Securing Device Communication by Predicate Logic Specifications*

Wolfgang Schreiner, Temur Kutsia
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at
Temur.Kutsia@risc.jku.at

Michael Krieger, Bashar Ahmad
RISC Software GmbH
Hagenberg, Austria
Michael.Krieger@risc-software.at
Bashar.Ahmad@risc-software.at

Helmut Otto, Martin Rummerstorfer
SecureGUARD GmbH
Linz, Austria
hotto@secureguard.at
mrummerstorfer@secureguard.at

Abstract*—We present a novel approach to the runtime monitoring of network traffic where from a high-level specification of security properties an executable monitor is generated; this monitor observes the network traffic in real time for violation of the specified properties in order to report respectively prevent these violations. The specification formalism is purely based on the classical notions of predicate logic and set theory with the corresponding level of expressiveness; compared to other more restricted formalisms it has thus much stronger capabilities to describe properties of interest. Its high level of flexibility makes our approach also applicable to other problem areas and engineering domains such as process control where it is important to guarantee that sequences of events conform to a particular protocol.** ****Supported by the Austrian Research Promotion Agency (FFG) in the frame of the BRIDGE program by the project 846003 "LogicGuard II".

Keywords*— runtime monitoring; network security; event streams; predicate logic.

I. 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. The information on which the decision is based is therefore quite limited and predetermined by the firewall.

The problem of monitoring network traffic, however, may also be seen in the broader context of *runtime verification*. Here the core idea is to specify in some formalism the intended behavior of a system and to automatically generate from this specification an executable monitor that observes the actual execution of the system and reports violations of the specification. In the case of network monitoring, the observed system may consist of one or more streams of messages captured at some network interface(s) and the specified behavior may describe various security properties that the message streams are expected to satisfy. Depending on the formalism, the specified behavior may take into account not only a single message but the relationship of this message to all the other messages that have been observed so far and thus describe complex properties that are beyond the realm of current firewalls.

Most attempts to runtime monitoring are based on specialized formalisms such as linear temporal logic [2],[12],[1], rule systems on sets of atomic formulas [3], regular expressions, context-free grammars, and associated automata models [6], or the event calculus [16]. While the advantage of these formalisms lies in their efficient operational interpretation as executable monitors, their level of expressiveness is still quite limited; moreover their application often requires special formal expertise which may not be readily available.

In this paper, we present the LogicGuard stream specification language [15] and its associated monitoring system which attempts to overcome these limitations by applying a formalism that is rich and well known, namely classical predicate logic and set theory (more specifically, the logical foundation of LogicGuard is monadic second order logic [5][7]). 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 thus be formulated. Furthermore, to raise the level of abstraction, higher-level streams may be constructed from lower-level streams by a notation analogous to classical set builders.

Due to the expressiveness and flexibility of the specification language, our approach is not limited to monitoring security violations in a network. Both the specification language and its implementation have no "built-in" knowledge of the origin and the nature of the streams that are monitored; from their point of view, streams are just abstract sequences of "events" that are triggered by external sources; thus our approach can be applied to monitoring any kind of system that exhibits its behavior by triggering observable events. The runtime system provides interfaces to various event sources; new interfaces can be easily added. Furthermore, all knowledge about the events is confined to user-defined functions and predicates that are just declared in a specification; their actual definitions (in the form of executable code) is dynamically linked to the runtime system; new kinds of events can be thus added at any time. The language and its implementation should be thus also applicable to any kind of application domain where sequences of events shall conform to some "expectations" that are subject to a formal description; we thus see a wide range of applicability e.g. in the area of process control.

The implementation of the LogicGuard language is based on Microsoft .NET technology and the programming languages C# and F#: a translator generates from a specification an executable monitor [9] and a static analysis determines from the specification whether the generated monitor only requires a finite number of past messages to be preserved in its local buffers [13],[10]; if yes, the monitor can operate with a bounded amount of stream history.

The remainder of this paper is organized as follows: in Section II we describe an introductory application that depicts some features of our approach in the particular application context of process control. In Section III we outline the major elements of the stream monitoring specification language based on which we sketch in Section IV more application examples. In Section V we give an overview on the implementation of the corresponding monitoring system. In Section VI we present our conclusions and discuss further work. From the home page of our research project [11] various reports can be derived that complement this presentation [14].

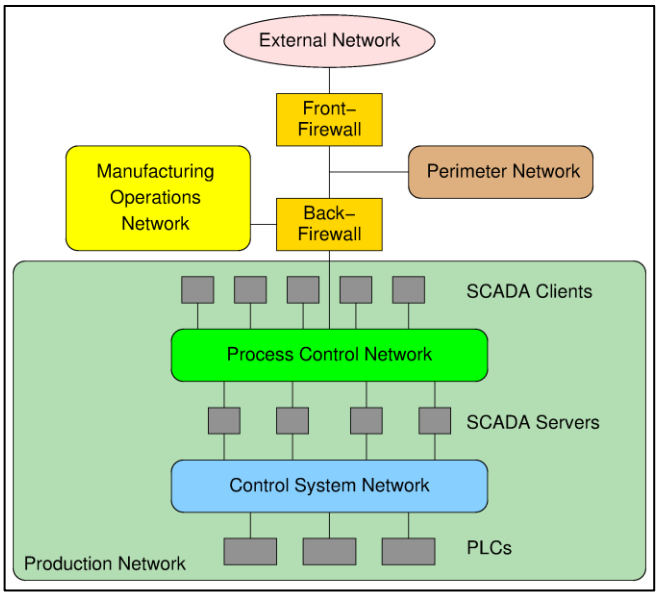

Fig. 1: The network of an industrial process plant.

II. An Application Scenario

Fig. 1 depicts the prototypical architecture of the corporate network of an industrial process plant. For security reasons the network is decomposed into multiple subnetworks that are separated by firewalls. The "front-firewall" protects the corporate network from the "external network" (the Internet) that is outside of the control of the corporation. Behind the front firewall there is the "perimeter network" of those hosts that are visible to the external network (the company's Web server and other published servers). The other company networks such as the "manufacturing operations network" are separated from the perimeter network by a "back-firewall" that prevents any access from the external network.

One of the company networks is the "production network" that encompasses the core facilities of the plant. Within this network there is the "process control network" to which the hosts of the "supervisory control and data acquisition" (SCADA) system are attached that operate and monitor the facilities [4]. Some of these hosts are clients that communicate via the process control network to servers that are linked via the "control system network" to programmable logic controllers (PLCs); these PLCs drive the actual machinery of the plant.

In such a corporate network the LogicGuard system may be applied on the front-firewall (respectively on any host connected to the perimeter network) in order to monitor the traffic flowing from/to the external network and to report suspicious activities that may represent *security violations*. However, while this is indeed a possible application of the system, we demonstrate the versatility of our approach by an application scenario where the system is applied on the back-firewall (respectively on any host connected to the process control network) in order to monitor the activities of the plant as reported by the SCADA system; the goal is to detect activities of the production process that deviate from specific norms and that may thus represent *safety violations*.

Our demonstration scenario is as follows: we assume that the PLCs operate a set of valves for which the SCADA system reports the following kinds of events:

- *Open*: a certain valve has been opened.
- *Close*: a certain valve has been closed.
- *Flow*: a certain volume of fluid has passed a valve.
- *Modify*: the flow rate of a valve has been modified.

This scenario can be modelled by the following declarations in the LogicGuard specification language:

```
type Event;
stream<Event> PCN;
logical Open(Event);
logical Close(Event);
logical Flow(Event);
logical Modify(Event);
```

The first two declarations introduce an abstract type *Event* of "events" and a stream *PCN* of such events. The subsequent four declarations introduce unary predicates that are true if and only if a given event is of the indicated type. Furthermore, a declaration

```
logical Same(Event,Event);
```

introduces a binary predicate *Same* that is true if two given events refer to the same valve. The concrete representation of the type *Event*, the source of the stream *PCN,* and the definitions of the various predicates are not part of the specification: the runtime system maps *Event* to some .NET type and provides an interface to the process control network that delivers a sequence of objects of this type under the stream name *PCN*; furthermore, the runtime system is dynamically linked to external .NET code that provides an executable implementation of the predicates.

In our first application scenario, we would like to check that every valve is closed not later than 100 time units after it has been opened. This is expressed by the following definition of a monitor:

```
monitor<PCN> Closed =
  monitor<PCN> x: Open(@x) =>
    exists<PCN> y with x<_<=#x+100 :
      Same(@x,@y) && Close(@y);
```

The declaration `monitor<PCN> Closed` introduces a monitor *Closed* that is interpreted over the stream *PCN*; this stream is conceptually an infinite sequence $e_0, e_1, e_2, \ldots$ of events that occur at stream positions $0,1,2,\ldots$ at some times $t_0 \leq t_1 \leq t_2 \leq \cdots$.

The definition of this monitor is provided by the expression `monitor<PCN> x:` … which introduces a locally bound variable x that denotes a position in stream *PCN*. The monitor checks whether for every value $0,1,2,\ldots$ for x the formula denoted by the monitor body … is true; every position for which the formula is false is reported as a "violation" of the specified property. In above definition, the formula body has shape `Open(@x)=>`… where `@x` denotes the event in stream *PCN* at position x and `=>` denotes logical implication. Thus only those positions are considered that denote *Open* events; violation reports thus indicate the positions of those events where valves have been opened that have not been closed in time.

The core formula `exists<PCN> y` … represents existential quantification: it asks for a position y in stream *PCN* that satisfies a certain property denoted by the body …; the formula is true if and only if there exists such a position. For every position x denoting an *Open* event, the monitor thus tries to find some position y that satisfies this property; if such a position can be found, the position x does not report a represent a violation of the specification; if the monitor can determine that no such position y exists, x is reported as a violation.

In more detail, the core formula has shape

```
exists<PCN> y …:
  Same(@x,@y) && Close(@y)
```

where `&&` denotes logical conjunction. This formula asks for a position y that denotes a *Close* event that refers to the same valve as the one in the *Open* event at position x. The monitor thus tries to find for every event that opens a valve another event that closes the same valve.

Furthermore, since the core formula has the clause

```
exists<PCN> y with x<_<=#x+100 :…
```

position y must be greater than x (the token _ represents the variable y introduced by the quantifier and < represents the strict ordering of stream positions); the event at position y must occur at a time that is not more than 100 time units after the time at which the event at position x has occurred (the token `<=#` denotes the non-strict time ordering and the phrase `x+100` denotes the time of the event at position x plus 100 time units). Thus, if 101 time units after the event at position x no suitable position y has been observed, the monitor can report x as a violating position. In other words, if a valve is not closed 101 time units after it has been opened, this fact can be reported as a violation of the specified property.

By this specification, the execution of the monitor at runtime essentially proceeds as follows: the monitor maintains a pool of positions that may represent violations of the specification; every such position is accompanied by a formula (actually the suspended state of its evaluation) that the position has to satisfy; whenever another *Open* event is observed on *PCN*, the corresponding position is added to the pool together with the initial state of the evaluation of the `exists` formula. For any event observed on *PCN,* every formula in the pool is further evaluated; if by this new observation the value of the formula can be decided, the corresponding position is removed from the pool and, if the formula has become false, reported as a violation; if the truth of the formula cannot be decided yet, the evaluation of the formula is suspended and the position and the state of the evaluation are returned to the pool.

In Section IV we will present more monitoring examples related to this scenario; more details on the implementation will be provided by Section V. Before, however, we will discuss the specification language in greater depth.

```
specification: (declaration ';')* ;

declaration: …
| 'stream' '<' typeid '>' streamid
| 'stream' '<' typeid '>' streamid '=' term
| 'monitor' '<' (streamid (',' streamid)* )? '>' monitorid '=' monitor ;

monitor: formula | 'monitor' variable monitor ;

formula: '(' formula ')' | 'true' | 'false' | 'logical' '?' | 'defined' formula | 'defined' term
| logicalid | logicalid '(' term (','term)* )? ')' | '!' formula
| formula '&&' ( '[' seq ']' )? formula | formula '||' ( '[' seq ']' )? formula
| formula '=>' ( '[' seq ']' )? formula | formula '<=>' ( '[' seq ']' )? formula
| 'if' ( '[' seq ']' )? formula 'then' formula 'else' formula
| 'forall' variable formula | 'exists' variable formula | binder ':' formula ;

term: '(' term ')' | 'value' '<' typeid '>' '?'
| 'stream' '<' typeid '>' '?' | 'stream' '<' typeid '>' 'empty'
| 'position' '<' streamid '>' '?' | 'zero' '<' streamid '>'
| 'old' | 'new' | ident | ident '(' (term (',' term)* )? ')'
| ( streamid )? '@' term | ( streamid )? '#' term
| 'if' ( '[' seq ']' )? formula 'then' term 'else' term
| 'min' variable formula | 'max' variable formula | 'num' variable formula
| 'value' '[' seq2 ',' term ',' valueid ']' variable term
| 'stream' '[' seq ']' variable term | 'stream' '[' seq2 ',' term ',' valueid ']' variable term
| 'merge' '[' seq ']' variable term | binder ':' term ;

seq: 'seq' | 'par' ;
seq2: 'seq' | 'par' | 'strict' ;

variable: '<' streamid '>' positionid
( 'with' bound ( 'and' bound )* )?
('satisfying' formula | binder )*
( ('until' | 'while' ) formula )? ':' ;

binder: 'logical' logicalid '=' formula
| 'position' '<' streamid '>' positionid '=' term
| 'value' '<' typeid '>' valueid '=' term ;
…
```

Fig. 2. Grammar of the LogicGuard language (excerpt).

III. The Language

Fig. 2 depicts an excerpt of the grammar of the LogicGuard stream specification language (in the syntax of the parser generator ANTLR); the full grammar is documented in [15]. In a nutshell, a specification consists of a sequence of the following elements:

- *External streams:* these are the “real” streams from which messages are fed into the monitor; the nature and origin of these streams are not part of the specification but are determined by the runtime system monitoring the specification.
- *Internal streams:* these are “virtual” streams that are constructed from other (external or internal) streams in order to raise the level of abstraction of the specification.
- *Monitors:* these are descriptions of properties that certain (external or internal) streams shall satisfy, separately or in combination.

The construction of internal streams and monitors may depend on user-defined functions and predicates. The language is statically typed: a specification may declare uninterpreted type names and external functions and predicates on these types; streams are typed with respect to the values they carry; language terms are typed with respect to the values they denote. Furthermore, stream positions are typed with respect to the identities to the streams to which they refer; thus dereferencing a position variable *pid* by the term *@pid* uniquely denotes a stream and the value carried by a particular message in that stream; correspondingly the term *#pid* denotes the (wall clock) time at which this message has arrived.

A. Formulas

The core of the specification language are formulas in a three-valued logic (true, false, undefined); in this logic we have the usual propositional connectives for negation, conjunction, disjunction, implication, and equivalence (denoted by the operators `!`, `&&`, `||`, =>, and <=>). The binary connectives may be annotated with the tags `[seq]` or `[par]` in order to indicate whether the evaluation of the first formula must terminate before the evaluation of the second formula can start or whether “parallel” formula evaluation is allowed; the latter is the default since monitoring becomes more efficient if the evaluation of formulas is delayed as little as possible (see also Section IV for more details).

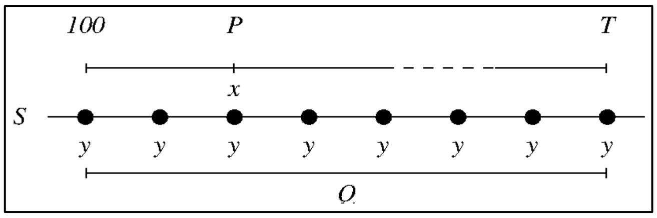

Fig. 3. Monitoring a stream.

However the most important formulas are the predicate logic formulas `forall` *variable formula* and `exists` *variable formula* which denote universal respectively existential quantification; the clause *variable* introduces an identifier *id* which denotes a position in some stream *S*; by the body *formula* the quantified formula can describe the values carried by a number of messages in *S*.

The positions assigned to *id* may be constrained by *variable* in multiple ways, namely

- by a clause _<*pos* (or _<=*pos)* which indicates that *id* must occur before (respectively not after) a particular position *pos* in the same stream;
- by a clause *pos*<_ (or *pos*<_*)* which indicates that *id* must occur after (respectively not before) a particular position *pos* in the same stream;
- by a clause _<#*pos*±*T* (or _<=*pos*±*T)* which indicates that the message at position *id* must occur at a time before (respectively not after) *T* time units plus/minus the time of the message at position *pos (*where *pos* may also refer to a position in another stream);
- by a corresponding clause *pos*±*T*<#_ (or *pos*±*T*<=*id)* that constrains the time of the message at position *id* from below;
- by a clause `until` *formula* which indicates that the quantification range terminates with the first position for which *formula* is true;
- by a clause `while` *formula* which indicates the quantification range terminates with the last position for which *formula* is true

as well as by combinations of these. Formulas represent the core of stream monitors; e.g. the monitor *M* defined as

```
monitor<S> M =
  monitor<S> x : P(@x) =>
    forall<S> y
      with x-100 <= _ until T(@y) :
      Q(@x,@y)
```

asks, for every message at some position *x* in stream *S* that satisfies property *P*, whether property *Q* is true for every message at some position *y* in *S* that occurs not earlier than 100 time units before *x*; for every message at *x* the monitoring stops with the first message at *y* for which property *T* holds. This relationship is illustrated in Fig. 3.

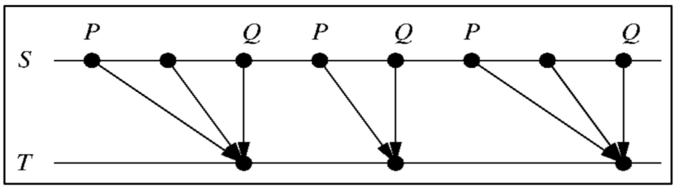

Fig. 4. Constructing a virtual stream.

B. Terms

The formula language embeds a term language that allows to extract (components from) the values carried by stream messages, to construct new values, and to combine values; since message values have uninterpreted types, the extraction and composition is ultimately based on external functions declared in the specification. The main role of the term language is to coordinate the composition of values that have emerged from messages that have arrived at different times respectively stream positions.

The core of the term language is represented by a number of quantified phrases such as `min` *variable formula*, `max` *variable formula* and `num` *variable formula* which denote the minimum/maximum position for which a formula is true as well as the number of positions for which this is the case. A more general kind of composition is denoted by the term pattern

`value` [s, b, f] *variable term*

which evaluates for all assignments of positions to the variable introduced by *variable* the denoted *term* yielding together with the base value *b* a non-empty sequence of values; by application of a binary function *f* these values are gradually combined to a single value that denotes the result of the term; the tag *s* indicates whether the evaluation must proceed in sequence or whether (because f is a commutative and associative operation) the order of combinations may be arbitrary. For instance, the term pattern

```
value[par,zero,add]<S> x … : f(@x)
```

describes the sum of the base value `zero` and of a bag of integers extracted from *n* messages in stream *S* (assuming that the user-defined function `add` denotes addition and `zero` denotes 0); its result is thus

$$f(x_1) + \cdots + f(x_n)$$

for some permutation $x_1, \ldots, x_n$ of these messages. Quantified terms can also denote streams, e.g.

```
stream<S> x … : f(@x)
```

denotes the stream $f(x_1), f(x_2), \ldots$ constructed from the messages $x_1, x_2, \ldots$ on stream *S*; the notation for stream construction mimics the classical set builder notation $\{f(x) \mid x \in S\}$. The term pattern

```
stream[par,zero,add]<S> x … : f(@x)
```

constructs the stream of values $s_1, s_2, \ldots$ where $s_i = f(x_1) + \cdots + f(x_i)$; i.e., its messages denote the partial results of the summation denoted by the `value` pattern shown above (the partial results are delivered in a non-deterministic order but using the keyword `seq` instead of `par` would make it deterministic by adding the elements in sequence).

Terms represent the core of the construction of virtual streams which allows to considerably raise the level of abstraction; e.g. the virtual stream *T* defined as

```
stream<M> T =
  stream<S> x satisfying P(@x) :
    value[s,b,f]<S>
      y with x <= _ until Q(@y): @y
```

consists of a sequence of values $v_1, v_2, ...$ of type *M* where each v_i is constructed from some value x_j on stream *S* that satisfies predicate *P*: the function *f* combines in v_i the values $b, x_j, x_{j+1}, ..., x_{j+n}$ where x_{j+n} is the first message for which property Q holds. This relationship between the original messages in *S* and the constructed messages in *T* is also illustrated in Fig. 4.

IV. Further Application Examples

We are now going to illustrate the language features presented in Section III by some more examples of monitors for the application scenario described in Section II.

A. *Checking Flow Rate Modifications*

First we would like to check that for no valve the number of flow rate modifications exceeds a certain bound before the valve is closed. For this purpose, we introduce another predicate

```
logical MaxModified(number);
```

which is true if a given number of events (denoted by the predefined type `number`) does not exceed a certain threshold. Then we can define the corresponding monitor as

```
monitor<PCN> Modified =
  monitor<PCN> x: Open(@x) =>
    forall<PCN> y with x < _
      satisfying Same(@x,@y)
      until Close(@y) : Modify(@y) =>
        value<number> n =
          num<PCN> z with x < _ <= y :
            Same(@x,@z)&&Modify(@z) :
        MaxModified(n);
```

For every valve that is opened as indicated by an *Open* event at some position *x* in the *PCN* stream, this monitor checks every subsequent position *y* of an event that refers to the same valve until a *Close* event indicates that the valve has been closed. If a *Modify* event at position *y* indicates a flow rate modification of this valve, the monitor determines the number *n* of such modifications that have occurred since the valve has been opened (i.e. the number of *Modify* events that have occurred at some position *z* after *x* but not later than *y*). If this number exceeds the allowed threshold as indicated by the predicate *MaxModified*, the position *x* at which the valve has been opened is reported as violating the specification. This specification thus illustrates the capability to deal with numerical properties that refer to a certain number of events.

B. *Checking Flow Amounts*

More general computations are required in our next example where we would like to check that for no valve the total amount of fluid that has passed the valve does not exceed a certain threshold. For this purpose we introduce by the declarations

```
type Flow;
value<Flow> Zero();
value<Flow> Add(Flow,Flow);
logical MaxFlow(Flow);
```

an abstract datatype *Flow* with a constant *Zero* and a binary operation *Add* for the computation of flow amounts; the predicate *MaxFlow* is true if the amount does not exceed a certain threshold. Furthermore the declaration

```
value<Flow> FlowValue(Event);
```

introduces a function *FlowValue* that extracts from a *Flow* event the amount of fluid reported by that event.

With these preliminaries we can now define the corresponding monitor as

```
monitor<PCN> Flowed =
  monitor<PCN> x: Open(@x) =>
    forall<PCN> y with x < _
      satisfying Same(@x,@y)
      until Close(@y) : Flow(@y) =>
        value<Flow> f =
          value[seq,Zero(),Add]<PCN> z
            with x < _ <= y satisfying
              Same(@x,@z)&&Flow(@z):
                FlowValue(@z)
        MaxFlow(f);
```

The definition of this monitor has the same shape as the previously introduced monitor *Modified*; the core difference is the computation of the flow value *f* rather than the event number *n*: For this computation we take into account all positions *z* of *Flow* events after the position *x* of the *Open* event up to the position *y* of the currently considered *Flow* event. From all these positions we extract the amount of liquid reported by the function *FlowValue*; by adding all these values with the help of the function *Add* starting with the base value *Zero*, we determine the total amount *f* of fluid that has passed the valve up to now and check whether it satisfies the threshold condition indicated by the predicate *MaxFlow*.

C. *Checking Flow Amounts (Revisited)*

While the previous examples have demonstrated the abilities of the specification language to handle event numbers and arbitrary computations with values derived from events, more complex specifications may become cumbersome to write and difficult to understand, if they have to be expressed solely in terms of the original event stream. We thus present an alternative solution for checking flow amounts where the level of abstraction of the monitor is raised by transforming the raw event stream into a virtual stream that carries the actual values of interest for the monitor. The core idea is to construct a pipeline

PCN → F → Flowed2

where

- the original event stream *PCN* is transformed to a stream *F* of "flow sums" that report the total amount of flow that has passed a valve so far, and
- the monitor *Flowed2* checks whether each flow sum report conforms to the required threshold.

The first step is achieved by the virtual stream definition

```
stream<Flow> F =
  merge[par]<PCN> x
      satisfying Open(@x):
    stream[par,Zero(),Add]<PCN> z
        with x < _ satisfying
        Same(@x,@z)&&
        (Flow(@z)||Close(@z))
        while !Close(@x,@z) :
      FlowValue(@z);
```

By the declaration

```
stream<Flow> F =
  merge[par]<PCN> x
    satisfying Open(@x): …
```

for every position *x* indicating an *Open* event a sub-stream of flow reports is generated; all sub-streams are merged to the result stream *F* in a "parallel" fashion, i.e., whenever a flow report is delivered by the sub-stream it is immediately forwarded to *F* which thus collects the flow reports of all open valves.

Each sub-stream is constructed by the term

```
stream[par,Zero(),Add]<PCN> z
    with x < _ satisfying
    Same(@x,@z)&&(Flow(@z)||Close(@z))
    while !Close(@x,@z) :
  FlowValue(@z);
```

which essentially proceeds like a corresponding term

```
value[par,Zero(),Add]<PCN> z …
```

by extracting and adding the flow amount reported by *FlowValue* from every *Flow* event that occurs after position *x* and refers to the same valve until the *Close* event for that valve is observed. However, in contrast to the `value` term which delivers a result only when all additions have ultimately been performed, the `stream` term delivers after every addition the partial result computed so far as an element of the result stream. Above stream thus delivers after every *Flow* event the total amount of flow reported so far for the valve whose opening was indicated by an *Open* event at position *x*; the merged stream *F* collects the reports for all valves.

With the help of the virtual stream *F* the task of the monitor becomes very simple; it can be expressed by the short definition

```
monitor<F> Flowed2 =
  monitor<F> x :
    MaxFlow(@x);
```

which operates on the virtual stream *F* by processing every flow report and checking whether it confirms to the threshold as indicated by the predicate *MaxFlow*.

The use of virtual streams may thus considerably simplify the definitions of monitors. It should be noted that the same virtual stream may be processed by multiple monitors and that the construction of virtual streams may be staged, i.e., from one virtual stream another virtual stream may be constructed.

D. Checking Manufacturing Requests

Our final example demonstrates that a monitor may also correlate multiple streams from different sources. We assume that the monitor is deployed on the back-firewall and has thus access not only to the process control network (PCN) but also to the manufacturing operations network (MON). The goal is to check that valve opening requests from the MON lead in time to an opening of the corresponding valve in the PCN. The additional declarations

```
type Request;
stream<Request> MON;
logical Same(Request,Event);
```

introduce an abstract datatype *Request*, a stream *MON* of such requests and a binary predicate *Same* that is true, if and only if the MON request and the PCN event refer to the same valve. The monitor

```
monitor<MON> Requested =
  monitor<MON> x:
    exists<PCN> y with
      x <=# _ <=# x+100:
        Same(@x,@y) && Open(@y)
```

checks for every *Request* observed in the stream *MON* at some position *x* whether it observes within 100 time units also an event in the stream *PCN* that honors this request, i.e., an *Open* event that refers to the same valve. If no such event is observed, the position *x* in *MON* is reported as violating the specification.

As this example demonstrates, also positions from different streams may be related by the absolute times (operator <=#) at which their corresponding events occur. However, since the positions refer to different streams, they must not be related by their relative order (operator <=). Actually, since stream positions are statically typed with respect to the streams to which they refer, any such attempt is reported as an error by the type checker and the subsequent translation of the specification to an executable monitor is refused.

V. The Implementation

As a first step towards an implementation of the specification language described in Section III we have defined (for an early version of language) a formal denotational semantics [8] and also formalized the translation of specifications described into this language into executable monitors that reports violations of the specified properties [9].

Based on this preparatory work we have implemented on top of Microsoft's .NET framework using the object-oriented programming language C# and the functional language F# a prototype of the LogicGuard language with the following components:

- A *parser* (whose C# code was produced from a BNF grammar with the help of the parser generator ANTLR) processes the text of the specification and generates an abstract syntax tree (AST).
- A *type checker* implemented in F# processes the AST and annotates it with type information; a *function inliner* implemented in F# replaces in the AST applications of user-defined functions and predicates by their definitions.
- An *analyzer* implemented in F# determines from the AST of the specification how much "stream history" the generated monitor requires for its operation (more on this below).
- A *translator* implemented in F# maps the resulting AST into an executable monitor. This monitor operates in a (potentially infinite) sequence of steps: in every step it accepts messages received from external streams, updates by these messages the status of its internal streams and of the specified properties, removes from its (external/internal stream) buffers those messages that according to the history analysis are not required any more, and returns information about the encountered violations of the specified properties.
- A *runtime system* implemented in C# dynamically links the .NET code of external functions and predicates that are used by the specification, feeds the messages received from external streams into the monitor, and reports the violations determined by the monitor. External messages may arise (via the SharpPcap packet capture framework for the .NET environment) from live network traffic or also from traffic captured in a log file in pcap format for post mortem analysis; for debugging purposes, also text files as stream sources are supported.

The core idea of the translation is that every formula and every stream is translated into a "step" function of type *FormulaStep* respectively *StreamStep*:

FormulaStep := Present → FormulaAnswer
FormulaAnswer := **done** of *Bool* + **next** of *FormulaStep*

StreamStep := Present → StreamAnswer
*StreamAnswer := Set(Message)×(***done** + **next** of *StreamStep)*

After every arrival of a message on an external stream the present state of all streams (including the newly arrived

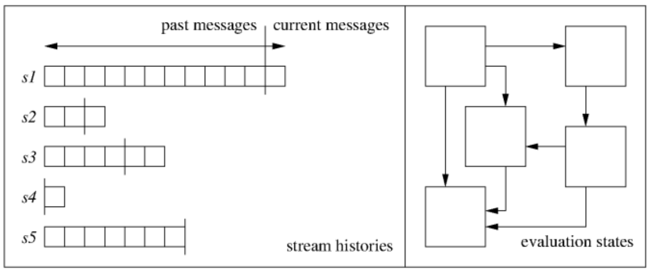

Fig. 5. The runtime state of a monitor.

message) is passed to these steps. A formula then either produces a Boolean answer or another step while a stream produces a set of messages and, if the stream has not yet terminated, another step; the resulting steps capture the suspended states of the evaluation of the phrases which are to be activated again by the arrival of another message on some external stream. The state of a monitor thus consists of the suspended states of the evaluation of all phrases in the specification (in particular formulas and stream terms) as is illustrated in the right part of Fig. 5. Additionally the monitor buffers all the messages produced on external and internal streams as is illustrated in the left part of Fig. 5; this state is used to prepare the new "present" which after the arrival of the next message is passed to the captured steps, thus re-activating the suspended execution.

However the buffering of the complete "histories" of all streams is clearly not sustainable: since monitors must be able to operate for an indefinite amount of time within a bounded amount of memory, the stream histories must be eventually pruned. For this purpose, we have devised a static analysis [13] which determines the amount of "history" that has to be preserved for every stream for the operation of the monitor. In a simplified form, this analysis is based on the logical judgment

$$e \vdash F\colon (h, d)$$

which determines for every formula (and in general for every phrase) F evaluated in environment e an upper bound h for the number of messages from the past ("history") that may have to be investigated for the evaluation of F and the number d of messages from the future ("delay") that may have to be awaited until the evaluation of F can be completed; the latter is needed since in a compound formula the result of one phrase F_1 may be required before the evaluation of another phrase F_2 can begin. The delay imposed by the evaluation of F_1 increases the history requirements of F_2 and thus of the whole phrase correspondingly (it is for this reason that the default for the evaluation of most compound phrases is the "parallel" evaluation of its components rather than the "sequential" one). Only specifications for which the analysis can impose an upper bound on the history requirements will be actually allowed for subsequent execution.

The information determined by the analysis is passed to the runtime system in the form of a two-dimensional table $H(s, t)$ that for arrival of a new message delivered to stream s determines how many messages need to be preserved on stream *t* after the message has been processed. The soundness of the analysis has been formally proved for a core version of the specification language [10]. Its implementation is currently under way; as soon as it will be completed, we will be able to commence with tests and benchmarks on perpetually running monitors generated from LogicGuard specifications.

VI. Conclusions

The LogicGuard stream specification language and monitoring system represent a novel approach to monitoring properties of streams of events, in particular (but not limited to) violations of security properties as determined by the sequence of messages observed at some network interface. This approach sets itself apart from other approaches such as the rule based specification formalisms of stateful firewalls by its flexibility and its high level of abstraction that are provided by its foundations in predicate logic and set theory.

By logical quantification over stream positions the language is able to describe properties that not only comprise a single message that is currently observed in the stream but that depend on multiple messages and that may thus take the history of the stream into account; the resulting monitor "loops" over the messages in the network and preserves in its state the information from the past of the stream relevant for processing future packages. By nested quantification "multi-level" properties may be described that depend on the relationship between messages on one level and messages on another level; for each message on one level an instance of the monitor is generated that processes the messages on the inner level. Furthermore, by constructing "virtual streams" that combine multiple messages from a lower-level stream into a single message of a higher-level stream, the layer of abstraction of the specification may be raised in analogy to a protocol stack where a corresponding transformation takes place.

The LogicGuard monitoring system mechanically generates from every specification an executable monitor; this monitor takes one message at a time and reports those violations of the specification that can be deduced from this message. Rather than manually coding programs for monitoring complex properties, which is a tedious and error-prone task, we leave the task of the generation of the monitor to an automatic system.

However, the generality of the LogicGuard specification language comes at a price: not for every specification that can be expressed in the language also an efficient monitor can be generated; in particular, for some specifications a monitor may have to preserve an unbounded amount of the history of the monitored stream in its buffer. We have thus devised an static analysis that allows to deduce from the text of a specification the amount of the stream history that needs to be buffered. Only specifications for which the analysis is able to determine a finite upper bound on the amount of history to be preserved are actually amenable to monitoring; at every step of the execution the monitor will prune from its buffer those messages that it will subsequently not require any more. In this form monitors may operate for an indefinite amount of time within a finite amount of memory.

Our future work will focus on evaluating our system on more and more application scenarios and thus exhibit its practical usefulness for these. Furthermore, we will work on a more in-depth analysis of the time and space requirements of the generated monitors; our goal is to devise classes of specifications and corresponding specification patterns for which efficient monitoring is always possible.

References

[1] A.M. Ahmed, „Online network intrusion detection system using temporal logic and stream data processing," Ph.D. thesis, University of Liverpool, UK, 2013.

[2] H. Barringer, A. Goldberg, K. Havelund, K. Sen, "Program monitoring with LTL in Eagle," IPDPS'04, 18th International Parallel and Distributed Processing Symposium — Workshop 16 PADTAD. Santa Fe, NM, USA, April 30, 2004.

[3] H. Barringer, D. Rydeheard, K. Havelund, "Rule systems for run-time monitoring: from Eagle to RuleR," Journal of Logic and Comput. 20(3), pp. 675–706, 2010.

[4] S. A. Boyer; „SCADA: Supervisory Control and Data Acquisition", ISA – The Instrumentation, Systems, and Automation Society, 2004.

[5] C. Büchi, "Weak second-order arithmetic and finite automata," Zeitschrift für mathematische Logik und Grundlagen der Mathematik 6, pp. 66–92, 1960.

[6] F. Chen, G. Rosu, "MOP: an efficient and generic runtime verification framework," 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07). pp. 569–588. ACM, New York, 2007.

[7] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, et al., "Mona: monadic second-order logic in practice," Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019, 1995.

[8] T. Kutsia, W. Schreiner, "LogicGuard abstract language". Technical Report 12-08, RISC, Johannes Kepler University, Linz, Austria, 2012.

[9] T. Kutsia, W. Schreiner, "Translation mechanism for the LogicGuard abstract language", Technical Report 12-11, RISC, Johannes Kepler University, Linz, Austria, 2012.

[10] T. Kutsia, W. Schreiner, "Verifying the soundness of resource analysis for LogicGuard monitors (revised version)", Technical Report 14-08, RISC, Johannes Kepler University, Linz, Austria, September 2014.

[11] LogicGuard II, http://www.risc.jku.at/projects/LogicGuard2, 2014.

[12] P. Naldurg, K. Sen, P. Thati, "A temporal logic based framework for intrusion detection," Formal Techniques for Networked and Distributed Systems (FORTE 2004), LNCS, vol. 3236, pp. 359–376. Springer, Berlin, 2004.

[13] W. Schreiner, T. Kutsia, "A resource analysis for LogicGuard monitors," Technical Report, RISC, Johannes Kepler University, Linz, Austria, December 2013.

[14] W. Schreiner, T. Kutsia, M. Krieger, B. Ahmad, H. Otto, M. Rumerstorfer, "Monitoring network traffic by predicate logic", Technical Report, RISC, Johannes Kepler University, Linz, Austria, September 2014.

[15] W. Schreiner, T. Kutsia, M. Krieger, B. Ahmad, H. Otto, M. Rumerstorfer, "The LogicGuard stream monitoring specification language: tutorial and reference manual", Technical Report, RISC, Johannes Kepler University, Linz, Austria, 2015 (to appear).

[16] G. Spandoudakis, C. Kloukindas, K. Mahbub: "The runtime monitoring frameworkof SERENITY," Security and Dependability for Ambient Intelligence, chapter 13, pp. 213–237, no. 13 in Information Security Series, Springer, 2009.