Translation Mechanism for the LogicGuard Abstract Language

Temur Kutsia Wolfgang Schreiner

RISC, Johannes Kepler University Linz

{schreine,kutsia}@risc.jku.at

October 23, 2012

Abstract

The LogicGuard project aims at developing a specification and verification formalism for runtime network monitoring based on predicate logic. This report describes the translation mechanism of the LogicGuard abstract language (LGAL). The translator accepts a formula that specifies the monitor and produces a program in functional style, seen as a runtime monitor.

1 Brief Overview

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. The specific focus is on computer and network security, concentrating on predicate logic specifications of security properties of network traffic. In the previous report [1], we described the syntax and semantics of the the LogicGuard Abstract Language (LGAL). LGAL represents a fragment of classical predicate logic. It 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. 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.

In this report we proceed further, describing the translation mechanism, which is supposed to automatically generate runtime monitors from the specifications. We do not give here the formal definitions, they can be found in the appendix. Rather, we try to informally explain what the mechanism is supposed to do.

We start the explanation by illustrating a single step in runtime network monitoring shown in Figure 1. Note that there is a difference from the basic model of runtime network monitoring described in [1]: Here we allow multiple global streams, called the external streams, and initialize some local, internal streams outside the monitor. (The latter does not forbid to create internal streams inside the monitor, though.)

The network traffic is modeled by the external streams. At each position, each stream contains a message, which is a pair (t, v) of the time stamp t and the value v. (Some of those streams can be finite.) The messages in the streams are linearly ordered according to the time stamp. In monitoring, not only the external streams, but also some internally created streams may be needed. The goal of the translation is to make the (modified version of the) basic model "executable" by translating the *monitor specification* into a program, which we call here the *runtime monitor.*

The monitor is described by its specification M = `monitor` $\mathtt{XP}_1$ `in` $\mathtt{SID}_1, \ldots, \mathtt{XP}_n$ `in` $\mathtt{SID}_n$: `F`. The monitoring formula `F` is interpreted over the streams $\mathtt{SID}_1, \ldots, \mathtt{SID}_n$, which refer to external or internal streams. Those streams are created and expanded outside of M. However, `F` may also

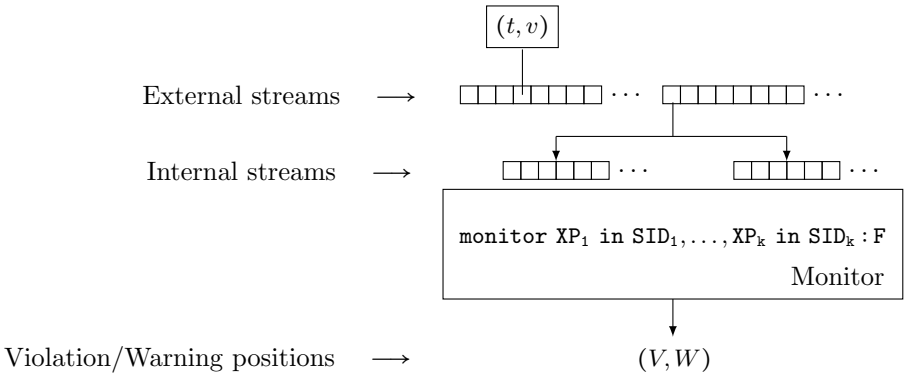

Figure 1: Monitoring Step.

contain some other internal streams, which are created and expanded there (i.e., inside M). The monitoring positions `XP`'s refer to the positions in the streams denoted by the corresponding `SID`'s.

Each time, we take a new message from one of the external streams (we say that a new message *arrives* to one of the external streams, which gets updated by this message), update the internal streams, and evaluate the monitoring formula on the updated external and internal streams. The result of the evaluation can be either true ($\mathbf{t}$), false ($\mathbf{f}$), error ($\perp_{\mathbf{F}}$), or the unknown value ($?_{\mathbf{F}}$). Then the verification system should report violations when the monitored formula evaluates to $\mathbf{f}$, and warnings when the monitored formula evaluates to $?_{\mathbf{F}}$. Violations are warnings are just positions in the streams $\mathtt{SID}_1, \ldots, \mathtt{SID}_n$, which show the values of the variables $\mathtt{XP}_1, \ldots, \mathtt{XP}_n$. In the figure, the sets of violations and warnings are denoted respectively by V and W.

Note that when a message arrives, the runtime monitor might not be able to determine the truth value of the monitoring formula: There might be simply not enough information on the streams. In this case, the runtime monitor waits for the next message to come.

The behavior of the runtime monitor can be described by executing the monitoring step at each message which arrives to the external streams, where the step itself performs the following actions: When a message arrives, try to evaluate the monitoring formula. If the answer is $\perp_{\mathbf{F}}$, abort the entire evaluation. If the answer is $\mathbf{f}$ or $?_{\mathbf{F}}$, report the position and wait for the next message. If the answer is $\mathbf{t}$ or if we can not get the answer at all, wait for the next message. This top-level control on the evaluation of the monitoring step in the translation is given by the pseudocode below. The translation of monitor specifications into runtime monitors can be found in the appendix.

```
...
state ← (stream-environment, monitoring-formula)
env-fun-pred ← environment-for-external-functions-and-predicates
loop
    receive (message, stream-id)
    (violations, warnings, state) =
        monitoring-step(message, stream-id, state, env-fun-pred)
    report violations
    report warnings
```

The loop is, in general, infinite. It only terminates if there are no new messages coming (i.e.,if the external streams are finite), or when the monitoring formula does not depend on the input streams. It may break if the monitoring step raises error.

The monitoring formula expresses a safety property, if every position for which the formula evaluates to false or unknown, belongs to some finite prefix of the input streams. Correctness of the

translation can be formulated, rather informally, as the soundness and completeness statements below, where the latter states completeness with respect to safety properties:

Soundness: If the monitor reports a position in an input stream as a violation (or warning), then the truth value of the monitoring formula is false (resp. unknown).

Completeness: If the monitoring formula expresses a safety property and its truth value is false (resp. unknown) at some position, then this position will eventually be reported as a violation (resp. warning) position during the monitoring loop.

This report describes the current state of the work-in-progress on the translation mechanism for LGAL. The translator is represented as a function whose definition is given in the appendix. Efficiency issues such as, e.g., history pruning, are not addressed. Formulas are evaluated sequentially, from left to right, in a lazy manner. Proving the correctness property remains as a future work.

Acknowledgments

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

References

[1] Temur Kutsia and Wolfgang Schreiner. LogicGuard Abstract Language. RISC Report Series 12-08, Research Institute for Symbolic Computation (RISC), University of Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, 2012.

APPENDIX: THE TRANSLATOR FOR LGAL

```
================================================================================
1. Translation of the Monitor
================================================================================

TYPES
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

================================================================================
STATE
================================================================================
State = EnvStream × Step[MK] × ℙ(PotentialViolation)

================================================================================
ENVIRONMENTS
================================================================================
Present =   Message × SID
          × EnvStreams × EnvFormula × EnvPosition × EnvValue
          × EnvFunctionsPredicates

EnvStream = ExtEnv × IntEnv

ExtEnv = ExSID → StreamInfo

IntEnv = InSID → StreamInfo

EnvFormula = XF → Step[MK]

EnvPosition = XP → Step[Position ∪ {?P}]

EnvValue = XV → Step[Value ∪ {?V}]

================================================================================
BINDINGS
================================================================================
Binder = Present → ((Present × Binder) ∪ {⊥E})

================================================================================
STREAMS
================================================================================
StreamInfo = StreamPast × StreamFuture

StreamPast = Message* × Nat

StreamFuture = NIL + ?S + Stream + External

Stream = Present → ⊥S + StreamInfo

================================================================================
STEP
================================================================================
Step[T] = Present → Answer[T]

Answer[T] = T + Step[T]

================================================================================
VIOLATIONS, WARNINGS
================================================================================
PotentialViolation = ((Position × SID)+ × Step[MK])

Violation = (Position × SID)+

Warning = (Position × SID)+

Message = Time × (Value ∪ {?V})

MessageStream = ℙ(Message) × Stream

MonitoringVarsSIDs = (XP × SID)*
   // Monitoring variables and corresponding streams
```

```
================================================================================
MK, VALUES, POSITIONS, CONSTRAINT DOMAINS, RANGES, ETC
================================================================================
Bool = {true,false}
MK = Bool ∪ {?F,⊥F}

Position? = Position ∪ {?P}
Poaition⊥ = Position ∪ {⊥P}
Position⊥? = Position ∪ {⊥P,?P}

Value? = Value ∪ {?V}
Value⊥ = Value ∪ {⊥V}
Value⊥? = Value ∪ {⊥V,?V}

VPS = Value? + Position? + Stream

ConstrDom = ⊥E + PresentMK
PresentMK = Present × MK

Range  =    Step[Position⊥?]
          × {"<","=<"} × XP × {"<","=<"}
          × Step[Position⊥? ∪ {∞}]

================================================================================
DATA CONSTRUCTORS:
================================================================================
isMK: MK → MK

isValue: Value → Value
isValue⊥? : Value⊥? → Value⊥?

isPosition: Position → Position
isPosition⊥?: Position⊥? → Position⊥?

isVPS: VPS → VPS
================================================================================

--------------------------------------------------------------------------------
Monitor :=  ExSID1 ... ExSIDn : M
          | ExSID1 ... ExSIDn, let InSID1 = TS1 ... InSIDn = TSn : M

M := monitor XP1 in SID1, ..., XPn in SIDn : F

SID := ExSID | InSID
ExSID := XS
InSID := XS
--------------------------------------------------------------------------------

Tr⟦Monitor⟧ : State × InSID* × ℙ(MonitoringVarsSIDs)

// Initializes the external and internal stream environments mapping
// the stream ids to the empty streams, translates the monitoring
// formula with free occurrences of monitored position variables, and
// creates the initial state consisting of the initial stream
// environment, tranlated monitoring formula, and the empty set
// of potential violations. Together with the initial state, it returns
// the sequence of internal stream ids and the set of pairs of those
// monitoring variables and the corresponding stream ids, which
// occur in the monitored formula.

Tr⟦Monitor⟧ =
  let (ExtID1,...,ExtIDn, let IntID1 = TS1,...,IntID_m = TS_m, M) = Monitor :
  let ext_env = {ExtID1 ↦ (ε,0,External), …, ExtIDn ↦ (ε,0,External)} :
  let s1 = Tr⟦TS1⟧,…, s_m = Tr⟦TS_m⟧ :
  let int_env = {IntID1 ↦ (ε,0,s1), …, IntID_m ↦ (ε,0,s_m)} :
  let stream_env = ext_env × int_env :
  let (monitor XP1 in SID1, ..., XPk in SIDk : F) = M :
  let f = Tr⟦F⟧ :
  let int_ids = (IntID1,...,IntID_m) :
  let monitoring_vars_ids =
        select_vars_occurring_in_formula((XP1,SID1)…,(XPk,SIDk),F) :
  let potential_violations = ∅ :
  let initial_state = (stream_env,f,potential_violations) :
  (initial_state,int_ids,monitoring_vars_ids)
```

```
149
150  ------------------------------------------------------------------------------
151  select_vars_occurring_in_formula :
152       MonitoringVarsSIDs × F → MonitoringVarsSIDs
153  ------------------------------------------------------------------------------
154
155  select_vars_occurring_in_formula((XP1,SID1)…,(XPk,SIDk),F) =
156     { (XP,SID) | (XP,SID) ∈ {(XP1,SID1)…,(XPk,SIDk)} and XP occurs freely in F }
157
158  // Requires an auxiliary function for checking free occurrences of a variable
159  // in a formula
160
161  ------------------------------------------------------------------------------
162  monitoring_step :
163     (Message + NIL + ?S) × ExSID × State × InSID* × ℙ(MonitoringVarsSIDs)
164                    × EnvFunctionsPredicates
165                → (ℙ(Violation) × ℙ(Warnings) × State) + MK
166  ------------------------------------------------------------------------------
167
168  // Performs one step of the monitoring process. Recieves a message
169  // from an external stream, updates the stream environment based
170  // on the obtained message, checks whether the potential violation
171  // steps become real violations/warnings, evaluates the monitoring
172  // formula getting more violation/warning/potential violation positions,
173  // and, finally, returns either
174  //  (a) a triple of whose first component is all violation position-sid pairs,
175  //      the second component is all warning position-sid pairs,
176  //      and the third component is the new state consisting of the updated
177  //      stream environment, monitoring formula, and all potential violation
178  //      position-step pairs collected together.
179  //  or
180  //  (b) a definite value true, false, ?F, or ⊥F. This eventually happens
181  //      when the set of monitoring variables occurring in the monitoring formula
182  //      is empty.
183
184
185  monitoring_step(message,sid,state,int_ids,mon_vars_ids,envfp) =
186    let (stream_env,f,pot_violations) = state :
187    let (ext_env,int_env) = stream_env :
188    let (new_stream_env,ids_of_updated_streams) =
189           update_stream_environment(stream_env,message,sid,envfp,int_ids) :
190    if new_stream_env = ⊥E then
191       ⊥F
192    let envf = envp = envv = EmptyEnvironment :
193    let present = (message,sid,new_stream_env,envf,envp,envv,envfp) :
194    if mon_vars_ids = ∅ then
195       let f0 = f(present)
196       case f0 of
197          isMK(mk) : f0
198          isStep(st) : (∅,∅,(stream_env,f,pot_violations))
199    else
200       let v = check_potential_violations(pot_violations,present) :
201       if v = ⊥ then
202          ⊥F
203       else
204          let (violations,warnings,pot_violations0) = v :
205          let v1 = generate_and_evaluate_instances(f,present,
206                                  mon_vars_ids,ids_of_updated_streams,int_ids) :
207          if v1 = ⊥ then
208              ⊥F
209          else
210              let (violations1,warnings1,pot_violations1) = v1 :
211              let new_violations = violations ∪ violations1 :
212              let new_warnings = warnings ∪ warnings1 :
213              let new_pot_violations = pot_violations0 ∪ pot_violations1 :
214              let new_state = (new_stream_env,f,new_pot_violations) :
215              (new_violations,new_warnings,new_state)
216
217  ------------------------------------------------------------------------------
218  generate_and_evaluate_instances :
219          Step[MK] × Present × ℙ(MonitoringVarsSIDs) × ℙ(SID) × InSID*
220              → (ℙ(Violation) × ℙ(Warnings) × ℙ(PotentialViolation)) ∪ {⊥}
221  ------------------------------------------------------------------------------
222
```

```
// Generates instances of the monitor, obtained by evaluating
// the monitoring formula at present, with the addition that
// the monitored variables in the monitoring formula are
// instantiated with the combination
// of positions from the updated and non-updated streams.
// The instances with the monitoring formula evaluated to false
// are collected in violations, those with ?F - in warnings,
// and those that are steps - in potential violations (potvals).


generate_and_evaluate_instances(f,present,monitoring_vars_ids,
                                ids_of_updated_streams,int_ids) =
   let {(xp₁,sid1,r11,r12),…,(xp_k,sid_k,rk1,rk2)} =
       get_ranges(present,monitoring_vars_ids,ids_of_updated_streams,int_ids) :

// The value of instances below is ⌀ if one of the SIDs is the empty stream.

   let instances = {((p1,SID1),…,(p_k,SID_k),f(present[X1↦p1,…,XPk↦pk]) |
                     r11 ≤ p1 ≤ r12, …, r_k1 ≤ p_k ≤ r_k2 and
                     f(present[X1↦p1,…,XPk↦pk]) ≠ true}
   If contains_error(instance) = true then
      ⊥
   else
      let violations = {((p1,SID1),…,(p_k,SID_k)) |
                        ((p1,SID1),…,(p_k,SID_k),false) ∈ instances} :
      let warnings = {((p1,SID1),…,(p_k,SID_k)) |
                        ((p1,SID1),…,(p_k,SID_k),?F) ∈ instances} :
      let potviols = intances \ (violations ∪ warnings) :
      (violations,warnings,potviols)

-------------------------------------------------------------------------------
get_ranges : Present × ℙ(MonitoringVarsSIDs) × ℙ(SID) × InSID*
             → ℙ(XP × SID × Position × Position)
-------------------------------------------------------------------------------

// Collects ranges for position variables in updated and non-updated
// streams. In the updated streams the collected positions range
// from the 1 + maximal position before updating, to the maximal
// position after updating. In non-updated streams the collected
// positions are all the positions in the stream.


get_ranges(present,monitoring_vars_ids,ids_of_updated_streams,int_ids) =
    let ext_env = (present↓3)↓1 :
    let int_env = (present↓3)↓2 :
    let ranges_for_updated = {(xp,sid,p1,p2) |
                              (xp,sid) ∈ monitoring_vars_ids,
                              sid ∈ ids_of_updated_streams,
                              p1 = if sid ∈ int_ids then
                                      let number_of_new_messages =
                                          (int_env(sid)↓1)↓2 :
                                      maximal_position(int_env(sid)↓1)-
                                       - number_of_new_messages + 1,
                                   else
                                     let number_of_new_messages =
                                          (ext_env(sid)↓1)↓2 :
                                      maximal_position(ext_env(sid)↓1)-
                                       - number_of_new_messages + 1,
                                      maximal_position(ext_env(sid)↓1)
                              p2 = if sid ∈ int_ids then
                                      maximal_position(int_env(sid)↓1)
                                   else
                                      maximal_position(ext_env(sid)↓1)}
    let ranges_for_non_updated = {(xp,sid,p1,p2) |
                                   (XP,sid) ∈ monitoring_vars_ids,
                                   sid ∉ ids_of_updated_streams,
                                   p1 = 0,
                                   p2 = if sid ∈ int_ids then
                                      maximal_position(int_env(sid)↓1)
                                   else
                                      maximal_position(ext_env(sid)↓1)}
    let ans = ranges_for_updated ∪ ranges_for_non_updated :
    ans
```

```
297  ------------------------------------------------------------------------------
298  maximal_position : StreamInfo → Nat
299  ------------------------------------------------------------------------------
300
301  maximal_position(info) =
302    len(info↓1)-1
303
304  ------------------------------------------------------------------------------
305  len : Message* → Nat
306  ------------------------------------------------------------------------------
307  len is a function that takes a sequence of messages and
308  returns its length
309
310  ------------------------------------------------------------------------------
311  contains_error : ℙ((Position × SID)+ × (false + ?F + ⊥F + Step[MK]))
312                   → Bool
313  ------------------------------------------------------------------------------
314
315  contains_error(set) =
316     if set = ∅ then
317        false
318     else
319        let elem = select_an_element(set) :
320        let (pos_sid_pairs,b) = elem :
321        if b = ⊥F then
322           true
323        else
324           let new_set = delete_an_element(set,elem) :
325           contains_error(new_set)
326
327  ------------------------------------------------------------------------------
328  update_stream_environment :
329      EnvStreams × (Message + NIL + ?S) × ExSID
330                 × EnvFunctionsPredicates × InSID*
331      → (EnvStreams ×  ℙ(SID)) ∪ {⊥E}
332  ------------------------------------------------------------------------------
333
334  update_stream_environment(stream_env,msg,sid,envfp,int_ids) =
335     let (ext_env,int_env) = stream_env :
336     let old_past = ext_env(sid)↓1 :
337     let (messages,lastlenght) = old_past :
338     let new_stream =
339         case msg of
340             NIL          : (messages,0,NIL)
341             ?S           : (messages,0,?S)
342             isMessage(m) : (messages||m,1,External)
343     let new_ext_env = ext_env[sid ↦ new_stream] :
344     let new_stream_env=(new_ext_env,int_env) :
345     let e = EmptyEnvironment :
346     let present = (msg,sid,new_stream_env,e,e,e,envfp) :
347     let ints = update_int_stream_env(present,int_ids,e,∅) :
348     if ints = ⊥E then
349         ⊥E
350     else
351         let (new_int_env,updated_int_ids) = ints :
352         let updated_ids = {sid}∪ updated_int_ids :
353         let updated_stream_env = (new_ext_env,new_int_env) :
354         (updated_stream_env,updated_ids)
355
356  ------------------------------------------------------------------------------
357  update_int_stream_env :
358     Present × InSID* × IntEnv ×  ℙ(SID) → (IntEnv ×  ℙ(SID)) ∪ {⊥E}
359  ------------------------------------------------------------------------------
360
361  update_int_stream_env(present, int_ids, env, ids_of_updated_streams) =
362     if int_ids = ε then
363       (env,ids_of_updated_streams)
364     else
365        let (id1,rest) = int_ids :
366        let old_ext_env = (present↓3)↓1 :
367        let old_int_env = (present↓3)↓2:
368        let (str_past,str_future) = old_int_env(id1) :
369        case str_future of
370           NIL :
```

```
            let new_env = env[id1 ↦ (str_past,str_future)] :
            update_int_stream_env(present,rest,new_env,ids_of_updated_streams)
         ?S :
            let new_env = env[id1 ↦ (str_past,str_future)] :
            update_int_stream_env(present,rest,new_env,ids_of_updated_streams)
         isStream(s) :
            let (past_messages,lastlength) = str_past :
            let s_streaminfo = s(present) :
            let (s_current_messages,s_future) = new_streaminfo :
                     // s_current_messages has the form of streampast:
                     // Message* × Nat
            case s_future of
               ⊥S : ⊥E
               NIL :
                   let new_past = (past_messages,0) :
                   let new_future = NIL :
                   let new_env = env[id1 ↦ (new_past, new_future)] :
                   let updated_present =
                         update_present_with_int_stream(present,id1,
                                                  new_past,new_future) :
                   update_int_stream_env(updated_present, rest,
                                         new_env, {id1}∪ids_of_updated_streams)
               ?S  :
                   let new_past = (past_messages,0) :
                   let new_future = ?S :
                   let new_env = env[id1 ↦ (new_past, new_future)] :
                   let updated_present =
                       update_present_with_int_stream(present,id1,
                                                  new_past,new_future) :
                   update_int_stream_env(updated_present, rest,
                                         new_env, {id1}∪ids_of_updated_streams)
               isStream(ms) :
                   let new_past = (past_messages||s_current_messages↓1,
                                   s_current_messages↓2) :
                   let new_env = env[id1 ↦ (new_past, s_future)] :
                   let updated_present =
                         update_present_with_int_stream(present,id1,
                                                   new_past,s_future) :
                   update_int_stream_env(updated_present, rest,
                                         new_env, {id1}∪ids_of_updated_streams)

-------------------------------------------------------------------------------
check_potential_violations :
   ℙ(PotentialViolation) × Present
     → ℙ(Violation) × ℙ(Warnings) × ℙ(PotentialViolation) ∪ {⊥}
-------------------------------------------------------------------------------

check_potential_violations_acc(pot_violations,present) =
    let new_set = {(p_1,SID_1,…,(p_k,SID_k),f(present)) |
                     (p_1,SID_1,…,(p_k,SID_k),f) ∈ pot_violations and
                      f(present) ≠ true}
    if contains_error(new_set) = true then
       ⊥
    else
       let violations = {((p1,SID1),…,(p_k,SID_k)) |
                           ((p1,SID1),…,(p_k,SID_k),false) ∈ new_set} :
       let warnings = {((p1,SID1),…,(p_k,SID_k)) |
                           ((p1,SID1),…,(p_k,SID_k),?F) ∈ new_set} :
       let potviols = intances \ (violations ∪ warnings) :
       (violations,warnings,potviols)

-------------------------------------------------------------------------------
update_present_with_int_stream :
      Present × InSID × StreamPast × StreamFuture  → Present
-------------------------------------------------------------------------------

update_present_with_int_stream(message,sid,envs,envf,envp,envv,envfp,
                               id1,new_past,new_future) =
    let (ext_env,int_env) = envs :
    let new_envs = (ext_env,int_env[id1↦ (new_past,new_future)]) :
    (message,sid,new_envs,envf,envp,envv,envfp)
```

```
================================================================================
2. Translation of Formulas
================================================================================

TYPES
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Tr⟦F⟧ = step

Input:  a formula F
Output: a function "step" that takes as input the present situation, i.e.

 message      … the current message
 stream_id    … the current stream name
 envs         … stream environment
 envf         … formula environment
 envp         … position environment
 envv         … value environment
 envfunpr     … environment for external functions and predicates
                (mapping from the names of nonlogical function and predicate
                 names to external functions and preducates)

and returns as an answer

* either true, false, ⊥F, or ?F
* or another step to be applied to the next situation.
================================================================================

--------------------------------------------------------------------------------
Tr⟦F⟧: Step[MK]

F ::= XF | BIND F
     | true | false | PV(T1,...,Tn)
     | ~ F  | F1 /\ F2 | F1 \/ F2 | F1 => F2 | F1 <=> F2
     | forall XP in SID with RAN : F | exists XP in SID with RAN: F
--------------------------------------------------------------------------------

Tr⟦XF⟧(message,stream_id,envs,envf,envp,envv,envfunpr) =
    inAnswer[MK](envf(XF))

Tr⟦BIND : F⟧  =
   let f: Binder × Step[MK] → Step[MK]
       f[bind0,f0](present) =
          let bind1 = bind0(present) :
          if bind1 = ⊥E then
             inAnswer[MK](⊥F)
          else
             let f1 : f0(bind1↓1) :
             case f1 of
                isMK(mk)      : f1
                isStep[MK](mkstep) : f[bind1,mkstep]
   : f[Tr⟦BIND⟧,Tr⟦F⟧]

Tr⟦true⟧(present) =
   inAnswer[MK](true)

Tr⟦false⟧(present) =
   inAnswer[MK](false)

Tr⟦PV(T1,…,Tn)⟧ =
 let f: (Present → VPSⁿ →  MK) × Step[VPS]^n → Step[MK] :
   f[pv,t1,…,tn](present) =
     let r1 = t1(present),…,rn = tn(present):
       if r1 = ⊥V ∨ ⋯ ∨ rn = ⊥V ∨ r1 = ⊥P ∨ ⋯ ∨ rn = ⊥P then
         inAnswer[MK](⊥F)
       else
         let i₁,…,i_k be the maximal subsequence of 1,…,n such that
             r_i₁ = isVPS(v₁) ∧ ⋯ ∧ r_{i_k} = isVPS(v_k) :
         let j₁,…,j_{n-k} be the subequence of 1,…,n such that
             r_j₁ = isStep[VPS](s₁) ∧ ⋯ ∧ r_{j_{n-k}} = isStep[VPS](s_{n-k}) :
         if k = n then
             let m = pv(present)(v₁,…,v_k) :
             inAnswer[MK](m)
         else
```

```
            let g₁,…,gk : Step[VPS]
              g₁(present)  = inAnswer[VPS](v₁),…,
              g_k(present) = inAnswer[VPS](vk}) :
              let ter₁,…,ter_n be the sequence such that for all 1≤i≤n,
                  if i ∈ {i₁,…,i_k} then
                     ter_i = g_i
                  else
                     ter_i = r_i :
            inAnswer[MK](f[ter₁,…,ter_n])
  : f[Tr⟦PV⟧, Tr⟦T1⟧,…,Tr⟦Tn⟧ ]

Tr⟦~F⟧ =
  let f: Step[MK] → Step[MK],
    f[f0](present) =
      let r = f0(present):
      case r of:
         isMK(m):
            if m = false then
               inAnswer[MK](true)
            else if m = true then
               inAnswer[MK](false)
            else if m = ?F then
               inAnswer[MK](?F}
            else
               inAnswer[MK](⊥F)
         isStep[MK](s):
            inAnswer[MK](f[s])
  : f[Tr⟦F⟧]

Tr⟦F1 /\ F2⟧ =
  let f: Step[MK] × Step[MK] → Step[MK],
     f[f1,f2](present) =
       let r1 = f1(present):
       case r1 of
         isMK(m1):
            if m1 = ⊥F ∨ m1 = false then
              r1
            else
              let r2 = f2(present):
              case r2 of
                isMK(m2) :
                  if m2 = true then
                     r1
                  else
                     r2
                isStep[MK](s2):
                   inAnswer[MK](f[f1,s2])
         isStep[MK](s1):
            inAnswer[MK](f[s1,f2])
    : f[Tr⟦F1⟧,Tr⟦F2⟧]

Tr⟦F1 \/ F2⟧ =
  let f: Step[MK] × Step[MK] → Step[MK],
    f[f1,f2](present) =
       let r1 = f1(present):
       case r1 of
         isMK(m1):
            if m1 = ⊥F ∨ m1 = true then
              r1
            else
              let r2 = f2(present):
              case r2 of
                isMK(m2):
                  if m2 = false then
                     r1
                  else
                     r2
                isStep[MK](s2)
                  if m1 = false then
                     r2
                  else
                     inAnswer[MK](f[f1,s2])
         isStep[MK](s1):
            inAnswer[MK](f[s1,f2])
```

```
593     : f[Tr⟦F1⟧,Tr⟦F2⟧]
594
595   Tr⟦F1 => F2⟧ =
596     let f: Step[MK] × Step[MK] → Step[MK],
597       f[f1,f2](present) =
598         let r1 = f1(present):
599         case r1 of
600           isMK(m1):
601             if m1 = ⊥F then
602               r1
603             else if m1 = false then
604               inAnswer[MK](true)
605             else
606               let r2 = f2(present)
607               case r2 of
608                 isMK(m2):
609                   if m2 = false then
610                     if m1 = true then
611                       false
612                     else
613                       ?F
614                   else
615                     r2
616                 isStep[MK](s2):
617                   if m1 = true then
618                     r2
619                   else
620                     inAnswer[MK](f[f1,s2])
621           isStep[MK](s1):
622             inAnswer[MK](f[s1,f2])
623     : f[Tr⟦F1⟧,Tr⟦F2⟧]
624
625   Tr⟦F1 <=> F2⟧ =
626     let f: Step[MK] × Step[MK] → Step[MK],
627       f[f1,f2](present) =
628         let r1 = f1(present):
629         case r1 of
630           isMK(m1):
631             if m1 = ⊥F then
632               r1
633             else if m1 = ?F then
634               r1
635             else
636               let r2 = f2(present):
637               if m1 = true then
638                  r2
639               else
640                  case r2 of
641                    isMK(m2):
642                      if m2 = true then
643                         inAnswer[MK](false)
644                      else if m2 = false then
645                         inAnswer[MK](true)
646                      else
647                         r2
648                    isStep[MK](s2):
649                         inAnswer[MK](f[false,s2])
650             isStep[MK](s1):
651               inAnswer[MK](f[s1,f2])
652       : f[Tr⟦F1⟧,Tr⟦F2⟧]
653
654   // Type checker should guarantee that the positions in RAN
655   // are positions in the stream SID
656
657   Tr⟦forall XP in SID with RAN : F⟧ =
658     all_ex["forall"](XP,TS,RAN,F)
659
660   Tr⟦exists XP in SID with RAN : F⟧ =
661     all_ex["exists"](XP,TS,RAN,F)
662
663   ------------------------------------------------------------------------------
664   all_ex : {"forall", "exists"}
665            → XP × SID × RAN × F
666            → Step[MK]
```

```
667  ------------------------------------------------------------------------------
668  // all_ex first fixes the boundaries of the range
669  // then computes the truth value of the quantified formula.
670  // If boundaries are unknown or error position, it returns
671  // unknown and error truth values, respectively.
672  // If a boundary is a step, all_ex returns a step.
673  // If both boundaries are definite positions (finite or infinite)
674  // it tries to compute the truth value.
675
676  all_ex[forall_exists](XP,SID,RAN,F) =
677    let ae : {"forall", "exists"}
678          → XP × SID × (Present → Range ∪ {?P,⊥P}) × F
679          → Step[MK]
680    ae[al_ex](xp,sid,ran,formula)(present) =
681      let r = ran(present) :
682      if r = ?P then
683         inAnswer[MK](?F)
684      else if r = ⊥P then
685         inAnswer[MK](⊥F)
686      else
687         let (tp1,pp1,xp0,pp2,tp2) = r :
688         if xp0 ≠ xp then
689            inAnswer[MK](⊥F)
690         else case tp1 of
691            isStep[Position⊥?](s1) :
692               let ran1 = (s1,pp1,xp0,pp2,tp2) :
693               inAnswer[MK](ae[forall_exists](xp,sid,ran1,formula))
694            isPosition(pos1) :
695               case tp2 of
696                  isStep[Position⊥?](s2) :
697                     let ran2 = (pos1,pp1,xp0,pp2,s2) :
698                     inAnswer[MK](ae[forall_exists](xp,sid,ran1,formula))
699                  isPosition(pos2) :
700                     let (from,to0) =
701                              precise_range_boundaries(pos1,pp1,pp2,pos2) :
702                     let default =
703                         if al_ex = "forall" then
704                            true
705                         else
706                            false :
707                     forall_exists_pos_pos[default,default]
708                                    (from,xp,to,sid,formula)(present)
709  : ae[forall_exists](XP,SID,Tr⟦RAN⟧,F)
710
711  ------------------------------------------------------------------------------
712  forall_exists_pos_pos : Bool∪{?F} × Bool
713                        → Position × XP × Position × SID × F
714                        → Step[MK]
715  ------------------------------------------------------------------------------
716
717  forall_exists_pos_pos[cur,def](from,xp,to,sid,fml)(present) =
718        if to > from then
719           inAnswer[MK](cur) : // does not match semantics
720        else
721           let streampast = get_stream_past(sid,present) :
722           if streampast = ⊥S then
723              inAnswer[MK](⊥F)
724           else
725              let max_pos = maximal_position(streampast) :
726              if max_pos < from then
727                 inAnswer[MK](forall_exists_pos_pos[cur,def](from,xp,to,sid,fml))
728              else
729                 if max_pos < to then
730                    let upper = max_pos :
731                    let flag = "incomplete" :
732                 else
733                    let upper = to :
734                    let flag = "complete" :
735                 let f =
736                      forall_exists_history[cur,def](from,xp,upper,fml)(present) :
737                 case f of
738                    isMK(mk) :
739                       if flag = "complete" then
740                          mk
```

```
                        else if mk = ⊥F ∨ mk = ¬def then
                           mk
                        else
                           inAnswer[MK](forall_exists_pos_pos[mk,def]
                                          (upper,xp,to,fml))
                     isPositionBool?(pos_b) :
                        let (p,b) = pos_b :
                        inAnswer[MK](forall_exists_pos_pos[b,def](p,xp,to,fml))

  ------------------------------------------------------------------------------
  forall_exists_history : Boolu{?F} × Bool
                           → Position × XP × Position × Step[MK]
                           → Present
                           → MK + PositionBool?

  PositionBool? = Position × Bool ∪ {?F}
  ------------------------------------------------------------------------------

  forall_exists_history[current,default](from,xp,to,formula)(present) =
   if from > to then
      inMK(current)
   else
      let f = formula(present[xp↦from]) :
      case f of
         isMK(mk) :
            if mk = ⊥F ∨ mk = ¬default then
               inMK(mk)
            else if mk = default then
               forall_exists_history[current,default]
                                    (from+1,xp,to,formula)(present)
            else // mk = ?F
               forall_exists_history[mk,default]
                                 (from+1,xp,to,formula)(present)
         isStep[MK](st) :
            let pb = (from,current) :
            inPositionBool?(pb)

  ------------------------------------------------------------------------------
  precise_range_boundaries :
          Position × {"≤","<"} × {"≤","<"} × Position
          → Position × Position
  ------------------------------------------------------------------------------

  precise_range_boundaries(pos1,pp1,pp2,pos2) =
     let from =
         if pp1 = "≤" then
            pos1
         else
            pos1 + 1
     let to0 =
         if pp2 = "≤" ∨ pos2 = "∞" then
            pos2
         else
            pos2 - 1
  : (from,to)

  ------------------------------------------------------------------------------
  Tr⟦TP⟧: Step[Position⊥?]
  TP ::= XP | BIND : TP
         | 0 | increment(SID⏲TP,N) | decrement(SID@TP,N)
         | max XP in SID with RAN : F | min XP in SID with RAN : F
  ------------------------------------------------------------------------------

  Tr⟦XP⟧(message,stream_id,history,envf,envp,envv,envfunpr) =
     inAnswer[Position⊥?](envp(XP))

  Tr⟦ BIND : TP ⟧ =
     let f: Binder × Step[Position⊥?] → Step[Position⊥?]
         f[bind0,f0](present) =
            let bind1 = bind0(present) :
            if bind1 = ⊥E then
               inAnswer[Position⊥?](⊥P)
            else
               let p : f0(bind1↓1) :
```

```
              case p of
                 isPosition⊥?(p1)          : p1
                 isStep[Position⊥?](pstep) : f[bind1,pstep]
    : f[Tr⟦BIND⟧,Tr⟦TP⟧]

Tr⟦0⟧ = inAnswer[Position](0)

Tr⟦increment(SID◷TP,N)⟧ =
  let f: SID × TP × Value → Step[Position⊥?] :
      f[sid0, pos0, v0](present) =
         let t = time_or_content(sid0,pos0,"time")(present) :
         case t of
            isValue⊥?(time)    :
               if time = ⊥V then
                  inAnswer[Position⊥?](⊥P)
               else if time = ?V then
                  inAnswer[Position⊥?](?P)
               else
                  let streampast = get_stream_past(sid0,present) :
                  if streampast = ⊥S then
                     inAnswer[Position⊥?](⊥P)
                  else
                     let max_pos = maximal_position(streampast) :
                     let (tmax,_) = extract_message(streampast,max_pos,0) :
                     if tmax ≥ time + v0 then
                        let p1 = min p :
                               extract_message(streampast,p,0)↓1 ≥ time + v0 :
                        inAnswer[Position⊥?](p1)
                     else
                        inAnswer[Position⊥?](f(sid0, pos0, v0))
            isStep(Value⊥?)(v) :
               inAnswer[Position⊥?](f(sid0, pos0, v0))
  : f(SID,TP,Tr⟦N⟧)

Tr⟦decrement(SID◷TP,N)⟧ =
  let f: SID × Step[Position⊥?] × Value → Step[Position⊥?] :
      f[sid0, pos0, v0](present) =
         let t = time_or_content(stream0,pos0,"time")(present) :
         case t of
            isValue⊥?(time)      :
               if time = ⊥V then
                  inAnswer[Position⊥?](⊥P)
               else if time = ?V then
                  inAnswer[Position⊥?](?P)
               else
                  let tmin = time - v0 :
                     if tmin < 0 then
                        inAnswer[Position⊥?](⊥P)
                     else
                        let streampast = get_stream_past(sid0,present) :
                        if streampast = ⊥S then
                           inAnswer[Position⊥?](⊥P)
                        else
                           let p1 = min p :
                                extract_message(streampast,p,0)↓1 ≥ time - v0 :
                           inAnswer[Position⊥?](p1)
            isStep(Value⊥?)(v) :
                  inAnswer[Position⊥?](f(sid0, pos0, v0))
  : f(SID,Tr⟦TP⟧,Tr⟦N⟧)


Tr⟦min XP in SID with RAN : F⟧ =
  min_max["min"](XP,TS,RAN,F)

Tr⟦max XP in SID with RAN : F⟧ =
  min_max["max"](XP,TS,RAN,F)

-------------------------------------------------------------------------------
min_max : {"min", "max"}
        → XP × SID × RAN × F
        → Step[Position⊥?]
-------------------------------------------------------------------------------

min_max[min_or_max](XP,SID,RAN,F) =
```

```
  let minmax :   {"min", "max"}
               → XP × SID × (Present → Range ∪ {?P,⊥P}) × F
               → Step[Position⊥?]
  minmax[mm](xp,sid,ran,formulas)(present) =
     let r = ran(present) :
     if r = ?P then
        inAnswer[Position⊥?](?P)
     else if r = ⊥P then
        inAnswer[Position⊥?](⊥P)
     else
        let (tp1,pp1,xp0,pp2,tp2) = r :
        if xp0 ≠ xp then
           inAnswer[Position⊥?](⊥P)
        else case tp1 of
           isStep[Position⊥?](s1) :
              let ran1 = (s1,pp1,xp0,pp2,tp2) :
              inAnswer[Position⊥?](minmax[mm](xp,sid,ran1,formula))
           isPosition(pos1) :
              case tp2 of
                 isStep[Position⊥?](s2) :
                    let ran2 = (pos1,pp1,xp0,pp2,s2) :
                    inAnswer[Position⊥?](minmax[mm](xp,sid,ran1,formula))
                 isPosition(pos2) :
                    let (from,to) =
                          precise_range_boundaries(pos1,pp1,pp2,pos2) :
                    let default = ?P:
                    min_max_pos_pos[mm,default]
                          (from,xp,to,sid,formula)(present))
: minmax[min_or_max](XP,SID,Tr⟦RAN⟧,F)

-------------------------------------------------------------------------------
min_max_pos_pos :   {"min","max"} × Position∪{?P}
                  → Position × XP × Position × SID × F
                  → Step[Position⊥?]
-------------------------------------------------------------------------------

min_max_pos_pos[min_or_max,p](from,xp,to,sid,formula)(present) =
  if from > to then
     inAnswer[Position⊥?](p)
  else
     let streampast = get_stream_past(sid,present) :
     if streampast = ⊥S then
        inAnswer[Position⊥?](⊥P)
     else
        let max_pos = maximal_position(streampast) :
        if max_pos < from then
           inAnswer[Position⊥?]
             (min_max_pos_pos[min_or_max,p](from,xp,to,sid,formula))
        else
           if max_pos < to then
             let upper = max_pos :
             let flag = "incomplete" :
           else
             let upper = to :
             let flag = "complete" :
           let pos = min_max_history[min_or_max,p](from,xp,upper,sid,formula)
                                                  (present):
           case pos of
             isPosition⊥?(pos0) :
                if flag = "complete" then
                   inAnswer[Position⊥?](pos0)
                else
                   if pos0 = ⊥P then
                       inAnswer[Position⊥?](⊥P)
                   else if pos0 = ?P then
                       inAnswer[Position⊥?]
                               (min_max_pos_pos[min_or_max,pos0]
                                   (upper,xp,to,sid,formula))
                   else if min_max = "min" then
                       inAnswer[Position⊥?](pos0)
                   else   // min_max = "max" ∧ flag = "incomplete"
                       inAnswer[Position⊥?]
                               (min_max_pos_pos[min_or_max,pos0]
                                   (upper,xp,to,sid,formula))
```

```
              isPositionPos?(pos_pos)
                 let (bound,p) = pos_pos :
                 inAnswer[Position⊥?]
                     (min_max_pos_pos[min_or_max,p](bound,xp,to,sid,formula))

-------------------------------------------------------------------------------
min_max_history :   {"min","max"} × Position ∪ {?}
                  → Position × XP × Position × F
                  → Present
                  → Position⊥? + PositionPos?

PositionPos? = Position × (Position ∪ {?P})
-------------------------------------------------------------------------------

min_max_history[min_max,pos](from,xp,to,formula)(present) =
 if from > to then
    inPosition⊥?(pos)
 else
    let f = formula(present[xp↦from]) :
    case f of
       isMK(mk) :
          if mk = ⊥F then
             inPosition⊥?(⊥P)
          else if mk = true then
             if min_max = "min" then
                inPosition⊥?(from)
             else
                min_max_history[min_max,from](from+1,xp,to,formula)(present)
          else  // if mk=false ∨ mk=?F
             min_max_history[min_max,pos](from+1,xp,to,formula)(present)
       isStep[MK](st) :
          let pp = (from,pos) :
          inPositionPos?(pp)

-------------------------------------------------------------------------------
Tr⟦TV⟧: Step[Value⊥?]

TV ::= XV | BIND : TV
       | SID⏲TP | SID@TP
       | FV(TV1,...,TVn) | num XP in SID with RANB : F
       | complete combine[TV0, FV] XP in SID with RAN CONSTR until F : TV1
-------------------------------------------------------------------------------

Tr⟦XV⟧(message,stream_id,envs,envf,envp,envv,envfunpr) =
   inAnswer[Value⊥?](env(XV))

Tr⟦ BIND : TV ⟧ =
     let f: Binder × Step[Value⊥?] → Step[Value⊥?]
       f[bind0,f0](present) =
          let bind1 = bind0(present) :
          if bind1 = ⊥E then
             inAnswer[Value⊥?](⊥V)
          else
             let p : f0(bind1↓1) :
             case p of
                isValue⊥?(v1)          : v1
                isStep[Value⊥?](vstep) : f[bind1,vstep]
   : f[Tr⟦BIND⟧,Tr⟦TV⟧]

// Below the type checker should make sure that TP is a position
// in the stream referred by SID

Tr⟦SID⏲TP⟧ =
   let tp = Tr⟦TP⟧ :
   time_or_content[SID,tp,"time"]

Tr⟦SID@TP⟧ =
   let tp = Tr⟦TP⟧ :
   time_or_content[SID,tp,"content"]

-------------------------------------------------------------------------------
time_or_content : SID × Step[Position⊥?] × {"time", "content"}
                  → Step[Value⊥?]
```

```
-------------------------------------------------------------------------------

time_or_content[SID,tp,time_content] =
  let f: SID × Step[Position⊥?] × {"time", "content"}
         → Step[Value⊥?]
    f[sid,tp0,tv](present) =
           let streampast = get_stream_past(sid,present)
           if streampast = ⊥S then
             inAnswer[Value⊥?](⊥V)
           else
             let p = tp0(present) :
             case p of
               isPosition⊥?(p0) :
                  if p0 = ⊥P  then
                     inAnswer[Value⊥?](⊥V)
                  else if p0 = ?P  then
                     inAnswer[Value⊥?](?V)
                  else if maximal_position(streampast) < p0 then
                     inAnswer[Value⊥?](f(sid,tp0,tv))
                  else
                     let m = extract_message(streampast,p0) :
                     let (t,content) = m :
                     if tv = "time" then
                         inAnswer[Value⊥?](t)
                     else
                         inAnswer[Value⊥?](content)
               isStep[Position⊥?](tp1) :
                  inAnswer[Value⊥?](f[sid,tp1,tv])
  : f[SID,tp,time_content]

-------------------------------------------------------------------------------
get_stream_past : SID × Present → StreamPast ∪ {⊥S}
-------------------------------------------------------------------------------

get_stream_past(id,present) =
   let ext_env = (present↓3)↓1 :
   if id ∈ dom(ext_env) then
      ext_env(id)
   else
     let int_env = (present↓3)↓2 :
     if id ∈ dom(int_env) then
        int_env(id)↓1
     else
        ⊥S

-------------------------------------------------------------------------------
extract_message : StreamPast × Position → Message
-------------------------------------------------------------------------------

extract_message(streampast,p) =
    let (m,rest) = streampast :
    if p = 0 then
       m
    else
       let p1 = p - 1
       extract_message(rest,p1)

Tr⟦FV(T1,…,Tn)⟧ =
 let f: (Present → VPSⁿ → Value⊥?) × Step[VPS]ⁿ
        → Step[Value⊥?] :
   f[fv,t1,…,tn](present) =
     let r1 = t1(present),…,rn = tn(present):
       if r1 = ⊥V ∨ ⋯ ∨ rn = ⊥V ∨ r1 = ⊥P ∨ ⋯ ∨ rn = ⊥P then
         inAnswer[Value⊥?](⊥F)
       else
         let i₁,…,i_k be the maximal subsequence of 1,…,n such that
             r_i₁ = isVPS(v₁) ∧ ⋯ ∧ r_{i_k} = isVPS(v_k) :
         let j₁,…,j_{n-k} be the subequence of 1,…,n such that
             r_j₁ = isStep[VPS](s₁) ∧ ⋯ ∧ r_{j_{n-k}} = isStep[VPS](s_{n-k}) :
         if k = n then
             let m = fv(present)(v₁,…,v_k) :
             inAnswer[Value⊥?](m)
         else
           let g₁,…,gk : Step[VPS]
```

```
              g₁(present) = inAnswer[VPS](v₁),…,
              g_k(present) = inAnswer[VPS](vk}) :
            let ter₁,…,ter_n be the sequence such that for all 1≤i≤n,
                  if i ∈ {i₁,…,i_k} then
                     ter_i = g_i
                  else
                     ter_i = r_i :
            inAnswer[Value⊥?](f[ter₁,…,ter_n])
  : f[Tr⟦FV⟧,Tr⟦T1⟧,…,Tr⟦Tn⟧]

// Type checking should guarantee that the positions in RAN refer to
// the stream SID

Tr⟦num XP in SID with RAN : F⟧ =
  let number : XP × SID × Range ∪ {?P,⊥P}) × F → Step[Value⊥?]
  number(xp,sid,ran,formula)(present) =
    let r = ran(present) :
    if r = ?P then
       inAnswer[Value⊥?](?V)
    else if r = ⊥P then
       inAnswer[Value⊥?](⊥V)
    else
       let (tp1,pp1,xp0,pp2,tp2) = r :
       if xp0 ≠ xp then
           inAnswer[Value⊥?](⊥V)
       else case tp1 of
           isStep[Position⊥?](s1) :
              let ran1 = (s1,pp1,xp0,pp2,tp2) :
              inAnswer[Value⊥?](number(xp,sid,ran1,formula))
           isPosition(pos1) :
              isStep[Position⊥?](s2) :
                  let ran2 = (pos1,pp1,xp0,pp2,s2) :
                  inAnswer[Value⊥?](number(xp,sid,ran1,formula))
              isPosition(pos2) :
                    let (from,to) =
                          precise_range_boundaries(pos1,pp1,pp2,pos2) :
                    let default = 0 :
                    num_pos_pos[default](from,xp,to,sid,formula)(present))
: number(XP,SID,Tr⟦RAN⟧,F)

-------------------------------------------------------------------------------
num_pos_pos :   Value
                → Position × XP × Position × SID × F
                → Step[Value⊥?]
-------------------------------------------------------------------------------

num_pos_pos[v](from,xp,to,sid,formula)(present) =
  if from > to then
     inAnswer[Value⊥?](v)
  else
     let streampast = get_stream_past(sid,present) :
     if streampast = ⊥S then
        inAnswer[Value⊥?](⊥V)
     else
        let max_pos = maximal_position(streampast) :
        if max_pos < from then
           inAnswer[Value⊥?](num_pos_pos[v](from,xp,to,sid,formula))
        else
           if max_pos < to then
             let upper = max_pos :
             let flag = "incomplete" :
           else
             let upper = to :
             let flag = "complete" :
           let val = num_history[v](from,xp,upper,formula)(present):
           case val of
              isValue⊥?(val0) :
                 if flag = "complete" then
                    inAnswer[Value⊥?](val0)
                 else
                    if val0 = ⊥V then
                       inAnswer[Value⊥?](⊥V)
                    else  // val0 is never ?V. Hence, it is a number here
                       inAnswer[Value⊥?](num_pos_pos[val0]
```

```
                                                    (upper,xp,to,sid,formula))
               isPositionVal?(pos_val) :
                  let (bound,v1) = pos_val :
                  inAnswer[Value⊥?](num_pos_pos[v1](bound,xp,to,sid,formula))

-------------------------------------------------------------------------------
num_history :   Value
              → Position × XP × Position × formula
              → Present
              → Value⊥? + PositionVal?

PositionVal? = Position × Value∪{?V}
-------------------------------------------------------------------------------

num_history[val](from,xp,to,formula)(present) =
 if from > to then
    inValue⊥?(val)
 else
    let f = formula(present[xp↦from]) :
    case f of
       isMK(mk) :
          if mk = ⊥F then
             inValue⊥?(⊥V)
          else if mk = true then
             num_history[val+1](from+1,xp,to,formula)(present)
          else
             num_history[val](from+1,xp,to,formula)(present)
       isStep[MK](st) :
          let pv = (from,v) :
          inPositionVal?(pv)

Tr⟦complete combine[TV0, FV] XP in SID with RAN CONSTR until F : TV1⟧ =
 let f : Step[Value⊥?] × (Present → VPSⁿ → Value⊥?)
           × XP × SID
           × Present → Range ∪ {?P,⊥P}
           × Step[ConstrDom]
           × Step[MK]
           × Step[Value⊥?]
         → Step[Value⊥?]
 f[v0,fv,xp,sid,r,c,f,v1](present) =
   let r_loc = r(present) :
   if r_loc = ⊥P then
     ⊥V
   else if r_loc = ?P then
     ?V
   else
     let (p1,pp1,xp1,pp2,p2) = r_loc :
     if xp1 ≠ xp then
        ⊥V
     else case p1 of
        isStep[Position⊥?](pos1) :
             let r1 = (pos1,pp1,xp1,pp2,p2) :
             inAnswer[Value⊥?](f[v0,fv,xp,sid,r1,c,f,v1])
        isPosition⊥?(pos1)       :
            if pos1 = ⊥P then
               ⊥V
            else if pos1 = ?P then
               ?V
            else
               case p2 of
                 isStep[Position⊥?](pos2) :
                   let r1 = (pos1,pp1,xp1,pp2,pos2) :
                   inAnswer[Value⊥?](f[v0,fv,xp,sid,r1,c,f,v1])
                 isPosition⊥?(pos2)  :
                   if pos2 = ⊥P then
                      ⊥V
                   else if pos2 = ?P then
                      ?V
                   else
                      let streampast = get_stream_past(sid,present) :
                      if streampast = ⊥S then
                         ⊥V
                      else
                         let r1 = (pos1,pp1,xp1,pp2,pos2) :
```

```
                          let max_pos = maximal_position(streampast) :
                          let (from,to) =
                                 precise_range_boundaries(pos1,pp1,pp2,pos2) :
                          if max_pos < to then
                              inAnswer[Value⊥?](f[v0,fv,xp,sid,r1,c,f,v1])
                          else
                             let acc0 = v0(present) :
                             case acc0 of
                                isStep[Value⊥?](val)   :
                                   inAnswer[Value⊥?](f[val,fv,xp,sid,r,c,f,v1])
                                isValue⊥?(acc) :
                                   if acc = ⊥V ∨ acc = ?V then
                                      inAnswer[Value⊥?](acc)
                                   else
                                      let combine_values = fv(present) :
                                      value_comb[xp,sid,c,f,v1,combine_values]
                                                 (from, to, acc)
 : f[Tr⟦TV0⟧,Tr⟦FV⟧,XP,SID,Tr⟦RAN⟧,Tr⟦CONSTR⟧,Tr⟦F⟧,Tr⟦TV1⟧]

-------------------------------------------------------------------------------
value_comb :
    (XP × SID × Step[ConstrDom] × Step[MK] × Step[Value⊥?]
        × (Value ∪ {?V} × Value ∪ {?V} → Value⊥?)
    → Position × Position × Value
    → Step[Value⊥?]
-------------------------------------------------------------------------------

value_comb[xp,sid,constr,formula,val,combine_values](from, to, acc)
          (present) =
  if from > to then
     inAnswer[Value⊥V](acc)
  else
     let c0 = constr(present[xp↦from]) :
     case c0 of
        isStep(ConstrDom)(c) :
           value_comb[xp,sid,c,formula,val,combine_values](from, to, acc)
        isConstrDom(c)       :
           case c of
              ⊥E : inAnswer[Value⊥?](⊥V)
              isPresentMK(pres_mk) :
                  let (pres,mk) = pres_mk :
                  if mk = ⊥F then
                     inAnswer[Value⊥V](⊥V)
                  else if mk = ?F ∨ mk = false then
                     let from1 = from + 1 :
                     value_comb[xp,constr,formula,val,combine_values]
                               (from1,to,acc)
                  else
                     let v1 = val(pres) :
                     case v1 of
                        isStep(Value⊥?)(v) :
                          inAnswer[Value⊥?](value_comb[xp,sid,constr,
                                                f,v,combine_values])(from,to,acc)
                        isValue⊥?(v)       :
                           if v = ⊥v then
                              inAnswer[Value⊥V](⊥V)
                           else
                              let newacc = combine_values(acc,v) :
                              if newacc = ⊥V then
                                 inAnswer[Value⊥V](⊥V)
                              else
                                 let f0 = formula(pres[THIS↦newacc]) :
                                         // [THIS↦newacc] Affects the envv
                                         // component of present
                                 case f0 of
                                     isStep[MK](f1) :
                                        inAnswer[Value⊥?]
                                          (value_comb[xp,sid,constr,f1,val,
                                                combine_values](from,to,acc))
// The call to value_comb above will use acc, not newacc
// in order not to keep increasing newacc while
// staying within the same range (from,to) as long as f1 is the step.
                                     isMK(f1)   :
                                        if f1 = ⊥F then
```

```
                                          inAnswer[Value⊥V](⊥V)
                                       else if f1 = ?F ∨ f1 = false then
                                          let from1 = from + 1 :
                                          inAnswer[Value⊥?]
                                              (value_comb[xp,sid,constr,
                                                 formula,val, combine_values]
                                                     (from1, to, newacc))
                                       else
                                          inAnswer[Value⊥V](newacc)

--------------------------------------------------------------------------------
Tr⟦TS⟧: Stream

TS ::= ES | IS

ES ::= ExSID

IS ::= InSID | BIND : IS
       | FS(T1,...,Tn)
       | partial combine[TV0, FV] XP in SID with RAN CONSTR until F : TV
         Default for F in 'until F' : false
       | construct XP in SID with RAN CONSTR : TV
       | construct XP in SID with RAN CONSTR : IS

SID ::= ExSID | InSID
--------------------------------------------------------------------------------

Tr⟦XS⟧(message,stream_id,envs,envf,envp,envv,envfunpr) =
    envs(XS)

Tr⟦ BIND : IS ⟧ =
   let f: Binder × Stream → Stream
       f[bind0,s0](present) =
          let bind1 = bind0(present) :
          if bind1 = ⊥E then
             ⊥S
          else
             let present1 = bind1↓1 :
             let s1 = s0(present1) :
             case s1 of
                is⊥S : ⊥S
                isStreamInfo(info) :
                     let info = (streampast,streamfuture) :
                     let (messages,l) = streampast :
                        case streamfuture of
                           isNIL :
                              info
                           is?S :
                              info
                           isExternal :
                              f[bind1,info]
                           isStream(s) :
                              f[bind1,info]
    : f[Tr⟦BIND⟧,Tr⟦IS⟧]

Tr⟦FS(T1,…,Tn)⟧ =
  let f:  (Present → VPSⁿ → Stream) × Step[VPS]ⁿ → Stream :
   f[fs,t1,…,tn](present) =
     let r1 = t1(present),…,rn = tn(present):
       if r1 = ⊥V ∨ ⋯ ∨ rn = ⊥V ∨ r1 = ⊥P ∨ ⋯ ∨ rn = ⊥P then
         ⊥S
       else
         let i₁,…,i_k be the maximal subsequence of 1,…,n such that
             r_i₁ = isVPS(v₁) ∧ ⋯ ∧ r_{i_k} = isVPS(v_k) :
         let j₁,…,j_{n-k} be the subequence of 1,…,n such that
             r_j₁ = isStep[VPS](s₁) ∧ ⋯ ∧ r_{j_{n-k}} = isStep[VPS](s_{n-k}) :
         if k = n then
             (fs(present)(v₁,…,v_k))(present)
         else
           let g₁,…,gk : Step[VPS]
             g₁(present) = inAnswer[VPS](v₁),…,
             g_k(present) = inAnswer[VPS](vk}) :
             let ter₁,…,ter_n be the sequence such that for all 1≤i≤n,
                 if i ∈ {i₁,…,i_k} then
```

```
                    ter_i = g_i
                  else
                    ter_i = r_i :
            f[ter₁,…,ter_n](present)
  : f[Tr⟦FS⟧,Tr⟦T1⟧,…,Tr⟦Tn⟧]

Tr⟦partial combine[TV0, FV] XP in SID with RAN CONSTR until F : TV1⟧ =
   let f : Step[Value⊥?] × (Present → VPSⁿ → Value⊥?)
            × XP × SID
            × Present → Range ∪ {?P,⊥P}
            × Step[ConstrDom]
            × Step[MK]
            × Step[Value⊥?]
           → Stream
   f[v0,fv,xp,sid,r,c,f,v1](present) =
       let r_loc = r(present) :
       if r_loc = ⊥P then
         ⊥S
       else if r_loc = ?P then
         let past = (ε,0) :
         (past,?S)
       else
         let (p1,pp1,xp1,pp2,p2) = r_loc :
         if xp1 ≠ xp then
           ⊥S
         else case p1 of
            isStep[Position⊥?](pos1) :
               let r1 = (pos1,pp1,xp1,pp2,p2) :
               let past = (ε,0) :
               (past,f[v0,fv,xp,sid,r1,c,f,v1])
            isPosition⊥?(pos1)       :
                if pos1 = ⊥P then
                   ⊥S
                else if pos1 = ?P then
                   let past = (ε,0) :
                   (past,?S)
                else
                   case p2 of
                     isStep[Position⊥?](pos2) :
                        let r1 = (pos1,pp1,xp1,pp2,pos2) :
                        let past = (ε,0) :
                        (past,f[v0,fv,xp,sid,r1,c,f,v1])
                     isPosition⊥?(pos2)       :
                       if pos2 = ⊥P then
                          ⊥S
                       else if pos2 = ?P then
                          let past = (ε,0) :
                          (past,?S)
                       else
                          let streampast = get_stream_past(sid,present) :
                          if streampast = ⊥S then
                             ⊥S
                          else
                             let r1 = (pos1,pp1,xp1,pp2,pos2) :
                             let max_pos = maximal_position(streampast) :
                             let (from,to) =
                                precise_range_boundaries(pos1,pp1,pp2,pos2) :
                              if from > max_pos then
                                f[v0,fv,xp,sid,r1,c,f,v1]
                              else
                                let v = v0(present)
                                case v of
                                   isStep[Value⊥?](v') :
                                     f[v',fv,xp,sid,r1,c,f,v1]
                                   isValue⊥?(val) :
                                     if val = ⊥V then
                                        ⊥S
                                     else if val = ?V then
                                        let past = (ε,0) :
                                        (past,?S)
                                     else
                                        let current_time = (present↓1)↓1 :
                                        let start_message = (current_time,v) :
                                        let acc = (start_message,1) :
```

```
                                          let combine_values = fv(present) :
                                          if max_pos < to then
                                             let upper = max_pos :
                                             let flag = "incomplete" :
                                          else
                                             let upper = to :
                                             let flag = "complete" :
                                          let strpast =
                                               streampast_comb[xp,sid,c,f,v1,
                                                 combine_values](from,upper,acc) :
                                          case strpast of
                                              ⊥S : ⊥S
                                              ?S : ?S
                                              isStreamPast(pst) :
                                                let streaminfo =
                                                  if flag = "complete" then
                                                     (pst,NIL)
                                                  else
                                                     let r2=(upper,pp1,xp,pp2,to):
                                                     (pst,f[v0,fv,xp,sid,r2,
                                                            c,f,v1]) :
                                                 inStreamInfo(streaminfo)
   : f[Tr⟦TV0⟧,Tr⟦FV⟧,XP,SID,Tr⟦RAN⟧,Tr⟦CONSTR⟧,Tr⟦F⟧,Tr⟦TV1⟧]

-------------------------------------------------------------------------------
streampast_comb : (XP × SID × Step[ConstrDom] × Step[MK] × Step[Value⊥?]
                      × (Value ∪ {?V} × Value ∪ {?V} → Value⊥?)
                  → Position × Position × StreamPast
                  → Step[StreamPast⊥?]

StreamPast⊥? = StreamPast ∪ {⊥S,?S}
-------------------------------------------------------------------------------

streampast_comb[xp,sid,constraint,formula,tv,combine_values](from,to,acc)
                (present) =
   let constr = contraint(present[xp↦from]) :
   case constr of
     isStep(ConstrDom)(c0) :
       inAnswer[StreamPast⊥?]
          (streampast_comb[xp,sid,c0,formula,tv,combine_values](from,to,acc))
     isConstrDom(c)       :
       case c of
         ⊥E :         inAnswer[StreamPast⊥?](⊥S)
         isPresentMK(pres_mk) :
            let pres_mk = (pres,mk)  :
            if b1 = ⊥F then
               inAnswer[StreamPast⊥?](⊥S)
            else if mk ≠ true then
               let from1 = from + 1 :
               inAnswer[StreamPast⊥?]
                  (streampast_comb[xp,sid,c,formula,tv,combine_values]
                                  (from1,to,acc))
            else
               let v1 = tv(pres) :
               case v1 of
                  isStep(Value⊥?)(v) :
                     inAnswer[StreamPast⊥?]
                       (streampast_comb[xp,sid,c,formula,tv,combine_values]
                                       (from,to,acc))
                  isValue⊥?(v)       :
                     if v = ⊥v then
                       inAnswer[StreamPast⊥?](⊥S)
                     else
                       let newacc = combine_and_join(acc,v,combine_values) :
                       if newacc = ⊥S then
                          inAnswer[StreamPast⊥?](⊥S)
                       else
                          let b = formula(pres[THIS↦newacc]) :
                                       // Affects the envs
                                       // component of present
                          case b of
                             isStep[MK](f1) :
                                inAnswer[StreamPast⊥?]
                                   (streampast_comb[xp,sid,c,f1,tv,
```

```
                                                        combine_values](from,to,acc))
// The call to f above will uses acc and not newacc
// in order not to keep increasing newacc while
// staying within the same range (from,to) as long as f1 is the step.
                              isMK(b2)   :
                                 if b2 = ⊥F then
                                    inAnswer[StreamPast⊥?](⊥S)
                                 else if b2 ≠ false then
                                    inAnswer[StreamPast⊥?](newacc)
                                 else if from ≥ to then
                                    inAnswer[StreamPast⊥?](newacc)
                                 else
                                    let from1 = from + 1 :
                                    inAnswer[StreamPast⊥?]
                                       (streampast_comb[xp,c,formula,tv,
                                              combine_values](from1,to,newacc))

-------------------------------------------------------------------------------
combine and join :
    StreamPast
      × Value ∪ {?V}
      × (Value ∪ {?V} × Value ∪ {?V} → Value⊥?)
     → StreamPast ∪ {⊥V}
-------------------------------------------------------------------------------

combine and join(past, v, combine_values) =
  let cur_time = (past↓1)↓1 :
  let (messages,l) = past :
  if v = ?V then
     let newmessages = messages||(cur_time, v) :
     (newmessages,1)
   else
     let k = max i : extract_message(s, i, 0) ≠ (_,?V) :
     let (tk,vk) = extract_message(s, k, 0) :
     let newval = combine_values(vk, v) :
     if newval = ⊥V then
        ⊥S
     else
        let newmessages = messages||(cur_time, v) :
        (newmessages,1)

Tr⟦construct XP in SID with RAN CONSTR : TS⟧ =
  let f : XP × SID
           × Present → Range ∪ {?P,⊥P}
           × Step[ConstrDom]
           × Stream
           × StreamInfo*
          → Stream
  f[xp,sid,r,c,ts,strinfos_acc](present) =
    let r_loc = r(present) :
    let past = (ε,0) :
    if r_loc = ⊥P then
       in⊥S(⊥S)
    else if r_loc = ?P then
       let info = (past,?S) :
       inStreamInfo(info)
    else
       let (p1,pp1,xp1,pp2,p2) = r_loc :
       if xp1 ≠ xp then
         in⊥S(⊥S)
       else case p1 of
          isStep[Position⊥?](pos1) :
             let r1 = (pos1,pp1,xp1,pp2,p2) :
             inStreamInfo((past,f[xp,sid,r1,c,ts,strinfos_acc]))
          isPosition⊥?(pos1)       :
             if pos1 = ⊥P then
                in⊥S(⊥S)
             else if pos1 = ?P then
                let info = (past,?S) :
                inStreamInfo(info)
             else
                case p2 of
                  isStep[Position⊥?](pos2) :
                     let r1 = (pos1,pp1,xp1,pp2,pos2) :
```

```
                        let info = (past, f[xp,sid,r1,c,ts,strinfos_acc])
                        inStreamInfo(info)
                     isPosition⊥?(pos2)       :
                        if pos2 = ⊥P then
                           in⊥S(⊥S)
                        else if pos2 = ?P then
                           let info = (past,?S) :
                           inStreamInfo(info) :
                        else
                           let streampast = get_stream_past(sid,present) :
                           if streampast = ⊥S then
                              in⊥S(⊥S)
                           else
                              let r1 = (pos1,pp1,xp1,pp2,pos2) :
                              let max_pos = maximal_position(streampast) :
                              let (from,to) =
                                   precise_range_boundaries(pos1,pp1,pp2,pos2) :
                              if from > max_pos then
                                 let info = (past, f[xp,sid,r1,c,ts,strinfos_acc])
                                 inStreamInfo(info)
                              else
                                 if max_pos < to then
                                    let upper = max_pos :
                                    let flag = "incomplete" :
                                 else
                                    let upper = to :
                                    let flag = "complete" :
                                 let new_strinfos_acc =
                                     update_acc_strinfos[xp,sid,c,ts,strinfos_acc]
                                                          (from,to) :
                                 case new_strinfos_acc of
                                     isStep[StreamInfoStar⊥](step)  :
                                         let info =
                                             (past, f[xp,sid,r1,c,ts,strinfos_acc])
                                         inStreamInfo(info)
                                     isStreamInfoStar⊥(new_streaminfos) :
                                         if new_streaminfos = ⊥S then
                                            in⊥S(⊥S)
                                         else
                                            let pst= mergeall(new_streaminfos,ε) :
                                            let info =
                                                if flag = "complete" then
                                                   (pst,NIL)
                                                else
                                                   let r2=(upper,pp1,xp,pp2,to):
                                                   (pst, f[xp,sid,r1,c,ts,
                                                           new_streaminfos]) :
                                            InStreamInfo(info)
: f[XP,SID,Tr⟦RAN⟧,Tr⟦CONSTR⟧,Tr⟦TS⟧,ε]

-------------------------------------------------------------------------------
update_acc_strinfos : XP × SID × Step[ConstrDom] × Stream × StreamInfo*
                      → Position × Position
                      → Step[StreamInfoStar⊥]

StreamInfoStar⊥ = StreamInfoStar* ∪ {⊥S}
-------------------------------------------------------------------------------

update_acc_strinfos[xp0,sid0,contraint0,ts0,acc_strinfos0](from0,to0) =
 let f : XP × SID × Step[ConstrDom] × Stream × StreamInfo*
         → Position × Position
         → Step[StreamInfoStar⊥]
 f[xp,sid,contraint,ts,acc_strinfos](from,to)(present) =
   if from > to then
      inAnswer[StreamPast⊥](acc_strinfos)
   else
      let constr = contraint(present[xp↦from]) :
      case constr of
        isStep(ConstrDom)(c0) :
          inAnswer[StreamPast⊥]
           (update_acc_strinfos[xp,sid,c0,ts,acc_strinfos](from,to))
        isConstrDom(c)       :
          case c of
            ⊥E                   :  inAnswer[StreamInfoStar⊥](⊥S)
```

```
             isPresentMK(pres_mk) :
                let pres_mk = (pres,mk)  :
                if mk = ⊥F then
                   inAnswer[StreamInfoStar⊥](⊥S)
                else if mk ≠ true then
                   let from1 = from + 1 :
                   let newacc_strinfos =
                       update_acc_strinfos_aux(acc_strinfos,xp,from1,present,ε):
                   if newacc_strinfos = ⊥S then
                      in⊥S(⊥S)
                   else
                      inAnswer[StreamInfoStar⊥]
                        (update_acc_strinfos[xp,sid,c,ts,newacc_strinfos]
                                             (from1,to))
                else
                   let s1 = is(pres[xp↦from]) :
                   case s1 of
                      ⊥S : in⊥S(⊥S)
                      isStreamInfo(strinfo) :
                        let updated_strinfos =
                           update_acc_strinfos_aux(acc_strinfos,xp,from,pres,ε):
                        if newacc_strinfos = ⊥S then
                           in⊥S(⊥S)
                        else
                           let newacc_strinfos = acc_strinfos||strinfo :
                           inAnswer[StreamInfoStar⊥]
                              (update_acc_strinfos[xp,sid,c,ts,newacc_strinfos]
                                                   (from1,to))
  : f[xp0,sid0,contraint0,ts0,acc_strinfos0](from0,to0)

-------------------------------------------------------------------------------
update_acc_strinfos_aux :
         StreamInfo* × XP × Position × Present × StreamInfo*
         → StreamInfoStar⊥
-------------------------------------------------------------------------------

update_acc_strinfos_aux(strinfos,xp,pos,present,acc) =
   if strinfos = ε then
      acc
   else
      let (s,rest) = strinfos :
      let (past,future) = s:
      case future of
        NIL      : let newacc = acc||s
        ?S       : let newacc = acc||s
        External : let newacc = acc||s
        isStream(str) :
           let s1 = str(present[xp↦pos])
           case s1 of
             ⊥S : in⊥S(⊥S)
             isStreamInfo(strinfo) :
               let (past1,future1) = strinfo :
               let (messages, _) = past :
               let (messages1,l) = past1 :
               let new_messages = messages||messages1 :
               let new_past = (new_messages,l) :
               let new_strinfo = (new_past,future1) :
               let newacc = acc||new_strinfo :
               update_acc_strinfos_aux(rest,xp,pos,newacc)

-------------------------------------------------------------------------------
mergeall : StreamInfo* × StreamPast → StreamPast
-------------------------------------------------------------------------------

mergeall(streaminfos, acc) =
  if streaminfos = ε then
     acc
  else
     let (strinfo,rest) = streaminfos :
     let (messages,l) = strinfo :
     let (messages_acc,_) = acc :
     let newmessages = messages_acc||messages :
     let newacc = (newmessages,l) :
     mergeall(rest,newacc)
```

```
Tr⟦construct XP in SID with RAN CONSTR : TV⟧ =
  let f : XP × SID
            × Present → Range ∪ {?P,⊥P}
            × Step[ConstrDom]
            × Step[Value⊥?]
           → Stream
  f[xp,sid,r,c,tv](present) =
       let r_loc = r(present) :
       let past = (ε,0) :
       if r_loc = ⊥P then
          in⊥S(⊥S)
       else if r_loc = ?P then
         let info = (past,?S) :
         inStreamInfo(info)
       else
         let (p1,pp1,xp1,pp2,p2) = r_loc :
         if xp1 ≠ xp then
           in⊥S(⊥S)
         else case p1 of
            isStep[Position⊥?](pos1) :
               let r1 = (pos1,pp1,xp1,pp2,p2) :
                inStreamInfo((past,f[xp,sid,r1,c,tv]))
            isPosition⊥?(pos1)       :
                if pos1 = ⊥P then
                   in⊥S(⊥S)
                else if pos1 = ?P then
                   let info = (past,?S) :
                   inStreamInfo(info)
                else
                   case p2 of
                     isStep[Position⊥?](pos2) :
                        let r1 = (pos1,pp1,xp1,pp2,pos2) :
                        inStreamInfo(past,f[xp,sid,r1,c,tv]))
                     isPosition⊥?(pos2)       :
                       if pos2 = ⊥P then
                          in⊥S(⊥S)
                       else if pos2 = ?P then
                          let info = (past,?S) :
                          inStreamInfo(info)
                       else
                          let streampast = get_stream_past(sid,present) :
                          if streampast = ⊥S then
                             in⊥S(⊥S)
                          else
                             let r1 = (pos1,pp1,xp1,pp2,pos2) :
                             let max_pos = maximal_position(streampast) :
                             let (from,to) =
                                precise_range_boundaries(pos1,pp1,pp2,pos2) :
                              if from > max_pos then
                                 let info = (past,f[xp,sid,r1,c,tv]) :
                                 inStreamInfo(info)
                              else
                                 if max_pos < to then
                                    let upper = max_pos :
                                    let flag = "incomplete" :
                                 else
                                    let upper = to :
                                    let flag = "complete" :
                                 let acc = (ε,0),
                                 let strpast =
                                      construct_past_from_value[xp,sid,c,tv]
                                            (from,to,acc) :
                                 case strpast of
                                     isStep[StreamPast⊥](step) :
                                        let info = (past,f[xp,sid,r1,c,tv]) :
                                        inStreamInfo(info)
                                     isStreamPast⊥(pst) :
                                        if pst = ⊥S then
                                           in⊥S(⊥S)
                                        else
                                           let info =
                                               if flag = "complete" then
                                                  (pst,NIL) :
```

```
                                                  else
                                                     let r2 =
                                                           (upper,pp1,xp,pp2,to) :
                                                     (pst,f[xp,sid,r2,c,tv]) :
                                              inStreamInfo(info)
: f[XP,SID,Tr⟦RAN⟧,Tr⟦CONSTR⟧,Tr⟦TV⟧]

-------------------------------------------------------------------------------
construct_past_from_value :
                  XP × SID × Step[ConstrDom] × Step[Value⊥?]
                  → Position × Position × StreamPast
                  → Step[StreamPast⊥]

StreamPast⊥ = StreamPastU{⊥S}
-------------------------------------------------------------------------------

construct_past_from_value[xp0,sid0,contraint0,tv0](from0,to0,acc0) =
 f : XP × SID × Step[ConstrDom] × Step[Value⊥?]
     → Position × Position × StreamPast
     → Step[StreamPast⊥]
 f[xp,sid,contraint,tv](from,to,acc)(present) =
   if from > to then
      inAnswer[StreamPast⊥](acc)
   else
      let constr = contraint(present[xp↦from]) :
      case constr of
        isStep(ConstrDom)(c0) :
          inAnswer[StreamPast⊥](f[xp,sid,c0,tv](from,to,acc))
        isConstrDom(c)       :
          case c of
            ⊥E                 :  inAnswer[StreamPast⊥](⊥S)
            isPresentMK(pres_mk) :
               let pres_mk = (pres,mk)  :
               if mk = ⊥F then
                  inAnswer[StreamPast⊥](⊥S)
               else if mk ≠ true then
                  let from1 = from + 1 :
                  inAnswer[StreamPast⊥](f[xp,sid,c,tv](from1,to,acc))
               else
                  let v1 = tv(pres[xp↦from]) :
                  case v1 of
                    isStep(Value⊥?)(v) :
                       inAnswer[StreamPast⊥](f[xp,sid,c,tv](from,to,acc))
                    isValue⊥?(v)       :
                       if v = ⊥v then
                          inAnswer[StreamPast⊥](⊥S)
                       else
                          let current_time = (present↓1)↓1 :
                          let (messages,_) = acc :
                          let newmessages = messages||(current_time,v) :
                          let newacc = (newmessages,1) :
                          let from1 = from + 1 :
                          inAnswer[StreamPast⊥](f[xp,sid,c,tv](from1,to,newacc))
: f[[xp0,sid0,contraint0,tv0](from0,to0,acc0)

-------------------------------------------------------------------------------
Tr⟦BIND⟧:  Binder // Present → (Present × Binder) ∪ {⊥E}
BIND ::= formula XF = F  | position XP = TP
         | value XV = TV | stream XS = SID
-------------------------------------------------------------------------------

Tr⟦formula XF = F⟧ =
  let f : XF × Step[MK] → Binder,
     f[xf,f0](present) =
       let b0 = f0(present) :
       case b0 of
         isMK(b) :
             if b = ⊥F then
                ⊥E
             else
                let f1 : Step[MK]
                    f1(present) = inAnswer(b0) :
                    (present[XF↦b0],f[xf,f1])
         isStep[MK](s) :
```

```
              (present[XF↦b0], f[xf,b0])
    : f[XF, Tr⟦F⟧]

  Tr⟦position XP = TP⟧ =
    let f : XP × Step[Position⊥?] → Binder,
       f[xp,tp0](present) =
         let p0 = tp0(present) :
         case p0 of
           isPosition⊥?(p) :
               if p = ⊥P then
                  ⊥E
               else
                  let f1 : Step[Position⊥?]
                      f1(present) = inAnswer(p) :
                      (present[XP↦p],f[xf,f1])
           isStep[Position⊥?](tp1) :
               (present[XF↦p0], f[xf,p0])
    : f[XP, Tr⟦TP⟧]

  Tr⟦value XV = TV⟧ =
    let f : XV × Step[Value⊥?] → Binder
       f[xv,tv0](present) =
         let v0 = tv0(present) :
         case v0 of
           isValue?⊥(v) :
               if v = ⊥V then
                  ⊥E
               else
                  let f1 : Step[Value⊥?]
                      f1(present) = inAnswer(v) :
                      (present[xv↦v],f[xv,f1])
           isStep[Value⊥?](tv1) :
               (present[XF↦v0], f[xf,v0])
    : f[XV, Tr⟦TV⟧]

  Tr⟦stream SID = TS⟧ =
    let f : SID × Stream → Binder,
      f[sid,stream](present) =
         let s0 = stream(present) :
         case s0 of
            ⊥S : ⊥E
            isStreamInfo(strinfo) :
               let strinfo = (streampast,streamfuture) :
               case streamfuture of
                  isNIL :
                     let s1 : Stream
                         s1(present) = inStreamInfo(strinfo) :
                     inBinder(present[sid↦s0],f[sid,s1])
                  is?S :
                     let s1 : Stream
                         s1(present) = inStreamInfo(strinfo) :
                     inBinder(present[sid↦s0],f[sid,s1])
                  isExternal :
                     let s1 : Stream
                         s1(present) = inStreamInfo(strinfo) :
                     inBinder(present[sid↦s0],f[sid,s1])
                  isStream(st) :
                     inBinder(present[sid↦s0],f[sid,st])
   :  f[SID,Tr⟦TS⟧]

  ------------------------------------------------------------------------------
  Tr⟦CONSTR⟧: Step[ConstrDom]  // ConstrDom = ⊥E + (Present × MK)
  CONSTR  ::= ε                          Empty
            | satisfying F CONSTR
            | BIND CONSTR
  ------------------------------------------------------------------------------

  Tr⟦ε⟧(present) =
    let pres_tr = (present,true) :
    inAnswer[ConstrDom](pres_tr)

  Tr⟦satisfying F CONSTR⟧ =
    let f : Step[MK] × Step[ConstrDom] → Step[ConstrDom]
```

```
      f[f0,c0](present) =
         let b0 = f0(present) :
         case b0 of
            isMK(b) :
              if b ≠ true then
                let ans = (present,b) :
                inAnswer[ConstrDom](ans)
              else
                let ans = c0(present) :
                inAnswer[ConstrDom](ans)
            isStep[MK](f1) :
               inAnswer[ConstrDom](f[f1,c0])
 : f[Tr⟦F⟧,Tr⟦CONSTR⟧]

Tr⟦BIND CONSTR⟧(present) =
   let f : Binder  × Step[ConstrDom] → Step[ConstrDom]
     f[bind0,c0](present) =
        let b0 = bind0(present) :
         if  e0 = ⊥E then
             ⊥E
         else
            let b0 = (present1,binder1) :
            let ans = c0(present1) :
            inAnswer[ConstrDom](ans)
 : f[Tr⟦BIND⟧, Tr⟦CONSTR⟧]

-------------------------------------------------------------------------------
Tr⟦RAN⟧: Present → Range ∪ {?P,⊥P}
RAN ::= TP1 PP1 XP PP2 TP2 | TP PP XP
-------------------------------------------------------------------------------

// Type checker should guarantee that TP1 and TP2 are positions in the
// same stream XP refers to.

Tr⟦TP1 PP1 XP PP2 TP2⟧ =
  let r : Range → Present → Range ∪ {?P,⊥P}
    r(tp1,pp1,xp,pp2,tp2)(present) =
      let p1 = tp1(present) :
      case p1 of
        is?P  : ?P
        is⊥P : ⊥P
        isPosition(pos1) :
           let p2 = tp2(present) :
           case p2 of
              is?P  : ?P
              is⊥P : ⊥P
              isPosition(pos2)                : (pos1,PP1,xp,PP2,pos2)
              isStep[Position⊥? ∪ {∞}](s2)    : (pos1,PP1,xp,PP2,s2)
        isStep[Position](s1) :
           let p2 = tp2(present) :
           case p2 of
              is?P  : ?P
              is⊥P : ⊥P
              isPosition(p2)                  : (s1,PP1,xp,PP2,pos2)
              isStep[Position⊥? ∪ {∞}](s2)    : (s1,PP1,xp,PP2,s2)

  : r(Tr⟦TP1⟧,PP1,XP,PP2,Tr⟦TP2⟧)

Tr⟦TP PP XP⟧  =
  let r : Range → Present → Range ∪ {?P,⊥P}
    r(tp1,pp1,xp,pp2,tp2) =
      let p = tp1(present) :
      case p of
         is?P                  :  ?P
         is⊥P                  :  ⊥P
         isPosition(pos)       :  (pos,pp1,xp,pp2,tp2)
         isStep[Position](s)   :  (s,pp1,xp,pp2,tp2)
  : r(Tr⟦TP⟧,PP,XP,"<",∞)

-------------------------------------------------------------------------------
Tr⟦PV⟧ : Present → VPSⁿ →  MK
-------------------------------------------------------------------------------
```

```
Tr⟦PV⟧(message,stream_id,envs,envf,envp,envv,envfunpr) = envfunpr(PV)

-------------------------------------------------------------------------------
Tr⟦FV⟧ : Present → VPSⁿ → Value⊥?
-------------------------------------------------------------------------------

Tr⟦FV⟧(message,stream_id,envs,envf,envp,envv,envfunpr) = envfunpr(FV)

-------------------------------------------------------------------------------
Tr⟦FS⟧ : Present → VPSⁿ → Stream
-------------------------------------------------------------------------------

Tr⟦FS⟧(message,stream_id,envs,envf,envp,envv,envfunpr) = envfunpr(FS)
```