,\&\, f_2) &= c_f(f_1) + c_f(f_2) \\ c_g(\forall_{V\in[b_1,b_2]}\colon f) &= 1 & c_g(\forall^I_{V\leq p}\colon f) &= 1 + \sum_{g\in I} c_i(g) \\ c_g(\forall_{V\in[p_1,p_2]}\colon f) &= 1 & c_i((n, f, c)) &= c_f(f) \end{array}$$

Our notion of size thus only considers the quantifier structure of a monitor and its formulas that are being evaluated and ignores their propositional contents (because it is this structure that dominates the space complexity).

Now we can define a relation which determines the maximum size of the runtime representation of a monitor encountered during its execution.

Definition 2. *We define the relation $\multimap \subseteq \mathcal{M} \times \mathbb{N} \times \{\top, \bot\}^* \times \mathbb{N} \times \mathbb{N}$ inductively as follows:*

$$\begin{array}{l} M \multimap_{p,s,0} S' \leftrightarrow S' = c_m(M) \\ M \multimap_{p,s,(n+1)} S' \leftrightarrow \\ \quad \big(\exists R.\ (M \to_{p,s,s(p),R} M') \wedge (M' \multimap_{p+1,s,n} S) \wedge S' = \max\{c_m(M), S\}\big) \end{array}$$

Essentially, $M \multimap_{p,s,n} S$ states that S is the maximum size of the representation of monitor m during the execution of n transitions over the stream s starting at position p. Our goal is to compute/bound the value of S by a static analysis, i.e., without having to actually perform the transitions. We will later in Theorem 2 formalize the connection between our analysis and the relation given above. In a nutshell, this analysis proceeds as follows:

1. We compute from a monitor $M \in \mathbb{M}$ the *dominating monitor* $M' = D(M) \in \mathbb{M}$ whose space requirements on the one hand bound the requirements of M and on the other hand can be determined exactly by the subsequent analysis.
2. We translate $M' \in \mathbb{M}$ into a *quantifier tree* $qt = QT(M')$ which contains the essential information required for the analysis.
3. We translate qt into an *annotated quantifier tree* $aqt = AQT(qt)$ which labels every node with the maximum interval bound of the corresponding subtree.

Algorithm 1 Space Requirements of an Annotated Quantifier Tree

```
 1: function SR(aqt)                  ▷ aqt is an annotated quantifier tree (A, a, b, qt')
 2:     if A = ∞ then
 3:         return ∞
 4:     else
 5:         return Σ_{i=0}^{A−1} SR(aqt, i)
 6:     end if
 7: end function
 8:
 9: function SR(aqt,i)         ▷ aqt is an annotated quantifier tree (A, a, b, Q), i < A
10:     cil ← 1 + min {i, b} − a
11:     if cil ≤ 0 & b ≥ a then
12:         return 1
13:     else
14:         return 0
15:     end if
16:     if i ≥ b then
17:         inst ← 0
18:     else
19:         inst ← 1
20:     end if
21:     for all aqt' = (A', a', b', Q') ∈ Q do
22:         if i < A' then
23:             inst ← inst + cil · SR(aqt', i)
24:         end if
25:     end for
26:     return inst
27: end function
```

4. Finally, we compute $SR(aqt) \in \mathbb{N}$ by application of Algorithm 1.

While the various steps will be explained in the following subsections, we will give a short account on the rationale behind this algorithm.

The core idea is that, if the monitor has a limit on the size of its runtime representation, it has also a limit on the number of instances stored in that representation. This limit will be reached in a finite number A of steps determined by the maximum distance that any subformula of the monitor will "look forward" in the stream in relation to the position of the message that is currently being processed. It then suffices, for every distance i in the interval $[0, A-1]$, to determine the number $N(i)$ of instances that are created by the monitor instance $M(p)$ at position $p+i$; every monitor instance $M(p+A)$ then behaves identical to $M(p)$. In particular, if $p \geq A$ and the upper limit of the number of instances is reached, for every new instance added to some instance set another instance is removed. Thus it suffices to compute the sum of all $N(i)$ to determine the maximum space requirements of the monitor, which in essence explains the top-level function $SR(aqt)$ in the algorithm.

In the auxiliary function $SR(aqt, i)$ of the algorithm, we first determine the "current interval length" *cil* which essentially denotes the number of steps that have already been performed for the monitoring of the currently considered quantified formula. If $cil < 0$, the monitoring has not yet started, and the space requirements are 0. Otherwise, if i is less than the upper bound b of the quantifier interval, one more instance of the formula may be created at position i and stored for processing in future steps. Anyway, for every quantified subformula aqt', the number of instances $SR(aqt', i)$ has to be determined and multiplied with *cil*, since for every previous position that number of instances has been created.

After this short exposition, the following sections will elaborate the formal details of the analysis and also justify its soundness.

4 Dominating Monitor Transformation

A concept introduced in [3], *the Dominating Monitor formula*, allows us to restrict our analysis to quantified formulas whose variable intervals only depend on the outermost monitor variable, i.e. the size of every interval is the same for every value of the monitor variable.

Definition 3 (Dominating Monitor/Formula Transformation). *Let* $\mathbb{A} = \mathbb{V} \to^{part.} \mathbb{N}$ *be the domain of assignments that map variables to natural numbers. Then the* dominating monitor transformation $D : \mathbb{M} \to \mathbb{M}$ *respectively* formula transformation $D' : \mathbb{F} \times \mathbb{A} \times \mathbb{A} \to \mathbb{F}$ *are defined as follows:*

$$\begin{aligned}
D(\forall_{0 \le V} \colon F) &= \forall_{0 \le V} \colon D'(F, [V \mapsto 0], [V \mapsto 0]) \\
D'(@V, a_l, a_h) &= @V \\
D'(\neg F, a_l, a_h) &= \neg D'(F, a_l, a_h) \\
D'(F_1 \,\&\, F_2, a_l, a_h) &= D'(F_1, a_l, a_h) \,\&\, D'(F_2, a_l, a_h) \\
D'(F_1 \wedge F_2, a_l, a_h) &= D'(F_1, a_l, a_h) \wedge D'(F_2, a_l, a_h) \\
D'(\forall_{V \in [B_1, B_2]} \colon F, a_l, a_h) &= \forall_{V \in [h_L(B_1), h_H(B_2)]} \colon D'(F, a_l', a_h')
\end{aligned}$$

In the last equation we have $a_l' = a_l[V \mapsto h_L(B_1)]$, $a_h' = a_h[V \mapsto h_H(B_2)]$, $h_L(B_1) = \min\{[\![B_1]\!]^{a_l}, [\![B_1]\!]^{a_h}\}$, $h_H(B_2) = \max\{[\![B_2]\!]^{a_l}, [\![B_2]\!]^{a_h}\}$ *and* $[\![B]\!]^a$ *denotes the result* n *of the evaluation of bound expression* B *for assignment* a*; actually, if* B *contains the monitor variable* x*, the result shall be the expression* $x + n$ *(we omit the formal details, see the example below).*

The relationship, in terms of the maximum size of instance sets, between a monitor M and its dominating form $D(M)$ is summarized in the following theorem.

Theorem 1. *Let* $M \in \mathbb{M}$*. Then for all* $p, n, S, S' \in \mathbb{N}$ *and* $s \in \{\top, \bot\}^\omega$ *such that* $T(M) \multimap_{p,s,n} S$ *and* $T(D(M)) \multimap_{p,s,n} S'$*, we have* $S \le S'$*.*

Proof. The correctness of this theorem follows from Def. 2 and 3. Because the dominating formula considers the smallest lower bound and the largest upper

bound only and creates a constant interval over which the quantifier is defined. The constant interval is the largest interval considered by the initial formula. The initial might have also considered smaller intervals which are a subset of this largest interval, thus we are checking extra instances. If $M = D(M)$, i.e., the monitor is already in its dominating form, then we have $S = S'$ because there are no extra instances being checked.

Example 2. Consider the following monitor M:

$$\forall_{0 \leq x}\colon \forall_{y \in [x+1,x+5]}\colon ((\forall_{z \in [y,x+3]}\colon \neg @z \,\&\, @z) \,\&\, G(x,y))$$
$$G(x,y) = \forall_{w \in [x+2,y+2]}\colon (\neg @y \,\&\, (\forall_{m \in [y,w]}\colon \neg @x \,\&\, @m))$$

The dominating form $D(M)$ of M is the following:

$$\forall_{0 \leq x}\colon \forall_{y \in [x+1,x+5]}\colon ((\forall_{z \in [x+1,x+3]}\colon \neg @z \,\&\, @z) \,\&\, G(x,y))$$
$$G(x,y) = \forall_{w \in [x+2,x+7]}\colon (\neg @y \,\&\, (\forall_{m \in [x+1,x+7]}\colon \neg @x \,\&\, @m))$$

Notice that additional instances are needed for the evaluation of $D(M)$.

Dominating monitors are used in the construction of *annotated quantifier trees* Which allow for a simpler space analysis of the core language. Thm. 1 makes it clear that space complexity results derived for dominating monitor provide upper bounds for the space complexity of general monitors.

5 Quantifier Trees

In this section we introduce the concept of *quantifier trees*. A quantifier tree represents the skeleton of a monitor that only describes its quantifier structure without the propositional connectives.

Definition 4 (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.*

Definition 5 (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:*

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

By this transformation, every node of a quantifier tree consists of the variable bound by a quantifier, the interval bounds of the variable, and a set of nodes that represent the quantified subformulas. Thus a quantifier tree describes that internal structure of a monitor which essentially influences its space complexity.

In our analysis, we take a monitor $M \in \mathbb{M}$ and compute the quantifier tree $QT(D(M))$ of its dominating form $D(M)$. Every interval bound in a node

of that tree can be only ∞, a constant c, or a term $x + c$ where x denotes the monitor variable. We may thus annotate each node of the tree with the maximum constant occurring in the bounds of the subtree rooted at that node (except in the cases of lower bound being infinity or lower bound constant and upper bound variable). The following definition formalizes this annotation.

Definition 6 (Size Annotation). *We define the size annotation* $A : \mathbb{QT} \rightarrow^{part.} \mathbb{Z} \cup \{\infty\}$ *(whose domain is the set of quantifier trees resulting from the dominating form of a monitor) recursively as follows:*

$$
\begin{aligned}
A((V, \infty, B, qt)) &= 0 \\
A((V, c_1, x + c_2, qt)) &= \begin{cases} \max\{c_1, c_2\}, & \text{if } \forall q \in qt.\ A(q) \leq 0 \\ \infty, & \text{otherwise} \end{cases} \\
A((V, x + c_1, c_2, qt)) &= A((V, x + c_1, x + c_2, qt)) = A((V, c_1, c_2, qt)) \\
&= \max\{c_1, c_2, \max_{q \in qt}\{A(q)\}\} \\
A((V, x + c_1, \infty, qt)) &= A((V, c_1, \infty, qt)) = \infty
\end{aligned}
$$

Notice that the annotation takes care of the cases when the evaluation of a formula requires an infinite amount of memory. There are three such cases, the most complex one being $(V, c_1, x + c_2, qt)$: here the amount of memory needed increases over time if qt requires a positive amount of memory, because every time we generate a new monitor instance the interval increases. This occurs while we are still evaluating the previous instances. These two factors together result in an unbounded number of instances.

The point of this annotation is to indicate at what position a monitor instance's runtime representation will have size zero. Assume we are dealing with monitor instance $x = m$, when this instance is evaluated at position $A + n$ for $m \leq n$, the runtime representation is of size zero. When $m \geq n$ the runtime representation will have a size greater than zero. When $m > A + n$, the monitor instance cannot be evaluated at all and we end up with a runtime representation with size one. Our Alg. 1 considers monitor instance such that $n < m \leq A + n$.

Definition 7 (Annotated Quantifier Trees). *An* annotated quantifier tree *is inductively defined to be either* $\emptyset$ *or a tuple of the form* (a, b_1, b_2, Q) *where* $a \in \mathbb{Z} \cup \{\infty\}$*,* $b_1, b_2 \in \mathbb{Z} \cup \{\infty\}$ *and* Q *is a set of annotated quantifier trees. Let* $\mathbb{AQT}$ *be the set of all annotated quantifier trees.*

Definition 8 (Annotated Quantifier Tree Transformation). *We define* $AQT : \mathbb{QT} \rightarrow^{part.} \mathbb{AQT}$ *(whose domain is the set of quantifier trees where only*

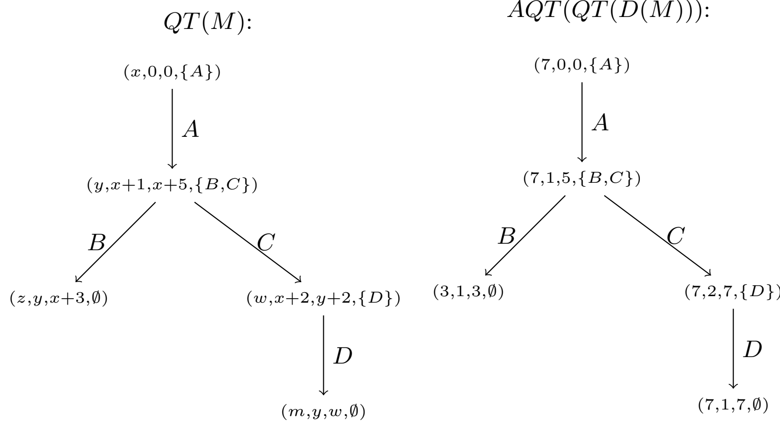

Fig. 3. (Annotated) Quantifier Trees

the monitor variable x occurs in bounds) recursively as follows:

$$\begin{aligned}
AQT((V, x+c_1, x+c_2, qt)) &= (A((V, x+c_1, x+c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q)) \\
AQT((V, c_1, c_2, qt)) &= (A((V, c_1, c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q)) \\
AQT((V, x+c_1, c_2, qt)) &= (A((V, x+c_1, c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q)) \\
AQT((V, x+c_1, \infty, qt)) &= (A((V, x+c_1, \infty, qt)), c_1, \infty, \cup_{q \in qt} AQT(q)) \\
AQT((V, c_1, \infty, qt)) &= (A((V, c_1, \infty, qt)), c_1, \infty, \cup_{q \in qt} AQT(q)) \\
AQT((V, \infty, x+c_1, qt)) &= (A((V, \infty, x+c_1, qt)), \infty, c_1, \cup_{q \in qt} AQT(q)) \\
AQT((V, \infty, c_1, qt)) &= (A((V, \infty, c_1, qt)), \infty, c_1, \cup_{q \in qt} AQT(q)) \\
AQT((V, c_1, x+c_2, qt)) &= (A((V, c_1, x+c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q)) \\
AQT((V, c_1, c_2, qt)) &= (A((V, c_1, c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q))
\end{aligned}$$

Notice that if any subtree of an annotated quantifier tree requires infinite memory, then the uppermost node of the tree, i.e. the root, will have an annotation of ∞. Also, if the monitor represented by the annotated quantifier tree is completely backwards looking, then the annotation at the root will be 0. Thus, in these two cases no further computation is necessary to compute the space complexity of the monitor. This can be seen in function $SR(\cdot)$ of Alg. 1. Also note that we drop the variable from the bounds. This means that the bounds c_1 and $x + c_1$ are treated the same. This is not problematic being that our algorithm only considers the case when x maps to zero. To deal with cases $x \geq 0$, we consider the monitor instance created at $x = 0$ at various future positions.

Example 3. Let us consider the monitor specification M from Ex. 2. Then $QT(M)$ and $AQT(QT(D(M)))$ are as depicted in Figure 3.

We are now ready to formally state the soundness of our analysis.

Theorem 2. *Let $M \in \mathbb{M}$ and $aqt = AQT(QT(D(M)))$. Then for all $n, p, S \in \mathbb{N}$ and $s \in \{\top, \bot\}^\omega$ such that $T(M) \multimap_{p,s,n} S$, we have $S \leq SR(aqt)$.*

We informally sketch the argument for the correctness of this theorem.

Ignoring the special cases that the algorithm considers, i.e. the annotation of infinite memory, or subtrees which evaluate instantly, the heart of the algorithm is the observation that the quantifiers in dominating monitors can be treated the same independently of their position in the formula. This is not the case for non-dominating monitors because there is dependence between the intervals.

A second important observation is that the evaluation of the runtime monitors is independent of the position of the stream. Thus, we can take a single monitor instance and evaluate it as different positions to understand how all instances of the monitor will evaluate. See Section 3 for more detail.

Going back to the first observation and Def. 1 & 2, we can consider the evaluation of a monitor M with a single quantifier whose interval is $[x+a, x+b]$, where $a \leq b$ and $a, b \in \mathbb{N}$. For $n \geq b$ it is easy to compute that $T(M) \multimap_{p,s,n} (b-a)+1$. However, at positions $a \leq n < b$, $T(M) \multimap_{p,s,n} (n-a)+1$. These results can already be found in [3]. Since the instance production of quantifiers is independent of their location in a formula, we can use these two basic results to compute the number of instances of the quantified formula produced. An elementary but tedious inductive argument leads to a soundness result and proof of Thm. 2. The argument would be, take a monitor with m quantifiers and construct a new monitor such that the monitor's formula has an additional quantifier added on top. The rest of the proof is just checking cases. We now give an asymptotic bound on the space complexity.

Theorem 3. *Let* $aqt = (A, b_1, b_2, aqt') \in \mathbb{AQT}$. *Then* $SR(aqt) = O\left(A^n\right)$ *where* $n = d(aqt)$ *is the quantifier depth of aqt inductively defined by* $d(\emptyset) = 0$ *and* $d(a, b_1, b_2, Q) = 1 + \max_{aqt \in Q} d(aqt)$.

Proof (sketch.). It is well known that $\sum_{i=0}^{A-1} i^n = O(A^n)$. If every quantifier in aqt has an interval $[0, A]$, then this summation accurately represents the computation of this algorithm: the outer $SR(\cdot)$ function represents the summation and the inner function $SR(\cdot,\cdot)$ computes the nth degree polynomial.

This result improves the $O\left(A^{2n}\right)$ space complexity bound presented in [3].

6 Experimental Results

We have experimentally validated the predictions of our analysis for the following monitors where (1a) and (2a) represent the dominating forms of the monitors (1b) and (2b), respectively:

$$\forall_{0 \leq x}\colon \forall_{y \in [x, x+80]}\colon \forall_{z \in [x, x+80]}\colon @z \quad (1a) \qquad \forall_{0 \leq x}\colon \forall_{y \in [x, x+40]}\colon \forall_{z \in [x, x+80]}\colon @z \quad (2a)$$
$$\forall_{0 \leq x}\colon \forall_{y \in [x, x+80]}\colon \forall_{z \in [x, y]}\colon @z \quad (1b) \qquad \forall_{0 \leq x}\colon \forall_{y \in [x, x+40]}\colon \forall_{z \in [x, y+40]}\colon @z \quad (2b)$$

The diagram in Figure 4 displays on the vertical axis the number of formula instances reported by the LogicGuard runtime system for corresponding monitors in the real specification language; the horizontal axis displays the number

of messages observed so far on the stream. The monitors are defined such that the body of the innermost quantifier always evaluates to true and thus always the full quantifier range is monitored and the worst-case space complexity is exhibited. One should note that the runtime system reports the number of formula instances while our analysis determines a measure for the size of the monitor's runtime representation (which is difficult to determine in the real system); however, for monitors with less than three nested quantifiers, such as the ones given above, the results coincide (the z-quantifier does not store any instances, since its body is propositional; the y quantifier contains instances of size 1; the runtime system reports the number of these instances which is identical to the total size of these instances determined by our analysis).

As expected, we can observe that the number of instances eventually reaches, after the startup phase, an upper bound. For the dominating monitors 1a and 2a, the predictions 1 (3320) and 2 (2459) reported by the analysis accurately match the observations. As also expected, however, these predictions overestimate the number of instances observed for the non-dominating monitors 1b (160) and 2b (1659), from which the dominating monitors were derived. Interestingly, the overapproximation for monitor 2b (by a factor of 1.5) is much less than for monitor 1b (by a factor of 21). It seems that our analysis is better at predicting the number of instances for certain quantifier configurations. This would imply that quantifier configurations which we cannot predict well (i.e., where the difference between the actual space requirements and that of the dominating form is large) may have better performance in real-world scenarios. This is a topic that we are going to investigate further in future work.

We have also tested our algorithm with the following more realistic monitoring scenario which is based on the full language sketched in Section 2:

```
type int; type message; stream<int> IP;
stream<int> S = stream<IP> x satisfying @x>=0 :
      value[seq,@x,plus]<IP> y with x < _ <=# x+10000: @y;
monitor<S> M = monitor<S> x :
      forall<S> y with x < _ <=# x+15000:
        exists<S> z with y < _ <=# y+4000: IsEven(#z);
```

The predicate `IsEven(#z)` is true only when the message arrives at an even time. Notice that the internal quantifier of the monitor depends on the external quantifier, which as shown in Fig. 5 yields a less accurate prediction than for independent quantifiers.

7 Conclusions

In this paper we have studied the space complexity of runtime monitor execution. The monitors are written in the core version of the LogicGuard specification language. For this purpose, we have abstracted every monitor formula into a tree structure which contains only those aspects of the formula that influence the size of the runtime representation, which is determined by the number of instances.

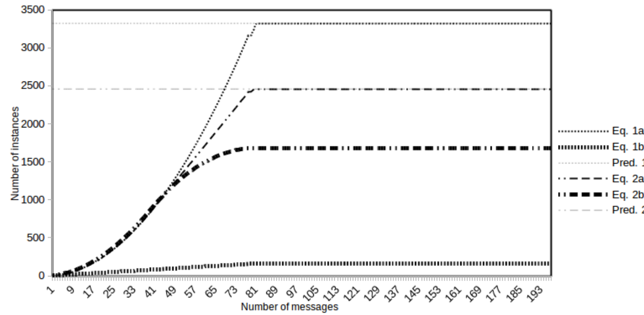

Fig. 4. Experimental results versus predictions

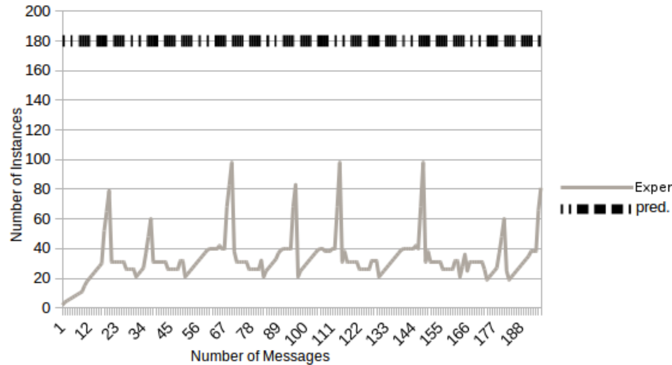

Fig. 5. Experimental results versus predictions

Using this structure, we developed an algorithm determining an upper bound for the number of formula instances that a monitor stores during execution. An essential part of this algorithm is the dominating monitor transformation, which over-approximates the actual number of instances stored. In our experimental results, it was shown that there are monitors whose instance number is accurately approximated by the algorithm's upper bound and monitors where the upper bound is far too conservative. The algorithm presented hints at a possible optimization when considered in conjunction with the technical report [2]. The ordering of quantifiers seems to greatly influence space complexity, that is larger quantifiers first implies lower space complexity. We plan to investigate optimizations based on quantifiers commutativity.

Another point we would like to address in future work is the variety of ways one can calculate the space complexity of a monitor specification. In Section 6, we brought up the subtle differences between our space calculation and the one used in the actual runtime system. The two measures diverge for quantifier depth

three or greater. We plan to perform a similar analysis using this alternative approach to space complexity measures. On a similar note, both measures so far mentioned are closely related to possible time complexity measures. In the case of time complexity, we would like to count each individual step of the operational semantics. We want to develop a time complexity measure based on the space complexity measure devised here.

References

1. Julius Richard Büchi. Weak SO Arithmetic and Finite Automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.
2. David Cerna. Space Complexity of LogicGuard Revisited. Technical report, RISC, JKU, Linz, October 2015.
3. 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, editors, *7th International Symposium on Symbolic Computation in Software Science*, volume 39 of *EPiC Computing*, pages 29–41, 2016.
4. 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.
5. 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.
6. Bernd Finkbeiner and Lars Kuhtz. Monitor Circuits for LTL with Bounded and Unbounded Future. In *RV, 9th International Workshop, RV 2009*, volume 5779 of *LNCS*, pages 60–75, Grenoble, France, June 26–28, 2009. Springer, Berlin.
7. Markus Frick and Martin Grohe. The Complexity of FO and Monadic SO Logic Revisited. *Annals of Pure and Applied Logic*, 130(1–3):3–31, 2004.
8. IEEE Std 1850-2007:Standard for Property Specification Language (PSL), 2007.
9. 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 *LNAI*, pages 302–316, Phnom Penh, Cambodia, November 13–17, 2006. Springer, Berlin, Germany.
10. Temur Kutsia and Wolfgang Schreiner. Verifying the Soundness of Resource Analysis for LogicGuard Monitors. Technical Report 14-08, RISC, JKU, Linz, 2014.
11. LogicGuard II, November 2015. http://www.risc.jku.at/projects/LogicGuard2/.
12. Oded Maler, Dejan Nickovic, and Amir Pnueli. Real Time Temporal Logic: Past, Present, Future. In Paul Pettersson and Wang Yi, editors, *FORMATS*, volume 3829 of *LNCS*, pages 2–16, Uppsala, Sweden, September 26–28, 2005. Springer.
13. Robert McNaughton and Seymour Papert. *Counter-Free Automata*, volume 65 of *Research Monograph*. MIT Press, Cambridge, MA, USA, 1971.
14. 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, RISC, JKU, Linz, November 2015.
15. Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *(LICS '86), Cambridge, Massachusetts, USA, June 16-18*, pages 332–344. IEEE Computer Society, 1986.