A Resource Analysis for LogicGuard Monitors*

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

December 17, 2013

Abstract

We describe a static analysis that we have devised to determine whether a specification expressed in the LogicGuard language gives rise to a monitor that can operate with a finite amount of resources, notably with finite *histories* of the streams that are monitored. This information can be passed to the runtime system of the monitor such that after every execution step the histories of the monitored streams can be appropriately pruned and the monitor can operate for an indefinite amount of time with a limited amount of memory. First, the analysis is presented for an abstract core language that monitors a single stream; the soundness of the analysis with respect to a formal operational semantics is verified in a companion paper. Second, we extend the analysis to an extended version of the language that can monitor multiple streams and supports the construction of virtual streams. This version already resembles the concrete LogicGuard language that is supported by the current prototype implementation. For the purpose of the analysis, several features of the language differ between the abstract and the concrete form; before the analysis can be implemented, the concrete language has to be correspondingly revised.

*This work was funded by the Austrian Research Promotion Agency (FFG) in the frame of the BRIDGE program by the project 832207 "LogicGuard: The Efficient Checking of Time-Quantified Logic Formulas with Applications in Computer Security".

Contents

1. Introduction

The goal of the LogicGuard project [4] is to demonstrate how classical first order predicate logic can be used to describe properties of network traffic that can be effectively monitored by a program generated from the specification. The core idea is to interpret the formulas over infinite sequences (streams) of messages where quantified variables denote positions in these sequences; given a sequence and a position, an atomic function can extract the corresponding message from the sequence. Furthermore, from a given external stream internal (virtual) streams can be constructed such that specifications can be expressed on these virtual streams on a higher level of abstraction than on the concrete stream. The core ideas have been presented in [6] and [5]; in [1], the syntax and semantics of (an early abstract form of) the specification language are given; in [2], the translation of a specification to an executable monitor is described. A prototype of the translator and of the corresponding runtime system have been implemented and are operational.

However, the current implementation still suffers from a severe deficiency: it assumes that the whole "history" of a stream is preserved, i.e., that all received messages are stored in memory; thus the memory requirements of a monitor continuously grow. In practice, however, we are only interested in monitors that operate for an indefinite amount of time within a bounded amount of memory. The purpose of this report is to fill this gap by presenting a static analysis that

1. is able to determine whether a given specification can be monitored with a finite amount of history (and that may consequently generate a warning/error message, if not) and that

2. generates corresponding information in an easily accessible form such that after each execution step the runtime system of the monitor may appropriately prune the histories of the streams on which it operates.

This report is organized in two parts: in Section 2, we present the main ideas of the analysis by an abstract core language, which is only a skeleton of the real language; in particular it only monitors a single stream and does not support the construction of virtual streams. In a companion paper [3], we use this language to formalize the operational semantics of the monitor execution and prove the soundness of the analysis presented in this report with respect to that semantics.

The rest of the report is dedicated to the description and analysis of a much more extended version of the abstract language which can be used to monitor multiple streams and supports the construction of virtual streams; this version is already quite similar to the actually implemented concrete language. In Section 3, this language is introduced and illustrated by examples. In Section 4, we extend the analysis of the core language to the extended language and show how the results of the analysis can be applied at runtime to the pruning of stream histories. Section 5 presents our conclusions and describes the next steps of our work. Appendices A–F present the complete definitions of the language, its analysis, and its application to history pruning.

2. The Core Language

The basic idea of our analysis is illustrated by the core language depicted in Figure 1. This language is interpreted over a single stream of messages carrying truth values. We assume that

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

Figure 1: The Core Language

a monitor M in this language is executed as follows: whenever a new message arrives on the stream, an instance $F[p/X]$ of the monitor body F is created where p denotes the position of the message in the stream. All instances are evaluated on every subsequently arriving message which may or may not let the instance evaluate to a definite truth value; whenever an instance evaluates to such a value, this instance is discarded from the set; the positions of instances with negative truth values are reported as "violations" of the monitor.

A formula F in a monitor instance is evaluated as follows:

- the predicate `@`X is immediately evaluated to the truth value of the message at position X of the stream (see below for further explanation);
- `~`F first evaluates F and then negates the result;
- F_1 `&&` F_2 first evaluates F_1 and, if the result is true, then also evaluates F_2;
- F_1 `/\` F_2 evaluates both F_1 and F_2 "in parallel" until the value of one subformula determines the value of the total formula;
- `forall` X `in` B_1`..`B_2 `:` F first determines the bounds of the position interval $[B_1, B_2]$; it then creates for every position p in the interval, as soon as the messages in the stream reach that position, an instance $F[p/X]$ of the formula body. All instances are evaluated on the subsequently arriving messages until all instances have been evaluated to "true" (and no more instances are to be generated) or some instance has been evaluated to "false".

We assume that the monitoring formula M is closed, i.e., every occurrence of a position variable X in it is bound by a quantifier `monitor` or `forall`. Since by the evaluation strategies for these quantifiers, a formula instance is created only when the messages have reached the position assigned to the quantified variable, every occurrence of predicate `@`X can be immediately evaluated without delay.

For instance, let us consider the monitor

```
monitor X: @X =>
  exists Y in X-1..X+2 : !@Y
```

which can be expressed in our core language as

	0	1	2	3	4	5	6	7	8	9	…
	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	…
$F[0]$	○										
$F[1]$	×	○									
$F[2]$		×	⊗	×							
$F[3]$				○							
$F[4]$				×	○						
$F[5]$					×	⊗	×	×			
$F[6]$						×	⊗	×	×		
$F[7]$							×	⊗	×		
$F[8]$									○		
$F[9]$										○	
⋮											

Figure 2: Monitor Instances

```
monitor X: ~(@X /\
  forall 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"). Execution proceeds by evaluating the instances $F[X]$ illustrated in Figure 2 which are created at those stream positions indicated by ○; the positions on which the corresponding `exists` formulas are evaluated are indicated by occurrences of ×. The only instances $F[X]$ that do not immediately evaluate to "true" are those where the stream holds at position X the value $\top$, i.e., in above example $F[1],F[2],F[4],F[5],F[6],F[7]$. All these instances evaluate the `exists` formula in the interval $[X-1,X+2]$ searching for a position Y where the stream holds value $\bot$; they terminate when the first such position is reached or when Y becomes $X+2$. Only for $F[5]$ this search remains unsuccessful such that position 5 is reported as a violation of the monitor.

We are interested in determining bounds for the resources used by the monitor, i.e., in particular in the following questions:

1. From the position where a monitor instance is created, how many "look-back" positions are required to evaluate the formula? This value determines the size of the "history" of past messages that have to be preserved in an implementation of the monitor.

2. How many instances can be active at the same time? This value determines the size that has to be reserved for the set of instances in the implementation of the monitor.

The basic idea for the analysis is a sort of "abstract interpretation" of the monitor where in a top-down fashion every position variable X is annotated as $X^{(l,u)}$ where the interval $[p+l,p+u]$ denotes those positions that the variables can have in relation to the position p of the "current" message of the stream; in a bottom up step, we then annotate every formula F with a pair (h,d) where h is (an upper bound of) the size of the "history" (the number of past messages) required

for the evaluation of F and d is (an upper bound of) the number of future messages that may be required such that the evaluation of F may be "delayed" by this number of steps.

In our example, the analysis yields the following annotation of variables (where the corresponding annotations for formulas are shown to the right):

```
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)
```

X denotes the position of the "current" message and is thus annotated as $(0,0)$, from this we can derive the range $(-1,2)$ for Y. The evaluation of `@`X proceeds immediately and is thus annotated as $(0,0)$; since the body of the `forall` formula can be thus immediately evaluated and evaluations are needed in the range $(-1,2)$, we determine an annotation $(1,2)$ indicating a history of 1 and a delay of 2. Combining this with the annotation $(0,0)$ for the formula `@`X we get an annotation $(1,2)$ for the conjunction and thus also for the negation and for the whole monitor. The monitor thus needs to preserve at most 1 message from the past (in addition to the current message) and at most 2 old instances (in addition to the current instance). In Figure 2, this is indicated by the fact that to the left of every circle (indicating the creation of a new instance) there is at most one marker (indicating the evaluation of the instance on messages from the past of the stream) and that to the top of every circle that there are at most two markers (indicating old instances that are still alive for evaluation with the current message).

The basic idea is formalized in Figures 3 and 4 by a rule system with three kinds of judgements:

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

We have $(h,d) \in \mathbb{N}^\infty \times \mathbb{N}^\infty$ where $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$; a value of ∞ indicates that the corresponding resource (history/instance set) cannot be bounded by the analysis. We have $e(X) \in \mathbb{Z}^\infty \times \mathbb{Z}^\infty$ where $\mathbb{Z}^\infty = \mathbb{Z} \cup \{\infty, -\infty\}$; a value of ∞ respectively $-\infty$ indicates that the position cannot be bounded from above respectively from below by the analysis. We have $(l,u) \in \mathbb{Z}^\infty \times \mathbb{Z}^\infty$; a value of ∞ for u indicates that the corresponding interval has no upper bound; a value of $-\infty$ for l indicates that the interval has no lower bound.

The effect of the delay of the evaluation of a formula can be illustrated by comparing the evaluation of two similar examples:

$$\vdash M : \mathbb{N}^\infty \times \mathbb{N}^\infty \quad \mathit{Environment} \vdash F : \mathbb{N}^\infty \times \mathbb{N}^\infty \quad \mathit{Environment} \vdash B : \mathbb{Z}^\infty \times \mathbb{Z}^\infty$$

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

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

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

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

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

$$e \vdash \texttt{0} : (0,0) \quad e \vdash \texttt{infinity} : (\infty,\infty) \quad \frac{[[\![X]\!] \notin \mathit{domain}(e)}{e \vdash X : (0,0)} \quad \frac{[[\![X]\!] \in \mathit{domain}(e)}{e \vdash X : e([[\![X]\!])}$$

$$\frac{e \vdash B : (l,u)}{e \vdash B\texttt{+}N : (l +^\infty [[\![N]\!], u +^\infty [[\![N]\!])} \qquad \frac{e \vdash B : (l,u)}{e \vdash B\texttt{-}N : (l -^\infty [[\![N]\!], u -^\infty [[\![N]\!])}$$

Figure 3: The Analysis of the Core Language

$$Environment := Variable \to \mathbb{Z}^\infty \times \mathbb{Z}^\infty$$
$$\mathbb{N}^\infty := \mathbb{N} \cup \{\infty\}, \mathbb{Z}^\infty := \mathbb{Z} \cup \{\infty, -\infty\}$$

$$max^\infty : \mathbb{N} \times \mathbb{N}^\infty \to \mathbb{N}^\infty$$
$$max^\infty(n_1, n_2) := \text{ if } n_2 = \infty \text{ then } \infty \text{ else } \max(n_1, n_2)$$

$$+^\infty : \mathbb{N}^\infty \times \mathbb{N}^\infty \to \mathbb{N}^\infty$$
$$n_1 +^\infty n_2 := \text{ if } n_1 = \infty \vee n_2 = \infty \text{ then } \infty \text{ else } n_1 + n_2$$

$$-^\infty : \mathbb{N}^\infty \times \mathbb{N} \to \mathbb{N}^\infty$$
$$n_1 -^\infty n_2 := \text{ if } n_1 = \infty \text{ then } \infty \text{ else } max(0, n_1 - n_2)$$

$$-^\infty : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$$
$$-^\infty i := \text{ if } i = \infty \text{ then } -\infty \text{ else if } i = -\infty \text{ then } \infty \text{ else } -i$$

$$\mathbb{N} : \mathbb{Z}^\infty \to \mathbb{N}^\infty$$
$$\mathbb{N}(i) := \text{ if } i = -\infty \vee i < 0 \text{ then } 0 \text{ else } i$$

Figure 4: The Semantic Algebras of the Analysis

1. In the first example, the evaluation of the inner quantifier `forall Y` is not delayed by the preceding evaluation of `@X`, thus the result of its analysis determines the result of the analysis of the whole formula:

```
monitor X^(0,0) :                          (3,5)
  @X &&                                    (0,0)
  forall Y^(0,5) in X..X+5 :               (3,5)
    forall Z^(-3,4) in Y-3..Y-1 :          (3,4)
      @Z                                   (0,0)
```

2. In the second example, the evaluation of `forall Y` is delayed by the preceding evaluation of `forall W` which needs three "future" messages; the evaluation of the whole formula thus needs three more messages from the "past", i.e., the history increases by three:

```
monitor X^(0,0) :                          (6,5)
  (forall W^(0,3) in X..X+3 : @W) &&       (0,3)
  forall Y^(0,5) in X..X+5 :               (3,5)
    forall Z^(-3,4) in Y-3..Y-1 :          (3,4)
      @Z                                   (0,0)
```

Above elaboration has presented the basic ideas for the analysis of the evaluation of the monitor in a simple core language. A forthcoming companion paper gives a proof of the soundness of the analysis based on a formal operational semantics of the monitor execution. In the following section, we generalize the basic ideas to a more complete version of the monitor language.

```
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,...,Tn)
     | 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
```

Figure 5: The Extended Language

3. The Extended Language

While the core language in the previous chapter demonstrates the essential ideas of the language and its analysis (and serves as the basis of the proof of the soundness of the analysis), that language is not of much practical interest. We are actually interested in a much richer version of the language which we are now going to describe.

3.1. Features

Figure 5 (also Appendix A) depicts a version of the abstract language which is considerably extended compared to the version presented in Section 2. In this language, we can write a specification such as

```
stream IP;
```

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

This specification specifies the monitoring of a stream *IP*; from this stream another stream *S* is constructed by combining every message from *IP* satisfying a predicate "Start" with all subsequent messages from the same sender (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* runs over every message in *S* 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 most essential features of the language and its interpretation model are as follows:

- A *stream* in this model carries a sequence of messages each of which has both 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.
- A specification is in general interpreted over *multiple* streams each of which is identified by a stream variable *XS*. Streams are introduced at the beginning of the specification; they are followed by a sequence of monitors which are evaluated over these streams.
- 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 specification language has multiple kinds of phrases denoting different kinds of entities and which may be bound to corresponding kinds of variables. There are those phrases that denote monitors (monitor formula *MF* and variable *XM*), those that denote truth values (formula *F* and variable *XF*), those that denote values (value term *TV* and variable *XV*), those that denote positions (position term *TP* and variable *XP*), and those that denote streams (stream term *TS* and variable *XS*).

  The corresponding binding constructs are `monitor XM=TM`, `formula XF=F`, `value XV=TV`, `position XP=TP`, `stream XS=TS`. Stream bindings may appear only at the beginning of the monitor at the top-level; this is also the only place where a phrase denoting a stream can appear (thus every stream can be uniquely identified by a global stream variable).

  The language supports external predicate constants *PC*, constants *FV* for value functions, and constants *FS* for stream functions; since we have to statically analyze positions, there are no external functions for position functions.

- 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.

- The domains provide various kinds of quantified phrases where the quantified variable runs over positions: quantified phrases denoting monitors (`position X:MX`), formulas (`forall X:F`, `exists X:F`), positions (`min X:F`, `max X:F`) and also values:

  - `num X:F` denotes the number of positions in the range denoted by constraint X in which formula F is true.
  - `complete combine[TV0,FV] X:TV1` constructs (in analogy to set construction $\{TV1|X\}$) a sequence of values $TV1_1,\ldots,TV_n$ which are folded by a base value TV_0 and a combining function FV into the result value

$$FV(FV(\ldots FV(FV(TV0,TV1_0),TV1_1)\ldots),TV1_n)$$

  There are also quantified constructs related to streams which will be explained below.

- A specification is in general interpreted over *external* streams that are introduced by declarations of the form `stream XS`. Additionally, *internal* (virtual) streams can be introduced by definitions of the form `stream XS=...`. These streams are constructed from other streams and only internally visible within a specification; they may be used to raise the layer of abstraction by combining messages from a lower-level stream to messages on a higher-level stream. For this purpose, multiple stream constructions are available:

  - `construct X:TV` constructs a stream of values by evaluating a value term TV over multiple bindings for a variable where the variable and the bindings are determined by a constraint X.
  - `partial combine[TV0,FV] X:TV1` constructs a stream by computing (similar to `construct`) a sequence of values $TV1_1,\ldots$ which are folded by a base value TV_0 and a combining function FV into the actual stream messages $TV0$, $FV(TV0,TV1_0)$, $FV(FV(TV0,TV1_0),TV1_1),\ldots$.
  - `construct X:TS` constructs a collection of streams by evaluating the stream term TS over multiple bindings for a variable where the variable and the bindings are determined by a constraint X). The multiple streams are merged into a single result stream by ordering the messages according to the times when they were constructed.

- A quantification constraint X has in general a form such as

$$XP \texttt{ in } XS \texttt{ with } TP_1 \texttt{ <= } XP \texttt{ <= } TP_2 \texttt{ satisfying } F_1 \texttt{ until } F_2$$

  where a bound position variable XP ranging over a stream XS is introduced. The values of the variable range over all positions denoted by the interval $[TP_1,TP_2]$ for which the formula F_1 yields true; the interval may end prematurely with the first position for which F_2 becomes true.

Intervals may be open on one or both sides (operation < rather than <=), they may be constrained by time bounds rather than position bounds (operations <T and <=T) and time bounds may be shifted by constants (bounds $TP+N$ and $TP-N$).

There may be multiple `satisfying` clauses and they may be separated by bindings introducing new local variables.

This version of the abstract specification language has been defined based on the experience with the actual implementation with the concrete language (which was in turn based on an earlier version of the abstract language). The new design reflects a better understanding of the necessary and desired features of the language; as a consequence, the abstract language differs in several aspects from the currently implemented concrete language.

Some of the differences are minor, but some are also crucial for the soundness of the analysis. In the following, we therefore outline the differences of the abstract language and the concrete language and discuss which aspects of the implementation of the concrete language have to be modified to make the analysis presented in this paper applicable to the actual implementation.

3.2. Differences to Concrete Language

We describe the differences between the abstract language and the concrete language in the order of importance:

- Most important, while the abstract language implements the evaluation of quantifiers by parallel evaluation of the instances, the concrete version of the language implements the evaluation of quantifiers sequentially; during the time that instance i of a quantified phrase is still under evaluation, the evaluation of instance $i+1$ will not be started.

  This difference is a crucial deficiency of the current implementation from the practical point of view as well as from the theoretical one[1]: the analysis in this paper is fundamentally based on the parallel evaluation of quantifier instances, i.e., it assumes that the evaluation of instance i is started when position i is reached in that stream that is associated to the position of the quantified position variable. To apply history pruning, thus the current implementation has to be revised.

  Furthermore, the current implementation of message access always reads from the underlying stream; the analysis of the abstract language assumes that messages identified by quantified position variables (i.e., by terms of form `XS@XP` and `XS#XP`) retrieve the messages from a separate cache in the environment of the monitor. To apply history pruning, thus the implementation has to be correspondingly revised.

- While the current implementation allows the definition of virtual streams such that a monitor can observe rather than an external stream an internal one, monitors that simultaneously operate on multiple streams are not yet adequately supported:

[1] Actually, an early version of the abstract language on which the concrete language was to be based used parallel evaluation; however, this feature was inadvertently lost during subsequent design states.

1. the `position` quantifier does not yet allow any constraints such as are supported by the other quantifiers; this makes monitors with more than one nested position variable practically unusable, since all pairs of positions have to be investigated and history pruning is not applicable to any of the nested streams.
2. there is no distinction between position comparison and time comparison; the only supported comparison operators are < and <= where position terms $TP{+}N$ and $TP{-}N$ denote positions that are identified by times; since the comparison of all positions is type-checked with respect to the streams on which they occur (i.e., only positions referring to the same stream may be compared), there is no time-based comparison of messages from different streams possible;
3. the handling of multiple `position` quantifiers is not adequate; if multiple such quantifiers occur, they are directly handled by the top-level execution mechanism which generates a new instance for every combination of the variables introduced by `position` quantifiers; the analysis assumes a handling of the `position` quantifier in a way that is analogous to the other quantifiers, i.e., by a recursive treatment of nested quantification.

To support monitors operating simultaneously over multiple streams, the implementation has to be correspondingly revised.

- The concrete language implements the logical connectives by sequential evaluation while the abstract implements it by parallel evaluation and uses the if-then-else construct to enforce sequential evaluation.

  This difference is of minor importance and reflects a shift of emphasis from sequential connectives to parallel ones.

- The concrete language supports different forms of constraints for different quantified constructs while the abstract language provides for all such constructs the same constraint domain `X`.

  This difference makes the current implementation less useful from the practical point of view (e.g. the `until` clause is only supported for the combining quantifiers, but not for the classical mathematical/logical quantifiers) and also more difficult to maintain. The implementation should be revised to match the generality of the abstract language.

- The abstract language supports at the top-level only stream and monitor definitions; the concrete language also allows other bindings. However, it should be not difficult to generalize the analysis to cover also this extension.

- In the concrete language, a position term can be qualified by a time difference N as `XP`$\pm$`N` while in the abstract language such qualifications are bound to the range descriptions in position variable quantification. This represents a conceptual hole in the concrete language, since a position term should denote a position while a position term with a time qualification represents (since multiple messages may have the same time) a range of positions. The difference is minor from the point of the analysis, but should be fixed in the concrete language.

- The concrete language is strongly typed. The domain of values supports undefined values with a test predicate for "undefinedness"; this allows the `min` and `max` quantifiers to return an undefined value if no position satisfies the desired constraint. Corresponding, the domain of truth values is three-valued supporting a truth value "undefined"; the various logical operations have been correspondingly extended.

  These differences have no influence on the analysis presented in this paper and can be preserved without change.

Among these differences, many are of less importance. Only the first issue is absolutely crucial; the implementation has to be revised to make the analysis described in the following section applicable. The second issue is important, if monitors observing multiple streams in a simultaneous fashion are to be supported.

4. Analysis

The analysis of the extended language is based on and generalizes the ideas sketched in Section 2 for the analysis of the core language. The major generalization is the appropriate consideration of *multiple* (external as well as virtual internal) streams in the analysis and the subsequent pruning of stream histories.

The consideration of the potential of pruning from the operational understanding of formula evaluation can be quite challenging; take for example the evaluation of the quantified formula

```
forall X in T :
  forall Y in S with X -T N <= Y:
    P(T@X,S@Y)
```

Here for every new message in stream T an instance of the formula binding X is created which looks at most N time units back into the history of S. If a message arrives on T, the history of T can be completely pruned and the history of S can be reduced to those messages that are not older than N time units.

On the contrary, take another specification

```
forall Y in S :
  forall X in T with X <= Y +T N:
    P(T@X,S@Y)
```

which is denotationally equivalent to the previous formula, since it considers the same pairs of messages from S and T. Operationally, however, the evaluation of this formula behaves quite differently: for every new message in stream S an instance of the formula binding Y is created which looks back into the full history of stream T. Thus, if a new message arrives in T, pruning T is not possible, because there may be still instances for Y to be created that need access to the history of T for the evaluation of the X formula. While pruning S on an arrival of a message in T is possible, it is not helpful, because already by every arrival of a message on S, the history of S can be completely pruned, i.e., no history need to be maintained for S at all.

As a consequence of these and other considerations, the forthcoming analysis will trigger the pruning of a stream only if a message arrives on the stream associated to the outermost quantifier

of the monitor (the *reference stream* of the monitor); for every position on that stream an instance evaluating the formula is generated which may require for its evaluation access to the histories of the streams referenced within the instance. The analysis determines those situations when the histories of these streams can be pruned by the arrival of a message on the reference stream. Due to this evaluation mechanism, the processing of the outermost quantifier itself does not require history on the reference stream (it may be however the case that the generated instance requires such a history because the reference stream is referenced within the instance itself).

We see that logically equivalent formulas may have different operational interpretation and potential for pruning; this creates the opportunity to investigate *formula optimizations* by transforming a formula into a logically equivalent but operationally more desirable form. However, for the moment we will contend ourselves with the analysis of the pruning potential for a given formula under the default evaluation mechanism. For this purpose, we have devised three steps:

1. **Abstract Interpretation:** a static analysis is performed that determines for every "engine" of a specification (the definition of a monitor or of a virtual stream) the information which portions of which streams are relevant for the evaluation of that engine; this information is expressed in a symbolic form.

2. **History Computation:** the symbolic information determined by the abstract interpretation is transformed into a two-dimensional numerical table indexed by pairs of streams s and t; each such entry describes the history requirement for stream s if a new message arrives on stream t.

3. **History Pruning:** at every step of the execution of the monitor (triggered by the arrival of messages on one or more external streams) the table generated by the history computation is consulted. Considering all messages that have arrived in the current execution step (messages observed on the external streams as well as messages generated from these observations on the internal streams), the information in the table is appropriately combined to determine which portion of the history has to be preserved on every external and internal stream: all messages beyond this portion are discarded.

In the following, we sketch these three steps in turn; the complete analysis is described in the appendices.

4.1. Abstract Interpretation

The core of the abstract interpretation is a judgement

$$e \vdash F : sr$$

which annotates every formula (in general every phrase) F with a set of stream ranges sr, i.e., with the portions required from every stream to determine the value of F. The judgement depends on an environment e that maps every free position variables in F to such a range.

Every stream range is described doubly

1. by a closed interval of message positions, and

2. by a closed interval of message times

both relative to the time/position of the "current" message in the *reference stream* of the monitor in which F occurs; as already stated, this reference stream is the stream associated to the outermost quantifier in which F occurs.

Each interval has form $[L,U]$ where L and U are symbolic expressions that capture the lower respectively upper bound of the denoted position respectively time range. The core of each such expression is a stream variable XS which denotes the position/time of the last message that has been received on the reference stream stream XS. If the expression denotes a position, it may be incremented/decremented by 1 using the operation ± 1 (a consequence of the fact that positions may be compared by strict inequality operator $<$); if it denotes a time, it may be incremented/decremented by a value N using the operation $\pm_{\mathrm{T}} N$ (a consequence of the fact that in time comparisons the specification language supports corresponding increments and decrements). Additionally, the minimum of two lower bounds may be determined by an operation MIN and the maximum of two upper bounds may be determined by an operation MAX (arising from safe over-approximations in the resource analysis). Special constants 0 and ∞ are used to denote left unbounded or right unbounded intervals.

For instance, for a reference stream XS the annotation pair

$$[XS+1,\infty],[XS,XS+_{\mathrm{T}}100]$$

indicates the portion of XS that starts at the position after the "current" position on that stream and is not constrained with respect to the upper position but with respect to the upper time; this time must be not farther than 100 time units from the "current" time. Likewise,

$$[0,XS],[0,\infty]$$

indicates the initial portion of the stream up to the "current" position with no constraint on time.

In the specification language positions are *typed* with respect to the streams to which they refer such that a position can be only compared with respect to another position in the same stream, as in

```
monitor M =
  position X in XS :
    forall Y in XS with Y <= X : P(XS@Y)
```

Consequently, the analysis of the monitor can limit only portions of the reference stream by a position interval; any other stream accessed by the monitor will thus be described by the unconstrained position interval $0\ldots\infty$. However, it is possible to relate messages to arbitrary other streams by the *times* at which they have occurred in their respective streams such as in

```
monitor M =
  position X in XS :
    forall Y in YS with XS-50 <=T Y : P(YS@Y)
```

The analysis can use this information to determine a corresponding time interval: in above monitor, the analysis derives for stream YS the annotation pair

$$[0,\infty],[XS-_{\mathrm{T}}50,XS]$$

which indicates that access to this stream is not constrained via positions in the reference stream *XS* but by the fact that the relevant messages *YS* are not farther in the past than 50 time units from the time of the current message in *XS*.

As a concrete example, take the specification

```
stream S;
stream T =
  construct X in S :
    combine[B,C] Y in IP with X-10 <=T Y <= X : F(IP@X,IP@Y) ;
monitor M =
  position P in T :
    position Q in T = min R in T with P <= R <=T P+20 : P(IP@R) :
    forall U in T with Q <= U until D(IP@U) : Q(IP@U) ;
```

where from an external stream S an internal stream T is constructed which is observed by monitor M; in this monitor, the reference stream for T is S and the reference stream for M is T.

The analysis correspondingly assigns to variable X the position/time ranges $[S,S],[S,S]$ and to Y the corresponding ranges $[0,S],[S-_{\mathrm{T}}10,S]$; the analysis of engine T then also yields the information that only stream S is relevant in range $[0,S],[S-_{\mathrm{T}}10,S]$;

Furthermore, the analysis assigns to P the ranges $[T,T],[T,T]$, to R and consequently to Q ranges $[T,\infty],[T,T+_{\mathrm{T}}20]$, and to U ranges $[T,\infty],[T,\infty]$; taking into account that the evaluation of Q precedes the evaluation of U, the analysis of engine M yields the information that only stream T is relevant in range $[0,\infty],[T-_{\mathrm{T}}20,\infty]$.

Appendix B defines the domains and operations used in the abstract interpretation; Appendix C lists the various kinds of judgements of which the analysis is composed; Appendix D gives the rules for deriving each kind of judgements.

4.2. History Computation

In the second step of the analysis, the symbolic information determined from the abstract interpretation is transformed into a two-dimensional table H indexed by streams s and t such that $H(s)(t)=(p,n)$ with $p,n\in\mathbb{N}\cup\{\infty\}$ is interpreted as follows: if in the current execution step of the monitor a message arrives at stream t, then

- p is an upper bound on the number of messages that must be preserved on stream s and
- n is an upper bound on the time that messages must be preserved on stream s.

These bounds do not denote absolute limits for pruning: if (apart from the message from stream t) also a message arrives on another stream t' with smaller values in $H(s)(t')$, then more pruning is possible.

The transformation from symbolic expression is conceptually easy: for every engine (monitor or virtual stream definition) e of the specification, we derive the position/time interval $L\ldots U$ associated to stream s; if $L=t-N$, i.e., the relevant portion of stream s starts at the position/time of the current message on t minus N, then

- if L indicates the lower bound of the position interval, then $p=t-L=t-(t-N)=N$,

- if L indicates the lower bound of the time interval, then $n = t - L = t - (t - N) = N$.

In other words, we have to perform a symbolic subtraction of the stream variable t and the term L. If the result can be bound from above by a constant N, this constant determines p respectively n; if no such constant bound N can be determined (because e.g., the term L contains another stream variable than t), the bound is taken as ∞. The overall result $H(s,t)$ is taken as the *maximum* of the bounds derived for s and t from all engines e.

It should be noted that we cannot simply perform an inverted analysis with respect to the upper bound U: even if we derive for a stream t the upper bound $U = s + N$, we cannot deduce that the relevant portion of stream t for the evaluation of s starts at the position/time of the current message of t minus N. The reason is that, as already explained at the beginning of this section, for instance the evaluation of a quantified formula

```
forall X in S :
  forall Y in S with X -T N <= Y:
    P(S@X,S@Y)
```

behaves operationally differently from the evaluation of the logically equivalent formula

```
forall Y in S :
  forall X in S with X <= Y +T N:
    P(S@X,S@Y)
```

In the first case (in which the analysis determines for S the relevant time interval $[S -_{\mathrm{T}} N, \infty]$), only those messages have to preserved in S that are not older than N time units; in the second case (in which the analysis determines for S the relevant time interval $[0, S +_{\mathrm{T}} N]$), all messages have to be preserved. We thus must not generally treat in our analysis an upper bound $s +_{\mathrm{T}} N$ on a stream t as the same as a lower bound $t +_{\mathrm{T}} N$ on a stream s.

As an example of history computation, take the specification

```
stream S;
stream T =
  construct X in S :
    combine[B,C] Y in IP with X-10 <=T Y <= X : F(IP@X,IP@Y) ;
monitor M =
  position P in T :
    position Q in T = min R in T with P <= R <=T P+20 : P(IP@R) :
    forall U in T with Q <= U until D(IP@U) : Q(IP@U) ;
```

for which we have derived in the previous section that

- for engine S only stream S is relevant in range $[0,S], [S -_{\mathrm{T}} 10, \infty]$ and that
- for engine M only stream T is relevant in range $[0,\infty], [T -_{\mathrm{T}} 20, \infty]$.

We thus can determine

- $H(S,S) = (\infty, 10)$
- $H(S,T) = (\infty, \infty)$

- $H(T,S) = (\infty, \infty)$
- $H(T,T) = (\infty, 20)$

Appendix E gives the concrete transformation algorithm.

4.3. History Pruning

The history table determined from the symbolic analysis can be applied at runtime to determine to which extent a stream s can be pruned in the following way: for every stream t on which a message is observed, the value of $H(s)(t)$ is determined; the minimum of these values (with respect to position and time) determines the history that has to be preserved at s (with respect to position and time). If p and n denote the minimum position and the minimum time requirement, then in s only those messages have to be preserved that have arisen in the last p positions and that are not older than n time units.

Taking the example specification

```
stream S;
stream T =
  construct X in S :
    combine[B,C] Y in IP with X-10 <=T Y <= X : F(IP@X,IP@Y) ;
monitor M =
  position P in T :
    position Q in T = min R in T with P <= R <=T P+20 : P(IP@R) :
    forall U in T with Q <= U until D(IP@U) : Q(IP@U) ;
```

for which in the previous section the table

- $H(S,S) = (\infty, 10)$
- $H(S,T) = (\infty, \infty)$
- $H(T,S) = (\infty, \infty)$
- $H(T,T) = (\infty, 20)$

was derived, we can deduce:

- On arrival of a message from stream S, all messages on S older than 10 time units can be pruned from the history;
- On arrival of a message on stream T, all messages on T older than 20 time units can be pruned from the history.

If messages arrive both on S and on T, then both pruning operations can be performed.

Appendix E gives the concrete pruning method.

5. Conclusions and Future Work

We have presented in this report a first version of a static analysis of specifications in the LogicGuard language which enables to apply at runtime *history pruning* on the streams over which a specification operates. This is a necessary prerequisite to put a limit on the memory consumption of the monitor and thus allow perpetually running monitors.

The core ideas were presented with the help of a core abstract language; based on an operational semantics of this core language, in an accompanying paper [3], the soundness of the analysis is proved.

We have also formalized in detail the analysis for an extended version of the abstract language. While this language resembles the currently implemented concrete language, it also differs in several crucial features that we have thoroughly described. Before actually implementing the analysis, the concrete language and its implementation have to be correspondingly revised.

Even with the analysis, the implementation will be quite inefficient for the evaluation of certain patterns of nested quantifiers where the work for the evaluation of inner quantifiers may overlap for different instances of the outer quantifier; also monotonicity properties of the special quantifiers `min`, `max`, and `num` have not been appropriately taken into account. We have already sketched an optimized runtime system that will allow different formula instances share intermediate results in order to avoid re-computations; the detailed elaboration and implementation of that runtime system will be our next goal.

Finally, we will turn our attention to a more elaborate static analysis with the goal of optimizing specifications by transforming specifications into logically equivalent but operationally more efficient ones.

References

[1] Temur Kutsia and Wolfgang Schreiner. LogicGuard Abstract Language. Technical Report 12-08, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2012. http://www.risc.jku.at/publications/download/risc_4552/2012-LG-abstract-language.pdf.

[2] Temur Kutsia and Wolfgang Schreiner. Translation Mechanism for the LogicGuard Abstract Language. Technical Report 12-11, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, October 2012. http://www.risc.jku.at/publications/download/risc_4601/2012-LG-translation-TR.pdf.

[3] Temur Kutsia and Wolfgang Schreiner. Verifying the Soundness of Resource Analysis for LogicGuard Monitors. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2013. To appear.

[4] LogicGuard — The Efficient Checking of Time-Quantified Logic Formulas with Applications in Computer Security, 2013. http://www.risc.jku.at/projects/LogicGuard/.

[5] Wolfgang Schreiner. Applying Predicate Logic to Monitoring Network Traffic. Invited talk at PAS 2013 - Second International Seminar on Program Verification, Automated Debugging and Symbolic Computation, Beijing, China, October 23-25, 2013.

[6] Wolfgang Schreiner. Generating Network Monitors from Logic Specifications. Invited Talk at FIT 2012, 10th International Conference on Frontiers of Information Technology, Islamabad, Pakistan, December 17-19, 2012.

A. The Extended Language

```
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,...,Tn)
     | 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
```

B. Types and Operations

```
// a pair of position range and time range
Range  := RangeCore × RangeCore

// position order
position: Range → RangeCore
position((p,t)) := p

// time order (implied by position order but not vice versa)
time: Range → RangeCore
time((p,t)) := t

// a range of stream positions/times
// * XS stands for current one in stream XS
```

```
// * +T/-T is only used if range denotes times
//   (might introduce separate types in implementation)
RangeCore ::= Lower..Upper
Lower  ::= XS | 0 | Lower+1 |
          MIN(Lower1,Lower2) | Lower +T N | Lower -T N
Upper  ::= XS | ∞ | Upper-1 |
         | MAX(Upper1,Upper2) | Upper +T N | Upper -T N

lower: RangeCore → Lower
lower(l..u) = l

upper: RangeCore → Upper
upper(l..u) = u

// mappings (partial functions) from positions/streams to ranges
PositionRange := PositionVariable →p Range
StreamRange   := StreamVariable →p Range

// update a position mapping by a new binding
update: PositionRange × PositionVariable × Range → PositionRange
update(pr,XP,r) := pr \ {(XP,r) | r ∈ Range} ∪ {(XP,r)}

// an environment (position ranges and streams for which
// some position variable has been declared)
Environment := PositionRange × ℙ(StreamVariable)

// empty environment
newEnvironment : Environment := (∅,∅)

// has environment a range for variable?
hasRange ⊆ Environment × PositionVariable
hasRange((pr,s),v) :⇔ v ∈ domain(pr)

// has environment some position variable?
anyStream ⊆ Environment
anyStream((pr,s)) :⇔ s ≠ ∅

// has environment some position variable for stream?
hasStream ⊆ Environment × StreamVariable
hasStream((pr,s),XS) :⇔ XS ∈ s

// give range of variable in environment
range: Environment × PositionVariable → PositionRange
range((pr,s),v) := pr(v)

// update environment by a new range for position variable in stream
update: Environment × PositionVariable × StreamVariable × Range → Environment
update((pr,s),XP,XS,r) := (update(pr,XP,r),s ∪ {XS})
```

```
// combine stream ranges to a safe approximation
combine: StreamRange × StreamRange → StreamRange
combine(sr1,sr2) :=
  let xs1 = domain(sr1) :
  let xs2 = domain(sr2) :
  { (x,r) |
     (x ∈ xs1 ∧ x ∉ xs2 ∧ r = sr1(x)) ∨
     (x ∈ xs2 ∧ x ∉ xs1 ∧ r = sr2(x)) ∨
     (x ∈ xs1 ∧ x ∈ xs2 ∧
        let r1 = sr1(x),r2 = sr2(x) :
        let p1 = position(r1), p2 = position(r2) :
        let t1 = time(r1), t2 = time(r2) :
        r = (min(lower(p1),lower(p2))..max(upper(p1),upper(p2)),
             min(lower(t1),lower(t2))..max(upper(t1),upper(t2)))) }

// the minimum of two lower bounds
min : Lower × Lower → Lower
min(l1,l2) :=
  if lesseq(l1,l2) then
    l1
  else if lesseq(l2,l1) then
    l2
  else
    MIN(l1,l2)

// if true,then l1 is less equal l2 (but not vice versa)
lesseq ⊆ Lower x Lower
lesseq(l1,l2) :⇔
  case l1 of
    XS1 :
      case l2 of
        XS2          : XS1=XS2 // must be true
        0            : false
        l+1          : lesseq(l1,l)
        MIN(la,lb)   : lesseq(l1,la) ∧ lesseq(l1,lb)
        l +T N       : lesseqal(l1,l)
        l -T N       : false
    0 :
      true
    l+1 :
      case l2 of
        XS2          : false // XS2 denotes unknown position
        0            : false
        l'+1         : lesseq(l,l')
        MIN(la,lb)   : lesseq(l1,la) ∧ lesseq(l1,lb)
        l' +T N'     : lesseqal(l1,l')
        l' -T N'     : false
    MIN(la,lb) :
      lesseq(la,l2) ∨ lesseq(lb,l2)
```

```
    l +T N :
      case l2 of
        l' +T N' : lesseq(l,l') ∧ N ≤ N'
        default  : false
    l -T N :
      case l2 of
        l' -T N' : lesseq(l,l') ∧ N' ≤ N
        default  : lesseq(l,l2)

// the maximum of two upper bounds
max : Upper × Upper → Upper
max(u1,u2) :=
  if lesseq(u1,u2) then
    u2
  else if lesseq(u2,u1) then
    u1
  else
    MAX(u1,u2)

// if true,then u1 is less equal u2 (but not vice versa)
lesseq ⊆ Upper x Upper
lesseq(u1,u2) :⇔
  case u1 of
    XS1 :
      case u2 of
        XS2          : XS1=XS2 // must be true
        ∞            : true
        u-1          : false
        MAX(la,lb)   : lesseq(u1,ua) ∨ lesseq(u1,ub)
        u +T N       : lesseqal(u1,u)
        u -T N       : false
    ∞ :
      case u2 of
        XS2          : false
        ∞            : true
        u-1          : false
        MAX(la,lb)   : lesseq(u1,ua) ∨ lesseq(u1,ub)
        u +T N       : lesseq(u1,u)
        u -T N       : false
    u-1 :
      case u2 of
        XS2          : false // XS2 denotes unknown position
        ∞            : true
        u'-1         : lesseq(u,u')
        MAX(ua,ub)   : lesseq(u1,ua) ∨ lesseq(u1,ub)
        u' +T N'     : lesseqal(u1,u')
        u' -T N'     : false
    MAX(ua,ub) :
      lesseq(ua,u2) ∧ lesseq(ub,u2)
```

```
    u +T N :
      case u2 of
        u' +T N' : lesseq(u,u') ∧ N ≤ N'
        default  : false
    u -T N :
      case u2 of
        u' -T N' : lesseq(u,u') ∧ N' ≤ N
        default  : lesseq(u,u2)

// shift sr2 because of evaluation after that of sr1
shift: StreamRange × StreamRange → StreamRange
shift(sr1,sr2) :=
  let xs1 = domain(sr1) :
  if xs1 = ∅ then
    sr2
  else
    // fold max/min over set of position/time bounds
    let x0 in xs1, p0=upper(position(sr1(x0))), t0=upper(time(sr1(x0))) :
    let maxP := (p0,max) {upper(position(sr1(x))) | x ∈ xs1\{x0}}
    let maxT := (t0,max) {upper(time(sr1(x)))     | x ∈ xs1\{x0}}
    let xs2 = domain(sr2) :
    { (x,r) | x ∈ xs2 ∧
        let r2 = sr2(x) :
        let p2 = position(r2), t2 = time(r2) :
        r = (cut(maxP,lower(p2))..max(maxP,upper(p2)),
             cut(maxT,lower(t2))..max(maxT,upper(t2))) }

// cut(XS+N1,XS-N2) = XS-(N1+N2)
cut: Upper × Lower → Lower
cut(u,l) :=
  let XS = base(l) :
  if XS = ⊥ then
    0
  else
    subtract(XS,l,u)

// base variable of lower bound (if any)
base: Lower → StreamVariable ∪ {⊥}
base(l) :=
  case l1 of
    XS   : XS
    0    : ⊥
    l+1  : base(l)
    MIN(la,lb) :
      let xa = base(la), xb = base(lb) :
      if xa = xb then xa else ⊥
    l +T N : base(l)
    l -T N : base(l)
```

```
// subtract(XS,XS-N1,XS+N2) = XS-(N1+N2)
// subtract(XS,XS-N1,YS+N2) = ⊥
substract: StreamVariable × Lower × Upper → Lower ∪ {⊥}
subtract(XS,l,u) :=
  case u of
    XS0 : if XS0 = XS then l else ⊥
    ∞   : 0
    u+1 : subtract(XS,l-1,u)
    MAX(ua,ub) : MIN(subtract(XS,l,ua), subtract(XS,l,ub))
    u +T N     : subtract(XS,l -T N, u)
    u -T N     : subtract(XS,l +T N, u)

// information for an engine
EngineVariable := StreamVariable ∪ MonitorVariable
EngineInfo := EngineVariable →p StreamRange

// add one engine information to another
// in the analysis we actually assume that no two
// engines have the same name
add: EngineInfo x EngineInfo → EngineInfo
add(info1,info2) := (info1 \ { (x,i) | x in domain(info2) }) ∪ info2
```

C. Judgements

```
// determine engine information for M
⊢ M : EngineInfo

// determine engine information for MS
⊢ MS : EngineInfo

// determine engine information for MF
⊢ MF : StreamRange

// determine ranges in which streams are touched in MX
Environment ⊢ MX : StreamRange

// determine ranges in which streams are touched in F
Environment ⊢ F : StreamRange

// determine ranges of streams that were touched in term evaluation
Environment ⊢ T : StreamRange

// determine position range and ranges of streams that were touched
Environment ⊢ TP : Range × StreamRange

// determine ranges of streams that were touched
Environment ⊢ TV : StreamRange
```

```
// determine ranges of streams that were touched
Environment ⊢ TS : StreamRange

// update position ranges and determine ranges of streams that were touched
Environment ⊢ B : PositionRange × StreamRange

// determine variable and its range and ranges of streams that were touched
// and update environment by local bindings that were generated
Environment ⊢ X :
  PositionVariable × StreamVariable × Range × StreamRange × Environment

// determine positon range and ranges of streams that were touched
Environment × StreamVariable ⊢ R : Range × StreamRange

// adjust lower/upper bounds of range depending on position predicate
// use denoted default range if predicate constrains only time, not position
Range × Lower × Lower ⊢ P : Lower × Lower
Range × Upper × Upper ⊢ P : Upper × Upper

// determine position range and ranges of streams that were touched
Environment ⊢ Q : Range × StreamRange

// update environment and determine ranges of streams that were touched
Environment ⊢ C : Environment × StreamRange

// determine ranges of streams that were touched
Environment ⊢ U : StreamRange
```

D. Rules

```
Monitors
========
M  ::= MS MF

// determine engine information for M
⊢ M : EngineInfo

⊢ MS : info0
⊢ MF : info1
info = add(info0,info1)
-----------------------
⊢ MS MF : info

Monitor Streams
===============
MS ::= _ | stream XS ; MS | stream XS = TS ; MS
```

```
// determine engine information for MS
⊢ MS : EngineInfo

⊢ _ : ∅

⊢ MS : info
-----------------------
⊢ stream XS ; MS : info

∅ ⊢ TS : sr
⊢ MS : info
info' = add({XS,sr},info)
-----------------------------
⊢ stream XS = TS ; MS : info'

Monitor Formulas
================
MF ::= _ | monitor XM = MX ; MF

// determine engine information for MF
⊢ MF : EngineInfo

⊢ _ : ∅

∅ ⊢ MX : sr
⊢ MF: info
info' = add({XM,sr},info)
-----------------------------
⊢ monitor XM = MX ; MF : info'

Monitor Variables
=================
MX ::= F | position X : MX

// determine ranges in which streams are touched in MX
Environment ⊢ MX : StreamRange

e ⊢ F : sr
----------
e ⊢ F : sr

e ⊢ X : (XP,XS,r,sr,e')
e' ⊢ MX : sr'
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
-------------------------------------------------
e ⊢ position X : MX : sr''

Formulas
========
```

```
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

// determine ranges in which streams are touched in F
Environment ⊢ F : StreamRange

e ⊢ XF : ∅

e  ⊢ B : (e',sr')
e' ⊢ F : sr''
sr = combine(sr',shift(sr',sr''))
---------------------------------
e ⊢ (B : F) : sr

e ⊢ T1 : sr1
...
e ⊢ Tn : srn
sr = combine(sr1,shift(sr1,
       combine(...,shift(...,combine(...,shift(...,srn))...))))
-------------------------------------------------------------
e ⊢ PC(T1,...,Tn) : sr

e ⊢ true : ∅

e ⊢ false : ∅

e ⊢ F1 : sr1
e ⊢ F2 : sr2
e ⊢ F3 : sr3
sr = combine(sr1, combine(shift(sr1,sr2), shift(sr1,sr3)))
----------------------------------------------------------
e ⊢ if F1 then F2 else F3 : sr

e ⊢ F : sr
------------
e ⊢ ~F : sr

e ⊢ F1 : sr1
e ⊢ F2 : sr2
sr = combine(sr1,sr2)
---------------------
e ⊢ F1 /\ F2 : sr

e ⊢ F1 : sr1
e ⊢ F2 : sr2
sr = combine(sr1,sr2)
---------------------
```

```
e ⊢ F1 \/ F2 : sr

e ⊢ F1 : sr1
e ⊢ F2 : sr2
sr = combine(sr1,sr2)
---------------------
e ⊢ F1 => F2 : sr

e ⊢ F1 : sr1
e ⊢ F2 : sr2
sr = combine(sr1,sr2)
---------------------
e ⊢ F1 <=> F2 : sr

e ⊢ X : (XP,XS,r,sr,e')
e' ⊢ F : sr'
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
-------------------------------------------------
e ⊢ (forall X : F) : sr''

e ⊢ X : (XP,XS,r,sr,e')
e' ⊢ F : sr'
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
-------------------------------------------------
e ⊢ (exists X : F) : sr''

Terms
=====
T ::= TP | TV | TS

// determine ranges of streams that were touched in term evaluation
Environment ⊢ T : StreamRange

e ⊢ TP : (r,rs)
---------------
e ⊢ TP : rs

e ⊢ TV : rs
-----------
e ⊢ TV : rs

e ⊢ TS : rs
-----------
e ⊢ TS : rs

Position Terms
==============
TP ::= XP | B : TP | max X : F | min X : F
```

```
// determine position range and ranges of streams that were touched
Environment ⊢ TP : Range × StreamRange

hasRange(e,XP)
r = range(e,XP)
---------------
e ⊢ XP : (r,∅)

e  ⊢ B : (e',sr')
e' ⊢ TP : (r,sr'')
sr = combine(sr',shift(sr',sr''))
---------------------------------
e ⊢ (B : TP) : (r,sr)

e ⊢ X : (XP,XS,r,sr,e')
e' ⊢ F : sr'
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
------------------------------
e ⊢ (max X : F) : (r,sr'')

e ⊢ X : (XP,XS,r,sr,e')
e' ⊢ F : sr'
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
------------------------------------------------
e ⊢ (min X : F) : (r,sr'')

Value Terms
===========
TV ::= XV | THIS | NEXT | B : TV | FV(T1,...,Tn) | XS@TP | XS#TP
     | num X : F | complete combine[TV0,FV] X : TV1

// determine ranges of streams that were touched
Environment ⊢ TV : StreamRange

e ⊢ XV : ∅

e ⊢ THIS : ∅

e ⊢ NEXT : ∅

e  ⊢ B : (e',sr')
e' ⊢ TV : sr''
sr = combine(sr',shift(sr',sr''))
---------------------------------
e ⊢ (B : TV) : sr

e ⊢ T1 : sr1
...
e ⊢ Tn : srn
```

```
sr = combine(sr1,shift(sr1,
       combine(...,shift(...,combine(...,shift(...,srn)...)))))
-------------------------------------------------------------
e ⊢ FV(T1,...,Tn) : sr

e ⊢ TP : (r,sr)
---------------
e ⊢ XS@TP : sr

e ⊢ TP : (r,sr)
---------------
e ⊢ XS#TP : sr

e ⊢ X : (XP,XS,r,sr,e')
e' ⊢ F : sr'
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
-------------------------------------------------
e ⊢ (num X : F) : sr''

e ⊢ TV0 : sr0
e ⊢ X : (XP,XS,r,sr1,e0)
e0 ⊢ TV1 : sr2
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
-------------------------------------------------
e ⊢ (complete combine[TV0,FV] X : TV1): sr

Stream Terms
============
TS ::= XS | B : TS | FS(T1,...,Tn)
     | construct X : TV | build X : TS
     | partial combine[TV0,FV] X : TV1

// determine ranges of streams that were touched
Environment ⊢ TS : StreamRange

e ⊢ XS : ∅

e  ⊢ B : (e',sr')
e' ⊢ TS : sr''
sr = combine(sr',shift(sr',sr''))
---------------------------------
e ⊢ (B : TS) : sr

e ⊢ T1 : sr1
...
e ⊢ Tn : srn
sr = combine(sr1,shift(sr1,
       combine(...,shift(...,combine(...,shift(...,srn)...)))))
-------------------------------------------------------------
```

```
e ⊢ FS(T1,...,Tn) : sr

e ⊢ X : (XP,XS,r,sr,e')
e' ⊢ TV : sr'
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
-------------------------------------------------
e ⊢ (construct X : TV): sr''

e ⊢ X : (XP,XS,r,sr,e')
e' ⊢ TS : sr'
sr'' = combine(sr,shift(sr,combine({(XS,r)},sr')))
-------------------------------------------------
e ⊢ (build X : TS): sr''

e ⊢ TV0 : sr0
e ⊢ X : (XP,XS,r,sr1,e0)
e0 ⊢ TV1 : sr2
sr = combine(sr0,shift(sr0,combine(sr1,shift(sr1,combine({(XS,r)},sr2)))))
-------------------------------------------------------------------------
e ⊢ (partial combine[TV0,FV] X : TV1): sr

Binders
=======
B ::= formula XF = F | position XP IN XS = TP | value XV = TV

// update position ranges and determine ranges of streams that were touched
Environment ⊢ B : Environment × StreamRange

e ⊢ F : sr
---------------------------
e ⊢ formula XF = F : (e,sr)

e ⊢ TP : (r,sr)
e' = update(pr,XP,XS,r)
-----------------------------------
e ⊢ position XP IN XS = TP : (e',sr)

e ⊢ TV : sr
--------------------------
e ⊢ value XV = TV : (e,sr)

Variables
=========
X ::= XP in XS R C U

// determine variable and its range and ranges of streams that were touched
// and update environment by local bindings that were generated
Environment ⊢ X :
  PositionVariable × StreamVariable × Range × StreamRange × Environment
```

```
(e,XS) ⊢ R : (r,sr0)
e0 = update(e,XP,XS,r)
e0 ⊢ C : (e1,sr1)
e0 ⊢ U : sr2
sr = combine(sr0,shift(sr0,combine(sr1,shift(sr1,sr2))))
-------------------------------------------------------
e ⊢ XP in XS R C U : (XP,XS,r,sr,e)

Ranges
======
R ::= _ | with Q P XP | with XP P TQ | with Q1 P1 XP P2 Q2

// determine positon range and ranges of streams that were touched
Environment × StreamVariable ⊢ R : Range × StreamRange

p = if anyStream(e) then 0..∞ else XS..XS
t = if anyStream(e) then 0..∞ else XS..XS
r = (p,t)
r' = (r,r)
-----------------------------------------
(e,XS) ⊢ _ : (r',∅)

e ⊢ Q : (r0,sr)
p = if anyStream(e) then 0..∞ else XS..XS
r = if anyStream(e) then 0..∞ else XS..XS
(r,lower(position(r0)),lower(time(r0))) ⊢ P : (lp,lt)
r' = (lp..∞,lt..∞)
-------------------------------------------------------
(e,XS) ⊢ with Q P XP : (r',sr)

e ⊢ Q : (r0,sr)
p = if anyStream(e) then 0..∞ else XS..XS
t = if anyStream(e) then 0..∞ else XS..XS
r = (p,t)
(r,upper(position(r0)),upper(time(r0))) ⊢ P : (up,ut)
r' = (0..up,0..ut)
-------------------------------------------------------
(e,XS) ⊢ with XP P Q : (r',sr)

e ⊢ Q1  : (r1,sr1)
e ⊢ Q2  : (r2,sr2)
p = if anyStream(e) then 0..∞ else XS..XS
t = if anyStream(e) then 0..∞ else XS..XS
r = (p,t)
(r,lower(position(r1)),lower(time(r1))) ⊢ P1 : (lp,lt)
(r,upper(position(r2)),upper(time(r2))) ⊢ P2 : (up,ut)
r' = (lp..up,lt..ut)
sr' = combine(sr1,shift(sr1,sr2))
```

```
-----------------------------------------------------
(e,XS) ⊢ with Q1 P1 XP P2 Q2 : (r',sr')

Position Predicates
===================
P   ::= < | <= | <T | <=T

// adjust lower/upper bounds of range depending on position predicate
// use denoted default range if predicate constrains only time, not position
Range × Lower × Lower ⊢ P : Lower × Lower
Range × Upper × Upper ⊢ P : Upper × Upper

// adjusting position does not imply adjusting time
(r,lp,lt) ⊢ <  : (lp+1,lt)
(r,lp,lt) ⊢ <= : (lp,lt)

// adjusting position does not imply adjusting time
(r,up,ut) ⊢ <  : (up-1,ut)
(r,up,ut) ⊢ <= : (up,ut)

// while time is adjusted, position remains unconstrained
(r,lp,lt) ⊢ <T  : (lower(r),lt+1)
(r,lp,lt) ⊢ <=T : (lower(r),lt)

// while time is adjusted, position remains unconstrained
(r,up,ut) ⊢ <T  : (upper(r),ut-1)
(r,up,ut) ⊢ <=T : (upper(r),ut)

Position Bounds
===============
Q ::= TP | TP + N | TP - N

// determine position range and ranges of streams that were touched
Environment ⊢ Q : Range × StreamRange

e ⊢ TP : (r,sr)
---------------
e ⊢ TP : (r,sr)

e ⊢ TP : (r,sr)
lp = lower(position(r))
up = upper(position(r))
lt = lower(time(r)) +T N
ut = upper(time(r)) +T N
N > 0
-----------------------------------
e ⊢ TP + N : ((lp+1..∞,lt..ut),sr)

e ⊢ TP : (r,sr)
```

```
lp = lower(position(r))
up = upper(position(r))
lt = lower(time(r)) -T N
ut = upper(time(r)) -T N
N > 0
-----------------------
e ⊢ TP - N : ((0..up-1,lt..ut),sr)

Constraints
===========
C  ::= _ | satisfying F C | B C

// update position ranges and determine ranges of streams that were touched
Environment ⊢ C : Environment × StreamRange

e ⊢ _ : (e,∅)

e ⊢ F : sr0
e ⊢ C: (e',sr1)
sr = combine(sr0,shift(sr0,sr1))
--------------------------------
e ⊢ satisfying F C : (e',sr)

e  ⊢ B : (e0,sr0)
e0 ⊢ C : (e',sr1)
sr = combine(sr0,shift(sr0,sr1))
--------------------------------
e ⊢ B C : (e',sr)

Until
=====
U ::= _ | until F

// determine ranges of streams that were touched
Environment ⊢ U : StreamRange

e ⊢ _ : ∅

e ⊢ F: sr
---------------
e ⊢ until F : sr
```

E. History Computation

```
// number and age of messages to be kept in stream
History := (ℕ ∪ {∞}) × (ℕ ∪ {∞})
```

```
// -------------------------
// determine stream histories
// -------------------------
Histories(S,E):
  Input:
    S ∈ P(StreamVariable)
      * the set of streams used
    E ∈ EngineInfo
      * domain(E) is the set of all engines used by the monitor.
      * domain(E(e)) is the set of all streams touched by engine e.
      * E(e)(s) is the range in which engine e touches stream s,
        i.e. a pair of position range and time interval.
  Output:
    H ∈ S → (S → History)
      * H(s)(t) = (p,n) is interpreted as follows:
        on an t-step (a message arrives on stream t), for stream s
        - p is (an upper bound on) the number of messages that must be preserved,
        - n is (an upper bound on) the time that messages must be preserved.
        Consequently, we need to preserve from the last p messages
        only those that are not older than time n.

  // start with minimum history
  for s in S do
    for t in S do
      H(s)(t) := ZeroHistory
    end
  end

  // increase history as required by every engine e
  for e in domain(E) do

    // the stream range for e
    r := E(e)

    // the streams touched by e
    V := domain(r)

    // assume maximum history for e
    for s in S do
      for t in S do
        H'(s)(t) := AllHistory
      end
    end

    // constrain history as determined by equations
    for s in V do
      for t in V do
        // t-N <=T s: at an t-step, preserve on s messages up to age N
        H'(s)(t) := MinimumHistory(H'(s)(t), LowerHistory(r(s),t))
```

```
        // not correct for the currently considered execution model
        // t <=T s+N: at an t-step, preserve on s messages up to age N
        // H'(s)(t) := MinimumHistory(H'(s)(t), UpperHistory(r(t),s))
      end
    end

    // combine history information
    for s in S do
      for t in S do
        H(s)(t) := MaximumHistory(H(s)(t), H'(s)(t))
      end
    end

  end

  return H
end History

// ----------------------
// history constants
// ----------------------
ZeroHistory ∈ History
ZeroHistory := (0,0)

AllHistory ∈ History
AllHistory := (∞,∞)

// -------------------------
// determine minimum history
// -------------------------
MinimumHistory(h1,h2):
  Input:
    h1,h2 ∈ History
  Output
    h ∈ History
  (p1,n1) := h1
  (p2,n2) := h2
  p := if p1 = ∞ then p2 else if p2 = ∞ then p1 else min(p1,p2)
  n := if n1 = ∞ then n2 else if n2 = ∞ then n1 else min(n1,n2)
  return (p,n)
end MinimumHistory

// -------------------------
// determine maximum history
// -------------------------
MaximumHistory(h1,h2):
  Input:
    h1,h2 ∈ History
```

```
  Output
    h ∈ History
  (p1,n1) := h1
  (p2,n2) := h2
  p := if p1 = ∞ ∨ p2 = ∞ then ∞ else max(p1,p2)
  n := if n1 = ∞ ∨ n2 = ∞ then ∞ else max(n1,n2)
  return (p,n)
end MaximumHistory

// ------------------------------------------------------------------
// determine from lower bound of range r of some stream
// the history required for that stream by a t-step
// ------------------------------------------------------------------
LowerHistory(r,t)
  Input
    r ∈ Range
    t ∈ StreamVariable
  Output
    h ∈ History
  p := lower(position(r))
  n := lower(time(r))
  return (LowerPosition(p,t), LowerTime(n,t))
end LowerHistory

// ----------------------------------------------------------
// determine (upper bound of) cutoff position for some stream
// from lower bound l of that stream and stream t
// (i.e., the value of "t-l", if this value can be denoted
// by a number, and ∞, otherwise)
// ----------------------------------------------------------
LowerPosition(l,t):
  Input
    l ∈ Lower
    t ∈ StreamVariable
  Output
    c ∈ ℕ ∪ {∞}
  case l of
    x:
      if x = t then c := 0 else c := ∞ end
    0:
      c := ∞:
    l0+1:
      c := LowerPosition(l0,t)
      if c ≠ ∞ then c := max(0,c-1) end
    MIN(l1,l2):
      c1 := LowerPosition(l1,t)
      c2 := LowerPosition(l2,t)
      if c1 = ∞ ∨ c2 = ∞ then
        c := ∞
```

```
      else
        c := max(c1,c2)
      end
    MAX(l1,l2):
      c1 := LowerPosition(l1,t)
      c2 := LowerPosition(l2,t)
      if c1 = ∞ then
        c := c2
      else if c2 = ∞ then
        c := c1
      else
        c := min(c1,c2)
      end
    l0 +T N:
      c := ∞
    l0 -T N
      c := ∞
  end
  return c
end LowerPosition

// ------------------------------------------------------
// determine (upper bound of) cutoff time for some stream
// from lower bound l of that stream and stream t
// (i.e., the value of "t-l", if this value can be denoted
// by a number, and ∞, otherwise)
// ------------------------------------------------------
LowerTime(l,t):
  Input
    l ∈ Lower
    t ∈ StreamVariable
  Output
    c ∈ ℕ ∪ {∞}
  case l of
    x:
      if x = t then c := 0 else c := ∞ end
    0:
      c := ∞:
    l0+1:
      c := ∞:
    MIN(l1,l2):
      c1 := LowerTime(l1,t)
      c2 := LowerTime(l2,t)
      if c1 = ∞ ∨ c2 = ∞ then
        c := ∞
      else
        c := max(c1,c2)
      end
    MAX(l1,l2):
```

```
      c1 := LowerTime(l1,t)
      c2 := LowerTime(l2,t)
      if c1 = ∞ then
        c := c2
      else if c2 = ∞ then
        c := c1
      else
        c := min(c1,c2)
      end
    l0 +T N:
      c := LowerTime(l0,t)
      if c ≠ ∞ then
        c := max(0,c-N)
      end
    l0 -T N
      c := LowerTime(l0,t)
      if c ≠ ∞ then
        c := c+N
      end
  end
  return c
end LowerTime

// ------------------------------------------------
// determine from upper bound of some stream range r
// the history required by an t-step for stream t
// ------------------------------------------------
UpperHistory(r,t)
  Input
    r ∈ Range
    t ∈ StreamVariable
  Output
    h ∈ History
  p := upper(position(r))
  n := upper(time(r))
  return (UpperPosition(p,t), UpperTime(n,t))
end UpperHistory

// ------------------------------------------------------
// determine (upper bound) of cutoff position for stream s
// from upper bound u of some stream range
// (i.e., the value of "u-t", if this value can be
// denoted by a number, and ∞, otherwise)
// ------------------------------------------------------
UpperPosition(l,t):
  Input
    u ∈ Upper
    t ∈ StreamVariable
  Output
```

```
    c ∈ ℕ ∪ {∞}
  case u of
    x:
      if x = t then c := 0 else c := ∞ end
    ∞:
      c := ∞:
    u0-1:
      c := Position(u0,t)
      if c ≠ ∞ then c := max(0,c-1) end
    MIN(u1,u2):
      c1 := UpperPosition(u1,t)
      c2 := UpperPosition(u2,t)
      if c1 = ∞ then
        c := c2
      else if c2 = ∞ then
        c := c1
      else
        c := min(c1,c2)
    MAX(u1,u2):
      c1 := UpperPosition(u1,t)
      c2 := UpperPosition(u2,t)
      if c1 = ∞ ∨ c2 = ∞ then
        c := ∞
      else
        c := max(c1,c2)
      end
    u0 +T N:
      c := ∞
    u0 -T N
      c := ∞
  end
  return c
end UpperPosition

// ------------------------------------------------
// determine (upper bound) of cutoff time for stream t
// from upper bound u of some stream range
// (i.e., the value of "u-t", if this value can be
// denoted by a number, and ∞, otherwise)
// ------------------------------------------------
UpperTime(u,t):
  Input
    u ∈ Upper
    t ∈ StreamVariable
  Output
    c ∈ ℕ ∪ {∞}
  case l of
    x:
      if x = t then c := 0 else c := ∞ end
```

```
    ∞:
      c := ∞:
    u0-1:
      c := ∞:
    MIN(u1,u2):
      c1 := UpperTime(u1,t)
      c2 := UpperTime(u2,t)
      if c1 = ∞ then
        c := c2
      else if c2 = ∞ then
        c := c1
      else
        c := min(c1,c2)
    MAX(u1,u2):
      c1 := UpperTime(u1,t)
      c2 := UpperTime(u2,t)
      if c1 = ∞ ∨ c2 = ∞ then
        c := ∞
      else
        c := max(c1,c2)
      end
    u0 +T N:
      c := UpperTime(u0,t)
      if c ≠ ∞ then c := c+N end
    u0 -T N
      c := UpperTime(u0,t)
      if c ≠ ∞ then c := max(0,c-N) end
  end
  return c
end UpperTime
```

F. History Pruning

```
// ------------------------------------------------
// on arrival of messages in T, prune streams in S
// according to history requirements H
// ------------------------------------------------
Prune(S,H,T)
  Input
    S ∈ P(StreamVariable)
      the set of all streams
    H ∈ S → (S → History)
      H(s)(t) ... the history required for stream s
      if a message arrives at stream t
    T ∈ P(StreamVariable)
      the set of (external/internal) streams on which
      some message has arrived (been generated) in the last step
```

```
  Effect
    prunes the histories of all streams in S

   for s in S do
     // fold history
     h := AllHistory
     for t in T do
       h := MinimumHistory(h, H(s)(t))
     end

     // apply result
     p := position(h)
     n := time(h)

     // keep in the history of s only those messages
     /7 * that have arised in the last p positions and
     // * that are not older than n time ticks
   end
end Prune
```