Monitoring Network Traffic by Predicate Logic[⋆]

Wolfgang Schreiner[1], Temur Kutsia[1], Michael Krieger[2], Bashar Ahmad[2], Helmut Otto[3], and Martin Rummerstorfer[3]

[1] Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at, Temur.Kutsia@risc.jku.at
[2] RISC Software GmbH, Hagenberg, Austria
Michael.Krieger@risc-software.at, Bashar.Ahmad@risc-software.at
[3] SecureGUARD GmbH, Linz, Austria
hotto@secureguard.at, mrummerstorfer@secureguard.at

Abstract. We present an approach to the runtime monitoring of network traffic for the violation of properties specified in classical predicate logic. The properties are expressed by quantified formulas which are interpreted over sequences of messages, i.e., the quantified variable denotes a position in the sequence. Using the ordering of stream positions and nested quantification, complex properties can be formulated. To raise the level of abstraction, we allow the definition of a higher-level stream from a lower-level stream by a notation analogous to classical set builders. A translator generates from the specification an executable monitor; a static analysis determines whether the generated monitor only requires a finite number of past messages to be preserved.

Keywords: runtime verification, predicate logic, network security

1 Introduction

Today the security of computer networks is primarily established by the application of firewalls. Originating from simple packet filters that inspect individual packets on the network layer and having further evolved to stateful filters that operate on the connection layer and take into account the connection to which a packet belongs, firewalls nowadays also operate on the application layer and protect access to resources and services by inspecting the contents of network traffic. While thus their inspection capabilities have substantially increased, their language in which to describe legal network traffic has not changed so much: in essence a specification still consists of a set of filtering rules where each rule, based on the header information of a package, the connection to which it belongs, the protocol that is used for the connection and (if residing on a host) the kind of process from/to which the connection is established, decides whether to allow a package/connection request or not.

[⋆] Supported by the Austrian Research Promotion Agency (FFG) in the frame of the BRIDGE program by the project 832207 "LogicGuard".

Network monitoring, however, may also be seen as an application of runtime verification where various kinds of formalisms are used to constrain the observable behavior of a computing system. Typically specifications are expressed in languages that are richer than simple filtering rules, for instance finite state machines, regular expressions, context-free patterns, linear temporal logic, and combinations/extensions/variations of these. The main advantage of these formalisms is that they can be effectively checked at execution time, typically by an automata generated from the specification. On the other hand, their expressive power is still limited and, even if a property can be theoretically formulated in a particular language, this formulation may become practically quite tedious.

In this paper, we present a specification language for monitoring network traffic which is based on a formalism that is rich and well known, namely classical predicate logic and set theory. Properties are expressed by quantified formulas interpreted over sequences of messages; the quantified variable denotes a position in the sequence. Using the ordering of stream positions and nested quantification, complex properties can be formulated. Furthermore, to raise the level of abstraction, a higher-level stream may be constructed from a lower-level stream by a notation analogous to classical set builders. A translator generates from the specification an executable monitor. A static analysis is devised which determines whether the generated monitor only requires a finite number of past messages to be preserved in its local buffer; then the monitor can operate with a finite amount of stream history. The home page of our research project is [10]; from there the reports [7, 8, 12, 9] can be derived that complement this presentation.

The rest of this paper is structured as follows: in Section 2, we give an overview of the specification language and its application to monitoring network traffic. In Section 3, we sketch the translation of the language to an executable monitor. In Section 4, we present a static analysis to bound the resources required by the execution of the monitor. The prototype implementation of the language is described in Section 5. Section 6 discusses related work and Section 7 concludes.

2 The Specification Language

Figure 1 depicts the abstract syntax of the specification language (essentially an untyped and regularized form of the concrete language whose implementation is sketched in Section 5). In this language, we can write a specification such as

```
stream S;
stream T =
   construct X in S value M = S@X satisfying Start(M) :
     combine[Base,Fold] Y in S with X <= Y value N = S@Y
         satisfying SameSender(M,N) until End(N) : N;
monitor M = position X in T
  value N = (number Y in T with X <= Y <=T X+1000 : Request(T@Y)) :
  Allowed(N);
```

This specification monitors an external stream `S`; from this external stream another internal stream `T` is constructed by combining every message from `S` satisfying a predicate `Start` with all subsequent messages from the same sender

```
M  ::= MS MF
MS ::= _ | stream XS ; MS | stream XS = TS ; MS
MF ::= _ | monitor XM = MX ; MF
MX ::= F | position X : MX

F ::= XF | B : F | PC(T1,...,TN) | true | false |
    | if F1 then F2 else F3
    | ~F  | F1 /\ F2 | F1 \/ F2 | F1 => F2 | F1 <=> F2
    | forall X : F | exists X : F

T ::= TP | TV | TS
TP ::= XP | B : TP | max X : F | min X : F
TV ::= XV | THIS | NEXT | B : TV | FV(T1,...,TN) | XS@TP | XS#TP
     | num X : F | complete combine[TV0,FV] X : TV1
TS ::= XS | B : TS | FS(T1,...,TV)
     | construct X : TV | build X : TS
     | partial combine[TV0,FV] X : TV1

B ::= formula XF = F | position XP IN XS = TP | value XV = TV
X ::= XP in XS R C U
R ::= _ | with Q P XP | with XP P Q | with Q1 P1 XP P2 Q2
P ::=  < | <= | <T | <=T
Q ::= TP | TP + N | TP - N
C ::= _ | satisfying F C | B C
U ::= _ | until F
```

Fig. 1. The Specification Language

(as determined by a predicate `SameSender`) until an end condition `End` is encountered; the combination starts with a `Base` value by application a binary `Fold` operation. The monitor `M` surveils every message in `T` and determines how many messages within 1000 time units satisfy the predicate `Request`; it reports a violation if the predicate `Allowed` rejects this number. The definitions of the functions and predicates applied in the specification are delegated to an external layer not visible in the specification.

The example depicts two essential features of the specification language:

- Monitors are described by quantified phrases where the bound variable ranges over positions of a stream; a stream position is reported as a violation of the specification, if it is the value of a variable introduced by a monitor's `position` quantifier such that the body formula becomes false.
- To raise the level of abstraction of a specification, monitors need not be interpreted over the external streams; they may be also interpreted over internal streams that are (directly or indirectly) derived from external ones. Such internal streams are typically constructed by combining messages from other streams by the application of externally defined combination functions.

In somewhat more detail, the core features of the language are as follows:

- The specification language has multiple kinds of phrases which denote different kinds of entities and which may be bound to corresponding kinds of variables; such an entity can be a monitor `M`, a formula `F`, a position term `TP`, a value term `TV`, and a stream term `TS`.
- Each phrase is interpreted in an environment of streams each of which is identified by a stream variable. Streams are introduced by the clause `MS` at the beginning of the specification; they are followed by a sequence of monitors `MF` which are evaluated over these streams.
- A stream is a finite or infinite sequence of messages each of which consists of a *value* (of an unspecified domain) and a *time* (a natural number); messages are ordered according to a (not necessarily strictly) increasing order of their times. The basic operations on a stream `XS` are the extraction of the value of a message at the position denoted by a term `TP` (operation `XS@TP` and of the time when this message has arrived (operation `XS#TP`; a time is also considered as a value.
- The formula language supports the usual logical connectives; we assume that in the binary connectives the evaluation of the formulas proceeds in parallel; if sequential evaluation is desired, the `if-then-else` construct may be used which evaluates its first argument before attempting the evaluation of either the second or the third one.
- Streams are processed by various kinds of quantified phrases whose bound variable runs over positions of the stream: quantified phrases denoting monitors (`position X:MX`), formulas (`forall X:F`, `exists X:F`), positions (`min X:F`, `max X:F`) and values (`num X:F`, `complete combine[TV0,FV] X:TV1`). Most of these follow conventional mathematical/logical practice; the quantifier `complete combine` folds the sequence of values denoted by the base term `TV1` by a binary function `FV` starting with base value `TV0`.
- Internal streams can be constructed by quantified phrases derived from a generalization of the set builder operation $\{TV|X\}$ where the constraint X introduces a bound variable x such that the term T is evaluated for a collection of bindings for x and the values are collected to the resulting set. The quantifier `construct X:V` is essentially interpreted in the same way except that the results are placed in the resulting stream; the quantifier `partial combine[TV0,FV] X:TV1` behaves like `complete combine` except that every partial combination result is placed on the resulting stream; the quantifier `build X:TS` constructs a collection of streams and merges the messages into the resulting stream by ordering them according their construction times.
- A constraint `X` has general form `XP in XS R C U`; it introduces a position variable `XP` in stream `XS`; the range of stream positions assigned to `XP` may be constrained via `R` by the order of the position with respect to other positions on the same stream (comparison operators `<` and `<=`) or by the order of the time associated to the message at the position with respect to the times of other positions on any stream (comparison operators `<T` and `<=T`). Clause `C` removes from the candidate sequence of positions those that fail to satisfy some guard condition; `U` constrains the sequence to the initial segment that is terminated by the first position that satisfies a termination condition.

The design of the specification language was shaped by the requirements of our industrial project partner who provided us with a challenge to express in the language a complex application scenario, that of a *download monitor* that checks multi-part file downloads in various protocols for malware contents. From the analysis of this scenario, it became quickly clear that the properties of interest could not be described directly on the abstraction layer of individual packets as provided by the IP layer of the Internet protocol suite. To cope with this scenario we extended the originally designed pure formula language by the additional feature of virtual stream constructions sketched above. With this extension, it became possible to adequately capture the given scenario:

– At the lowest layer we assume a TCP-aware runtime systems that provides to the monitor a stream `ip` that presents the IP packages of every TCP/IP connection in the logically correct order without omissions or duplications (it was deemed neither practical nor necessary to model within our specifications the transport layer of the Internet Protocol).
– From the external `ip` stream, an internal stream `conn` is constructed where every message represents a new state in some TCP/IP connection; this stream is constructed by application of the `build` quantifier (merging the messages arising from different connections) to the result of a `partial combine` quantifier (combining the packets from an individual connection); every new packet arriving on `ip` leads to a state change in some connection and thus to a new message in `conn`.
– From the internal `conn` stream, by application of the `construct` quantifier virtual streams for different file transfer protocols (e.g., `http` and `ftp`) are constructed; every message in such a stream contains a ranges of bytes of some file downloaded via the individual protocol; the various streams are merged into a single protocol-independent stream `ranges`.
– The messages on `ranges` are combined by application of `complete combine` to a stream `parts` of parts of files individually derived from different servers. These parts are subsequently combined to a stream `files` of complete files which are ultimately scanned by the monitor for occurrences of malware (the scenario assumes that only a complete file may be scanned because it may be compressed in a format that does not support on-the-fly decompression).

We deliberately chose this "heavy-weight" scenario as the starting point of language development, because it was considered to be much more difficult to start with a restricted language to which potentially later unforeseen capabilities would have to be added than to start with an expressive language and then potentially focus on the efficient implementation of some subset. In the following we have also worked on the specification and monitoring of much more "light-weight" security properties such as *ping attacks* (ping of death, ping flood, smurf); a catalogue of corresponding specifications is under development. Ultimately, we also plan the development of a "specification pattern" catalogue and supporting tool sets which simplifies the construction of monitors for typical classes of security properties.

3 Translation

In [7], a formal denotational semantics is given for (an earlier version of) the language presented in Section 2. It is based on valuation functions

$$\begin{aligned}
[\![\ ]\!] &: \mathit{Monitor} \to \mathit{Environment} \to \mathbb{P}(\mathit{Position})\\
[\![\ ]\!] &: \mathit{Formula} \to \mathit{Environment} \to \mathit{Bool}\\
[\![\ ]\!] &: \mathit{Stream} \to \mathit{Environment} \to \mathit{Bool}
\end{aligned}$$

where

$$\begin{aligned}
\mathit{Environment} &:= \mathit{Variable} \overset{\textsf{part.}}{\to} (\mathit{Stream} + \ldots)\\
\mathit{Stream} &:= \mathit{Message}^{\omega} \cup \mathit{Message}^{*}\\
\mathit{Position} &:= \mathit{Identifier} \times \mathbb{N}
\end{aligned}$$

These functions map, for a given environment e of streams to be monitored, the specification of a monitor M to the set $[\![M]\!](e)$ of stream positions violating the specification, a formula F to its truth value $[\![F]\!](e)$ (in a four-valued logic including ? for "unknown" and $\perp$ for "error"), and a stream term S to a stream, i.e., a finite or infinite sequence of messages.

In [8], the translation the specification of a monitor to an executable program is defined. This translation is in essence based on functions

$$\begin{aligned}
T &: \mathit{Monitor} \to \mathit{MonitorStep}\\
T &: \mathit{Formula} \to \mathit{FormulaStep}\\
T &: \mathit{Stream} \to \mathit{StreamStep}
\end{aligned}$$

with the following domains:

$$\begin{aligned}
\mathit{MonitorStep} &:= \mathit{Present} \to \mathit{MonitorResult}\\
\mathit{MonitorResult} &:= \mathbb{P}(\mathit{Position}) \times (\textsf{\textbf{done}} \ + \textsf{\textbf{next}} \textsf{ of } \mathit{MonitorStep})\\
\mathit{FormulaStep} &:= \mathit{Present} \to \mathit{FormulaResult}\\
\mathit{FormulaResult} &:= \textsf{\textbf{done}} \textsf{ of } \mathit{Bool} \ + \textsf{\textbf{next}} \textsf{ of } \mathit{FormulaStep}\\
\mathit{StreamStep} &:= \mathit{Present} \to \mathit{StreamResult}\\
\mathit{StreamResult} &:= \mathbb{P}(\mathit{Message}) \times (\textsf{\textbf{done}} \ + \textsf{\textbf{next}} \textsf{ of } \mathit{StreamStep})
\end{aligned}$$

The evaluation of monitors, formulas, and streams proceeds in *steps* where the current situation ("present") p is passed to the current state of the evaluator: the application $T(M)(p)$ returns those violating stream positions that could be determined from the present and, if the monitor has not yet terminated, its new state; $T(F)(p)$ may return either the truth value of formula F or return the new state of the formula evaluator; $T(S)(p)$ returns the set of messages produced on stream S in the current step and, if the stream is not closed, its new state. The present p contains the environment mapping free variables to their denotations, the new message observed on a stream, and the identifier of that stream. However, it also contains the *histories* of all streams, i.e., all messages observed so far. In Section 4 we will devise a static analysis that can be applied to *prune* message histories, such that certain specifications can be monitored with a bounded amount of stream history as it is needed for the perpetual execution of the monitor.

As an example, the translation for a sequential conjunction operator `&&` has been formalized in [8] as:

$$T : \mathit{Formula} \rightarrow \mathit{FormulaStep}$$
$$T(F_1 \;\texttt{\&\&}\; F_2) := \mathit{and}(T(F_1), T(F_2))$$

```
$\mathit{and} : \mathit{FormulaStep} \times \mathit{FormulaStep} \rightarrow \mathit{FormulaStep}$
$\mathit{and}(f_1, f_2)(p) :=$
    let $r_1 = f_1(p)$
    case $r_1$ of
        $\textbf{done}(b_1)$ :
            if $b_1 = \bot \vee b_1 =$ false then $r_1$ else
            let $r_2 = f_2(p)$
            case $r_2$ of
                $\textbf{done}(b_2)$ : if $r_2 =$ true then $r_1$ else $r_2$
                $\textbf{next}(s_2)$ : $\textbf{next}(f_1, s_2)$
        $\textbf{next}(s_1)$ : $\textbf{next}(\mathit{and}(s_1, f_2))$
```

We now define the driver operation

```
$\mathit{run} : \mathit{MonitorStep} \times \mathit{Present} \times \mathit{Stream} \times \mathbb{N} \rightarrow \mathbb{P}(\mathit{Position})$
$\mathit{run}(M, p, s, n) :=$
    if $n = 0$ then $\emptyset$ else
        let $p' := \mathit{update}(p, \mathit{head}(s))$
        case $M(p')$ of
            $(rs, \textbf{done})$ : $rs$
            $(rs, \textbf{next}(M'))$ : $rs \cup \mathit{run}(M', p', \mathit{tail}(s), n-1)$
```

which applies the translation of of a monitor to n messages of a sequence s. We can then state the following result.

Theorem 1 (Soundness of Translation). *The translation of monitors is sound in the sense that only violating positions are reported. Formally:*

$$\forall M \in \mathit{Monitor}, e \in \mathit{Environment}, s \in \mathit{Stream}, n \in \mathbb{N} :$$
$$\mathit{run}(T(M), \mathit{present}(e), s, n) \subseteq [\![\, M \,]\!](e)(s)$$

where present : *Environment* $\rightarrow$ *Present translates a given environment into the initial monitor situation.*

The converse is not necessarily true: while the specification language also allows the description of *liveness properties* (which are only violated by infinite streams in their totality), the monitor can of course only detect the violation of *safety properties* (which are already violated by a finite prefix of a stream).

4 Resource Analysis and Optimization

The basic translation of the monitor presented in the Section 3 is naive in the sense that it assumes that all messages of the monitors are preserved in memory such that at a later time every instance of the monitor has access to the full history of the stream. Practically, however, we are of course only interested in monitors that can operate for an indefinite amount of time with a limited amount of memory resources. We have therefore devised a static analysis which is

1. able to determine whether a monitor in the given execution model only needs a limited number of messages from the history of a stream such that
2. the runtime system is able to *prune* the history of that stream such that at no time more than the required number of messages is preserved.

In [12], this analysis is described for the abstract language presented in Section 2; we will however present the essential ideas on the basis of the much simpler core language depicted at the top of Figure 2. This language is the basis of [9] where the soundness of the analysis is verified (the proof is currently being completed).

A specification in the core language describes a single monitor that controls a single stream of Boolean values where the atomic predicate `@`X denotes the value on the stream at the position X, `~`X denotes negation, F_1 `&&` F_2 denotes sequential conjunction (the evaluation of F_2 is delayed until the value of F_1 becomes available), F_1 `/\` F_2 describes parallel evaluation (both formulas are evaluated simultaneously until becomes false or both become true) and `forall` X `in` B_1`..`B_2 `:` F evaluates F at all positions in the range denoted by the interval $B_1 \ldots B_2$ until one instance becomes false or all instances have become true; the creation of a new instance $F[n]$ is triggered by the arrival of the message number n on the stream.

The middle part of Figure 1 sketches the formalization of the operational interpretation of a monitor by a translation $T : \mathit{Monitor} \rightarrow \mathit{TMonitor}$ from the abstract syntax domain *Monitor* to a domain *TMonitor* denoting the runtime representation of the monitor. Apart from the quantified position variable X and the translation $f = T(F)$ of the body of this monitor, this representation maintains the set *fs* of instances of f which for certain values of X could not yet be evaluated to a truth value. The execution of the monitor is formalized by an operational semantics with a small step transition relation $\rightarrow_{n,ms,m,rs}$ where n is the index of the next message m arriving on the stream, *ms* denotes the sequence of messages that have previously arrived (the stream history), and *rs* denotes the set of those positions for which it can be determined by the current step that they violate the specification. In this step, first a new instance mapping X to the pair (n, m) is created and added to the instance set and all instances in this set are evaluated; *rs* becomes the set of positions of those instances yielding "false", the new instance set fs_1 preserves all those instances that could not yet be evaluated to a definite truth value. This formalization of the monitor operation is based on a corresponding translation of the abstract syntax of formulas to their runtime representation and an operational semantics of formula evaluation sketched at the bottom of Figure 1; the details are presented in [9].

$$\begin{aligned} M &::= \texttt{monitor}\ X\ \texttt{:}\ F \\ F &::= \texttt{@}X \mid \texttt{\~{}}F \mid F_1\ \texttt{\&\&}\ F_2 \mid F_1\ \texttt{/\textbackslash}\ F_2 \mid \texttt{forall}\ X\ \texttt{in}\ B_1\texttt{..}B_2\ \texttt{:}\ F \\ B &::= \texttt{0} \mid \texttt{infinity} \mid X \mid B\texttt{+}N \mid B\texttt{-}N \\ N &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ \ldots \\ X &::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ \ldots \end{aligned}$$

$$T : \mathit{Monitor} \rightarrow \mathit{TMonitor}$$
$$T(\texttt{monitor}\ X\ \texttt{:}\ F) := \mathit{TM}(X, T(F), \emptyset)$$

$$\begin{aligned} \mathit{TMonitor} &:= \mathit{TM}\ \textsf{of}\ \mathit{Variable} \times \mathit{TFormula} \times \mathbb{P}(\mathit{TInstance}) \\ \mathit{TInstance} &:= \mathbb{N} \times \mathit{TFormula} \times \mathit{Context} \\ \mathit{Context} &:= (\mathit{Variable} \overset{\textsf{part.}}{\rightarrow} \mathbb{N}) \times (\mathit{Variable} \overset{\textsf{part.}}{\rightarrow} \mathit{Message}) \end{aligned}$$

$$\mathit{TMonitor} \rightarrow_{\mathbb{N}, \mathit{Message}^\omega, \mathit{Message}, \mathbb{P}(nat)} \mathit{TMonitor}$$

$$\frac{\begin{array}{l} \mathit{fs}_0 = \mathit{fs} \cup \{(n, f, [X \mapsto (n, m)])\} \\ rs = \{n \in \mathbb{N} \mid \exists g \in \mathit{TFormula}, c \in \mathit{Context} : (n, g, c) \in \mathit{fs}_0 \wedge \\ \qquad\qquad \vdash g \rightarrow_{n,ms,m,c} \textsf{done}(\text{false})\} \\ \mathit{fs}_1 = \{(n, g_0, c) \in \mathit{TInstance} \mid \exists g \in \mathit{TFormula} : (n, g, c) \in \mathit{fs}_0 \wedge \\ \qquad\qquad \vdash g \rightarrow_{n,ms,m,c} \textsf{next}(g_1)\} \end{array}}{\mathit{TM}(X, f, \mathit{fs}) \rightarrow_{n,ms,m,rs} \mathit{TM}(X, f, \mathit{fs}_1)}$$

$$T : \mathit{Formula} \rightarrow \mathit{TFormula}$$
$$T(\ldots) := \ldots$$

$$\begin{aligned} \mathit{TFormula} &:= \mathit{done}\ \textsf{of}\ \mathit{Bool} \mid \mathit{next}\ \textsf{of}\ \mathit{TFormulaCore} \\ \mathit{TFormulaCore} &:= \ldots \end{aligned}$$

$$\mathit{TFormula} \rightarrow_{\mathbb{N}, \mathit{Message}^\omega, \mathit{Message}, \mathit{Context}} \mathit{TFormula}$$

$$\frac{\ldots}{\textsf{next}(\ldots) \rightarrow_{n,ms,m,c} \ldots}$$

Fig. 2. The Core Language and Its Operational Semantics (Excerpt)

	0	1	2	3	4	5	6	7	8	9	...
	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	...
$F[0]$	$\bigcirc$										
$F[1]$	$\times$	$\bigcirc$									
$F[2]$		$\times$	$\otimes$	$\times$							
$F[3]$				$\bigcirc$							
$F[4]$				$\times$	$\bigcirc$						
$F[5]$					$\times$	$\otimes$	$\times$	$\times$			
$F[6]$						$\times$	$\otimes$	$\times$	$\times$		
$F[7]$							$\times$	$\otimes$	$\times$		
$F[8]$									$\bigcirc$		
$F[9]$										$\bigcirc$	
$\vdots$											

Fig. 3. Monitor Instances

In this core language, we are able to express a specification such as

```
monitor X: ~(@X /\ forall Y in X-1..X+2 : @Y)
```

where the core formula could be written with the help of an implication operator and an existential quantifier as `@X => exists Y in X-1..X+2:~@Y`. Let us assume that this monitor is executed on the stream $[\bot, \top, \top, \bot, \top, \top, \top, \top, \bot, \bot, \ldots]$ (where $\bot$ denotes "false" and $\top$ denotes "true"). The execution proceeds by evaluating the instances $F[X]$ illustrated in Figure 3 which are created at those stream positions indicated by $\bigcirc$; the positions on which the corresponding `exists` formulas are evaluated are indicated by occurrences of $\times$. The table illustrates that the monitor needs to preserve at most 1 message from the past (to the left of every circle there is at most one marker) and there are at most two instances alive at the same time (to the top of every circle that there are at most two markers). This is exactly the information determined by the static analysis depicted in Figure 4.

This analysis is presented in inference style by three kinds of judgements:

- $\vdash M : (h, d)$ states that the evaluation of monitor M requires at most h messages from the past of the stream and at most d old monitor instances.
- $e \vdash F : (h, d)$ states that evaluating formula F requires at most h messages from the past and at most d messages from the future of the stream.
- $e \vdash B : (l, u)$ determines the lower bound l and upper bound u for the position denoted by an interval bound B.

In the inferences, e denotes a partial mapping of variables to pairs (l, u) denoting the lower bound and upper bound of the interval relative to the position of the "current" message. If the derivation yields $h = \infty$ or $d = \infty$, the analysis could not determine a bound. The crucial rules in the analysis are the following:

1. In an application of the sequential conjunction operator `&&`, the delay d_1 caused by the evaluation of formula F_1 increases the history h_2 required for the evaluation of F_2 to $h_2 + d_1$.

$$\vdash M : \mathbb{N}^\infty \times \mathbb{N}^\infty$$

$$\frac{[\llbracket X \rrbracket \mapsto (0,0)] \vdash F : (h,d)}{\vdash (\texttt{monitor}\ X\ \texttt{:}\ F) : (h,d)}$$

$$\mathit{Environment} \vdash F : \mathbb{N}^\infty \times \mathbb{N}^\infty$$

$$e \vdash \texttt{@}X : (0,0) \qquad \frac{e \vdash F : (h,d)}{e \vdash \texttt{\~{}}F : (h,d)}$$

$$\frac{e \vdash F_1 : (h_1,d_1),\ \ e \vdash F_2 : (h_2,d_2)}{e \vdash F_1\ \texttt{\&\&}\ F_2 : (\max^\infty(h_1, h_2 +^\infty d_1), \max^\infty(d_1,d_2))}$$

$$\frac{e \vdash F_1 : (h_1,d_1),\ \ e \vdash F_2 : (h_2,d_2)}{e \vdash F_1\ \texttt{/\textbackslash}\ F_2 : (\max^\infty(h_1, h_2), \max^\infty(d_1,d_2))}$$

$$\frac{\begin{array}{l} e \vdash B_1 : (l_1,u_1),\ \ e \vdash B_2 : (l_2,u_2) \\ e[\llbracket X \rrbracket \mapsto (l_1,u_2)] \vdash F : (h',d') \\ h = \max^\infty(h', \mathbb{N}^\infty(-^\infty l_1)) \\ d = \max^\infty(d', \mathbb{N}^\infty(u_2)) \end{array}}{e \vdash (\texttt{forall}\ X\ \texttt{in}\ B_1\texttt{..}B_2\ \texttt{:}\ F) : (h,d)}$$

$$\mathit{Environment} \vdash B : \mathbb{Z}^\infty \times \mathbb{Z}^\infty$$

$$e \vdash \texttt{0} : (-\infty, 0) \quad e \vdash\ \texttt{infinity} : (\infty,\infty) \quad \frac{\llbracket X \rrbracket \notin \mathit{domain}(e)}{e \vdash X : (-\infty, 0)} \quad \frac{\llbracket X \rrbracket \in \mathit{domain}(e)}{e \vdash X : e(\llbracket X \rrbracket)}$$

$$\frac{e \vdash B : (l,u)}{e \vdash B\texttt{+}N : (l +^\infty \llbracket N \rrbracket, u +^\infty \llbracket N \rrbracket)} \quad \frac{e \vdash B : (l,u)}{e \vdash B\texttt{-}N : (l -^\infty \llbracket N \rrbracket, u -^\infty \llbracket N \rrbracket)}$$

$$\mathit{Environment} := \mathit{Variable} \overset{\textsf{part.}}{\to} \mathbb{Z}^\infty \times \mathbb{Z}^\infty$$
$$\mathbb{N}^\infty := \mathbb{N} \cup \{\infty\}, \mathbb{Z}^\infty := \mathbb{Z} \cup \{\infty, -\infty\}$$

$$\begin{aligned}
max^\infty :& \mathbb{N} \times \mathbb{N}^\infty \to \mathbb{N}^\infty, max^\infty(n_1,n_2) := \ \textbf{if}\ n_2 = \infty\ \textbf{then}\ \infty\ \textbf{else}\ \max(n_1,n_2) \\
+^\infty :& \mathbb{N}^\infty \times \mathbb{N}^\infty \to \mathbb{N}^\infty, n_1 +^\infty\ n_2 := \ \textbf{if}\ n_1 = \infty \lor n_2 = \infty\ \textbf{then}\ \infty\ \textbf{else}\ n_1 + n_2 \\
-^\infty :& \mathbb{N}^\infty \times \mathbb{N} \to \mathbb{N}^\infty, n_1 -^\infty\ n_2 := \ \textbf{if}\ n_1 = \infty\ \textbf{then}\ \infty\ \textbf{else}\ max(0, n_1 - n_2) \\
-^\infty :& \mathbb{Z}^\infty \to \mathbb{Z}^\infty, -^\infty i := \ \textbf{if}\ i = \infty\ \textbf{then}\ -\infty\ \textbf{else if}\ i = -\infty\ \textbf{then}\ \infty\ \textbf{else}\ -i \\
\mathbb{N} :& \mathbb{Z}^\infty \to \mathbb{N}^\infty, \mathbb{N}(i) := \ \textbf{if}\ i = -\infty \lor i < 0\ \textbf{then}\ 0\ \textbf{else}\ i
\end{aligned}$$

Fig. 4. The Analysis of the Core Language

2. To an application of the stream access operator `@`X we assign history 0 and delay 0, while in the analysis of the `forall` quantifier with bound variable X we superimpose the result of the analysis of body F with the over-approximation $[-l_1, u_2]$ of the interval over which X ranges. This is due to the fact that the monitor arising from the translation of the quantifier immediately transfers the message at the position denoted by X from the stream to the runtime context as soon as this position is reached such that further stream access via this variable does not require history or cause delay.

For the example above, the analysis yields

```
monitor X^(0,0):                             (1,2)
  ~(                                         (1,2)
    @X /\                                    (0,0)
    forall Y^(-1,2) in X-1..X+2 :            (1,2)
      !                                      (0,0)
      @Y)
```

where variables are annotated with the values (l, u) in the corresponding environments and in the column to the right the derived values (h, d) are shown. The analysis yields $h = 1$ and $d = 2$, as expected from our previous consideration.

In the following, we generalize the monitor's small step transition relation $\rightarrow_{n,ms,m,rs}$ to a big step relation $\rightarrow^*_{n,s,rs}$, which describes the processing the first n messages in stream s by which the violations rs are reported:

$$Mt \rightarrow^*_{0,s,\emptyset} Mt \qquad \frac{\begin{array}{l} n > 0 \\ Mt \rightarrow^*_{0,s,rs_1} Mt' \\ Mt' \rightarrow_{n-1,s\downarrow(n-1),s(n-1),rs_2} Mt'' \end{array}}{Mt \rightarrow^*_{n,s,rs_1\cup rs_2} Mt''}$$

This relation passes to transition $\rightarrow$ in $s \downarrow (n-1)$ the full history of s before the current position n. A corresponding relation $\rightarrow^*_{n,s,rs,h}$ can be defined that behaves like $\rightarrow^*_{n,s,rs}$ except that passes to $\rightarrow$ only the last h messages from the history of s. We can then formalize the central claim of our static analysis.

Theorem 2 (Soundness of Resource Analysis). *The resource analysis of the core monitor language is sound with respect to its operational semantics, i.e., if the analysis yields for monitor M natural numbers h and d, then the execution does not maintain more than d monitor instances and does not require more than the last h messages from the stream. Formally:*

$$\begin{array}{l} \forall M \in \mathit{Monitor}, Mt \in \mathit{TMonitor}, n \in \mathbb{N}, s \in \mathit{Message}^\omega, rs \in \mathbb{P}(\mathbb{N}), d, h \in \mathbb{N}^\infty : \\ \quad \vdash M : (h, d) \Rightarrow \\ \qquad (d \in \mathbb{N} \Rightarrow (\vdash T(M) \rightarrow^*_{n,s,rs} Mt \Rightarrow |\mathit{instances}(Mt)| \leq d)) \wedge \\ \qquad (h \in \mathbb{N} \Rightarrow (\vdash T(M) \rightarrow^*_{n,s,rs} Mt \Leftrightarrow \vdash T(M) \rightarrow^*_{n,s,rs,h} Mt)) \\ \quad \textsf{where } \mathit{instances}(\mathit{TM}(X, f, \mathit{fs})) := \mathit{fs} \end{array}$$

A major part of the proof of this theorem is presented in [9]; the remainder of the proof is currently under completion.

5 Prototype Implementation

We have implemented a prototype of the concrete language which is in essence derived from the abstract language presented in Section 2. The concrete language differs from the abstract version in that it is strongly typed, based on a three-valued logic with values "true", "false", and "undefined", and various other features which are described in more detail in [12].

The prototype has been implemented in the programming languages C# and F#. The C# part of the implementation consists of a parser, type-checker, function inliner, and a runtime system which provides a network interface via the SharpPcap packet capture framework for the .NET environment; a specification may use atomic functions and predicates implemented by external C# code. The translator has been implemented in the functional language F# (a variant of Objective CAML) in a style that very closely resembles the abstract formulation of the translation that has been sketched in Section 3 and is presented in more detail in [8]. For instance, the translation of the sequential conjunction operator is denoted by the following F# function:

```
let rec And (f1: FormulaStep) (f2: FormulaStep): FormulaStep =
  (fun (present: Present) ->
    let r1 = f1 present
    match r1 with
    | Now(b1) ->
        if b1 = BoolU.Bool(false) then r1 else
        let r2 = f2 present
        match r2 with
        | Now(b2) ->
            if b2 = BoolU.Bool(true) then r1 else r2
        | Next(s2) ->
            if b1 = BoolU.Bool(true) then r2 else Next(And f1 s2)
    | Next(s1) ->
         Next(And s1 f2))
```

Using the prototype implementation, we have tested the various features of our specification language with numerous smaller examples. As a more realistic test case we have also specified the already mentioned "download monitor" that checks multi-part file downloads in various protocols for malware contents; this example will also serve as a test for the adequacy of our approach for the long-term purposes of our industrial project partner.

The resource analysis presented in Section 4 has not yet been implemented, most notably because the prototype implementation does not yet fully conform to the assumptions made for the static analysis with respect to the evaluation of quantifiers and the handling of message access; as soon as the implementation has been revised, the analysis and the corresponding optimizations will be integrated. We will then also perform systematic experiments with respect to the execution performance and memory requirements of the generated monitors.

To protect the commercial interests of our industrial project partner, the implemented prototype is not publicly available.

6 Related Work

The core logic of the presented specification language is essentially monadic first order logic (MFO) whose expressive power is that of LTL; however the translation of an MFO formula into an equivalent LTL formula in general entails a non-elementary growth in the size of the formula, i.e., properties can be expressed in MFO much more succinctly than in LTL. Since also the translation of an MFO formula into a deterministic automaton that accepts exactly those words that satisfy the formula requires a non-elementary number of states [4], only few frameworks have made practical use of monadic logic as a specification language; a major exception is the MONA tool [6] which is based on monadic second order logic and has been applied e.g. in hardware verification.

The work closest in spirit to our approach is the Eagle framework which is based on a temporal fixed point logic; a translation from LTL to this logic is provided in [2] and has been applied to intrusion detection in [11]. However, the search for a more efficient core logic lead to the lower-level rule based system RuleR [3] which has a more operational flavor and is thus further from our own strive towards an expressive declarative specification language.

The Monitoring-Oriented Programming framework MOP [5] supports via plugins a variety of specification languages such as finite state machines, extended regular expressions, context free grammars, linear temporal logic, past time LTL, past time LTL with calls and returns, and string rewriting systems; a specification language based on predicate logic similar to the one presented in this paper is not provided.

In the frame of the EU project SERENITY the event reasoning toolkit EVEREST [13] was developed for the asynchronous monitoring of security properties in networked environments; the system uses specification language which is based on the event calculus that originates from artificial intelligence research. This approach is quite different from ours which deliberately aims to stay within a classical logical framework.

In the recent dissertation [1], an intrusion detection system is presented which combines the use of temporal logic with that of stream data processing. Temporal logic specifications are translated into stream queries that run on the stream database server and are continuously evaluated against network traffic to detect intrusions. The system reports very high performance but its logic is operating on a lower level than the one presented in this paper.

7 Conclusions

We have presented an approach to monitoring network traffic which is purely declarative and entirely based on the classical formalisms of predicate logic and set theory. The resulting specification language is very expressive and allows by the concept of "virtual streams" to raise the level of abstraction in the specification of a property of interest. A prototype implementation demonstrates how specifications can be translated from the declarative framework to executable monitors that surveil network traffic for violations of the specified properties. A static analysis is able to give upper bounds for the amount on the history of a stream required for monitoring a specification such that the runtime system may prune the stream history accordingly. As soon as this analysis will have been integrated into the implementation, we will be able to monitor certain specifications with a limited amount of memory for an indefinite amount of time.

However, there is much potential for further optimization: for example, the handling of nested quantifiers is still naive in that different instances of a quantified outer phrase may perform duplicate work in the monitoring of the inner instances. As a first attempt to overcome this problem we will improve the runtime system by the caching of information determined by evaluating the inner quantifier of one instance for reuse by another instance. Another direction of research will be the transformation of a specification into a logically equivalent form that can be monitored in a more effective way. The further development of a catalogue of practically interesting specifications will provide more ideas for further optimizations.

References

1. Ahmed, A.M.: Online Network Intrusion Detection System Using Temporal Logic and Stream Data Processing. Ph.D. thesis, University of Liverpool, UK (2013)
2. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Program Monitoring with LTL in Eagle. In: IPDPS'04, 18th International Parallel and Distributed Processing Symposium — Workshop 16 PADTAD. Santa Fe, NM, USA, April 30 (2004)
3. Barringer, H., Rydeheard, D., Havelund, K.: Rule Systems for Run-time Monitoring: From Eagle to RuleR. Journal of Logic and Comput. 20(3), 675–706 (2010)
4. Büchi, C.: Weak Second-Order Arithmetic and Finite Automata. Zeitschrift für mathematische Logik und Grundlagen der Mathematik 6, 66–92 (1960)
5. Chen, F., Roşu, G.: MOP: an Efficient and Generic Runtime Verification Framework. In: 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07). pp. 569–588. ACM, New York (2007)
6. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., al.: Mona: Monadic Second-order Logic in Practice. In: Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019 (1995)
7. Kutsia, T., Schreiner, W.: LogicGuard Abstract Language. Tech. Rep. 12-08, RISC, Johannes Kepler University, Linz, Austria (2012)
8. Kutsia, T., Schreiner, W.: Translation Mechanism for the LogicGuard Abstract Language. Tech. Rep. 12-11, RISC, Johannes Kepler Univ., Linz, Austria (2012)
9. Kutsia, T., Schreiner, W.: Verifying the Soundness of Resource Analysis for LogicGuard Monitors (Part 1). Tech. rep., RISC, J. Kepler Univ. Linz, Austria (2013)
10. LogicGuard Project (2013), http://www.risc.jku.at/projects/LogicGuard
11. Naldurg, P., Sen, K., Thati, P.: A Temporal Logic Based Framework for Intrusion Detection. In: Formal Techniques for Networked and Distributed Systems (FORTE 2004). LNCS, vol. 3236, pp. 359–376. Springer, Berlin (2004)
12. Schreiner, W., Kutsia, T.: A Resource Analysis for LogicGuard Monitors. Tech. rep., RISC, Johannes Kepler University, Linz, Austria (December 2013)
13. Spandoudakis, G., Kloukindas, C., Mahbub, K.: The Runtime Monitoring Framework of SERENITY. In: Security and Dependability for Ambient Intelligence, chap. 13, pp. 213–237. No. 13 in Information Security Series, Springer (2009)