LogicGuard Abstract Language*

Temur Kutsia Wolfgang Schreiner

RISC, Johannes Kepler University Linz

{schreine,kutsia}@risc.jku.at

Abstract

The LogicGuard project aims at developing a specification and verification formalism for runtime network monitoring based on predicate logic. This report documents the initial steps in this development, describing the syntax and semantics of the LogicGuard abstract language LGAL.

1 Introduction

Runtime verification is an approach to the analysis of computing systems that bridges the gap between ad-hoc informal testing and fully formal verification. Roughly, it extracts some traces of runs of executable systems and applies verification methods to those traces. This approach is more lightweight than fully formal verification, because it analyzes only certain traces against specific properties. There is no need to model and verify the entire system. On the other hand, it is more formal than testing, because it specifies and verifies the desired properties formally. In this way, runtime verification combines advantages of scaling up relatively well yet being formally rigorous: Covering relatively small part of the system reduces the complexity of the approach and contributes to its scalability, while dealing formally with those parts brings rigor and confidence in the results. On the back side, it might consume systems resources and reduce its performance, especially if the verification component is a part of the system itself.

Instead of static analysis of a system before its execution, runtime verification, as the name also indicates, performs dynamic, runtime system analysis. This becomes particularly important for the systems that can not be completely verified statically. In the process of runtime verification, the system is being continuously monitored to see whether the desired properties hold. In case of their violation, certain predefined steps are performed, such as, for instance, issuing a warning, logging violation details, correcting errors, etc.

In runtime verification, to establish a correctness result in a state of the runtime system based on the past or future states, one can not use methods that work for an isolated single state (such as, e.g., from [15]). We need a special tool, the “monitor”, which checks the sequence of states and reacts on the violation of the desired property. Such a tool can either be constructed by the user (imperative approach), or can be generated automatically from a specification of the correctness property in a suitable logic (declarative approach). The former one is more error-prone, since the user has to construct programs and prove its properties manually. In the declarative approach, the basic techniques for the generation of monitors from specifications of system runs originate from the area of model checking [11]. However, unlike model checking, in runtime specification there is no need to prove the correctness result for all possible executions of the system model: It is enough to specialize it to only one such path, the actual system execution.

Various formalisms for the specification of system runs have been developed and used in the context of model checking and runtime specification. We do not go into the details of their

*The project “LogicGuard: The Efficient Checking of Time-Quantified Logic Formulas with Applications in Computer Security” is sponsored by the FFG BRIDGE program, project No. 832207.

description here, but just mention several prominent representatives: Languages over infinitely long words, called ω-regular languages, and automata that recognize them [8, 9, 23]; star-free ω-regular languages and linear temporal logic LTL [20, 13]; extensions of LTL: ETL [24], FTL [1], μTL [3, 22], etc. Based on these formalisms, various runtime verification frameworks and systems have been developed: Eagle [5, 4], its rule-based version RuleR [6, 7], MOP (Monitoring-Oriented Programming) [10, 19], EVEREST [21], MaC [16], PathExplorer [14]. Temporal Rover [12], etc.

The goal of the LogicGuard project is to investigate to what extent classical predicate logic formulas are suitable as the basis for the specification and efficient runtime verification of system runs. In comparison to the above mentioned formalisms, predicate logic has a relatively intuitive semantics, which helps to describe complex relationships in a natural way, keeping the gap between an interesting property and its specification small. To address the problem of efficiency, certain restricted fragment of predicate logic will be considered. A prototype implementation will help to carry out systematic experiments and to indicate the directions of improvements and optimizations. The specific focus of the project is on computer and network security, concentrating on predicate logic specifications of security properties of network traffic. Specification formulas will be interpreted over streams of messages. Furthermore, a prototype implementation of the translation mechanism is planned, which is supposed to automatically generate runtime monitors from the specifications.

In this paper we describe the initial steps in this development, describing the syntax and semantics of the fragment of predicate logic we plan to use for runtime network monitoring. We call this abstract language the LogicGuard Abstract Language (LGAL). Its has four-valued semantics, which corresponds to our intuition behind monitoring: A property being monitored over the given stream can be either true, false, or the monitoring can be interrupted because of an error. These are, so to say, the "definite cases". Yet another possibility is that, at the given time point, it is not known whether the property is true or false or whether an error will occur.

The basic model of runtime network monitoring can be illustrated by the drawing in Figure 1: The network traffic is modeled by the global stream. At each position, the stream contains a message, which is a pair (t, v) of the time stamp t and the value v. The messages in the stream are linearly ordered according to the time stamp. Some messages may contain the unknown value $?_{\mathrm{V}}$ as the second component instead of some definite value. Such messages are interpreted as time "ticks", indicating the progress of time. The formula that should be monitored is interpreted over that stream. During monitoring some local streams may be generated, over which certain subformulas are checked. Their results are then combined[1] into the final result, which can be either true ($\mathbf{t}$), false ($\mathbf{f}$), error ($\perp_{\mathbf{F}}$), or the unknown value ($?_{\mathbf{F}}$), if the monitor did not manage to get one of the other three values. Then the verification system should give warnings etc. about violations, i.e., when the monitored formula evaluates to $\mathbf{f}$.

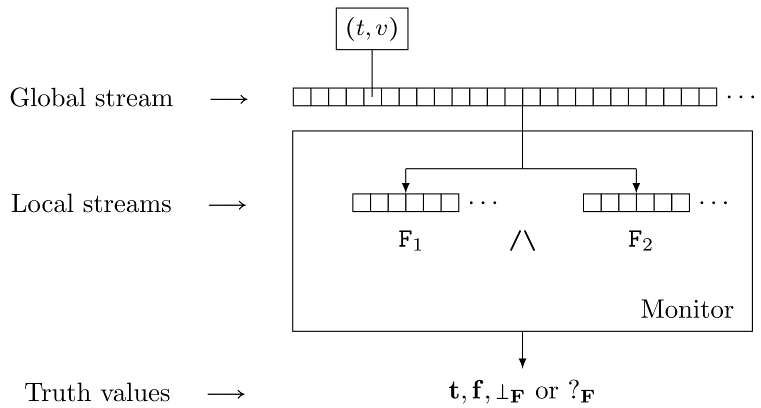

Figure 1: Basic Model of Runtime Network Monitoring.

[1] In Figure 1, we denoted the subformulas by $\mathtt{F}_1$ and $\mathtt{F}_2$ and used conjunction (/\) as an example of combination.

In the following two sections we describe and explain syntax and semantics of LGAL. In the last section we illustrate how a file download monitor can be specified in this language. The appendix contains the reference material.

2 Syntax

The language distinguishes three kinds of terms: for streams, positions, and values. Intuitively, stream terms are supposed to model the stream of messages on the network, position terms refer to particular positions in such streams, while value terms model the contents of messages. Formulas are constructed in the usual way.

More specifically, the alphabet of the LGAL consists of the sets of symbols given below. They are grouped into four classes: variables, logical symbols (whose semantics is fixed), nonlogical symbols (with no predefined semantics), and auxiliary symbols. The sets are pairwise disjoint.

The variables are:

- position variables, denoted `XP`[2],
- value variables `XV`, containing the variable `this`,
- stream variables `XS`,
- formula variables `XF`.

Logical symbols:

- Constant position function symbols `0`, `1`, ...,
- Binary position function symbols + and −,
- Binary value function symbols ◷ and @,
- Binary position predicate symbols < and =<,
- Connectives: `true`, `false`, `~`, `/\`, `\/`, `=>`, `<=>`.
- Quantifiers:
  - `forall`, `exists`, `monitor`,
  - term quantifiers: `max`, `min`, `num`, `complete combine`, `partial combine`, `construct`,
  - local binders: `formula`, `position`, `value`, `stream`.

Nonlogical symbols:

- Fixed arity function symbols:
  - value function symbols `FV`,
  - stream function symbols `FS`.
- Fixed arity predicate symbols:
  - value predicate symbols `PV`.

[2]Symbols in SANS SERIF font denote symbol categories in the alphabet. For concrete symbols, we use lower case letters, with or without indices, e.g. xp.

The auxiliary symbols are the specifiers `in`, `with`, `until`, `satisfying`, the parentheses (,), square brackets [,], comma, colon, =. (The latter does not stand for equality but, rather, is used with the binders.)

One can notice that the alphabet is quite similar to an alphabet of a (many-sorted) first-order language with the exception of formula variables, term quantifiers, and local binders. The purpose of term quantifiers is to construct stream terms, position terms, or value terms. Local binders are nothing else than the `let` construct (for formulas and for three kinds of terms), which is quite common in some programming languages. This also justifies the use of formula variables: They are used by binders. Table 2 gives a compact view of the notation for variables, function, and predicate symbols for the reference:

	Variables	Function symbols		Predicate symbols	
		Logical	Nonlogical	Logical	Nonlogical
Position	`XP`	`+`, `-`, `0`, `1`, ...		`<`, `=<`	
Value	`XV`	⏲, @	`FV`		`PV`
Stream	`XS`		`FS`		
Formula	`XF`				

Table 1: Notation for variables, function symbols, and predicate symbols.

To make the explanation more precise, we need to show how *formulas* and *terms* of LGAL are constructed. As we have already mentioned, there are three kinds of terms in LGAL: Position terms (`TP`), value terms (`TV`), and stream terms (`TS`). We use `T` to denote either of them: `T` := `TP` | `TV` | `TS`. Formulas are denoted by `F`. Monitor is denoted by `M`. Definitions of their grammars use auxiliary syntactic categories of *bindings* (`BIND`), *constraints* (`CONSTR`), and *position ranges* (`RAN`).

Formulas and Monitor:

$$
\begin{aligned}
\mathtt{F} ::= {} & \mathtt{XF} \mid \mathtt{BIND} : \mathtt{F} \\
& \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{PV}(\mathtt{T}_1, \ldots, \mathtt{T}_n) \\
& \mid \mathtt{\sim F} \mid \mathtt{F}_1 \mathbin{\mathtt{/\backslash}} \mathtt{F}_2 \mid \mathtt{F}_1 \mathbin{\mathtt{\backslash/}} \mathtt{F}_2 \mid \mathtt{F}_1 \mathbin{\mathtt{=>}} \mathtt{F}_2 \mid \mathtt{F}_1 \mathbin{\mathtt{<=>}} \mathtt{F}_2 \\
& \mid \mathtt{forall\ XP\ in\ TS\ with\ RAN} : \mathtt{F} \mid \mathtt{exists\ XP\ in\ TS\ with\ RAN} : \mathtt{F}
\end{aligned}
$$

`PV` is assumed to be n-ary with $n \geq 0$. As one can see, the formula syntax is more or less standard. The exact semantics will be explained in the next section.

The notions of bound and free variables, variable renaming, closed and open formula are defined as usual. Note that in our expressions variables can be bound not only by `forall` and `exists`, but also by term quantifiers and local binders discussed below.

A *monitor* is defined as

$$\mathtt{M} ::= \mathtt{monitor\ XP} : \mathtt{F}.$$

The quantifier `monitor` binds `XP` in the whole expression. The intuition behind a monitor formula is similar to a universally quantified one: It is supposed to be true if `F` is true for all `XP`. However, during monitoring, we are interested in those positions `XP` for which `F` is violated. Therefore, as we will see later, the semantics of `monitor XP : F` is defined in a special way to reflect this requirement.

Bindings:

$$\mathtt{BIND} ::= \mathtt{formula\ XF = F} \mid \mathtt{position\ XP = TP} \mid \mathtt{value\ XV = TV} \mid \mathtt{stream\ XS = TS}.$$

The intuitive reading of, for instance, `position XP = TP` is "let `XP` be the position term `TP`". We will consider only the bindings where variables `XP`, `XV`, `XS`, and `XF` do not occur in the right hand sides of the corresponding equalities, i.e., they do not occur, respectively, in `TP`, `TV`, `TS`, and `F`.

Constraints:

$$\texttt{CONSTR} ::= \epsilon \mid \texttt{satisfying F CONSTR} \mid \texttt{BIND CONSTR}.$$

Here ϵ stands for the empty constraint. We postpone explanation of constraints until their usage is considered.

Position Ranges:

$$\texttt{RAN} ::= \texttt{TP PP XP} \mid \texttt{TP}_1\ \texttt{PP}_1\ \texttt{XP PP}_2\ \texttt{TP}_2 \qquad \text{Default value for } \texttt{TP}_1\ \texttt{PP}_1 : \texttt{0 =<}$$
$$\texttt{PP} ::= \texttt{<} \mid \texttt{=<}$$

The range expressions are supposed to restrict the ranges for position variables (used in the quantifiers below). From above, the ranges can be bounded or unbounded. From below, they are always bounded. Default values help to avoid too verbose notation.

Position Terms:

$$\begin{aligned} \texttt{TP} ::=\ & \texttt{XP} \mid \texttt{BIND} : \texttt{TP} \\ & \mid \texttt{0} \mid \texttt{TP} + \texttt{N} \mid \texttt{TP} - \texttt{N} \\ & \mid \texttt{max XP in TS with RAN} : \texttt{F} \mid \texttt{min XP in TS with RAN} : \texttt{F} \\ \texttt{N} ::=\ & \texttt{0} \mid \texttt{1} \mid \cdots \end{aligned}$$

Given a position term `value XV = TV : TP`, we would read it as "let `XV` be the value term `TV` in the position term `TP`." The position term `max XP in TS with RAN : F` reads as "the maximal position `XP` in the stream `TS` within the range `RAN`, for which the formula `F` holds." As for the terms `TP + N` and `TP - N`, one of the motivations of using them is the possibility to look beyond the range bounds of a variable. For instance, to see whether there was some relevant activity recorded in the positions smaller than the lower bound of the range by a given fixed value. This can help, to some extent, model timeouts, when for certain time nothing happened.

Value Terms:

$$\begin{aligned} \texttt{TV} ::=\ & \texttt{XV} \mid \texttt{BIND} : \texttt{TV} \\ & \mid \texttt{TS}\circleddash\texttt{TP} \mid \texttt{TS@TP} \\ & \mid \texttt{FV}(\texttt{T}_1, \ldots, \texttt{T}_n) \mid \texttt{num XP in TS with RAN} : \texttt{F} \\ & \mid \texttt{complete combine}[\texttt{TV}_0, \texttt{FV}]\ \texttt{XP in TS with RAN CONSTR until F} : \texttt{TV}_1 \\ & \quad \text{Default value for F in ``until F''} : \texttt{false} \end{aligned}$$

The intuition behind the terms $\texttt{TS}\circleddash\texttt{TP}$ and `TS@TP` is, respectively, "the time stamp of the message at position `TP` in the stream `TS`" and "the content of the message at the position `TP` in the stream `TS`". (Our streams will consist of messages that have the time stamp and content.) The term `num XP in TS with RAN : F` reads as "the number of all positions `XP` in the stream `TS` within the range `RAN`, for which the formula `F` holds."

The combination term is more complex. It is supposed to construct a value term. The intended meaning is better understood if we explain it procedurally. Starting from the initial term $\texttt{TV}_0$, it should combine into a single value term (with the help of the combination function `FV`) all those value terms $\texttt{TV}_1$ that are selected for each position `XP`. These are the positions taken (incrementally) from the stream `TS` within the range `RAN` until `F` succeeds, for which the constraint `CONSTR` holds. The variable `this`, when it appears in `F`, should be instantiated with the value term constructed up to the moment when `F` is evaluated. It should be noted that the scopes of bindings that appear in `CONSTR` last till the end of the `complete combine` expression, including $\texttt{TV}_1$.

Stream Terms:

$$
\begin{aligned}
\mathtt{TS} ::= {} & \mathtt{XS} \mid \mathtt{BIND} : \mathtt{TS} \\
& \mid \mathtt{FS}(\mathtt{T}_1, \ldots, \mathtt{T}_n) \\
& \mid \mathtt{partial\ combine}[\mathtt{TV}_0, \mathtt{FV}]\ \mathtt{XP\ in\ TS\ with\ RAN\ CONSTR\ until\ F} : \mathtt{TV}_1 \\
& \quad \text{Default value for } \mathtt{F} \text{ in “}\mathtt{until\ F}\text{”} : \mathtt{false} \\
& \mid \mathtt{construct\ XP\ in\ TS\ with\ RAN\ CONSTR} : \mathtt{TV} \\
& \mid \mathtt{construct\ XP\ in\ TS}_1\ \mathtt{with\ RAN\ CONSTR} : \mathtt{TS}_2
\end{aligned}
$$

Partial combination is quite similar to complete combination, but instead of constructing a single value term, it constructs a stream of all intermediate value terms that are used in complete combination in the process of constructing the value term. The variable `this`, when it appears in `F`, refers to the stream constructed up to the moment when `F` is evaluated. `construct XP in TS with RAN CONSTR : TV` constructs a stream in the following way: For each position `XP` in the stream `TS` within the range `RAN` satisfying the constraint `CONSTR`, the value term `TV` is put in stream that is being constructed. Similarly, `construct XP in` $\mathtt{TS}_1$ `with RAN CONSTR :` $\mathtt{TS}_2$ constructs a stream by joining the $\mathtt{TS}_2$'s together for each position `XP` in the stream $\mathtt{TS}_1$ within the range `RAN` satisfying the constraint `CONSTR`.

We finish this section with an example illustrating various syntactic categories:

Example 1.

$$
\begin{aligned}
& \mathtt{stream}\ xs = \mathtt{construct}\ xp\ \mathtt{in}\ s_1\ \mathtt{with}\ \mathtt{0} =< xp \\
& \qquad\quad \mathtt{value}\ xv = s_1@xp\ \mathtt{satisfying}\ s_1 \circleddash xp < s_2 \circleddash \mathtt{0} \mathbin{/\backslash} p(xv) : xv \\
& : \\
& \mathtt{forall}\ \ xp\ \mathtt{in}\ xs\ \mathtt{with}\ \mathtt{0} =< xp : q(xs@xp) \Rightarrow r(xs@xp)
\end{aligned}
$$

In this formula s_1 and s_2 stand for stream constants, p, q, and r are unary value predicates, and $<$ is a binary value predicate. The formula states that for all positions $\mathtt{0} =< xp$ in the stream xs, if q holds for the value $xs@xp$ of xs at the position xp, then r holds for the same value. The stream xs is constructed by selecting those messages from the stream s_1 that chronologically precede any message in the stream s_2 and satisfy the predicate p. The syntax tree of the formula is shown in Figure 2. Note that except the specifiers, the auxiliary symbols of the alphabet are not displayed.

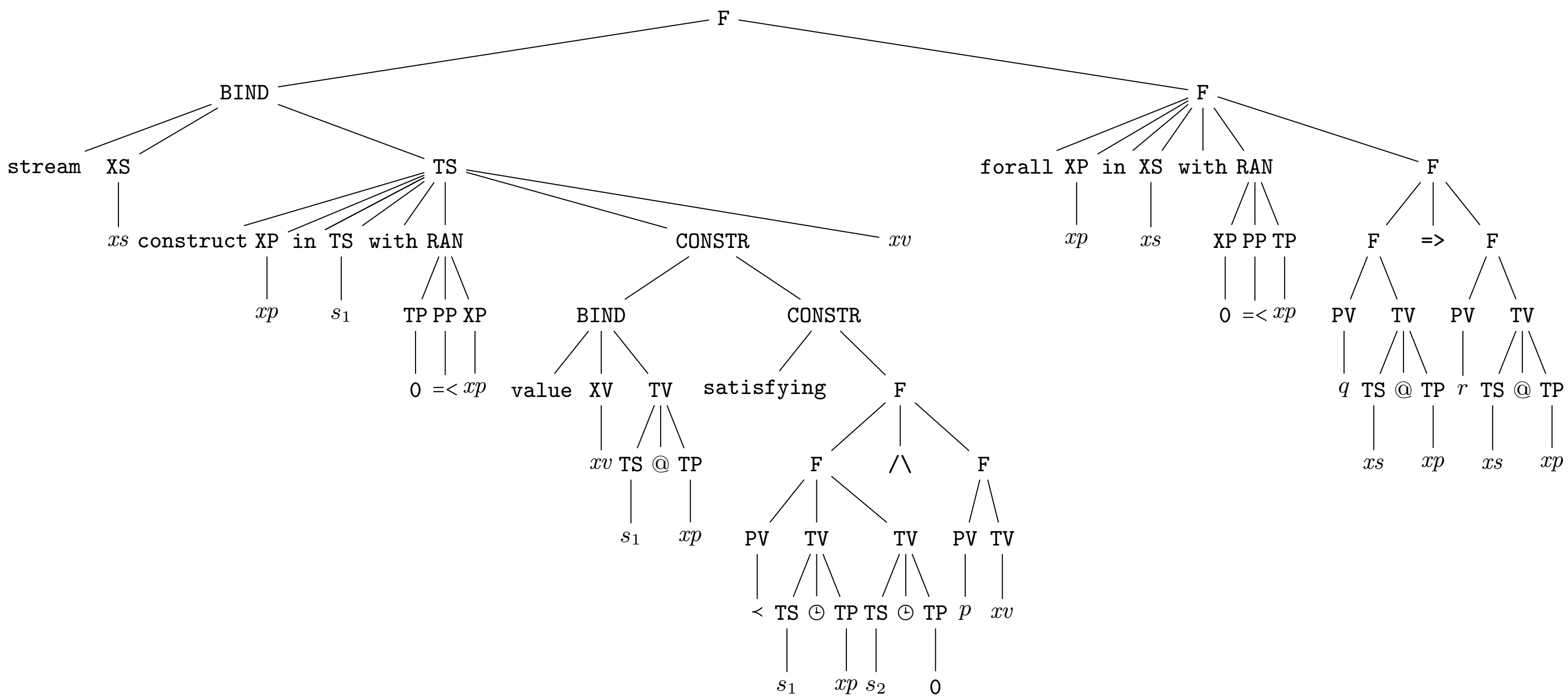

Figure 2: Syntax tree of the formula from Example 1.

3 Semantics

To define semantics of our language, we start with introducing semantic domains. After that, the syntactic constructs will be connected to these domains and to their corresponding operations with the help of the valuation function.

3.1 Semantic Domains and Environments

We choose mnemonic names for semantic domains, to underline their connection with syntactic categories. Their definitions and connections to syntactic constructs are given in Table 2.

Domain	Definition	Corresponding syntactic category
Stream	$\mathbb{N} \to \mathit{Message}$	Stream terms
Message	$(\mathit{Time} \times (\mathit{Value} \cup \{?_\mathrm{V}\})) \cup \{?_\mathrm{M}\}$	
Time	$\mathbb{N}$	Value terms (the time stamp of a message)
Value	$\mathbb{N} + \mathit{Char}^*$	Value terms (general)
Position	$\mathbb{N}$	Position terms
MK	$\{\mathbf{t}, \mathbf{f}, ?_\mathbf{F}, \bot_\mathbf{F}\}$	Formulas

Table 2: Semantic domains.

In the table, $\to, \times, +, {}^*$ are constructors of compound domains: $\to$ for the function domain, $\times$ for the product domain, $+$ for the sum domain, and * for the Kleene closure. $\mathbb{N}$ stands for the set of natural numbers, *Char* for characters, $?_\mathrm{V}$ is the unknown value, $?_\mathrm{M}$ is the unknown message, $?_\mathbf{F}$ is the unknown truth value, and $\bot_\mathbf{F}$ is the error truth value. ($\mathbf{F}$ in the index stands for "formula".) The name *MK* is an abbreviation of McCarthy-Kleene: The reason is that $?_\mathbf{F}$ and $\bot_\mathbf{F}$ behave like Kleene's undefined truth value [17] and McCarthy's error value [18], respectively. Below we will use also unknown and error positions $?_\mathrm{P}$ and $\bot_\mathrm{P}$ and error values $\bot_\mathrm{V}$.

We distinguish between unknown and error truth values having in mind the following difference: A formula that evaluates to the error value immediately propagates it to the entire context around it, forcing the context to get also the error value. To give an analogy with programming, it is a critical error that forces to abort the evaluation. On the other hand, the unknown truth value does not force the same behavior: A formula can still evaluate to $\mathbf{t}$ or $\mathbf{f}$ even if some of its subformula evaluates to $?_\mathbf{F}$.

The idea is to compute the truth value of the monitor formula over the given *Stream* domain. The elements of this domain are streams. They are infinite sequences of messages. Each message is either a pair of the time value and the message content, or an unknown message. We assume that all our streams s have the following properties:

- *Ascending Time Property:* For all $i \in \mathbb{N}$, if $s(i) \neq ?_\mathrm{M}$ and $s(i+1) \neq ?_\mathrm{M}$, then the time value at $s(i)$ does not exceed the time value at $s(i+1)$.
- *Continuous Unknown Messages Property:* For all $i \in \mathbb{N}$, if $s(i) = ?_\mathrm{M}$, then $s(i+1) = ?_\mathrm{M}$.

The evaluation of syntactic objects naturally depends on the assignment of values to the variables. This is what the environments are responsible for. They map identifiers to the corresponding values. In our case, these are mappings from variables to the corresponding semantic domains as it is shown in Table 3. One can notice that there are some peculiarities there: First, our environment can be erroneous, because the mapping might map a variable to the corresponding error value. To express such a possibility, we have the lifting with the erroneous environment $\bot_\mathrm{E}$. Moreover, we will use an extended environment, that records the "current time point" (a natural number). The reason is that the semantics of our language constructs will depend on the current time, which divides streams from the *Stream* semantic domain between finite *observable* initial part and infinite *non-observable* tail stream. As time progresses, more of the "non-observable" part becomes "observable". We do not have an explicit current time variable in our language. To evaluate a monitor (to compute its truth value) over a given stream (or streams), we proceed by evaluating it for all time points, starting from some minimal value (say, 0) for the current time and monotonically increasing it. Therefore, instead of an explicit mapping, *CurrTime* is added to the extended environment.

Environment	Definition
Env	$\mathit{EnvFormula} \times \mathit{EnvPosition} \times \mathit{EnvValue} \times \mathit{EnvStream}$
EnvExtended	$\mathit{EnvFormula} \times \mathit{EnvPosition} \times \mathit{EnvValue} \times \mathit{EnvStream} \times \mathit{CurrTime}$
EnvFormula	$\mathtt{XF} \to \{\mathbf{t}, \mathbf{f}, ?_{\mathbf{F}}\}$
EnvPosition	$\mathtt{XP} \to \mathit{Position}$
EnvValue	$\mathtt{XV} \to \mathit{Value}$
EnvStream	$\mathtt{XS} \to \mathit{Stream}$
CurrTime	*Time*
$\perp_{\mathrm{E}}$	Error environment

Table 3: Environments.

3.2 Valuation Function

The valuation function $[\![\cdot]\!]$ operates on a syntactic construct and returns a function from the environment to a semantic domain.

Auxiliary Functions. We start with auxiliary definitions that are used in several places later:[3]

$$\underline{\mathrm{time}} : \mathit{Message} \to \mathit{Time} \qquad \underline{\mathrm{value}} : \mathit{Message} \to \mathit{Value} \cup \{?_{\mathrm{V}}\}$$
$$\underline{\mathrm{time}}(m) = (\mathrm{let}\ (t, v) = m : t) \qquad \underline{\mathrm{value}}(m) = (\mathrm{let}\ (t, v) = m : v)$$

In the definitions below $\mathcal{P}$ stands for the powerset, $e{\downarrow}i$ for the i's projection of an environment e, and $e{\downarrow}i[\mathtt{X} \mapsto V]$ for the environment e' with the property that $e'(\mathtt{X}) = V$ and $e'(\mathtt{Y}) = e{\downarrow}i(\mathtt{Y})$ for all $\mathtt{Y} \neq \mathtt{X}$.

For an extended environment e, we define a function <u>current time</u> that simply returns the projection of e on its last component, the current time:

$$\underline{\text{current time}} : \mathit{EnvExtended} \to \mathit{CurrTime}$$
$$\underline{\text{current time}}(e) = e{\downarrow}5$$

The function <u>collect</u> will be used in the definitions of quantifier semantics. Given a position variable, semantics of a stream term, of a range, and of a formula, it returns a function from an extended environment to a set of pairs (*position*, *truth value*), defined as follows:

$$\begin{aligned}
\underline{\mathrm{collect}} : {} & \mathtt{XP} \times \mathtt{TS} \times (\mathit{EnvExtended} \to (\mathtt{XP} \times \mathit{Position} \times \mathit{Position}) \cup \{?_{\mathrm{P}}, \perp_{\mathrm{P}}\}) \\
& \times (\mathit{EnvExtended} \to \mathit{MK}) \\
& \to \mathit{EnvExtended} \\
& \to (\mathcal{P}(\mathit{Position} \times \mathit{MK}) \times \{\mathrm{complete_range}, \mathrm{incomplete_range}\}) \cup \{?_{\mathrm{P}}, \perp_{\mathrm{P}}\}
\end{aligned}$$

$$\begin{aligned}
& \underline{\mathrm{collect}}[\mathtt{XP}, \mathtt{TS}, [\![\mathtt{RAN}]\!], [\![\mathtt{F}]\!]](e) = \\
& \qquad \mathrm{let}\ r = [\![\mathtt{RAN}]\!](e) : \\
& \qquad \mathrm{if}\ r = \perp_{\mathrm{P}}\ \mathrm{then} \\
& \qquad\quad \perp_{\mathrm{P}} \\
& \qquad \mathrm{else\ if}\ r = ?_{\mathrm{P}}\ \mathrm{then} \\
& \qquad\quad ?_{\mathrm{P}}
\end{aligned}$$

[3] We denote auxiliary semantic functions with underlined identifiers.

else

  let $(\mathtt{XP}_1, \mathit{from}, \mathit{to}_0) = c$:

  if $\mathtt{XP}_1 \neq \mathtt{XP}$ then

    $\perp_{\mathrm{P}}$

  else

    let $p_0 = \max_{pos} : [\![\mathtt{TS}]\!](e)(pos) \neq ?_{\mathrm{M}} \wedge [\![\mathtt{TS} \ominus pos]\!](e) \leq \underline{\text{current time}}(e)$:

    let $\mathit{to} = \min(p_0, \mathit{to}_0)$:

    let flag =

      if $\mathit{to} = \mathit{to}_0$ then

        complete_range

      else

        incomplete_range :

    let $\mathit{pairs} = \{(p, [\![\mathtt{F}]\!](e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto p], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)) \mid \mathit{from} \leq p \leq \mathit{to}\}$:

      $(\mathit{pairs}, \mathit{flag})$.

From this definition one can notice that although RAN can be unbounded from above, the set collect computes is always finite. It is because we consider only those positions from TS, which contain messages with the time stamp not exceeding the current time point current time(e). The formula F is then evaluated for all those selected positions.

Semantics of Monitors and Formulas. Our logic is a four-valued logic where binary connectives are not commutative. The interpretations of the connectives are similar to [2] and are given by the following tables:

not	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
	f	**t**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$

and	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
t	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
f	**f**	**f**	**f**	**f**
$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$

or	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
t	**t**	**t**	**t**	**t**
f	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
$?_{\mathbf{F}}$	**t**	$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$

implies	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
t	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
f	**t**	**t**	**t**	**t**
$?_{\mathbf{F}}$	**t**	$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$

iff	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
t	**t**	**f**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
f	**f**	**t**	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$\perp_{\mathbf{F}}$
$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$	$\perp_{\mathbf{F}}$

We use these operators in the definition of the valuation function for the monitors and formulas. For a monitor M = monitor XP : F, the valuation $[\![\mathtt{M}]\!]$ collects positions of violations of $[\![\mathtt{F}]\!]$, where XP points to "current position":

▷ $[\![\mathtt{M}]\!] : \mathit{Env} \to \mathcal{P}(\mathit{Position})$

  $[\![\mathtt{monitor\ XP} : \mathtt{F}]\!](e) = \{p \in \mathit{Position} : [\![\mathtt{F}]\!](e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto p], e{\downarrow}3, e{\downarrow}4) = \mathbf{f}\}$.

The truth value of a formula F in the given environment is defined as $?_{\mathbf{F}}$ if it is $?_{\mathbf{F}}$ at all time points in the extended environment. If there is a time point at which the truth value of F is not $?_{\mathbf{F}}$, we take the minimal such time point, evaluate F in the extended environment at that time point, and take the obtained truth value as the result:

▷ $[\![\mathtt{F}]\!] : \mathit{Env} \to \mathit{MK}$

  $[\![\mathtt{F}]\!](e)$ =

    if $(\exists\ t \in \mathbb{N} : [\![\mathtt{F}]\!](e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, t) \neq ?_{\mathbf{F}})$ then

$$\text{let } t = \min_{t \in \mathbb{N}} : [\![\mathtt{F}]\!](e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, t) \neq ?_{\mathbf{F}} : [\![\mathtt{F}]\!](e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, t)$$

else

$?_{\mathbf{F}}$.

Truth value of formulas over an extended environment is defined as follows:

$\triangleright$ $[\![\mathtt{F}]\!] : \mathit{EnvExtended} \to \mathit{MK}$

$[\![\mathtt{XF}]\!](e) = e{\downarrow}1(\mathtt{XF})$.

$[\![\mathtt{BIND:F}]\!](e) = (\text{let } e_1 = [\![\mathtt{BIND}]\!](e) : \text{if } e_1 = \bot_{\mathrm{E}} \text{ then } \bot_{\mathbf{F}} \text{ else } [\![\mathtt{F}]\!](e_1))$.

$[\![\mathtt{true}]\!](e) = \mathbf{t}$.

$[\![\mathtt{false}]\!](e) = \mathbf{f}$.

$[\![\mathtt{PV}(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!](e) =$

let $v_1 = [\![\mathtt{T}_1]\!](e), \ldots, v_n = [\![\mathtt{T}_n]\!](e)$:

if $(\exists\, 1 \le i \le n : v_i = \bot_{\mathrm{V}} \vee v_i = \bot_{\mathrm{P}})$ then $\bot_{\mathbf{F}}$ else $[\![\mathtt{PV}]\!](v_1, \ldots, v_n)$.

$[\![\mathtt{\sim F}]\!](e) = (\text{let } b = [\![\mathtt{F}]\!](e) : \mathbf{not}(b))$.

$[\![\mathtt{F}_1 \mathbin{\mathtt{/\backslash}} \mathtt{F}_2]\!](e) = (\text{let } b_1 = [\![\mathtt{F}_1]\!](e), b_2 = [\![\mathtt{F}_2]\!](e) : \mathbf{and}(b_1, b_2))$.

$[\![\mathtt{F}_1 \mathbin{\mathtt{\backslash/}} \mathtt{F}_2]\!](e) = (\text{let } b_1 = [\![\mathtt{F}_1]\!](e), b_2 = [\![\mathtt{F}_2]\!](e) : \mathbf{or}(b_1, b_2))$.

$[\![\mathtt{F}_1 \mathbin{\mathtt{=>}} \mathtt{F}_2]\!](e) = (\text{let } b_1 = [\![\mathtt{F}_1]\!](e), b_2 = [\![\mathtt{F}_2]\!](e) : \mathbf{implies}(b_1, b_2))$.

$[\![\mathtt{F}_1 \mathbin{\mathtt{<=>}} \mathtt{F}_2]\!](e) = (\text{let } b_1 = [\![\mathtt{F}_1]\!](e), b_2 = [\![\mathtt{F}_2]\!](e) : \mathbf{iff}(b_1, b_2))$.

The above definitions are more or less straightforward. Just to emphasize on how the error truth value is obtained, we comment on the semantics of $\mathtt{BIND:F}$ and $\mathtt{PV}(\mathtt{T}_1, \ldots, \mathtt{T}_n)$. The valuation of `BIND` changes the environment (we will see this definition a bit later), because it binds variables with the corresponding values (position, value, stream terms, or formulas), which might raise an error. Therefore, the environment might become erroneous and it causes $\mathtt{BIND:F}$ to be evaluated to $\bot_{\mathbf{F}}$. In the case of $\mathtt{PV}(\mathtt{T}_1, \ldots, \mathtt{T}_n)$, error in the evaluation of the arguments makes the truth value of the formula equal to $\bot_{\mathbf{F}}$.

The valuation function for quantified formulas is defined as follows:

$[\![\mathtt{forall\ XP\ in\ TS\ with\ RAN\ :F}]\!](e) =$

let $\mathit{collected} = \underline{\text{collect}}[\mathtt{XP}, \mathtt{TS}, [\![\mathtt{RAN}]\!], \mathtt{F}](e)$:

if $\mathit{collected} = \bot_{\mathrm{P}}$ then

$\bot_{\mathbf{F}}$

else if $\mathit{collected} = ?_{\mathrm{P}}$ then

let $b_0 = [\![\mathtt{F}]\!](e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto ?_{\mathrm{P}}], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)$:

if $b_0 = \mathbf{f}$

$\mathbf{f}$

else if $b_0 = \bot_{\mathbf{F}}$

$\bot_{\mathbf{F}}$

else

$?_{\mathbf{F}}$

else

let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected}$:

if $(\forall\, (p, b) \in \mathit{pairs} : b = \mathbf{t})$ then

if $\mathit{flag} = \text{complete\_range}$ then

$\mathbf{t}$

else

$?_{\mathbf{F}}$

else if $(\forall\,(p,b) \in \mathit{pairs} : b = \mathbf{t} \vee b = ?_{\mathbf{F}})$ then

$?_{\mathbf{F}}$

else

let $p_1 = \min_p : (p,b) \in \mathit{pairs} \wedge (b = \mathbf{f} \vee b = \bot_{\mathbf{F}}) :$

let $(p_1, b_1) \in \mathit{pairs} :$

b_1.

$[\![\texttt{exists XP in TS with RAN : F}]\!](e) =$

let $\mathit{collected} = \underline{\text{collect}}[\texttt{XP}, \texttt{TS}, [\![\texttt{RAN}]\!], \texttt{F}](e) :$

if $\mathit{collected} = \bot_{\mathrm{P}}$ then

$\bot_{\mathbf{F}}$

else if $\mathit{collected} = ?_{\mathrm{P}}$ then

let $b_0 = [\![\texttt{F}]\!](e{\downarrow}1, e{\downarrow}2[\texttt{XP} \mapsto ?_{\mathrm{P}}], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5) :$

if $b_0 = \mathbf{t}$

$\mathbf{t}$

else if $b_0 = \bot_{\mathbf{F}}$

$\bot_{\mathbf{F}}$

else

$?_{\mathbf{F}}$

else

let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected} :$

if $(\forall\,(p,b) \in \mathit{pairs} : b = \mathbf{f})$ then

if flag = complete_range then

$\mathbf{f}$

else

$?_{\mathbf{F}}$

else if $(\forall\,(p,b) \in \mathit{pairs} : b = \mathbf{f} \vee b = ?_{\mathbf{F}})$ then

$?_{\mathbf{F}}$

else

let $p_1 = \min_p : (p,b) \in \mathit{pairs} \wedge (b = \mathbf{t} \vee b = \bot_{\mathbf{F}}) :$

let $(p_1, b_1) \in \mathit{pairs} :$

b_1.

Both definitions rely on the output of the collect function. Notice that this output may contain a pair $(p, \bot_{\mathbf{F}})$, i.e., evaluating `F` to $\bot_{\mathbf{F}}$ does not cause collect itself to become erroneous. The information contained in the set the collect function computes is used to establish the truth value of the quantified formula. And if for some position p in `TS` the formula `F` evaluates to $\bot_{\mathbf{F}}$, it does not automatically mean that, say, `forall XP in TS with RAN : F` to have the error truth value as well. It might be that for some position $p_0 < p$, `forall XP in TS with RAN : F` fails. Since our logic is sequential, in such a case the valuation function should return (for the given environment) $\mathbf{f}$ and not $\bot_{\mathbf{F}}$. This is in accordance to the intuition that an universally quantified formula can be seen, in general, as an infinite conjunction. (In our case it is finite, because the output of collect is always finite.) Hence, it should follow the rules for the interpretation of conjunction, which in our case is not commutative. Note that here and below the meta-quantifiers and meta-connectives $\forall, \exists, \wedge, \vee, \ldots$ in semantic functions are the ordinary two-valued (non-sequential) ones.

Semantics of Bindings, Constraints, and Position Ranges. Bindings change environments:

$\triangleright$ $[\![\mathtt{BIND}]\!] : \mathit{EnvExtended} \to \mathit{EnvExtended} \cup \{\bot_{\mathrm{E}}\}$

$[\![\mathtt{formula\ XF = F}]\!](e) =$

let $b = [\![\mathtt{F}]\!](e)$:

if $b = \bot_{\mathbf{F}}$ then $\bot_{\mathrm{E}}$ else $(e{\downarrow}1[\mathtt{XF} \mapsto b], e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)$.

$[\![\mathtt{position\ XP = TP}]\!](e) =$

let $p = [\![\mathtt{TP}]\!](e)$:

if $p = \bot_{\mathrm{P}}$ then $\bot_{\mathrm{E}}$ else $(e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto p], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)$.

$[\![\mathtt{value\ XV = TV}]\!](e) =$

let $v = [\![\mathtt{TV}]\!](e)$:

if $v = \bot_{\mathrm{V}}$ then $\bot_{\mathrm{E}}$ else $(e{\downarrow}1, e{\downarrow}2, e{\downarrow}3[\mathtt{XV} \mapsto v], e{\downarrow}4, e{\downarrow}5)$.

$[\![\mathtt{stream\ XS = TS}]\!](e) =$

let $s = [\![\mathtt{TS}]\!](e)$:

$(e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4[\mathtt{XS} \mapsto s], e{\downarrow}5)$.

The valuation of constraints returns a pair of an environment and a truth value:

$\triangleright$ $[\![\mathtt{CONSTR}]\!] : \mathit{EnvExtended} \to \mathit{EnvExtended} \times \mathit{MK}$

$[\![\epsilon]\!](e) = (e, \mathbf{t})$.

$[\![\mathtt{satisfying\ F\ CONSTR}]\!](e) =$

let $b = [\![\mathtt{F}]\!](e)$:

if $b \neq \mathbf{t}$ then (e, b) else $[\![\mathtt{CONSTR}]\!](e)$.

$[\![\mathtt{BIND\ CONSTR}]\!](e) =$

let $e_1 = [\![\mathtt{BIND}]\!](e)$:

if $e_1 = \bot_{\mathrm{E}}$ then $(\bot_{\mathrm{E}}, \bot_{\mathbf{F}})$ else $[\![\mathtt{CONSTR}]\!](e_1)$.

The valuation of a range gives a triple of a position variable and lower and upper bounds of its range (the bounds are positions). It may happen that the output is the error position or an unknown position.

$\triangleright$ $[\![\mathtt{RAN}]\!] : \mathit{EnvExtended} \to (\mathtt{XP} \times \mathit{Position} \times \mathit{Position} \cup \{\infty\}) \cup \{?_{\mathrm{P}}, \bot_{\mathrm{P}}\}$

$[\![\mathtt{TP}_1\ \mathtt{PP}_1\ \mathtt{XP}\ \mathtt{PP}_2\ \mathtt{TP}_2]\!](e) =$

let $p_1 = [\![\mathtt{TP}_1]\!](e)$:

let $p_2 = [\![\mathtt{TP}_2]\!](e)$:

if $p_1 = \bot_{\mathrm{P}}\ \lor\ p_2 = \bot_{\mathrm{P}}$ then

$\bot_{\mathrm{P}}$

else if $p_1 = ?_{\mathrm{P}} \lor p_2 = ?_{\mathrm{P}}$ then

$?_{\mathrm{P}}$

else

let *from* =

if $\mathtt{PP}_1$ = "<" then

$p_1 + 1$

else

p_2 :

let *to* =

if $\texttt{PP}_2$ = "<" then

$p_2 - 1$

else

p_2 :

$(\texttt{XP}, \mathit{from}, \mathit{to})$.

$[\![\texttt{TP PP XP}]\!](e)$ =

let $p = [\![\texttt{TP}]\!](e)$:

if $p = \bot_{\mathrm{P}}$ then

$\bot_{\mathrm{P}}$

else if $p = ?_{\mathrm{P}}$ then

$?_{\mathrm{P}}$

else

let *from* =

if $\texttt{PP}_1$ = "<" then

$p + 1$

else

p :

$(\texttt{XP}, \mathit{from}, \infty)$.

Semantics of Position Terms. The valuation function for position terms is defined as follows:

▷ $[\![\texttt{TP}]\!] : \mathit{EnvExtended} \to \mathit{Position} \cup \{?_{\mathrm{P}}, \bot_{\mathrm{P}}\}$

$[\![\texttt{XP}]\!](e) = e{\downarrow}2(\texttt{XP})$.

$[\![\texttt{BIND} : \texttt{TP}]\!](e) = (\text{let } e_1 = [\![\texttt{BIND}]\!](e) : \text{if } e_1 = \bot_{\mathrm{E}} \text{ then } \bot_{\mathrm{P}} \text{ else } [\![\texttt{TP}]\!](e_1))$.

$[\![\texttt{0}]\!](e) = 0, \quad [\![\texttt{1}]\!](e) = 1, \ldots.$

$[\![\texttt{TP} + \texttt{N}]\!](e) = [\![\texttt{TP}]\!](e) + [\![\texttt{N}]\!](e))$.

$[\![\texttt{TP} - \texttt{N}]\!](e) = \max([\![\texttt{TP}]\!](e) - [\![\texttt{N}]\!](e), 0)$.

$[\![\texttt{max XP in TS with RAN} : \texttt{F}]\!](e)$ =

let *collected* = $\underline{\text{collect}}[\texttt{XP}, \texttt{TS}, [\![\texttt{RAN}]\!], [\![\texttt{F}]\!]](e)$:

if *collected* = $\bot_{\mathrm{P}}$ then

$\bot_{\mathrm{P}}$

else if *collected* = $?_{\mathrm{P}}$ then

$?_{\mathrm{P}}$

else

let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected}$:

if $(\exists\ (p, b) \in \mathit{pairs} : b = \bot_{\mathbf{F}})$ then

$\bot_{\mathrm{P}}$

else if $(\forall\ (p, b) \in \mathit{pairs} : b \neq \mathbf{t})$ then

$?_{\mathrm{P}}$

else if *flag* = incomplete_range then

$?_{\mathrm{P}}$

else

$\max_p : (p, b) \in \mathit{pairs} \wedge b = \mathbf{t}$.

$[\![\texttt{min XP in TS with RAN} : \texttt{F}]\!](e)$ =

let $\mathit{collected} = \underline{\text{collect}}[\mathtt{XP}, \mathtt{TS}, [\![\mathtt{RAN}]\!], [\![\mathtt{F}]\!]](e)$:
if $\mathit{collected} = \bot_{\mathrm{P}}$ then
  $\bot_{\mathrm{P}}$
else if $\mathit{collected} = ?_{\mathrm{P}}$ then
  $?_{\mathrm{P}}$
else
  let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected}$:
  if $(\forall\ (p, b) \in \mathit{pairs} : b \neq \mathbf{t} \wedge b \neq \bot_{\mathbf{F}})$ then
    $?_{\mathrm{P}}$
  else if flag = incomplete_range then
    if $(\exists\ (p, b) \in \mathit{pairs} : b = \bot_{\mathbf{F}})$ then
      $\bot_{\mathrm{P}}$
    else
      $?_{\mathrm{P}}$
  else
    let $p_0 = \min_{p} : (p, b) \in \mathit{pairs} \wedge (b = \mathbf{t} \vee b = \bot_{\mathbf{F}})$:
    let $(p_0, b_0) \in \mathit{pairs}$:
    if $b_0 = \mathbf{t}$ then
      p_0
    else
      $\bot_{\mathrm{P}}$.

The definition should be self-explanatory. One can see how collect is used in the definition of semantics of quantified expressions.

Semantics of Value Terms. Except the terms involving `complete combine`, semantics of value terms can be defined easily. We first show this easy part and then consider the `complete combine` terms in detail.

▷ $[\![\mathtt{TV}]\!] : \mathit{EnvExtended} \to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\}$
$[\![\mathtt{XV}]\!](e) = e{\downarrow}3(\mathtt{XV})$.
$[\![\mathtt{BIND} : \mathtt{TV}]\!](e) = (\text{let } e_1 = [\![\mathtt{BIND}]\!](e) : \text{if } e_1 = \bot_{\mathrm{E}} \text{ then } \bot_{\mathrm{V}} \text{ else } [\![\mathtt{TV}]\!](e_1))$.
$[\![\mathtt{TS}⌚\mathtt{TP}]\!](e) = \underline{\text{time or value}}[\mathtt{TS}, \mathtt{TP}, \text{time}](e)$.
$[\![\mathtt{TS@TP}]\!](e) = \underline{\text{time or value}}[\mathtt{TS}, \mathtt{TP}, \text{value}](e)$.
$[\![\mathtt{FV}(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!](e) =$
  let $v_1 = [\![\mathtt{T}_1]\!](e), \ldots, [\![\mathtt{T}_n]\!](e)$:
  if $(\exists\ 1 \leq i \leq n : v_i = \bot_{\mathrm{V}} \vee v_i = \bot_{\mathrm{P}})$ then $\bot_{\mathrm{V}}$ else $[\![\mathtt{FV}]\!](v_1, \ldots, v_n)$.
$[\![\mathtt{num\ XP\ in\ TS\ with\ RAN} : \mathtt{F}]\!](e) =$
  let $\mathit{collected} = \underline{\text{collect}}[\mathtt{XP}, \mathtt{TS}, [\![\mathtt{RAN}]\!], [\![\mathtt{F}]\!]](e)$:
  if $\mathit{collected} = \bot_{\mathrm{P}}$ then
    $\bot_{\mathrm{V}}$
  else if $\mathit{collected} = ?_{\mathrm{P}}$ then
    $?_{\mathrm{V}}$
  else
    let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected}$:

$$
\begin{array}{l}
\text{if } (\exists(p,b) \in \mathit{pairs} : b = \bot_{\mathbf{F}}) \text{ then} \\
\quad \bot_{\mathrm{V}} \\
\text{else if } \mathit{flag} = \text{incomplete\_range then} \\
\quad ?_{\mathrm{V}} \\
\text{else} \\
\quad \#(p,b) \in \mathit{pairs} : b = \mathbf{t}.
\end{array}
$$

The auxiliary function time or value used above should return (for the given environment) either the time stamp or the content value from the message in the given stream at the given position:

$$
\begin{array}{l}
\underline{\text{time or value}} : \mathtt{TS} \times \mathtt{TP} \times \{\text{time}, \text{value}\} \to \mathit{EnvExtended} \to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\} \\
\underline{\text{time or value}}[\mathtt{TS}, \mathtt{TP}, \mathit{time\_or\_value}](e) = \\
\quad \text{let } p = [\![\mathtt{TP}]\!](e), s = [\![\mathtt{TS}]\!](e) : \\
\quad \text{if } p = \bot_{\mathrm{P}} \text{ then} \\
\quad\quad \bot_{\mathrm{V}} \\
\quad \text{else if } p = ?_{\mathrm{P}} \text{ then} \\
\quad\quad ?_{\mathrm{V}} \\
\quad \text{else if } s(p) = ?_{\mathrm{M}} \text{ then} \\
\quad\quad \bot_{\mathrm{V}} \\
\quad \text{else} \\
\quad\quad \text{if } \underline{\text{current time}}(e) < \underline{\text{time}}(s(p)) \text{ then} \\
\quad\quad\quad \bot_{\mathrm{V}} \\
\quad\quad \text{else} \\
\quad\quad\quad \text{if } \mathit{time\_or\_value} = \text{time then} \\
\quad\quad\quad\quad \underline{\text{time}}(s(p)) \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \underline{\text{value}}(s(p)).
\end{array}
$$

This function first checks whether the position itself is a valid one, i.e., whether it is neither $\bot_{\mathrm{P}}$ and $?_{\mathrm{P}}$. If this is not the case, then it looks at the stream message at that position. It may happen that this message is $?_{\mathrm{M}}$.[4] In that case the function gives back the unknown value. Otherwise, it is checked whether the position is within the currently observable part of `TS`. If not, again the unknown value is returned. If yes, then the time stamp or the content value is extracted from the message with the help of time and value functions, respectively.

Now we turn to `complete combine`. Its intuitive meaning has already been mentioned when value terms were introduced. Now we put the concrete definition of its semantics.

$$
\begin{array}{l}
[\![\mathtt{complete\ combine}[\mathtt{TV}_0, \mathtt{FV}]\ \mathtt{XP\ in\ TS\ with\ RAN\ CONSTR\ until\ F} : \mathtt{TV}_1]\!](e) = \\
\quad \text{let } r = [\![\mathtt{RAN}]\!](e) : \\
\quad \text{if } r = \bot_{\mathrm{P}} \text{ then} \\
\quad\quad \bot_{\mathrm{V}} \\
\quad \text{else if } r = ?_{\mathrm{P}} \text{ then} \\
\quad\quad ?_{\mathrm{V}} \\
\quad \text{else} \\
\quad\quad \text{let } (\mathtt{XP}_1, \mathit{from}, \mathit{to}_0) = r :
\end{array}
$$

[4]This can happen if the stream `TS` was constructed so that its finite prefix contain full messages, while the rest, by default, is filled in with $?_{\mathrm{M}}$. We will see stream construction in the section about semantics of stream terms.

if $\texttt{XP}_1 \neq \texttt{XP}$ then

$\perp_{\mathrm{V}}$

else

let $p_0 = \max_{pos} : [\![\texttt{TS}]\!](e)(pos) \neq ?_{\mathrm{M}} \wedge [\![\texttt{TS}\ominus pos]\!](e) \leq \underline{\text{current time}}(e)$:

let $to = \min(p_0, to_0)$:

if $to < from$ then

$\perp_{\mathrm{V}}$

else

let $flag$ =

if $to < to_0$ then

incomplete_range

else

complete_range :

let $acc = [\![\texttt{TV}_0]\!](e)$:

if $acc = \perp_{\mathrm{V}} \vee acc = ?_{\mathrm{V}}$ then

acc

else

let $combine\_values = [\![\texttt{FV}]\!](e)$:

$\underline{\text{value comb}}[\texttt{XP}, \texttt{CONSTR}, \texttt{F}, \texttt{TV}, combine\_values, flag, e](from, to, acc)$.

In this definition, the main task is delegated to the complete combine function $\underline{\text{value comb}}[\texttt{XP}, \texttt{CONSTR}, \texttt{F}, \texttt{TV}, combine\_values, flag, e]$ on the last line. (Its definition comes below.) Before that, some "preparatory work" is done, which involves computing the position range for the variable XP in the stream TS and "preparing" the accumulator, the start value, to which the next values are combined with the help of the combination function. We discuss both steps in more detail:

Computing the range: We need to evaluate RAN and see, whether it is $\perp_{\mathrm{P}}$, $?_{\mathrm{P}}$, or a triple $(\texttt{XP}_1, from, to_0)$ fixing the range of some position variable $\texttt{XP}_1$. In the first two cases the value of the `complete combine` term should be the $\perp_{\mathrm{V}}$ and $?_{\mathrm{V}}$, respectively. In the third case we check whether $\texttt{XP}_1$ is the same as XP, i.e., whether RAN gives the range for XP or for some other variable. If it is the other variable, it is considered to be an error and the value of the `complete combine` term is $\perp_{\mathrm{V}}$. Otherwise, we have the range $(from, to_0)$ for XP, which should be refined with respect to the current time $\underline{\text{current time}}(e)$: We find the maximal position in the stream TS such that the message at that position has the time stamp smaller than the current time. We can not go beyond that position: The "currently observable" part of the stream TS ends there. So, the upper bound of the range should be the minimum between that position and to_0, and at the same time not smaller than the lower bound $from$. If such an upper bound does not exist, then we again get the error value. Otherwise, it is denoted by to, and the range of XP is $(from, to)$. Besides, we pass the information to the function value comb whether the range was completely or incompletely observable. It will be needed for "external" functions (operating on values) to mark or to determine completeness of a particular value.

Preparing the accumulator: The accumulator is supposed to be the value of the term $\texttt{TV}_0$. If it gives an error, then the value of the `complete combine` term is an error as well. Otherwise, we compute $\underline{\text{value comb}}[\texttt{XP}, \texttt{CONSTR}, \texttt{F}, \texttt{TV}, combine\_values, flag, e]$. This is a function whose application to the already computed range of XP and the accumulator gives the result of the evaluation of the `complete combine` term.

The function value comb is defined as follows:

$\underline{\text{value comb}}$:

$$
\begin{array}{l}
(\mathtt{XP} \times \mathtt{CONSTR} \times \mathtt{F} \times \mathtt{TV} \times (\, \mathit{Value} \cup \{?_{\mathrm{V}}\} \times \mathit{Value} \cup \{?_{\mathrm{V}}\} \to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\}) \\
\quad \times \{\text{complete\_range}, \text{incomplete\_range}\} \times \mathit{Env}) \\
\to \mathit{Position} \times \mathit{Position} \times \mathit{Value} \\
\to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\}
\end{array}
$$

$\underline{\text{value comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}, \mathit{combine\_values}, \mathit{flag}, e](\mathit{from}, \mathit{to}, \mathit{acc}) =$

$$
\begin{array}{l}
\text{let } (e_1, b_1) = [\![\mathtt{CONSTR}]\!](e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto \mathit{from}], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5) : \\
\text{if } b_1 = \bot_{\mathbf{F}} \text{ then} \\
\quad \bot_{\mathrm{V}} \\
\text{else if } b_1 = ?_{\mathbf{F}} \text{ then} \\
\quad ?_{\mathrm{V}} \\
\text{else if } b = \mathbf{f} \text{ then} \\
\quad \mathit{acc} \\
\text{else} \\
\quad \text{let } v = [\![\mathtt{TV}]\!](e_1) : \\
\quad \text{if } v = \bot_{\mathrm{V}} \text{ then} \\
\quad\quad \bot_{\mathrm{V}} \\
\quad \text{else} \\
\quad\quad \text{let } \mathit{newacc} = \mathit{combine\_values}(\mathit{acc}, v) : \\
\quad\quad \text{if } \mathit{newacc} = \bot_{\mathrm{V}} \text{ then} \\
\quad\quad\quad \bot_{\mathrm{V}} \\
\quad\quad \text{else} \\
\quad\quad\quad \text{let } b_2 = [\![\mathtt{F}]\!](e_1{\downarrow}1, e_1{\downarrow}2, e_1{\downarrow}3[\mathtt{this} \mapsto \mathit{newacc}], e_1{\downarrow}4, e_1{\downarrow}5) : \\
\quad\quad\quad \text{if } b_2 = \bot_{\mathbf{F}} \text{ then} \\
\quad\quad\quad\quad \bot_{\mathrm{V}} \\
\quad\quad\quad \text{else if } b_2 = ?_{\mathbf{F}} \text{ then} \\
\quad\quad\quad\quad ?_{\mathrm{V}} \\
\quad\quad\quad \text{else if } b_2 = \mathbf{t} \text{ then} \\
\quad\quad\quad\quad \mathit{newacc} \\
\quad\quad\quad \text{else if } \mathit{from} \ \geq \mathit{to} \text{ then} \\
\quad\quad\quad\quad \text{if } \mathit{flag} = \text{complete\_range then} \\
\quad\quad\quad\quad\quad \mathit{newacc} \\
\quad\quad\quad\quad \text{else} \\
\quad\quad\quad\quad\quad ?_{\mathrm{V}} \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \text{let } \mathit{from}_1 = \mathit{from} + 1 : \\
\quad\quad\quad\quad \underline{\text{value comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}, \mathit{combine\_values}, \mathit{flag}, e](\mathit{from}_1, \mathit{to}, \mathit{newacc}).
\end{array}
$$

Hence, $\underline{\text{value comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}, \mathit{combine\_values}, e]$ is a recursive function that operates on a triple of two positions and a value and returns back a value (including unknown and error values). Given such a triple $(\mathit{from}, \mathit{to}, \mathit{acc})$, recursion is supposed to go through the numbers between *from* and *to*. At each step, the function evaluation either stops returning a value (including $?_{\mathrm{V}}$ and $\bot_{\mathrm{V}}$), or generates a new triple $(\mathit{from} + 1, \mathit{to}, \mathit{combine\_values}(\mathit{acc}, v))$ where v is obtained from `TV`, and proceeds with recursion. In fact, one can see that value comb is a primitive recursive function. The boundary cases are any of the following (*from* indicates the current position):

- when `CONSTR` does not evaluate to **t** in the environment updated by assigning *from* to `XP`,

- when $\texttt{TV}$ evaluates to $\bot_{\mathrm{V}}$ in the updated environment,
- when *combine_values* returns $\bot_{\mathrm{V}}$ while trying to compute the new accumulator *newacc*,
- when the `until` condition `F`, after replacing `this` with *newacc* in it, does not evaluate to **f** in the updated environment,
- when the upper bound of the range is reached, i.e., when *from* ≥ *to*.

If none of these conditions hold, evaluation proceeds by recursion. At the end, when either `CONSTR` fails, `F` succeeds, or *from* exceeds *to* when the range is complete, the result of function evaluation is a value of the form $\mathit{combine\_values}(\cdots(\mathit{combine\_values}(\mathit{combine\_values}(\mathit{acc}, v_1), v_2)\ldots, v_n)$ for some $n \geq 1$. In the other terminal cases the result is either $?_{\mathrm{V}}$ or $\bot_{\mathrm{V}}$.

Semantics of Stream Terms. Semantics of stream variables, bindings, and simple stream terms can be defined easily:

$$\triangleright \quad \texttt{TS} : \mathit{EnvExtended} \to \mathit{Stream}$$

$$[\![\texttt{XS}]\!](e) = e{\downarrow}4(\texttt{XS})$$

$$[\![\texttt{BIND} : \texttt{TS}]\!](e) = (\text{let } e_1 = [\![\texttt{BIND}]\!](e) : \text{if } e_1 = \bot_{\mathrm{E}} \text{ then } ?^{\omega}_{\mathrm{M}} \text{ else } [\![\texttt{TS}]\!](e_1)).$$

$$[\![\texttt{FS}(\texttt{T}_1, \ldots, \texttt{T}_n)]\!](e) = (\text{let } s_1 = [\![\texttt{T}_1]\!](e), \ldots, s_1 = [\![\texttt{T}_n]\!](e) : [\![\texttt{FS}]\!](s_1, \ldots, s_n)).$$

Note that for $\texttt{BIND} : \texttt{TS}$, if $[\![\texttt{BIND}]\!](e)$ is $\bot_{\mathrm{E}}$, then $[\![\texttt{BIND} : \texttt{TS}]\!](e)$ returns $?^{\omega}_{\mathrm{M}}$ and not an error value, unlike the valuations for formulas and position and value terms. The reason is that we do not have erroneous streams.

The semantics of `partial combine` is largely similar to `complete combine`. The main difference is that while the result of `complete combine` is a value of the form

$$\mathit{combine\_values}(\cdots(\mathit{combine\_values}(\mathit{combine\_values}(\mathit{acc}, v_1), v_2)\ldots, v_n),$$

the semantics of `partial combine` is a stream whose known messages have contents of the form

$$\mathit{acc}, \mathit{combine\_values}(\mathit{acc}, v_1), \mathit{combine\_values}(\mathit{combine\_values}(\mathit{acc}, v_1), v_2), \ldots,$$
$$\mathit{combine\_values}(\cdots(\mathit{combine\_values}(\mathit{combine\_values}(\mathit{acc}, v_1), v_2)\ldots, v_n),$$

followed by the stream of unknown messages $?^{\omega}_{\mathrm{M}}$. The known messages look like all partial results computed in the course of computing $\mathit{combine\_values}(\cdots(\mathit{combine\_values}(\mathit{combine\_values}(\mathit{acc}, v_1), v_2)\ldots, v_n)$. This is where the name `partial combine` comes from, compared to `complete combine` for the value case. The formal definition follows:

$$
\begin{array}{l}
[\![\texttt{partial combine[TV}_0\texttt{,FV] XP in TS with RAN CONSTR until F} : \texttt{TV}_1]\!](e) = \\
\quad \text{let } \mathit{current\_time} = \underline{\text{current time}}(e) : \\
\quad \text{if } \mathit{current\_time} = 0 \text{ then} \\
\qquad ?^{\omega}_{\mathrm{M}} \\
\quad \text{else} \\
\qquad \text{let } r = [\![\texttt{RAN}]\!](e) : \\
\qquad \text{if } r = \bot_{\mathrm{P}} \text{ then} \\
\qquad\quad ?^{\omega}_{\mathrm{M}} \\
\qquad \text{else if } r = ?_{\mathrm{P}} \text{ then} \\
\qquad\quad ((\mathit{current\_time}, ?_{\mathrm{V}}), ?^{\omega}_{\mathrm{M}}) \\
\qquad \text{else} \\
\qquad\quad \text{let } (\texttt{XP}_1, \mathit{from}, \mathit{to}_0) = r : \\
\qquad\quad \text{if } \texttt{XP}_1 \neq \texttt{XP} \text{ then}
\end{array}
$$

$?^{\omega}_{\mathrm{M}}$

else

let $p_0 = \max_{pos} : [\![\mathtt{TS}]\!](e)(pos) \neq ?_{\mathrm{M}} \wedge [\![\mathtt{TS}\ominus pos]\!](e) \leq \mathit{current\_time}$:

let $\mathit{to} = \min(p_0, \mathit{to}_0)$:

if $\mathit{to} < \mathit{from}$ then

$?^{\omega}_{\mathrm{M}}$

else

let $v = [\![\mathtt{TV}_0]\!](e)$:

if $v = \bot_{\mathrm{V}}$ then

$?^{\omega}_{\mathrm{M}}$

else

let $\mathit{acc} = ((\mathit{current\_time}, v), ?^{\omega}_{\mathrm{M}})$:

let $\mathit{combine\_values} = [\![\mathtt{FV}]\!](e)$:

let $s_1 = \underline{\text{stream comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}_1, \mathit{combine\_values}, e](\mathit{from}, \mathit{to}, \mathit{acc})$:

let $s_2 = [\![\mathtt{partial\ combine}[\mathtt{TV}_0,\mathtt{FV}]\ \mathtt{XP\ in\ TS\ with\ RAN\ CONSTR}$

$\mathtt{until\ F} : \mathtt{TV}_1]\!](e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, \mathit{current\_time} - 1)$:

$\underline{\text{diff and append}}(s_1, s_2)$.

According to this definition, first it is checked whether the current time point *current_time* is 0 or not. For simplicity, the time points are expressed with natural numbers and 0 indicates the starting time. Since we assume that at the starting time point no stream is observable, `partial combine` for the case $\mathit{current\_time} = 0$ should construct $?^{\omega}_{\mathrm{M}}$, the completely unobservable stream. If we are not at the starting time, then again, like in the case of `complete combine`, we prepare the input for the corresponding function stream comb in two steps: First, computing the range and, second, preparing the accumulator.

Computing the range: We need to evaluate `RAN` and see, whether it is $\bot_{\mathrm{P}}$, $?_{\mathrm{P}}$, or a triple $(\mathtt{XP}_1, \mathit{from}, \mathit{to}_0)$ fixing the range of some position variable $\mathtt{XP}_1$. In the first case the value of the `partial combine` term is the unobservable stream $?^{\omega}_{\mathrm{M}}$. In the second case, with the unknown position $?_{\mathrm{P}}$, we assume that only time tick is propagated to the result stream. That means, the propagated message has the current time stamp and the unknown value, hence the result stream has the form $((\mathit{current\_time}, ?_{\mathrm{V}}), ?^{\omega}_{\mathrm{M}})$. In the third case we check whether $\mathtt{XP}_1$ is the same as `XP`, i.e., whether `RAN` gives the range for `XP` or for some other variable. If it is the other variable, then the value of the `partial combine` term is $?^{\omega}_{\mathrm{M}}$. Otherwise, we have the range $(\mathit{from}, \mathit{to}_0)$ for `XP`, which should be refined with respect to the current time *current_time*: We find the maximal position in the stream `TS` such that the message at that position has the time stamp smaller than the current time. This is where the "currently observable" part of the stream `TS` ends. Hence, the upper bound of the range should be the minimum between that position and to_0, and at the same time not smaller than the lower bound *from*. If such an upper bound does not exist, then we again get the unobservable stream. Otherwise, it is denoted by *to*, and the range of `XP` is $(\mathit{from}, \mathit{to})$.

Preparing the accumulator: The accumulator is supposed to be the stream whose observable part is the message consisting of the current time stamp and the value of the term $\mathtt{TV}_0$. If the latter gives an error, then the value of the `partial combine` term is the unobservable stream $?^{+}_{\mathrm{M}}$. Otherwise, we compute $\underline{\text{stream comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F},\ \mathtt{TV}_1,\ \mathit{combine\_values}, e]$. This is a function whose application to the already computed range of `XP` and the accumulator gives the stream that is very close to the intended result of the valuation of `partial combine`: The right content is there, but the observable messages all carry the same time stamp, the current time *current_time*, no matter whether they have been put into the stream at *current_time* or could have been there at the same position also if the stream was constructed in earlier time points. To make the time stamps correct, we compute the value of the `partial combine` term recursively at the time point *current_time* − 1 and then compare that result with the current result: The difference between them are those messages that appear in the stream at the time point *current_time*. This is the job of diff and append (defined below): It takes the observable part of the stream at *current_time* − 1, appends to it the observable messages of the stream at *current_time* that were not there at *current_time* − 1, and forms the result stream.

Note that, unlike `complete combine`, there is no check in `partial combine` whether the range is complete or not. The reason is that `complete combine` is supposed to perform the complete combination of the values within the range. If the range is incomplete, complete combination is not possible. As for `partial combine`, it has to put on a stream the results of partial combinations within the range and there is no requirement for completeness of those combined values there.

The function stream comb is defined as follows:

$$
\begin{array}{l}
\underline{\text{stream comb}}: \\
\quad (\texttt{XP} \times \texttt{CONSTR} \times \texttt{F} \times \texttt{TV} \times (\mathit{Value} \cup \{?_\text{V}\} \times \mathit{Value} \cup \{?_\text{V}\} \rightarrow \mathit{Value} \cup \{?_\text{V}, \bot_\text{V}\}) \times \mathit{Env}) \\
\quad \rightarrow \mathit{Position} \times \mathit{Position} \times \mathit{Stream} \\
\quad \rightarrow \mathit{Stream} \\
\underline{\text{stream comb}}[\texttt{XP}, \texttt{CONSTR}, \texttt{F}, \texttt{TV}, \mathit{combine\_values}, e](\mathit{from}, \mathit{to}, \mathit{acc}) = \\
\qquad \text{let } (e_1, b_1) = [\![\texttt{CONSTR}]\!](e{\downarrow}1, e{\downarrow}2[\texttt{XP} \mapsto \mathit{from}], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5): \\
\qquad \text{if } b_1 \neq \mathbf{t} \text{ then} \\
\qquad\quad \mathit{acc} \\
\qquad \text{else} \\
\qquad\quad \text{let } v = [\![\texttt{TV}]\!](e_1): \\
\qquad\quad \text{if } v = \bot_\text{V} \text{ then} \\
\qquad\qquad \mathit{acc} \\
\qquad\quad \text{else} \\
\qquad\qquad \text{let } \mathit{newacc} = \underline{\text{combine and join}}(\mathit{acc}, v, \mathit{combine\_values}, \underline{\text{current time}}(e_1)): \\
\qquad\qquad \text{let } b_2 = [\![\texttt{F}]\!](e_1{\downarrow}1, e_1{\downarrow}2, e_1{\downarrow}3[\texttt{this} \mapsto \mathit{newacc}], e_1{\downarrow}4, e_1{\downarrow}5): \\
\qquad\qquad \text{if } b_2 \neq \mathbf{f} \text{ then} \\
\qquad\qquad\quad \mathit{newacc} \\
\qquad\qquad \text{else if } \mathit{from} \geq \mathit{to} \text{ then} \\
\qquad\qquad\quad \mathit{newacc} \\
\qquad\qquad \text{else} \\
\qquad\qquad\quad \text{let } \mathit{from}_1 = \mathit{from} + 1: \\
\qquad\qquad\quad \underline{\text{stream comb}}[\texttt{XP}, \texttt{CONSTR}, \texttt{F}, \texttt{TV}, \mathit{combine\_values}, e](\mathit{from}_1, \mathit{to}, \mathit{newacc}).
\end{array}
$$

Compared to value comb, one can see that there are fewer boundary cases here. The reason is that here we return a stream even if `CONSTR` or `F` (after replacing `this` with *newacc*) evaluates to error or unknown values, while value comb leads in such cases to an error or unknown result. Otherwise, the structures of the definitions of stream comb and value comb are pretty similar.

The actual work of putting messages on the stream in stream comb is done by the auxiliary function called combine and join, which a bit more involved that its counterpart value combination function from value comb. In fact, the value combination function *combine_values* is one of the parameters of combine and join. The other parameters are the stream *acc* itself, a value v to be put in *acc*, and the current time point *current_time*. Putting a new message on the stream corresponds to placing it after the last observable position. The time stamp is *current_time*. As for its content, it is either $?_{\mathrm{V}}$, or is obtained by combining the last value ($\neq ?_{\mathrm{V}}$) from the observable part of the stream *acc* with the value v. In this way, in the observable part of the new stream we keep the observable part of *acc* and add a new message obtained by extending the last known (i.e., whose content $\neq ?_{\mathrm{V}}$) message. This guarantees that the Ascending Time Property is retained.

$$
\begin{aligned}
&\underline{\text{combine and join}} :\\
&\quad \mathit{Stream} \times \mathit{Value} \cup \{?_{\mathrm{V}}\} \times (\mathit{Value} \cup \{?_{\mathrm{V}}\} \times \mathit{Value} \cup \{?_{\mathrm{V}}\} \to \mathit{Value} \cup \{?_{\mathrm{V}}, \perp_{\mathrm{V}}\}) \times \mathit{Time}\\
&\quad \to \mathit{Stream}\\
&\underline{\text{combine and join}}(s, v, \mathit{combine\_values}, \mathit{current\_time}) =\\
&\qquad \text{let } ((t_1, v_1), \ldots, (t_n, v_n), ?_{\mathrm{M}}^{\omega}) = s :\\
&\qquad \text{if } v = ?_{\mathrm{V}} \text{ then}\\
&\qquad\quad \underline{\text{stream join}}(s, ((\mathit{current\_time}, v), ?_{\mathrm{M}}^{\omega}))\\
&\qquad \text{else}\\
&\qquad\quad \text{let } k = \max_i : v_i \neq ?_{\mathrm{V}} :\\
&\qquad\quad \text{let } \mathit{newval} = \mathit{combine\_values}(v_k, v) :\\
&\qquad\quad \text{if } \mathit{newval} = \perp_{\mathrm{V}} \text{ then}\\
&\qquad\qquad \underline{\text{stream join}}(s, ((\mathit{current\_time}, ?_{\mathrm{V}}), ?_{\mathrm{M}}^{\omega}))\\
&\qquad\quad \text{else}\\
&\qquad\qquad \underline{\text{stream join}}(s, ((\mathit{current\_time}, \mathit{newval}), ?_{\mathrm{M}}^{\omega})).
\end{aligned}
$$

The auxiliary function $\underline{\text{stream join}}$ used above and also later, is defined easily:

$$
\begin{aligned}
&\underline{\text{stream join}} : \mathit{Stream} \times \mathit{Stream} \to \mathit{Stream}\\
&\underline{\text{stream join}}(s_1, s_2) =\\
&\qquad \text{let } ((t_1, v_1), \ldots, (t_n, v_n), ?_{\mathrm{M}}^{\omega}) = s_1 :\\
&\qquad ((t_1, v_1), \ldots, (t_n, v_n)) \| s_2.
\end{aligned}
$$

where $\|$ stands for prepending a finite sequence of messages to a stream. Note that $\underline{\text{stream join}}$ is always used with the first argument having the finite observable part.

To finish the definition of the semantics of `partial combine`, we need to define the auxiliary function $\underline{\text{diff and append}}$:

$$
\begin{aligned}
&\underline{\text{diff and append}} : \mathit{Stream} \times \mathit{Stream} \to \mathit{Stream}\\
&\underline{\text{diff and append}}(\mathit{new\_stream}, \mathit{old\_stream}) =\\
&\qquad \text{let } ((t, v_1), \ldots, (t, v_n), ?_{\mathrm{M}}^{\omega}) = \mathit{new\_stream} \ (n \geq 0) :\\
&\qquad \text{let } ((t'_1, v'_1), \ldots, (t'_m, v'_m), ?_{\mathrm{M}}^{\omega}) = \mathit{old\_stream} \ (m \geq 0) :\\
&\qquad ((t'_1, v'_1), \ldots, (t'_m, v'_m)) \| ((t, v_{m+1}), \ldots, (t, v_n), ?_{\mathrm{M}}^{\omega})
\end{aligned}
$$

The way how $\underline{\text{diff and append}}$ is used in `partial combine` guarantees that $n \geq m$ and $t \geq t'_m$. The resulting stream is obtained from *old_stream* by appending to its observable part those observable messages from the *new_stream* that are located in the positions greater that the position of (t'_m, v'_m) in *old_stream*. In this sense, we take a "difference" between *new_stream* and *old_stream* and append it to the end of (the observable part of) *old_stream*. Hence, we have the Ascending Time Property in the resulting stream.

The next stream terms are the `construct` terms. There are two versions of them, one that constructs a stream from the given value(s) and the other that joins together existing streams. We give a generic definition of their valuations below, where *value_or_stream* replaces `TV` and `TS`:

$$[\![\texttt{construct XP in TS with RAN CONSTR} : \mathit{value\_or\_stream}]\!](e) =$$

$$
\begin{array}{l}
\text{let } \mathit{current\_time} = \underline{\text{current time}}(e): \\
\text{if } \mathit{current\_time} = 0 \text{ then} \\
\quad ?^{\omega}_{\mathrm{M}} \\
\text{else} \\
\quad \text{let } r = [\![\mathtt{RAN}]\!](e): \\
\quad \text{if } r = \bot_{\mathrm{P}} \lor r = ?_{\mathrm{P}} \text{ then} \\
\quad\quad ?^{\omega}_{\mathrm{M}} \\
\quad \text{else} \\
\quad\quad \text{let } (\mathtt{XP}_1, \mathit{from}, \mathit{to}_0) = r: \\
\quad\quad \text{if } \mathtt{XP}_1 \neq \mathtt{XP} \text{ then} \\
\quad\quad\quad ?^{\omega}_{\mathrm{M}} \\
\quad \text{else} \\
\quad\quad \text{let } p_0 = \max\limits_{pos} : [\![\mathtt{TS}]\!](e)(\mathit{pos}) \neq ?_{\mathrm{M}} \land [\![\mathtt{TS}\circleddash \mathit{pos}]\!](e) \leq \mathit{current\_time}: \\
\quad\quad \text{let } \mathit{to} = \min(p_0, \mathit{to}_0): \\
\quad\quad \text{if } \mathit{to} < \mathit{from} \text{ then} \\
\quad\quad\quad ?^{\omega}_{\mathrm{M}} \\
\quad\quad \text{else} \\
\quad\quad\quad \text{let } s_1 = \underline{\text{stream construct}}[\mathtt{XP}, \mathtt{CONSTR}, \mathit{value\_or\_stream}, e](\mathit{from}, \mathit{to}, ?^{\omega}_{\mathrm{M}}): \\
\quad\quad\quad \text{let } s_2 = [\![\mathtt{construct\ XP\ in\ TS\ with\ RAN\ CONSTR} : \mathit{value\_or\_stream}]\!] \\
\quad\quad\quad\quad\quad\quad (e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, \mathit{current\_time} - 1): \\
\quad\quad\quad \underline{\text{diff and append}}(s_1, s_2).
\end{array}
$$

The meaning of this term is a stream constructed from the valuations of *value_or_stream*, taken for all XP's from TS within RAN satisfying CONSTR. Like the previous stream construction terms, also here we do the similar "preparatory" work before delegating the task to the recursive function stream construct, which constructs the first approximation of the result stream. The difference and append method discussed about helps also here to get the right time stamps on the messages in the final stream.

The function stream construct is defined in the following way:

$$
\begin{array}{l}
\underline{\text{stream construct}}: \\
\quad (\mathtt{XP} \times \mathtt{CONSTR} \times (\mathtt{TV} + \mathtt{TS}) \times \mathit{Env}) \to \mathit{Position} \times \mathit{Position} \times \mathit{Stream} \to \mathit{Stream} \\
\underline{\text{stream construct}}[\mathtt{XP}, \mathtt{CONSTR}, \mathit{value\_or\_stream}, e](\mathit{from}, \mathit{to}, \mathit{acc}) = \\
\quad\quad \text{let } (e_1, b_1) = [\![\mathtt{CONSTR}]\!](e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto \mathit{from}], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5): \\
\quad\quad \text{if } b_1 \neq \mathbf{t} \text{ then} \\
\quad\quad\quad \mathit{acc} \\
\quad\quad \text{else} \\
\quad\quad\quad \text{let } \mathit{v\_or\_s} = [\![\mathit{value\_or\_stream}]\!](e_1): \\
\quad\quad\quad \text{cases } \mathit{v\_or\_s} \text{ of} \\
\quad\quad\quad\quad \text{isValue}(v) \longrightarrow \text{if } v = \bot_{\mathrm{V}} \text{ then } \mathit{acc} \text{ else let } \mathit{str} = ((\underline{\text{current time}}(e_1), v), ?^{\omega}_{\mathrm{M}}): \\
\quad\quad\quad\quad \text{isStream}(s) \longrightarrow \text{let } \mathit{str} = s: \\
\quad\quad\quad \text{let } \mathit{newacc} = \underline{\text{stream join}}(\mathit{acc}, \mathit{str}): \\
\quad\quad\quad \text{if } \mathit{from} \geq \mathit{to} \text{ then} \\
\quad\quad\quad\quad \mathit{newacc} \\
\quad\quad\quad \text{else}
\end{array}
$$

let $\mathit{from}_1 = \mathit{from} + 1$

$\underline{\text{stream construct}}[\mathtt{XP}, \mathtt{CONSTR}, v\_or\_s, e](\mathit{from}_1, \mathit{to}, \mathit{newacc})$.

We think there is no need to explain this definition in detail, since it uses the constructions and ideas we have already explained earlier. The only new thing is related to the fact that since *value_or_stream* can be value or term, we need to have a recognizer to know in which case we are. This is achieved in the standard way, with the help of the cases statement and the disassemblers isValue and isStream, with their intended meaning.

Semantics of Nonlogical Symbols. We have not defined semantics of FV, FS, and PV above. Nonlogical symbols do not have a fixed semantics. Their meaning may vary from one interpretation to another. Therefore, we only fix the type of their valuation function:

$$[\![\mathtt{FV}]\!] : (\mathit{Value} \cup \{?_\mathrm{V}\} + \mathit{Position} \cup \{?_\mathrm{P}\} + \mathit{Stream})^* \to \mathit{Value} \cup \{?_\mathrm{V}, \bot_\mathrm{V}\}$$
$$[\![\mathtt{FS}]\!] : (\mathit{Value} \cup \{?_\mathrm{V}\} + \mathit{Position} \cup \{?_\mathrm{P}\} + \mathit{Stream})^* \to \mathit{Stream}$$
$$[\![\mathtt{PV}]\!] : (\mathit{Value} \cup \{?_\mathrm{V}\} + \mathit{Position} \cup \{?_\mathrm{P}\} + \mathit{Stream})^* \to \mathit{MK}$$

4 Example: File Download Monitor

In this section we describe an example illustrating how a file download monitor can be modeled in our language. It is supposed to monitor the input TCP/IP stream to detect viruses in multi-part/multi-file downloads. The scenario is as follows:

- The goal is to prevent download of files containing malware.
  - Monitor must not forward last TCP/IP package of the file.
  - In general, only the complete file can be analyzed for malware (because it may need to be decompressed).

    `file.zip → file`
- A file may be split into multiple parts.
  - Each part may be hosted on a different server under a different name.

    `file.zip.001, file.zip.002, ...`
- A part may be transferred in multiple downloads.
  - Each download consists of a range of bytes from the part.

    ```
    GET /file.zip.001 HTTP/1.1
    Range: bytes=500-999
    ```
  - Downloads may refer to different hosts and use different protocols.

    `HTTP, FTP, ...`

We suppose that the TCP/IP stream is preprocessed before it is passed to the monitor so that it can be adequately modeled by the stream construct of our language. Such a transformation includes, in particular, rearranging messages in the stream in such a way that they are ordered with respect to the time stamp they carry, to guarantee the Ascending Time Property. The preprocessing itself is not modeled in the language.

Typically, one would be interested to answer the questions:

- Related to *analyzing the file:*

  - Does the file contain a virus?

- Related to *combining parts to files:*
  - Does a certain part belong to a certain file?
  - Does a collection of parts represent the whole file?

- Related to *combining downloads to parts:*
  - Does a certain download belong to a certain part?
  - Does a collection of downloads represent the whole part?

To fix the terminology, we say that each message in the input TCP/IP stream is a pair of the time stamp and the message content (value) that we call the *packet*. When it does not cause confusion, we do not distinguish between a message and its content. Logically, the TCP/IP stream may be structured into different substreams consisting of messages exchanged between the same source and destination or vice versa. That means, two messages $m_1 = (t_1, packet_1)$ and $m_2 = (t_2, packet_2)$ belong to the same such substream, if the sources and destinations of m_1 and m_2 are the same, or if the source of m_1 is the destination of m_2 and the source of m_2 is the destination of m_1. We assume that each message carries the information about its source and destination in the *packet* part and there are means to extract it from there.

For the given TCP/IP stream, we would like to construct new streams that we call *connections*. One feature of connections is that all messages in the same connection share the same source/destination as discussed above. Let us illustrate a connection construction on an example. Let $(t_1, packet_1)$, $(t_2, packet_2)$, $(t_3, packet_3), \ldots$ be the input TCP/IP stream. Assume that the messages at every fourth position $(t_1, packet_1), (t_5, packet_5), (t_9, packet_9), \ldots$ form the substream whose messages share the same source and destination in the way described above. Let us denote this stream by s_0. Assume also that (the contents of) some message triples in s_0 can be combined into one coherent piece of information. The triples are formed from the messages at the following positions:

- $1, 3, 5$
- $2, 4, 6$
- $7, 9, 11$
- $8, 10, 12$
- $13, 15, 17$
- $\ldots$

That means that, for instance, one such a piece can be formed from the packets $packet_1, packet_9$, and $packet_{17}$. Let *current_time* be the current time and *comb* be the combination function. Then the connection *conn* should have the form:

$$
\begin{aligned}
&(current\_time, packet_1), (current\_time + 1, comb(packet_1, packet_9)),\\
&(current\_time + 2, comb(comb(packet_1, packet_9), packet_{17})),\\
&(current\_time + 3, packet_5), (current\_time + 4, comb(packet_5, packet_{13})),\\
&(current\_time + 5, comb(comb(packet_5, packet_{13}), packet_{21})),\\
&(current\_time + 6, packet_{25}), \ldots
\end{aligned}
$$

In this stream, the messages $comb(comb(packet_1, packet_9), packet_{17})$ and $comb(comb(packet_5, packet_{13}), packet_{21})$ are called the complete ones, because they represent that coherent piece of information we were talking about. The other messages are partial. We assume that there are means to determine whether a message is partial or complete.

Once we have connections, we can form new streams from them. One such stream can be the stream *http*: A message (t, v) from a connection stream is put in *http* if v forms a complete http download. Again, we assume that there are means to check whether a message content is a complete http download. We can form also the stream *ftp* of ftp downloads. This construction is slightly more involved than the *http* stream construction. Finally, we can merge the *http* and *ftp* streams together to form the *downloads* stream.

Now, in the *downloads* stream $(t_1^d, download_1), (t_2^d, download_2), (t_3^d, download_3), \ldots$, where each *download* is either a http or an ftp complete download, several messages may correspond to the same file part. For instance,

- combining $download_1$ and $download_3$ gives part 1 of the first file: `file1.zip.001`,
- combining $download_2$, $download_7$, $download_8$, and $download_9$ gives part 1 of the second file: `file2.zip.001`,
- combining $download_4$ and $download_6$ gives part 2 of the second file: `file2.zip.002`,
- $download_5$ gives part 2 of the first file `file1.zip.002`.

From the *downloads* stream, we can form the stream *parts* whose messages are such file parts: $(t_1^p, \texttt{file1.zip.001}), (t_2^p, \texttt{file2.zip.001}),\ (t_3^p, \texttt{file2.zip.002}),\ (t_4^p, \texttt{file1.zip.002}), \ldots$. These file parts themselves are complete.

Finally, the monitor will use the stream *parts* to combine file parts into files and check whether they contain virus. Of course, the actual check for the viruses is beyond the language.

Now we show how these ideas can be expressed in our language. Instead of writing one large formula for the monitor, we use definitions of some functions and predicates as a shorthand notation to make it more comprehensible. To ease reading, we also add keywords <u>predicate</u>, <u>function</u>, <u>stream</u>, etc. to those definitions. These defined symbols are written in serif, variables in *italic*. The external functions, whose definitions are not provided, are written in SMALL CAPS.

The top-level monitor is the formula:

$$
\begin{array}{l}
\texttt{monitor}\ current\_position : \\
\quad \texttt{value}\ part = \mathsf{parts}@current\_position \\
\qquad : \mathsf{start\_file}(current\_position, part)\ \texttt{=>} \\
\qquad\quad \texttt{value}\ file = \mathsf{get\_file}(current\_position, part) \\
\qquad\quad : \textsc{NoVirus}(file).
\end{array}
$$

The predicate start_file is true if no part in the past (restricted by the given TIMEOUT) referred to the same file as the current one:

$$
\begin{array}{l}
\underline{\text{predicate}}\ \mathsf{start\_file}(current\_position, part)\ \texttt{<=>} \\
\quad \textsc{CompletePart}(part)\ \texttt{/\textbackslash} \\
\quad \sim \texttt{exists}\ pos\ \texttt{in}\ \mathsf{parts}\ \texttt{with}\ current\_position - \textsc{Timeout} =< pos < current\_position : \\
\qquad \textsc{SameFile}(part, \mathsf{parts}@pos).
\end{array}
$$

The function get_file gets file whose first part has currently started:

$$
\begin{array}{l}
\underline{\text{function}}\ \mathsf{get\_file}(current\_position, part) = \\
\quad \texttt{value}\ set = \\
\qquad \texttt{complete combine}[\textsc{EmptySet}, \textsc{AddPart}] \\
\qquad\quad pos\ \texttt{in}\ \mathsf{parts} \\
\qquad\quad \texttt{with}\ current\_position =< pos \\
\qquad\quad \texttt{value}\ part_0 = \mathsf{parts}@pos
\end{array}
$$

$\texttt{satisfying}\ \textsc{SameFile}(part, part_0)$

$\texttt{until}\ \textsc{CompleteSet}(\texttt{this})$

$: part_0$

$: \textsc{File}(set).$

The monitor, as one can see, works on the parts stream. We give now its definition based on the ideas above:

$\underline{\text{stream}}\ \text{parts} =$

$\texttt{construct}$

$start\ \texttt{in}\ \text{downloads}$

$\texttt{with}\ 0 =< start$

$\texttt{value}\ download = \text{downloads}@start$

$\texttt{satisfying}\ \text{start\_part}(start, download)$

$: \texttt{complete}\ \texttt{combine}[\textsc{EmptyPart}, \textsc{AddDownload}]$

$now\ \texttt{in}\ \text{downloads}$

$\texttt{with}\ start =< now$

$\texttt{value}\ download_0 = \text{downloads}@now$

$\texttt{satisfying}\ \textsc{SamePart}(download, download_0)$

$\texttt{satisfying}\ \textsc{CompleteDownload}(download_0)$

$\texttt{until}\ \textsc{CompletePart}(\texttt{this})$

$: download_0.$

The predicate start_part is true if no download in the past referred to the current part:

$\underline{\text{predicate}}\ \text{start\_part}(start, download)\ \texttt{<=>}$

$\textsc{CompleteDownload}(download)\ \texttt{/\textbackslash}$

$\sim\ \texttt{exists}\ pos\ \texttt{in}\ \text{downloads}\ \texttt{with}\ start - \textsc{Timeout} < pos < start :$

$\textsc{SamePart}(download, \text{downloads}@pos).$

The downloads stream is obtained by merging the http and ftp downloads streams, where merging is an external function:

$\underline{\text{stream}}\ \text{downloads} = \textsc{Merge}(\text{http}, \text{ftp}).$

The http downloads stream is constructed from the connections stream relatively easily, we just extract complete http downloads from there:

$\underline{\text{stream}}\ \text{http} =$

$\texttt{construct}$

$current\_position\ \texttt{in}\ \text{connections}$

$\texttt{with}\ 0 =< current\_position$

$: \textsc{CompleteHTTPDownload}(\text{connections}@current\_position).$

Construction of the ftp stream is more involved. First, we construct an intermediate stream of ftp requests and then put requests and the corresponding ftp downloads together:

$\underline{\text{stream}}\ \text{ftprequests} =$

$\texttt{construct}$

$current\_position\ \texttt{in}\ \text{connections}$

with 0 =< *current_position*
value *request* = connections@*current_position*
satisfying COMPLETEFTPREQUEST(*request*)
: *request*.

<u>stream</u> ftp =
construct
current_position **in** ftprequests
with 0 =< *current_position*
value *request* = ftprequests@*current_position*
position *p* =
min
pos **in** connections
with 0 =< *pos*
: **value** *connection* = connections@*pos*
: COMPLETEFTPDOWNLOAD(*connection*) /\
ftprequests◷*current_position* ≤ connections◷*pos* /\
FTPMATCH(*request*, *connection*)
: FTPDOWNLOAD(*request*, connections@*p*).

Both http and ftp downloads stream use the connections stream, which is constructed from the given tcpip stream in the following way:

<u>stream</u> connections =
construct
start **in** tcpip
with 0 =< *start*
value $packet_0$ = tcpip@*start*
satisfying start_connection($packet_0$)
: **partial combine**[EMPTYCONNECTION, ADDPACKET]
pos **in** tcpip
with *start* =< *pos*
value $packet_1$ = tcpip@*pos*
satisfying same_connection($packet_0, packet_1$)
until end_connection($packet_0, packet_1$)
: *packet*.

Here we used the predicates start_connection, same_connection, and end_connection. Their definitions follow. start_connection succeeds on a packet that starts a connection:

<u>predicate</u> start_connection(*packet*) **<=>** SYN(*packet*) /\ ~ ACK(*packet*).

same_connection is true if both of its arguments belong to the same connection:

<u>predicate</u> same_connection($packet_0, packet_1$) **<=>**
(SOURCE($packet_0$) = SOURCE($packet_1$) /\ DEST($packet_0$) = DEST($packet_1$)) \/
(SOURCE($packet_0$) = DEST($packet_1$) /\ DEST($packet_0$) = SOURCE($packet_1$)).

The binary predicate $=$ is interpreted as the syntactic equality over values.

Finally, end_connection is true if its second argument closes the connection started by the first argument:

$$\underline{\text{predicate}}\ \text{end\_connection}(\mathit{packet}_0, \mathit{packet}_1)\ \texttt{<=>}$$
$$\textsc{Fin}(\mathit{packet}_1)\ \texttt{\textbackslash/}$$
$$(\textsc{Reset}(\mathit{packet}_1)\ \texttt{/\textbackslash}\ \textsc{Source}(\mathit{packet}_1) = \textsc{Dest}(\mathit{packet}_0))\ \texttt{\textbackslash/}$$
$$(\textsc{Syn}(\mathit{packet}_1)\ \texttt{/\textbackslash}\ \texttt{\~{}}\ \textsc{Ack}(\mathit{packet}_1)).$$

The first line in the body of the definition ($\textsc{Fin}(\mathit{packet}_1)$) corresponds to the normal end. The next line models the server reset, the last one – new start.

Acknowledgments

The authors thank the project partner companies: SecureGuard GmbH and RISC Software GmbH.

References

[1] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The forspec temporal logic: A new temporal property-specification language. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211. Springer, 2002.

[2] Arnon Avron and Beata Konikowska. Proof systems for reasoning about computation errors. *Studia Logica*, 91(2):273–293, 2009.

[3] Behnam Banieqbal and Howard Barringer. Temporal logic with fixed points. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1987.

[4] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program monitoring with LTL in EAGLE. In *IPDPS*. IEEE Computer Society, 2004.

[5] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[6] Howard Barringer, Klaus Havelund, David E. Rydeheard, and Alex Groce. Rule systems for runtime verification: A short tutorial. In Saddek Bensalem and Doron Peled, editors, *RV*, volume 5779 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2009.

[7] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3):675–706, 2010.

[8] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[9] J. Richard Büchi. On a decision method in restricted second order arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Proc. International Congress on Logic, Method, and Philosophy of Science*, pages 1–12, Stanford, CA, 1962. Stanford University Press.

[10] Feng Chen and Grigore Rosu. Mop: an efficient and generic runtime verification framework. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 569–588. ACM, 2007.

[11] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 2000.

[12] Doron Drusinsky. The temporal rover and the atg rover. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.

[13] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal basis of fairness. In Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors, *POPL*, pages 163–173. ACM Press, 1980.

[14] Klaus Havelund and Grigore Rosu. Monitoring java programs with java pathexplorer. *Electr. Notes Theor. Comput. Sci.*, 55(2):200–217, 2001.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[16] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *ECRTS*, pages 114–122. IEEE Computer Society, 1999.

[17] Stephen C. Kleene. *Introduction to Metamathematics.* North Holland, 1952.

[18] John McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[19] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the mop runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14(3):249–289, 2012.

[20] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[21] George Spanoudakis, Christos Kloukinas, and Khaled Mahbub. The serenity runtime monitoring framework. In Spyros Kokolakis, Antonio Maña Gómez, and George Spanoudakis, editors, *Security and Dependability for Ambient Intelligence*, volume 45 of *Advances in Information Security*, pages 213–237. Springer, 2009.

[22] Moshe Y. Vardi. A temporal fixpoint calculus. In Jeanne Ferrante and P. Mager, editors, *POPL*, pages 250–259. ACM Press, 1988.

[23] Moshe Y. Vardi. Alternating automata and program verification. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1995.

[24] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

A Syntax of the LGAL

A.1 Notation

The alphabet (terminal symbols)

`XP`	:	Position variables
`XV`	:	Value variables (containing the variable `this`)
`XS`	:	Stream variables
`XF`	:	Formula variables
`0`, `1`, ...,	:	Constant position function symbols (logical)
`+`, `-`	:	Binary position function symbols (logical)
⌚, @	:	Binary value function symbols (logical)
`FV`	:	Fixed arity value function symbols (nonlogical)
`FS`	:	Fixed arity stream function symbols (nonlogical)
`<`, `=<`	:	Binary position predicate symbols (logical)
`PV`	:	Fixed arity value predicate symbols (nonlogical)
`true`, `false`	:	Nullary Connectives
`~`	:	Unary connective
`/\`, `\/`, `=>`, `<=>`	:	Binary connectives
`forall`, `exists`, `monitor`	:	Formula quantifiers
`max`, `min`	:	Quantifiers for position terms
`num`, `complete combine`	:	Quantifiers for value terms
`partial combine`, `construct`	:	Quantifiers for stream terms
`formula`, `position`, `value`, `stream`	:	Local binders
`in`, `with`, `until`, `satisfying`	:	Specifiers (auxiliary symbols)
(,), [,], ",", ":"	:	Punctuation marks (auxiliary symbols)
=	:	Special symbol for local binding (auxiliary symbol)

Defined (nonterminal) symbols

`M`	:	Monitor
`F`	:	Formula
`BIND`	:	Binding
`CONSTR`	:	Constraint
`RAN`	:	Position range
`PP`	:	Position predicate
`N`	:	Position constant
`TP`	:	Position term
`TV`	:	Value term
`TS`	:	Stream term
`T`	:	Term

A.2 Definitions

`M` ::= `monitor XP : F`.

`F` ::= `XF` | `BIND : F`
  | `true` | `false` | `PV`($\mathtt{T}_1, \ldots, \mathtt{T}_n$)
  | `~F` | $\mathtt{F}_1$ `/\` $\mathtt{F}_2$ | $\mathtt{F}_1$ `\/` $\mathtt{F}_2$ | $\mathtt{F}_1$ `=>` $\mathtt{F}_2$ | $\mathtt{F}_1$ `<=>` $\mathtt{F}_2$
  | `forall XP in TS with RAN : F` | `exists XP in TS with RAN : F`

`BIND` ::= `formula XF = F` | `position XP = TP` | `value XV = TV` | `stream XS = TS`.

CONSTR ::= ϵ | satisfying F CONSTR | BIND CONSTR.

Remark: ϵ stands for the empty constraint.

RAN ::= TP PP XP | $\mathtt{TP}_1$ $\mathtt{PP}_1$ XP $\mathtt{PP}_2$ $\mathtt{TP}_2$ Default value for $\mathtt{TP}_1$ $\mathtt{PP}_1$: 0 =<

PP ::= < | =<

TP ::= XP | BIND : TP
| 0 | TP + N | TP − N
| max XP in TS with RAN : F | min XP in TS with RAN : F

N ::= 0 | 1 | $\cdots$

TV ::= XV | BIND : TV
| TS◷TP | TS@TP
| FV($\mathtt{T}_1, \ldots, \mathtt{T}_n$) | num XP in TS with RAN : F
| complete combine[$\mathtt{TV}_0$,FV] XP in TS with RAN CONSTR until F : $\mathtt{TV}_1$
Default value for F in “until F” : false

TS ::= XS | BIND : TS
| FS($\mathtt{T}_1, \ldots, \mathtt{T}_n$)
| partial combine[$\mathtt{TV}_0$,FV] XP in TS with RAN CONSTR until F : $\mathtt{TV}_1$
Default value for F in “until F” : false
| construct XP in TS with RAN CONSTR : TV
| construct XP in $\mathtt{TS}_1$ with RAN CONSTR : $\mathtt{TS}_2$

T ::= TP | TV | TS

B Semantics of the LGAL

B.1 Notation

Unknowns and Errors

$?_{\mathrm{P}}$	:	Unknown position
$\bot_{\mathrm{P}}$	:	Error position
$?_{\mathrm{V}}$	:	Unknown value
$\bot_{\mathrm{V}}$	:	Error value
$?_{\mathbf{F}}$	:	Unknown truth value
$\bot_{\mathbf{F}}$	:	Error truth value
$?_{\mathrm{M}}$	:	Unknown (unobservable) message
$?^{\omega}_{\mathrm{M}}$	:	Unknown (unobservable) stream
$\bot_{\mathrm{E}}$	:	Error environment

Domain Constructors

$\to$	:	Function domain
$\times$	:	Product domain
$+$	:	Sum domain
$*$	:	Kleene closure
$\mathcal{P}$	:	Powerset

Semantic Domains

Stream	:	$\mathbb{N} \to \mathit{Message}$, for stream terms
Message	:	$(\mathit{Time} \times (\mathit{Value} \cup \{?_{\mathrm{V}}\})) \cup \{?_{\mathrm{M}}\}$
Time	:	$\mathbb{N}$, for value terms (the time stamp of a message)
Value	:	$\mathbb{N} + \mathit{Char}^*$, for value terms (general)
Position	:	$\mathbb{N}$, for position terms
MK	:	$\{\mathbf{t}, \mathbf{f}, ?_{\mathbf{F}}, \bot_{\mathbf{F}}\}$, for formulas (McCarthy-Kleene)

Environments

Env	:	$\mathit{EnvFormula} \times \mathit{EnvPosition} \times \mathit{EnvValue} \times \mathit{EnvStream}$
EnvExtended	:	$\mathit{EnvFormula} \times \mathit{EnvPosition} \times \mathit{EnvValue} \times \mathit{EnvStream} \times \mathit{CurrTime}$
EnvFormula	:	$\mathtt{XF} \to \{\mathbf{t}, \mathbf{f}, ?_{\mathbf{F}}\}$
EnvPosition	:	$\mathtt{XP} \to \mathit{Position}$
EnvValue	:	$\mathtt{XV} \to \mathit{Value}$
EnvStream	:	$\mathtt{XS} \to \mathit{Stream}$
CurrTime	:	*Time*
$\bot_{\mathrm{E}}$	:	Error environment

Additional Operations

$\Vert$	:	$(m_1, \ldots, m_n)\Vert s$ stands for prepending the finite sequence of messages $m_1, \ldots, m_n$ to the stream s.

B.2 Valuation Function

▷ $[\![\mathtt{M}]\!] : \mathit{Env} \to \mathcal{P}(\mathit{Position})$

$[\![\mathtt{monitor}\ \mathtt{XP} : \mathtt{F}]\!](e) = \{p \in \mathit{Position} : [\![\mathtt{F}]\!](e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto p], e{\downarrow}3, e{\downarrow}4) = \mathbf{f}\}$

▷ $[\![\mathtt{F}]\!] : \mathit{Env} \to \mathit{MK}$

$[\![\mathtt{F}]\!](e) =$

$\quad$ if $(\exists\ t \in \mathbb{N} : [\![\mathtt{F}]\!](e\downarrow 1, e\downarrow 2, e\downarrow 3, e\downarrow 4, t) \neq ?_{\mathbf{F}})$ then

$\quad\quad$ let $t = \min_{t \in \mathbb{N}} : [\![\mathtt{F}]\!](e\downarrow 1, e\downarrow 2, e\downarrow 3, e\downarrow 4, t) \neq ?_{\mathbf{F}} : [\![\mathtt{F}]\!](e\downarrow 1, e\downarrow 2, e\downarrow 3, e\downarrow 4, t)$

$\quad$ else

$\quad\quad$ $?_{\mathbf{F}}$

$\triangleright$ $[\![\mathtt{F}]\!] : \mathit{EnvExtended} \to \mathit{MK}$

$[\![\mathtt{XF}]\!](e) = e\downarrow 1(\mathtt{XF})$.

$[\![\mathtt{BIND : F}]\!](e) = (\text{let } e_1 = [\![\mathtt{BIND}]\!](e) : \text{if } e_1 = \bot_{\mathrm{E}} \text{ then } \bot_{\mathbf{F}} \text{ else } [\![\mathtt{F}]\!](e_1))$.

$[\![\mathtt{true}]\!](e) = \mathbf{t}$.

$[\![\mathtt{false}]\!](e) = \mathbf{f}$.

$[\![\mathtt{PV(T}_1, \ldots, \mathtt{T}_n\mathtt{)}]\!](e) =$

$\quad$ let $v_1 = [\![\mathtt{T}_1]\!](e), \ldots, v_n = [\![\mathtt{T}_n]\!](e)$:

$\quad$ if $(\exists\, 1 \le i \le n : v_i = \bot_{\mathrm{V}} \vee v_i = \bot_{\mathrm{P}})$ then $\bot_{\mathbf{F}}$ else $[\![\mathtt{PV}]\!](v_1, \ldots, v_n)$.

$[\![\mathtt{\sim F}]\!](e) = (\text{let } b = [\![\mathtt{F}]\!](e) : \mathbf{not}(b))$.

$[\![\mathtt{F}_1 \mathtt{/\backslash} \mathtt{F}_2]\!](e) = (\text{let } b_1 = [\![\mathtt{F}_1]\!](e), b_2 = [\![\mathtt{F}_2]\!](e) : \mathbf{and}(b_1, b_2))$.

$[\![\mathtt{F}_1 \mathtt{\backslash/} \mathtt{F}_2]\!](e) = (\text{let } b_1 = [\![\mathtt{F}_1]\!](e), b_2 = [\![\mathtt{F}_2]\!](e) : \mathbf{or}(b_1, b_2))$.

$[\![\mathtt{F}_1 \mathtt{=>} \mathtt{F}_2]\!](e) = (\text{let } b_1 = [\![\mathtt{F}_1]\!](e), b_2 = [\![\mathtt{F}_2]\!](e) : \mathbf{implies}(b_1, b_2))$.

$[\![\mathtt{F}_1 \mathtt{<=>} \mathtt{F}_2]\!](e) = (\text{let } b_1 = [\![\mathtt{F}_1]\!](e), b_2 = [\![\mathtt{F}_2]\!](e) : \mathbf{iff}(b_1, b_2))$.

$[\![\mathtt{forall\ XP\ in\ TS\ with\ RAN\ : F}]\!](e) =$

$\quad$ let $\mathit{collected} = \underline{\text{collect}}[\mathtt{XP}, \mathtt{TS}, [\![\mathtt{RAN}]\!], \mathtt{F}](e)$:

$\quad$ if $\mathit{collected} = \bot_{\mathrm{P}}$ then

$\quad\quad$ $\bot_{\mathbf{F}}$

$\quad$ else if $\mathit{collected} = ?_{\mathrm{P}}$ then

$\quad\quad$ let $b_0 = [\![\mathtt{F}]\!](e\downarrow 1, e\downarrow 2[\mathtt{XP} \mapsto ?_{\mathrm{P}}], e\downarrow 3, e\downarrow 4, e\downarrow 5)$:

$\quad\quad$ if $b_0 = \mathbf{f}$ then

$\quad\quad\quad$ $\mathbf{f}$

$\quad\quad$ else if $b_0 = \bot_{\mathbf{F}}$ then

$\quad\quad\quad$ $\bot_{\mathbf{F}}$

$\quad\quad$ else

$\quad\quad\quad$ $?_{\mathbf{F}}$

$\quad$ else

$\quad\quad$ let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected}$:

$\quad\quad$ if $(\forall\ (p, b) \in \mathit{pairs} : b = \mathbf{t})$ then

$\quad\quad\quad$ if flag = complete_range then

$\quad\quad\quad\quad$ $\mathbf{t}$

$\quad\quad\quad$ else

$\quad\quad\quad\quad$ $?_{\mathbf{F}}$

$\quad\quad$ else if $(\forall\ (p, b) \in \mathit{pairs} : b = \mathbf{t} \vee b = ?_{\mathbf{F}})$ then

$\quad\quad\quad$ $?_{\mathbf{F}}$

$\quad\quad$ else

$\quad\quad\quad$ let $p_1 = \min_{p} : (p, b) \in \mathit{pairs} \wedge (b = \mathbf{f} \vee b = \bot_{\mathbf{F}})$:

$\quad\quad\quad$ let $(p_1, b_1) \in \mathit{pairs}$:

$\quad\quad\quad$ b_1.

$[\![\texttt{exists XP in TS with RAN :F}]\!](e) =$

$\quad$ let $\mathit{collected} = \underline{\text{collect}}[\texttt{XP}, \texttt{TS}, [\![\texttt{RAN}]\!], \texttt{F}](e)$:

$\quad$ if $\mathit{collected} = \bot_{\mathrm{P}}$ then

$\quad\quad \bot_{\mathbf{F}}$

$\quad$ else if $\mathit{collected} = ?_{\mathrm{P}}$ then

$\quad\quad$ let $b_0 = [\![\texttt{F}]\!](e{\downarrow}1, e{\downarrow}2[\texttt{XP} \mapsto ?_{\mathrm{P}}], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)$:

$\quad\quad$ if $b_0 = \mathbf{t}$ then

$\quad\quad\quad \mathbf{t}$

$\quad\quad$ else if $b_0 = \bot_{\mathbf{F}}$ then

$\quad\quad\quad \bot_{\mathbf{F}}$

$\quad\quad$ else

$\quad\quad\quad ?_{\mathbf{F}}$

$\quad$ else

$\quad\quad$ let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected}$:

$\quad\quad$ if $(\forall\, (p, b) \in \mathit{pairs} : b = \mathbf{f})$ then

$\quad\quad\quad$ if flag = complete_range then

$\quad\quad\quad\quad \mathbf{f}$

$\quad\quad\quad$ else

$\quad\quad\quad\quad ?_{\mathbf{F}}$

$\quad\quad$ else if $(\forall\, (p, b) \in \mathit{pairs} : b = \mathbf{f} \vee b = ?_{\mathbf{F}})$ then

$\quad\quad\quad ?_{\mathbf{F}}$

$\quad\quad$ else

$\quad\quad\quad$ let $p_1 = \min_p : (p, b) \in \mathit{pairs} \wedge (b = \mathbf{t} \vee b = \bot_{\mathbf{F}})$:

$\quad\quad\quad$ let $(p_1, b_1) \in \mathit{pairs}$:

$\quad\quad\quad b_1$.

▷ $[\![\texttt{BIND}]\!] : \mathit{EnvExtended} \to \mathit{EnvExtended} \cup \{\bot_{\mathrm{E}}\}$

$[\![\texttt{formula XF = F}]\!](e) =$

$\quad$ let $b = [\![\texttt{F}]\!](e)$:

$\quad$ if $b = \bot_{\mathbf{F}}$ then $\bot_{\mathrm{E}}$ else $(e{\downarrow}1[\texttt{XF} \mapsto b], e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)$.

$[\![\texttt{position XP = TP}]\!](e) =$

$\quad$ let $p = [\![\texttt{TP}]\!](e)$:

$\quad$ if $p = \bot_{\mathrm{P}}$ then $\bot_{\mathrm{E}}$ else $(e{\downarrow}1, e{\downarrow}2[\texttt{XP} \mapsto p], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)$.

$[\![\texttt{value XV = TV}]\!](e) =$

$\quad$ let $v = [\![\texttt{TV}]\!](e)$:

$\quad$ if $v = \bot_{\mathrm{V}}$ then $\bot_{\mathrm{E}}$ else $(e{\downarrow}1, e{\downarrow}2, e{\downarrow}3[\texttt{XV} \mapsto v], e{\downarrow}4, e{\downarrow}5)$.

$[\![\texttt{stream XS = TS}]\!](e) =$

$\quad$ let $s = [\![\texttt{TS}]\!](e)$:

$\quad (e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4[\texttt{XS} \mapsto s], e{\downarrow}5)$.

▷ $[\![\texttt{CONSTR}]\!] : \mathit{EnvExtended} \to \mathit{EnvExtended} \times \mathit{MK}$

$[\![\epsilon]\!](e) = (e, \mathbf{t})$.

$[\![\texttt{satisfying F CONSTR}]\!](e) =$

let $b = [\![\mathtt{F}]\!](e)$:

if $b \neq \mathbf{t}$ then (e, b) else $[\![\mathtt{CONSTR}]\!](e)$.

$[\![\mathtt{BIND\ CONSTR}]\!](e) =$

let $e_1 = [\![\mathtt{BIND}]\!](e)$:

if $e_1 = \bot_{\mathrm{E}}$ then $(\bot_{\mathrm{E}}, \bot_{\mathbf{F}})$ else $[\![\mathtt{CONSTR}]\!](e_1)$.

▷ $[\![\mathtt{RAN}]\!] : \mathit{EnvExtended} \to (\mathtt{XP} \times \mathit{Position} \times \mathit{Position} \cup \{\infty\}) \cup \{?_{\mathrm{P}}, \bot_{\mathrm{P}}\}$

$[\![\mathtt{TP}_1\ \mathtt{PP}_1\ \mathtt{XP}\ \mathtt{PP}_2\ \mathtt{TP}_2]\!](e) =$

let $p_1 = [\![\mathtt{TP}_1]\!](e)$:

let $p_2 = [\![\mathtt{TP}_2]\!](e)$:

if $p_1 = \bot_{\mathrm{P}} \ \vee \ p_2 = \bot_{\mathrm{P}}$ then

$\bot_{\mathrm{P}}$

else if $p_1 = ?_{\mathrm{P}} \vee p_2 = ?_{\mathrm{P}}$ then

$?_{\mathrm{P}}$

else

let *from* =

if $\mathtt{PP}_1 =$ "<" then

$p_1 + 1$

else

p_2 :

let *to* =

if $\mathtt{PP}_2 =$ "<" then

$p_2 - 1$

else

p_2 :

$(\mathtt{XP}, \mathit{from}, \mathit{to})$.

$[\![\mathtt{TP\ PP\ XP}]\!](e) =$

let $p = [\![\mathtt{TP}]\!](e)$:

if $p = \bot_{\mathrm{P}}$ then

$\bot_{\mathrm{P}}$

else if $p = ?_{\mathrm{P}}$ then

$?_{\mathrm{P}}$

else

let *from* =

if $\mathtt{PP} =$ "<" then

$p + 1$

else

p :

$(\mathtt{XP}, \mathit{from}, \infty)$.

▷ $[\![\mathtt{TP}]\!] : \mathit{EnvExtended} \to \mathit{Position} \cup \{?_{\mathrm{P}}, \bot_{\mathrm{P}}\}$

$[\![\mathtt{XP}]\!](e) = e{\downarrow}2(\mathtt{XP})$.

$[\![\mathtt{BIND} : \mathtt{TP}]\!](e) = (\text{let } e_1 = [\![\mathtt{BIND}]\!](e) : \text{if } e_1 = \bot_{\mathrm{E}} \text{ then } \bot_{\mathrm{P}} \text{ else } [\![\mathtt{TP}]\!](e_1))$.

$[\![\texttt{0}]\!](e) = 0, \quad [\![\texttt{1}]\!](e) = 1, \ldots.$

$[\![\texttt{TP} + \texttt{N}]\!](e) = [\![\texttt{TP}]\!](e) + [\![\texttt{N}]\!](e)).$

$[\![\texttt{TP} - \texttt{N}]\!](e) = \max([\![\texttt{TP}]\!](e) - [\![\texttt{N}]\!](e), 0).$

$[\![\texttt{max XP in TS with RAN} : \texttt{F}]\!](e) =$

$\quad$ let $\mathit{collected} = \underline{\mathrm{collect}}[\texttt{XP}, \texttt{TS}, [\![\texttt{RAN}]\!], [\![\texttt{F}]\!]](e)$:

$\quad$ if $\mathit{collected} = \bot_{\mathrm{P}}$ then

$\quad\quad \bot_{\mathrm{P}}$

$\quad$ else if $\mathit{collected} = ?_{\mathrm{P}}$ then

$\quad\quad ?_{\mathrm{P}}$

$\quad$ else

$\quad\quad$ let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected}$:

$\quad\quad$ if $(\exists\ (p, b) \in \mathit{pairs} : b = \bot_{\mathbf{F}})$ then

$\quad\quad\quad \bot_{\mathrm{P}}$

$\quad\quad$ else if $(\forall\ (p, b) \in \mathit{pairs} : b \neq \mathbf{t})$ then

$\quad\quad\quad ?_{\mathrm{P}}$

$\quad\quad$ else if $\mathit{flag} = \mathrm{incomplete\_range}$ then

$\quad\quad\quad ?_{\mathrm{P}}$

$\quad\quad$ else

$\quad\quad\quad \max\limits_{p} : (p, b) \in \mathit{pairs} \wedge b = \mathbf{t}.$

$[\![\texttt{min XP in TS with RAN} : \texttt{F}]\!](e) =$

$\quad$ let $\mathit{collected} = \underline{\mathrm{collect}}[\texttt{XP}, \texttt{TS}, [\![\texttt{RAN}]\!], [\![\texttt{F}]\!]](e)$:

$\quad$ if $\mathit{collected} = \bot_{\mathrm{P}}$ then

$\quad\quad \bot_{\mathrm{P}}$

$\quad$ else if $\mathit{collected} = ?_{\mathrm{P}}$ then

$\quad\quad ?_{\mathrm{P}}$

$\quad$ else

$\quad\quad$ let $(\mathit{pairs}, \mathit{flag}) = \mathit{collected}$:

$\quad\quad$ if $(\forall\ (p, b) \in \mathit{pairs} : b \neq \mathbf{t} \wedge b \neq \bot_{\mathbf{F}})$ then

$\quad\quad\quad ?_{\mathrm{P}}$

$\quad\quad$ else if $\mathit{flag} = \mathrm{incomplete\_range}$ then

$\quad\quad\quad$ if $(\exists\ (p, b) \in \mathit{pairs} : b = \bot_{\mathbf{F}})$ then

$\quad\quad\quad\quad \bot_{\mathrm{P}}$

$\quad\quad\quad$ else

$\quad\quad\quad\quad ?_{\mathrm{P}}$

$\quad\quad$ else

$\quad\quad\quad$ let $p_0 = \min\limits_{p} : (p, b) \in \mathit{pairs} \wedge (b = \mathbf{t} \vee b = \bot_{\mathbf{F}})$:

$\quad\quad\quad$ let $(p_0, b_0) \in \mathit{pairs}$:

$\quad\quad\quad$ if $b_0 = \mathbf{t}$ then

$\quad\quad\quad\quad p_0$

$\quad\quad\quad$ else

$\quad\quad\quad\quad \bot_{\mathrm{P}}.$

▷ $[\![\texttt{TV}]\!] : \mathit{EnvExtended} \to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\}$

$[\![\texttt{XV}]\!](e) = e{\downarrow}3(\texttt{XV})$.

$[\![\texttt{BIND} : \texttt{TV}]\!](e) = (\text{let } e_1 = [\![\texttt{BIND}]\!](e) : \text{if } e_1 = \bot_{\mathrm{E}} \text{ then } \bot_{\mathrm{V}} \text{ else } [\![\texttt{TV}]\!](e_1))$.

$[\![\texttt{TS}\text{◷}\texttt{TP}]\!](e) = \underline{\text{time or value}}[\texttt{TS}, \texttt{TP}, \text{time}](e)$.

$[\![\texttt{TS@TP}]\!](e) = \underline{\text{time or value}}[\texttt{TS}, \texttt{TP}, \text{value}](e)$.

$[\![\texttt{FV}(\texttt{T}_1, \ldots, \texttt{T}_n)]\!](e) =$

$\quad \text{let } v_1 = [\![\texttt{T}_1]\!](e), \ldots, [\![\texttt{T}_n]\!](e) :$

$\quad \text{if } (\exists\ 1 \le i \le n : v_i = \bot_{\mathrm{V}} \lor v_i = \bot_{\mathrm{P}}) \text{ then } \bot_{\mathrm{V}} \text{ else } [\![\texttt{FV}]\!](v_1, \ldots, v_n)$.

$[\![\texttt{num XP in TS with RAN} : \texttt{F}]\!](e) =$

$\quad \text{let } \mathit{collected} = \underline{\text{collect}}[\texttt{XP}, \texttt{TS}, [\![\texttt{RAN}]\!], [\![\texttt{F}]\!]](e) :$

$\quad \text{if } \mathit{collected} = \bot_{\mathrm{P}} \text{ then}$

$\quad\quad \bot_{\mathrm{V}}$

$\quad \text{else if } \mathit{collected} = ?_{\mathrm{P}} \text{ then}$

$\quad\quad ?_{\mathrm{V}}$

$\quad \text{else}$

$\quad\quad \text{let } (\mathit{pairs}, \mathit{flag}) = \mathit{collected} :$

$\quad\quad \text{if } (\exists (p, b) \in \mathit{pairs} : b = \bot_{\mathbf{F}}) \text{ then}$

$\quad\quad\quad \bot_{\mathrm{V}}$

$\quad\quad \text{else if } \mathit{flag} = \text{incomplete\_range then}$

$\quad\quad\quad ?_{\mathrm{V}}$

$\quad\quad \text{else}$

$\quad\quad\quad \#(p, b) \in \mathit{pairs} : b = \mathbf{t}$.

$[\![\texttt{complete combine}[\texttt{TV}_0, \texttt{FV}]\ \texttt{XP in TS with RAN CONSTR until F} : \texttt{TV}_1]\!](e) =$

$\quad \text{let } r = [\![\texttt{RAN}]\!](e) :$

$\quad \text{if } r = \bot_{\mathrm{P}} \text{ then}$

$\quad\quad \bot_{\mathrm{V}}$

$\quad \text{else if } r = ?_{\mathrm{P}} \text{ then}$

$\quad\quad ?_{\mathrm{V}}$

$\quad \text{else}$

$\quad\quad \text{let } (\texttt{XP}_1, \mathit{from}, \mathit{to}_0) = r :$

$\quad\quad \text{if } \texttt{XP}_1 \ne \texttt{XP} \text{ then}$

$\quad\quad\quad \bot_{\mathrm{V}}$

$\quad\quad \text{else}$

$\quad\quad\quad \text{let } p_0 = \max\limits_{\mathit{pos}} : [\![\texttt{TS}]\!](e)(\mathit{pos}) \ne ?_{\mathrm{M}} \land [\![\texttt{TS}\text{◷}\mathit{pos}]\!](e) \le \underline{\text{current time}}(e) :$

$\quad\quad\quad \text{let } \mathit{to} = \min(p_0, \mathit{to}_0) :$

$\quad\quad\quad \text{if } \mathit{to} < \mathit{from} \text{ then}$

$\quad\quad\quad\quad \bot_{\mathrm{V}}$

$\quad\quad\quad \text{else}$

$\quad\quad\quad\quad \text{let } \mathit{flag} =$

$\quad\quad\quad\quad\quad \text{if } \mathit{to} < \mathit{to}_0 \text{ then}$

$\quad\quad\quad\quad\quad\quad \text{incomplete\_range}$

$\quad\quad\quad\quad\quad \text{else}$

$\quad\quad\quad\quad\quad\quad \text{complete\_range} :$

$\quad\quad\quad\quad \text{let } \mathit{acc} = [\![\texttt{TV}_0]\!](e) :$

$$
\begin{array}{l}
\quad\quad\text{if } acc = \bot_{\mathrm{V}} \vee acc = ?_{\mathrm{V}} \text{ then} \\
\quad\quad\quad acc \\
\quad\quad\text{else} \\
\quad\quad\quad\text{let } combine\_values = [\![\mathtt{FV}]\!](e): \\
\quad\quad\quad\underline{\text{value comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}, combine\_values, flag, e](from, to, acc).
\end{array}
$$

$$
\begin{array}{l}
\underline{\text{value comb}}: \\
\quad(\mathtt{XP} \times \mathtt{CONSTR} \times \mathtt{F} \times \mathtt{TV} \times (\,Value \cup \{?_{\mathrm{V}}\} \times Value \cup \{?_{\mathrm{V}}\} \to Value \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\}) \\
\quad\quad \times \{\text{complete\_range}, \text{incomplete\_range}\} \times Env) \\
\quad \to Position \times Position \times Value \\
\quad \to Value \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\} \\
\underline{\text{value comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}, combine\_values, flag, e](from, to, acc) = \\
\quad\quad\text{let } (e_1, b_1) = [\![\mathtt{CONSTR}]\!](e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto from], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5): \\
\quad\quad\text{if } b_1 = \bot_{\mathbf{F}} \text{ then} \\
\quad\quad\quad \bot_{\mathrm{V}} \\
\quad\quad\text{else if } b_1 = ?_{\mathbf{F}} \text{ then} \\
\quad\quad\quad ?_{\mathrm{V}} \\
\quad\quad\text{else if } b = \mathbf{f} \text{ then} \\
\quad\quad\quad acc \\
\quad\quad\text{else} \\
\quad\quad\quad\text{let } v = [\![\mathtt{TV}]\!](e_1): \\
\quad\quad\quad\text{if } v = \bot_{\mathrm{V}} \text{ then} \\
\quad\quad\quad\quad \bot_{\mathrm{V}} \\
\quad\quad\quad\text{else} \\
\quad\quad\quad\quad\text{let } newacc = combine\_values(acc, v): \\
\quad\quad\quad\quad\text{if } newacc = \bot_{\mathrm{V}} \text{ then} \\
\quad\quad\quad\quad\quad \bot_{\mathrm{V}} \\
\quad\quad\quad\quad\text{else} \\
\quad\quad\quad\quad\quad\text{let } b_2 = [\![\mathtt{F}]\!](e_1{\downarrow}1, e_1{\downarrow}2, e_1{\downarrow}3[\mathbf{\mathtt{this}} \mapsto newacc], e_1{\downarrow}4, e_1{\downarrow}5): \\
\quad\quad\quad\quad\quad\text{if } b_2 = \bot_{\mathbf{F}} \text{ then} \\
\quad\quad\quad\quad\quad\quad \bot_{\mathrm{V}} \\
\quad\quad\quad\quad\quad\text{else if } b_2 = ?_{\mathbf{F}} \text{ then} \\
\quad\quad\quad\quad\quad\quad ?_{\mathrm{V}} \\
\quad\quad\quad\quad\quad\text{else if } b_2 = \mathbf{t} \text{ then} \\
\quad\quad\quad\quad\quad\quad newacc \\
\quad\quad\quad\quad\quad\text{else if } from \geq to \text{ then} \\
\quad\quad\quad\quad\quad\quad\text{if } flag = \text{complete\_range then} \\
\quad\quad\quad\quad\quad\quad\quad newacc \\
\quad\quad\quad\quad\quad\quad\text{else} \\
\quad\quad\quad\quad\quad\quad\quad ?_{\mathrm{V}} \\
\quad\quad\quad\quad\quad\text{else} \\
\quad\quad\quad\quad\quad\quad\text{let } from_1 = from + 1: \\
\quad\quad\quad\quad\quad\quad\underline{\text{value comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}, combine\_values, flag, e](from_1, to, newacc).
\end{array}
$$

$\triangleright\quad \mathtt{TS} : \mathit{EnvExtended} \to \mathit{Stream}$

$[\![\mathtt{XS}]\!](e) = e{\downarrow}4(\mathtt{XS})$

$[\![\mathtt{BIND} : \mathtt{TS}]\!](e) = (\text{let } e_1 = [\![\mathtt{BIND}]\!](e) : \text{if } e_1 = \bot_{\mathrm{E}} \text{ then } ?^{\omega}_{\mathrm{M}} \text{ else } [\![\mathtt{TS}]\!](e_1)).$

$[\![\mathtt{FS}(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!](e) = (\text{let } s_1 = [\![\mathtt{T}_1]\!](e), \ldots, s_1 = [\![\mathtt{T}_n]\!](e) : [\![\mathtt{FS}]\!](s_1, \ldots, s_n)).$

$[\![\mathtt{partial\ combine}[\mathtt{TV}_0, \mathtt{FV}]\ \mathtt{XP\ in\ TS\ with\ RAN\ CONSTR\ until\ F} : \mathtt{TV}_1]\!](e) =$

$$
\begin{array}{l}
\text{let } \mathit{current\_time} = \underline{\text{current time}}(e) : \\
\text{if } \mathit{current\_time} = 0 \text{ then} \\
\quad ?^{\omega}_{\mathrm{M}} \\
\text{else} \\
\quad \text{let } r = [\![\mathtt{RAN}]\!](e) : \\
\quad \text{if } r = \bot_{\mathrm{P}} \text{ then} \\
\quad\quad ?^{\omega}_{\mathrm{M}} \\
\quad \text{else if } r = ?_{\mathrm{P}} \text{ then} \\
\quad\quad ((\mathit{current\_time}, ?_{\mathrm{V}}), ?^{\omega}_{\mathrm{M}}) \\
\quad \text{else} \\
\quad\quad \text{let } (\mathtt{XP}_1, \mathit{from}, \mathit{to}_0) = r : \\
\quad\quad \text{if } \mathtt{XP}_1 \neq \mathtt{XP} \text{ then} \\
\quad\quad\quad ?^{\omega}_{\mathrm{M}} \\
\quad \text{else} \\
\quad\quad \text{let } p_0 = \max\limits_{\mathit{pos}} : [\![\mathtt{TS}]\!](e)(\mathit{pos}) \neq ?_{\mathrm{M}} \wedge [\![\mathtt{TS} \circleddash \mathit{pos}]\!](e) \leq \mathit{current\_time} : \\
\quad\quad \text{let } \mathit{to} = \min(p_0, \mathit{to}_0) : \\
\quad\quad \text{if } \mathit{to} < \mathit{from} \text{ then} \\
\quad\quad\quad ?^{\omega}_{\mathrm{M}} \\
\quad\quad \text{else} \\
\quad\quad\quad \text{let } v = [\![\mathtt{TV}_0]\!](e) : \\
\quad\quad\quad \text{if } v = \bot_{\mathrm{V}} \text{ then} \\
\quad\quad\quad\quad ?^{\omega}_{\mathrm{M}} \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \text{let } \mathit{acc} = ((\mathit{current\_time}, v), ?^{\omega}_{\mathrm{M}}) : \\
\quad\quad\quad\quad \text{let } \mathit{combine\_values} = [\![\mathtt{FV}]\!](e) : \\
\quad\quad\quad\quad \text{let } s_1 = \underline{\text{stream comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}_1, \mathit{combine\_values}, e](\mathit{from}, \mathit{to}, \mathit{acc}) : \\
\quad\quad\quad\quad \text{let } s_2 = [\![\mathtt{partial\ combine}[\mathtt{TV}_0, \mathtt{FV}]\ \mathtt{XP\ in\ TS\ with\ RAN\ CONSTR} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathtt{until\ F} : \mathtt{TV}_1]\!](e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, \mathit{current\_time} - 1) : \\
\quad\quad\quad\quad \underline{\text{diff and append}}(s_1, s_2).
\end{array}
$$

$\underline{\text{stream comb}} :$

$$
\begin{array}{l}
\quad (\mathtt{XP} \times \mathtt{CONSTR} \times \mathtt{F} \times \mathtt{TV} \times (\mathit{Value} \cup \{?_{\mathrm{V}}\} \times \mathit{Value} \cup \{?_{\mathrm{V}}\} \to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\}) \times \mathit{Env}) \\
\quad \to \mathit{Position} \times \mathit{Position} \times \mathit{Stream} \\
\quad \to \mathit{Stream}
\end{array}
$$

$\underline{\text{stream comb}}[\mathtt{XP}, \mathtt{CONSTR}, \mathtt{F}, \mathtt{TV}, \mathit{combine\_values}, e](\mathit{from}, \mathit{to}, \mathit{acc}) =$

$$
\begin{array}{l}
\text{let } (e_1, b_1) = [\![\mathtt{CONSTR}]\!](e{\downarrow}1, e{\downarrow}2[\mathtt{XP} \mapsto \mathit{from}], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5) : \\
\text{if } b_1 \neq \mathbf{t} \text{ then} \\
\quad \mathit{acc}
\end{array}
$$

else

let $v = [\![\texttt{TV}]\!](e_1)$:

if $v = \bot_\mathrm{V}$ then

acc

else

let $newacc = \underline{\text{combine and join}}(acc, v, combine\_values, \underline{\text{current time}}(e_1))$:

let $b_2 = [\![\texttt{F}]\!](e_1{\downarrow}1, e_1{\downarrow}2, e_1{\downarrow}3[\texttt{this} \mapsto newacc], e_1{\downarrow}4, e_1{\downarrow}5)$:

if $b_2 \neq \mathbf{f}$ then

newacc

else if $from \geq to$ then

newacc

else

let $from_1 = from + 1$:

$\underline{\text{stream comb}}[\texttt{XP}, \texttt{CONSTR}, \texttt{F}, \texttt{TV}, combine\_values, e](from_1, to, newacc)$.

$[\![\texttt{construct XP in TS with RAN CONSTR} : value\_or\_stream]\!](e) =$

let $current\_time = \underline{\text{current time}}(e)$:

if $current\_time = 0$ then

$?^\omega_\mathrm{M}$

else

let $r = [\![\texttt{RAN}]\!](e)$:

if $r = \bot_\mathrm{P} \vee r = ?_\mathrm{P}$ then

$?^\omega_\mathrm{M}$

else

let $(\texttt{XP}_1, from, to_0) = r$:

if $\texttt{XP}_1 \neq \texttt{XP}$ then

$?^\omega_\mathrm{M}$

else

let $p_0 = \max\limits_{pos} : [\![\texttt{TS}]\!](e)(pos) \neq ?_\mathrm{M} \wedge [\![\texttt{TS}\circleddash pos]\!](e) \leq current\_time$:

let $to = \min(p_0, to_0)$:

if $to < from$ then

$?^\omega_\mathrm{M}$

else

let $s_1 = \underline{\text{stream construct}}[\texttt{XP}, \texttt{CONSTR}, value\_or\_stream, e](from, to, ?^\omega_\mathrm{M})$:

let $s_2 = [\![\texttt{construct XP in TS with RAN CONSTR} : value\_or\_stream]\!]$

$(e{\downarrow}1, e{\downarrow}2, e{\downarrow}3, e{\downarrow}4, current\_time - 1)$:

$\underline{\text{diff and append}}(s_1, s_2)$.

$\underline{\text{stream construct}}$:

$(\texttt{XP} \times \texttt{CONSTR} \times (\texttt{TV} + \texttt{TS}) \times Env) \rightarrow Position \times Position \times Stream \rightarrow Stream$

$\underline{\text{stream construct}}[\texttt{XP}, \texttt{CONSTR}, value\_or\_stream, e](from, to, acc) =$

let $(e_1, b_1) = [\![\texttt{CONSTR}]\!](e{\downarrow}1, e{\downarrow}2[\texttt{XP} \mapsto from], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)$:

if $b_1 \neq \mathbf{t}$ then

$\quad\quad acc$
$\quad$ else
$\quad\quad$ let $v\_or\_s = [\![value\_or\_stream]\!](e_1)$:
$\quad\quad$ cases v_or_s of
$\quad\quad\quad$ isValue$(v) \longrightarrow$ if $v = \bot_{\mathrm{V}}$ then acc else let $str = ((\underline{\text{current time}}(e_1), v), ?^{\omega}_{\mathrm{M}})$:
$\quad\quad\quad$ isStream$(s) \longrightarrow$ let $str = s$:
$\quad\quad$ let $newacc = \underline{\text{stream join}}(acc, str)$:
$\quad\quad$ if $from \geq to$ then
$\quad\quad\quad newacc$
$\quad\quad$ else
$\quad\quad\quad$ let $from_1 = from + 1$
$\quad\quad\quad \underline{\text{stream construct}}[\mathtt{XP}, \mathtt{CONSTR}, v\_or\_s, e](from_1, to, newacc)$.

$[\![\mathtt{FV}]\!] : (\mathit{Value} \cup \{?_{\mathrm{V}}\} + \mathit{Position} \cup \{?_{\mathrm{P}}\} + \mathit{Stream})^* \to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\}$
$[\![\mathtt{FS}]\!] : (\mathit{Value} \cup \{?_{\mathrm{V}}\} + \mathit{Position} \cup \{?_{\mathrm{P}}\} + \mathit{Stream})^* \to \mathit{Stream}$
$[\![\mathtt{PV}]\!] : (\mathit{Value} \cup \{?_{\mathrm{V}}\} + \mathit{Position} \cup \{?_{\mathrm{P}}\} + \mathit{Stream})^* \to \mathit{MK}$

B.3 Auxiliary Functions

$\underline{\text{time}} : \mathit{Message} \to \mathit{Time}$
$\underline{\text{time}}(m) = (\text{let } (t, v) = m : t)$

$\underline{\text{value}} : \mathit{Message} \to \mathit{Value} \cup \{?_{\mathrm{V}}\}$
$\underline{\text{value}}(m) = (\text{let } (t, v) = m : v)$

$\underline{\text{current time}} : \mathit{EnvExtended} \to \mathit{CurrTime}$
$\underline{\text{current time}}(e) = e{\downarrow}5$

$\underline{\text{collect}} : \mathtt{XP} \times \mathtt{TS} \times (\mathit{EnvExtended} \to (\mathtt{XP} \times \mathit{Position} \times \mathit{Position}) \cup \{?_{\mathrm{P}}, \bot_{\mathrm{P}}\})$
$\quad\quad \times (\mathit{EnvExtended} \to \mathit{MK})$
$\quad\quad \to \mathit{EnvExtended}$
$\quad\quad \to (\mathcal{P}(\mathit{Position} \times \mathit{MK}) \times \{\text{complete\_range}, \text{incomplete\_range}\}) \cup \{?_{\mathrm{P}}, \bot_{\mathrm{P}}\}$
$\underline{\text{collect}}[\mathtt{XP}, \mathtt{TS}, [\![\mathtt{RAN}]\!], [\![\mathtt{F}]\!]](e) =$
$\quad\quad$ let $r = [\![\mathtt{RAN}]\!](e)$:
$\quad\quad$ if $r = \bot_{\mathrm{P}}$ then
$\quad\quad\quad \bot_{\mathrm{P}}$
$\quad\quad$ else if $r = ?_{\mathrm{P}}$ then
$\quad\quad\quad ?_{\mathrm{P}}$
$\quad\quad$ else
$\quad\quad\quad$ let $(\mathtt{XP}_1, from, to_0) = c$:
$\quad\quad\quad$ if $\mathtt{XP}_1 \neq \mathtt{XP}$ then
$\quad\quad\quad\quad \bot_{\mathrm{P}}$
$\quad\quad\quad$ else

$$
\begin{array}{l}
\text{let } p_0 = \max_{pos} : [\![\texttt{TS}]\!](e)(pos) \neq ?_{\mathrm{M}} \wedge [\![\texttt{TS} \circleddash pos]\!](e) \leq \underline{\text{current time}}(e) : \\
\text{let } \mathit{to} = \min(p_0, \mathit{to}_0) : \\
\text{let } \mathit{flag} = \\
\quad \text{if } \mathit{to} = \mathit{to}_0 \text{ then} \\
\quad\quad \text{complete\_range} \\
\quad \text{else} \\
\quad\quad \text{incomplete\_range} : \\
\text{let } \mathit{pairs} = \{(p, [\![\texttt{F}]\!](e{\downarrow}1, e{\downarrow}2[\texttt{XP} \mapsto p], e{\downarrow}3, e{\downarrow}4, e{\downarrow}5)) \mid \mathit{from} \leq p \leq \mathit{to}\} : \\
\quad (\mathit{pairs}, \mathit{flag}).
\end{array}
$$

$$
\begin{array}{l}
\underline{\text{time or value}} : \texttt{TS} \times \texttt{TP} \times \{\text{time}, \text{value}\} \to \mathit{EnvExtended} \to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\} \\
\underline{\text{time or value}}[\texttt{TS}, \texttt{TP}, \mathit{time\_or\_value}](e) = \\
\quad \text{let } p = [\![\texttt{TP}]\!](e), s = [\![\texttt{TS}]\!](e) : \\
\quad \text{if } p = \bot_{\mathrm{P}} \text{ then} \\
\quad\quad \bot_{\mathrm{V}} \\
\quad \text{else if } p = ?_{\mathrm{P}} \text{ then} \\
\quad\quad ?_{\mathrm{V}} \\
\quad \text{else if } s(p) = ?_{\mathrm{M}} \text{ then} \\
\quad\quad \bot_{\mathrm{V}} \\
\quad \text{else} \\
\quad\quad \text{if } \underline{\text{current time}}(e) < \underline{\text{time}}(s(p)) \text{ then} \\
\quad\quad\quad \bot_{\mathrm{V}} \\
\quad\quad \text{else} \\
\quad\quad\quad \text{if } \mathit{time\_or\_value} = \text{time then} \\
\quad\quad\quad\quad \underline{\text{time}}(s(p)) \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \underline{\text{value}}(s(p)).
\end{array}
$$

$$
\begin{array}{l}
\underline{\text{combine and join}} : \\
\quad \mathit{Stream} \times \mathit{Value} \cup \{?_{\mathrm{V}}\} \times (\mathit{Value} \cup \{?_{\mathrm{V}}\} \times \mathit{Value} \cup \{?_{\mathrm{V}}\} \to \mathit{Value} \cup \{?_{\mathrm{V}}, \bot_{\mathrm{V}}\}) \times \mathit{Time} \\
\quad \to \mathit{Stream} \\
\underline{\text{combine and join}}(s, v, \mathit{combine\_values}, \mathit{current\_time}) = \\
\quad \text{let } ((t_1, v_1), \ldots, (t_n, v_n), ?_{\mathrm{M}}^{\omega}) = s : \\
\quad \text{if } v = ?_{\mathrm{V}} \text{ then} \\
\quad\quad \underline{\text{stream join}}(s, ((\mathit{current\_time}, v), ?_{\mathrm{M}}^{\omega})) \\
\quad \text{else} \\
\quad\quad \text{let } k = \max_{i} : v_i \neq ?_{\mathrm{V}} : \\
\quad\quad \text{let } \mathit{newval} = \mathit{combine\_values}(v_k, v) : \\
\quad\quad \text{if } \mathit{newval} = \bot_{\mathrm{V}} \text{ then} \\
\quad\quad\quad \underline{\text{stream join}}(s, ((\mathit{current\_time}, ?_{\mathrm{V}}), ?_{\mathrm{M}}^{\omega})) \\
\quad\quad \text{else} \\
\quad\quad\quad \underline{\text{stream join}}(s, ((\mathit{current\_time}, \mathit{newval}), ?_{\mathrm{M}}^{\omega})).
\end{array}
$$

$\underline{\text{stream join}} : \mathit{Stream} \times \mathit{Stream} \to \mathit{Stream}$

$\underline{\text{stream join}}(s_1, s_2) =$

$\quad$ let $((t_1, v_1), \ldots, (t_n, v_n), ?^{\omega}_{\mathrm{M}}) = s_1$:

$\quad$ $((t_1, v_1), \ldots, (t_n, v_n)) \| s_2$.

Remark: In stream join, s_1 has the finite observable part.

$\underline{\text{diff and append}} : \mathit{Stream} \times \mathit{Stream} \to \mathit{Stream}$

$\underline{\text{diff and append}}(\mathit{new\_stream}, \mathit{old\_stream}) =$

$\quad$ let $((t, v_1), \ldots, (t, v_n), ?^{\omega}_{\mathrm{M}}) = \mathit{new\_stream}\ (n \geq 0)$:

$\quad$ let $((t'_1, v'_1), \ldots, (t'_m, v'_m), ?^{\omega}_{\mathrm{M}}) = \mathit{old\_stream}\ (m \geq 0)$:

$\quad$ $((t'_1, v'_1), \ldots, (t'_m, v'_m)) \| ((t, v_{m+1}), \ldots, (t, v_n), ?^{\omega}_{\mathrm{M}})$.

not	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
	f	**t**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$

and	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
t	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
f	**f**	**f**	**f**	**f**
$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$

or	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
t	**t**	**t**	**t**	**t**
f	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
$?_{\mathbf{F}}$	**t**	$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$

implies	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
t	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
f	**t**	**t**	**t**	**t**
$?_{\mathbf{F}}$	**t**	$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$

iff	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
t	**t**	**f**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
f	**f**	**t**	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$?_{\mathbf{F}}$	$\bot_{\mathbf{F}}$
$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$	$\bot_{\mathbf{F}}$