Formalising the Optimised Link State Routing Protocol

Ryan Barry
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
ryan.barry@unsw.edu.au

Rob van Glabbeek
Data61, CSIRO, Sydney, Australia
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
rvg@cs.stanford.edu

Peter Höfner
Research School of Computer Science
The Australian National University, Australia
Data61, CSIRO, Sydney, Australia
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
Peter.Hoefner@anu.edu.au

Routing protocol specifications are traditionally written in plain English. Often this yields ambiguities, inaccuracies or even contradictions. Formal methods techniques, such as process algebras, avoid these problems, thus leading to more precise and verifiable descriptions of protocols. In this paper we use the timed process algebra T-AWN for modelling the Optimised Link State Routing protocol (OLSR) version 2.

1 Introduction

Wireless Mesh Networks (WMNs) are a promising technology, having seen recent successes in the area of wireless communication. These multi-hop networks are designed to operate in a decentralised manner, with the responsibility of route discovery and packet forwarding being placed upon the nodes comprising a network. This necessitates the exchange of control messages in order to determine routes to available destinations.

A subset of WMNs known as Mobile Ad hoc Networks (MANETs) have received considerable attention for their use in vehicular communication[1], disaster relief and other emerging fields. Nodes in a MANET are distinguished by their high mobility when compared to other types of mesh networks. As a consequence of link breakages and fluctuations in signal quality, routes through these networks are subject to frequent changes. To further complicate matters, MANETs often require restrictions on bandwidth and power consumption due to the limitations of physical devices.

To address the challenges associated with ad hoc routing, a number of protocols specifically tailored to MANETs have been proposed in the literature. Of these protocols, recent research and development efforts by the Internet Engineering Task Force (IETF) have primarily targeted the Optimised Link State Routing protocol (OLSR) [9]. Originally specified in 2003, the protocol received numerous design modifications over the next decade which culminated in OLSR version 2 (OLSRv2) [8].

Despite its status as an IETF proposed standard, analyses of OLSRv2 have thus far been limited to simulations and testbed experiments. These techniques, although instrumental to the development of MANET protocols, cannot guarantee certain desirable properties, or the absence of certain undesirable

[1] Here, MANETs are often called Vehicular Ad hoc Networks (VANETs).

A. Fehnker and H. Garavel (Eds.):
Models for Formal Analysis of Real Systems (MARS 2020)
EPTCS 316, 2020, pp. 40–71, doi:10.4204/EPTCS.316.3

properties, in a system. Given that MANETs are increasingly deployed in safety-critical applications, stronger correctness guarantees ought to be made about the protocol.

Due to the ambiguous nature of natural languages, protocol specifications are often filled with ambiguities and contradictions that give rise to conflicting interpretations. To avoid such issues, we model OLSRv2 in a formalism known as the Timed Algebra for Wireless Networks (T-AWN) [6, 5].

T-AWN, and its untimed fragment AWN [12], have been used previously to formalise the routing protocol AODV [20]—see [15, 5]. This turned out to be a solid basis for analysing the protocol by means of model checking [11]. Additionally, based on this formalisation, crucial correctness properties of the protocol, including loop freedom, route correctness and route discovery, have been proven [15, 4, 5] or disproven [16, 13, 5], manually and with the support of interactive proof assistants. We expect the present formalisation of OLSRv2 to be equally useful for establishing correctness properties.

2 The Optimised Link State Routing Protocol

The Optimised Link State Routing protocol version 2 (OLSRv2) [8] is a link-state protocol tailored specifically to MANETs. In contrast to its 2003 counterpart, OLSRv1 [9], OLSRv2's offering of improved security, flexibility and scalability has solidified its status as the sole IETF "proposed standard" among MANET routing protocols.

Before describing the protocol's operation, we clarify some basic terminology.

Symmetric path: A sequence of nodes in a graph $ip_0 \ldots ip_n$ is a symmetric path iff for each pair of nodes (ip_i, ip_{i+1}) there exists an edge from ip_i to ip_{i+1} and vice-versa.

Symmetric n-hop neighbour: A node ip is a symmetric n-hop neighbour of another node ip' iff $ip \neq ip'$ and there exists a symmetric path of exactly n edges between ip and ip'. (So, a symmetric n-hop neighbour is also a symmetric $n+2$-hop neighbour.)

OLSRv2 is based on traditional link-state routing, and so inherits its basic characteristics. Each node maintains a graph representing the network topology, with each edge representing a pair of symmetric 1-hop neighbours. Optimal routes to all reachable destinations are then determined by applying some shortest path algorithm to this graph. Rather than waiting for data packets to arrive, the topology graph is proactively assembled by exchanging link-state information with neighbouring routers in the network, through broadcasts. These advertisements are scheduled periodically by each node using local timers.

Motivated by the strict limits on power and bandwidth consumption in MANETs, the designers of OLSR pioneered a number of optimisations to control traffic generation. The most significant optimisations come in the form of flooding reduction, where the number of broadcasts is reduced, and topology reduction, where the number of advertised links is minimised. Each router designates a subset of its symmetric 1-hop neighbours with one or both of these tasks in a process known as *multipoint relay* (MPR) selection.

MPR: A router (node) ip is an MPR of another router ip' iff ip has been designated the task of flooding reduction or topology reduction on behalf of ip'.

MPR selector: A router ip' is an MPR selector of another router ip iff ip is an MPR of ip'.

Routers in OLSRv2 maintain two sets of MPRs for flooding reduction and topology reduction, respectively. Flooding MPRs are responsible for forwarding link advertisements received from their flooding MPR selectors. To ensure that all routers receive link advertisements, each router must guarantee that all of its symmetric 2-hop neighbours are symmetric 1-hop neighbours of a flooding MPR, or symmetric 1-hop neighbours themselves. Figure 1 demonstrates this flooding reduction in practice. Just

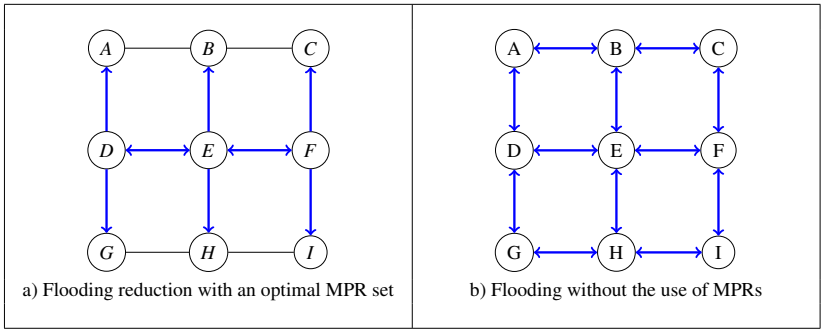


Figure 1: Flooding reduction

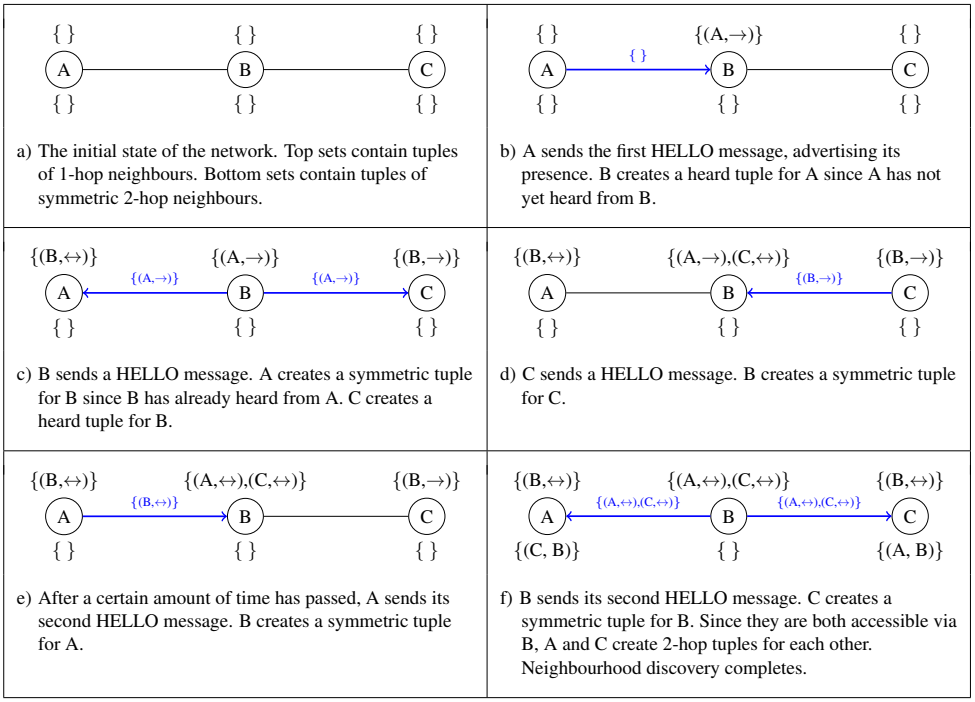
three broadcasts are needed to disseminate E's link advertisements when using optimal flooding MPR sets, whereas all nine nodes must perform a broadcast when flooding reduction is disabled. By contrast, routing MPRs are responsible for advertising links between themselves and their routing MPR selectors. Routing MPRs are chosen such that all symmetric 2-hop neighbours are accessible via a routing MPR in a minimal distance 1-hop or 2-hop route. In theory, the links advertised by routing MPRs are sufficient for all routers to construct shortest paths through the network.

Before links between routers can be advertised, each router must identify all of its symmetric 1-hop and 2-hop neighbours. The Neighbourhood Discovery Protocol (NHDP) formalised in RFC 6130 [7] is incorporated into and extended by OLSRv2 for this purpose. At a specific interval, a router will broadcast a HELLO message containing the addresses and statuses of its 1-hop neighbours. On the receiving end, the sending router is assigned a status of either symmetric, indicating a bidirectional link, or heard, indicating a unidirectional link. This status is determined by the receiver's inclusion in the HELLO message. Once symmetric links between routers are established, symmetric 2-hop neighbours can be inferred from the contents of current and future HELLO messages. All HELLO messages contain a validity time which determines when this information must be discarded, so care should be taken to avoid premature timeouts while links still exist. A simplified HELLO message exchange is detailed in Figure 2.

OLSRv2 extends NHDP with the inclusion of link metrics and MPR sets in its HELLO messages. Link metrics are assessed at the receiving end, and must therefore be propagated backwards to the sending router. Meanwhile, the flooding and routing MPR sets included in these messages indicate to the receiving routers whether or not they should engage in link advertisement on behalf of the sender.

After a node has identified its routing MPR selectors, it advertises all links between itself and these routers. The dissemination of this link-state information, when used in tandem with links discovered by NHDP, allows routers to establish shortest paths to all reachable destinations. The messages used to propagate advertised links are referred to as Topology Control (TC) messages. These messages are generated periodically and flooded through the network so that all reachable destinations may receive them.

TC messages contain a set of links to advertised routers, in addition to an advertising neighbour sequence number indicating how recent the message is. Unlike HELLO messages, which are processed but never forwarded, TC messages are flooded through the network by the flooding MPRs of each broadcasting router. When a router receives a TC message, it updates its topology graph with the advertised links provided that the advertising neighbour sequence number included in the message does not indicate out-of-date information. The message is then forwarded if it was received from a flooding MPR selector and was not forwarded in the past. Like HELLO messages, TC messages are prone to expiration if not received frequently.



a) The initial state of the network. Top sets contain tuples of 1-hop neighbours. Bottom sets contain tuples of symmetric 2-hop neighbours.

b) A sends the first HELLO message, advertising its presence. B creates a heard tuple for A since A has not yet heard from B.

c) B sends a HELLO message. A creates a symmetric tuple for B since B has already heard from A. C creates a heard tuple for B.

d) C sends a HELLO message. B creates a symmetric tuple for C.

e) After a certain amount of time has passed, A sends its second HELLO message. B creates a symmetric tuple for A.

f) B sends its second HELLO message. C creates a symmetric tuple for B. Since they are both accessible via B, A and C create 2-hop tuples for each other. Neighbourhood discovery completes.

Figure 2: A simple Hello message exchange

3 The Specification Language T-AWN

One of the standard tools for describing interactions, communications and synchronisations between a collection of agents, processes or network nodes is provided by process algebras. Process algebras are a family of approaches to modelling concurrent systems, as well as formally analysing said systems through algebraic laws. We choose to model OLSRv2 using T-AWN [6, 5], a *timed* process algebra designed for wireless networks in general and routing protocols in particular.

The reason for choosing T-AWN is two-fold: on the one hand, it is tailored to wireless protocols and therefore offers primitives such as **broadcast**; on the other hand, it defines the protocol in pseudo-code that is easily readable by any network or software researcher/engineer. The language itself is implementation independent.

The timed process algebra T-AWN is based on the (untimed) process algebra AWN (Algebra of Wireless Networks) [12, 13]. (T-)AWN's key operators are *conditional unicast*—allowing error handling in response to failed communications while abstracting from link layer implementations of the communication handling—and *local broadcast*—allowing a node to send messages to all its immediate neighbours as implemented by the physical and data link layer, i.e. to all neighbours within transmission range.

Every process algebra such as (T-)AWN is equipped with an operational semantics [12, 6]: once a model has been given, its (timed) behaviour is governed by the transitions allowed by the algebra's semantics. In this paper we abstain from a formal definition of the operational semantics. Instead, we employ a correspondence between the transitions of (T-)AWN processes and the execution of *actions*—subexpressions as occur in Entries 3–10 of Table 1—identified by line numbers in protocol specifications in (T-)AWN.

We use an underlying data structure (sketched in Section 4 and described in detail in Section A.1)

Table 1: process expressions [15]

$X(exp_1,\ldots,exp_n)$	process name with arguments
$P+Q$	choice between processes P and Q
$[\varphi]P$	conditional process (if-statement)
$[\![\texttt{var} := exp]\!]P$	assignment followed by process P
broadcast$(ms).P$	broadcast of message *ms* followed by P
groupcast$(dests,ms).P$	iterative unicast or multicast to all destinations *dests*
unicast$(dest,ms).P \blacktriangleright Q$	unicast *ms* to *dest*; if successful proceed with P; otherwise with Q
send$(ms).P$	synchronously transmit *ms* to parallel process on same node
deliver$(data).P$	deliver data to application layer
receive$(\texttt{msg}).P$	receive a message
ξ, P	process with valuation
$P \langle\langle Q$	parallel processes on the same node
$a:P:R$	node a running process P with range R
$N \| M$	parallel composition of nodes
$[N]$	encapsulation

with several types, variables ranging over these types, operators and predicates. First order predicate logic yields terms (or *data expressions*) and formulas to denote data values and statements about them. The (T-)AWN data structure must contain the types `DATA`, `MSG`, `IP` and $\mathscr{P}(\texttt{IP})$ of *application data*, *messages*, *IP addresses*—or any other node identifiers—and *sets of IP addresses*, respectively; in the case of T-AWN, the data structure also features the type `TIME` of time values. The rest of the data structure is customisable for any application of (T-)AWN.

In the process algebra at hand an entire network is modelled as an encapsulated parallel composition of network nodes; several processes can be executed on the same node. Nodes can only communicate with their direct neighbours, i.e. with nodes that are currently within transmission range. There are three different ways for nodes to perform internode communication: broadcast, unicast, or an iterative unicast/multicast (called *groupcast* in (T-)AWN).

The *process expressions* are given in Table 1. A process name X comes with a *defining equation*

$$X(\texttt{var}_1,\ldots,\texttt{var}_n) \stackrel{def}{=} P,$$

where P is a process expression, and the $\texttt{var}_i$ are data variables maintained by process X. Furthermore, φ is a condition, $\texttt{var} := exp$ an assignment of a data expression *exp* to a variable `var` of the same type, *dest*, *dests*, *data* and *ms* data expressions of types `IP`, $\mathscr{P}(\texttt{IP})$, `DATA` and `MSG`, respectively, and `msg` a data variable of type `MSG`.

Given a valuation of the data variables by concrete data values, the process $[\varphi]P$ acts as P if φ evaluates to `true`, and deadlocks if φ evaluates to `false`.[2] In case φ contains free variables that are not yet interpreted as data values, values are assigned to these variables in any way that satisfies φ, if possible. The process $[\![\texttt{var} := exp]\!]P$ acts as P, but under an updated valuation of the data variables. The process $P+Q$ acts either as P or as Q, depending on which of the two processes is able to act at all. In case both are able to act, the choice is non-deterministic. The process **broadcast**$(ms).P$ broadcasts (the data value bound to the expression) *ms* to all nodes within transmission range, and subsequently acts as P, whereas the process **unicast**$(dest,ms).P \blacktriangleright Q$ tries to unicast the message *ms* to the destination *dest*; if

[2] As operators we also allow *partial* functions with the convention that any atomic formula containing an undefined subterm evaluates to `false`.

successful it continues to act as P and otherwise as Q. The unicast is unsuccessful if the destination *dest* is out of transmission range of the node *ip* performing the unicast. The latter models an abstraction of an acknowledgment-of-receipt mechanism that is typical for unicast communication but absent in broadcast communication, as implemented by the link layer of wireless standards such as IEEE 802.11 [17]. The process **groupcast**$(dests,ms).P$ tries to transmit *ms* to all destinations *dests*, and proceeds as P regardless of whether any of the transmissions is successful. The process **send**$(ms).P$ synchronously transmits a message to another process running on the same network node; this action can occur only when the other process is ready to receive the message. The process **receive**$(\texttt{msg}).P$ receives a message m (of type `MSG`); the value m is then bound to the variable `msg` and the process proceeds as P. The received message stems either from another node, from another process running on the same node or from the application layer process on the local node. The latter is used to model the injection of data to the network, using the process **receive**$(\texttt{newpkt}(data,dip))$, where the function `newpkt` generates a message containing the application layer *data* and the intended destination address *dip*. Data is delivered back to the application layer by **deliver**$(data)$.

A state of a network node is modelled as a *valuated process* given as a pair (ξ,P) of a process expression P built from the above syntax, together with a (partial) *valuation* function ξ that specifies values of the data variables maintained by P. Finally, $P \langle\langle Q$ denotes a parallel composition of processes P and Q, with information piped from right to left; in our application Q will be a message queue.

In the full process algebras AWN [12] and T-AWN [6], *node expressions* $a:P:R$ are given by process expressions P, annotated with an *address* a and a set of nodes R that are within *transmission range* of a. A partial network is then modelled as a parallel composition of node expressions, using the operator $\|$, and a complete network is obtained by placing this composition in the scope of an encapsulation operator $[\_]$. The main purpose of the encapsulation operator is to prevent the receipt of messages that have never been sent by other nodes in the network—with the exception of messages $\texttt{newpkt}(data,dip)$ stemming from the application layer of a node.

When designing or formalising a protocol in (T-)AWN, an engineer should not be bothered with timing aspects, except for functions and procedures that schedule tasks depending on the current time. Because of this, the only difference between the syntax of AWN and the one of T-AWN is that the latter is equipped with a local timer `now`, which is of type `TIME`.

T-AWN assumes a discrete model of time, where each sequential process maintains the local variable `now` holding its local clock value—an integer. Only one clock for each sequential process is employed. All (sequential) processes in a network synchronise in taking time steps, and at each time step all local clocks are incremented by one time unit. For the rest, the variable `now` behaves as any other variable maintained by a process: its value can be read when evaluating guards, thereby making progress time-dependent, and any value can be assigned to it, thereby resetting the local clock.

Before describing our T-AWN-specification of OLSRv2, we want to point out two fundamental assumptions of T-AWN. (i) The underlying formal semantics of (T-)AWN is that any broadcast message *is* received by all nodes within transmission range. This abstraction allows us to interpret a failure of route discovery of a protocol (or any other property) as an imperfection in the protocol, rather than as a result of a chosen formalism not ensuring guaranteed receipt. The same holds for groupcast and unicast messages in case the destinations are within range. (ii) Only internode communication, i.e. transferring a message from one node in the network to another, takes time. This is justified as in wireless networks sending a packet takes multiple microseconds; compared to these "slow" actions, time spent for internal (intranode) computations, such as variable assignments, is negligible.

4 Modelling OLSRv2 in T-AWN

In this section, we present parts of our T-AWN model of OLSRv2. The model itself consists of five main processes implementing the OLSRv2 specification and a queue process to receive packets from other routers:

- The process `OLSR` constitutes the main protocol loop. It is responsible for receiving packets from the input queue and processing these packets according to their type. It it also responsible for periodically generating new HELLO and TC messages.
- The `UPDATE_INFO` process ensures that the protocol's information bases remain consistent with certain constraints.
- The `PROCESS_HELLO` and `PROCESS_TC` processes are responsible for recording information obtained through HELLO messages and TC messages in the relevant information bases.
- The `FORWARD_TC` process forwards TC messages received from the router's flooding MPR selectors, subject to a few side conditions.
- The `QUEUE` process receives packets from other routers in the network and delivers them to the `OLSR` process.

Due to a lack of space we only present the process `OLSR`, including the necessary data structure; the full specification of OLSRv2, including the full data structure and a detailed description can be found in Appendix A.

4.1 Data Structure

We now describe the data structure needed and functions used in modelling the process `OLSR`. In the remainder, we use the notation $\mathtt{x}_{1..n}$ as shorthand for the tuple $(\mathtt{x}_1,\mathtt{x}_2,\ldots,\mathtt{x}_n)$.

We take the type `TIME` isomorphic to $\mathbb{Z}\cup\{\infty,-\infty\}$. Additionally, we use the type synonym $\mathtt{SQN}=\mathbb{Z}$ for sequence numbers. Moreover, we assume the existence of a polymorphic data type for lists:

```
data [a] = [] | a:[a]
```

with the standard functions `concat` and `append` to concatenate two lists and to append a single element to a list, respectively.

The `MESSAGE` data type encompasses both HELLO messages and TC messages, the two communication primitives used by the protocol.

```
data MESSAGE =
  HELLO IP TIME 𝒫(IP × STATUS) 𝒫(IP × MPR) 𝒫(IP × METRIC) 𝒫(IP × METRIC)
| TC IP IP TIME SQN SQN 𝒫(IP × METRIC)
```

In our implementation, HELLO messages contain six elements: (*i*) an originator address, of the T-AWN basic type `IP`; (*ii*) a validity time detailing how long the message's contents should be considered valid for once the message is processed; (*iii*) a set of tuples assigning link statuses to each of the originating router's neighbours; (*iv*) a set of tuples indicating which neighbours the originating router has selected as flooding and routing MPRs; (*v*) a set of tuples representing the incoming metrics to the originating router from each of its neighbouring routers, and (*vi*) a set of tuples representing the outgoing metrics from the originating router to its neighbours.

TC messages also contain six elements: (*i*) an originator address; (*ii*) a sender address for the router that last broadcast the message; (*iii*) a validity time; (*iv*) a sequence number identifying the message;

(*v*) an advertising neighbour sequence number indicating how fresh the advertised information in the message is, and (*vi*) a set of advertised links between the originating router and its routing MPR selectors.

Routers combine these messages into packets when broadcasting them. We do not care about the packet header, and so take the `MSG` type of T-AWN to represent a list of `MESSAGE` values. We determine whether a message is a HELLO message or a TC message by invoking the `isHELLO` and `isTC` functions.

Routers must be able to generate new messages periodically. To this end, we use the functions `newHELLO` and `newTC`. Both make use of a set `ls` containing link tuples (elements of a data type `L`) that contain information such as an originator address and a validity time; functions of the form `L_*` are used to distil information from link tuples. The precise definitions are given in Section A.1.

$$\begin{array}{l}
\texttt{newHELLO} :: \texttt{IP} \times \texttt{TIME} \times \mathscr{P}(\texttt{L}) \times \texttt{TIME} \to \texttt{MESSAGE} \\
\texttt{newHELLO(ip,vtime,ls,now)} \equiv \\
\quad \textbf{let} \quad \texttt{statuses} = \{(\texttt{L_oip}(\texttt{lt}), \texttt{L_status}(\texttt{lt},\texttt{now})) \mid \texttt{lt} \in \texttt{ls}\} \\
\quad \textbf{and} \ \texttt{mprs} = \\
\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{FLOODING}) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_fmpr}(\texttt{lt}) \wedge \neg \texttt{L_rmpr}(\texttt{lt})\} \cup \\
\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{ROUTING}) \mid \texttt{lt} \in \texttt{ls} \wedge \neg \texttt{L_fmpr}(\texttt{lt}) \wedge \texttt{L_rmpr}(\texttt{lt})\} \cup \\
\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{FLOOD_ROUTE}) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_fmpr}(\texttt{lt}) \wedge \texttt{L_rmpr}(\texttt{lt})\} \\
\quad \textbf{and} \ \texttt{in_metrics} = \\
\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{L_in_metric}(\texttt{lt})) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_status}(\texttt{lt},\texttt{now}) \neq \texttt{LOST}\} \\
\quad \textbf{and} \ \texttt{out_metrics} = \\
\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{L_out_metric}(\texttt{lt})) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}\} \\
\quad \textbf{in} \quad \texttt{HELLO ip vtime statuses mprs in_metrics out_metrics}
\end{array}$$

$$\begin{array}{l}
\texttt{newTC} :: \texttt{IP} \times \texttt{TIME} \times \texttt{SQN} \times \texttt{SQN} \times \mathscr{P}(\texttt{L}) \times \texttt{TIME} \to \texttt{MESSAGE} \\
\texttt{newTC(ip,vtime,sqn,ansn,ls,now)} \equiv \\
\quad \textbf{let} \ \texttt{dests} = \{(\texttt{L_oip}(\texttt{lt}), \texttt{L_out_metric}(\texttt{lt})) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_rmpr_selector}(\texttt{lt}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}\} \\
\quad \textbf{in} \quad \texttt{TC ip ip vtime sqn ansn dests}
\end{array}$$

Our protocol maintains its state in a list of variables. Among others these include

- `ls`, a link set maintaining information about 1-hop neighbours and their statuses
- `2hs`, the 2-hop set maintaining information about 2-hop neighbours
- `arrs`, a remote router set containing information about routers which have advertised links
- `rts`, the router topology set containing advertised links
- `rs`, a routing set containing shortest known routes
- `ps`, the processed set identifying processed TC messages
- `rxs`, the received set identifying TC messages received and considered for forwarding
- `pkt`, a list of messages requiring sending, such as those generated by the router or forwarded by it
- `hello_time`, the time when the next HELLO message must be added to `pkt`
- `tc_time`, the time when the next TC message must be added to `pkt`
- `send_time`, the time when `pkt` must be broadcast
- `mqueue`, the queue of to-be-processed messages
- `sqn`, the sequence number identifying a TC message
- `ansn`, the advertising neighbour sequence number included in TC messages to indicate how recent the advertised contents are
- `prev_ls`, the previous link set used to check for updates.

These variables are modified during the protocol's execution. For our specification, presented below, we use the shortcut σ for these variables:

$$\sigma \equiv \texttt{ls}, \texttt{2hs}, \texttt{arrs}, \texttt{rts}, \texttt{rs}, \texttt{ps}, \texttt{rxs}, \texttt{pkt}, \texttt{hello\_time}, \texttt{tc\_time}, \texttt{send\_time}, \texttt{mqueue}, \texttt{sqn}, \texttt{ansn}, \texttt{prev\_ls}$$

The protocol also maintains variables that are not changed by the protocol itself, including parameters set by a network administrator. These are

- `ip`, the (unique) address of the router
- `hp_maxjitter`, the maximum jitter time for HELLO messages
- `tp_maxjitter`, the maximum jitter time for TC messages
- `h_hold_time`, the validity time for generated HELLO messages
- `t_hold_time`, the validity time for generated TC messages
- `l_hold_time`, the length that lost links should be kept for
- `hello_interval`, the period between HELLO message transmissions
- `tc_interval`, the period between TC message transmissions,

and are abbreviated by Γ. All variables contained in σ and Γ are also summarised in Table 2.

Besides these variables, we also maintain a variable `queue` of type `[MSG]` in our input queue process and a variable `msg` of type `MESSAGE` when processing or forwarding a message.

In our model, we localise all relevant information base updates in the single process `UPDATE_INFO` (Process 2 in the appendix). We use a condition `Updated`, also defined in the appendix, to check whether this process needs to be called. `Updated` holds iff the updates of Process `UPDATE_INFO` would not modify the protocol's information bases. This is the case iff the information bases are currently consistent, implying for instance that links whose expiration time has elapsed have been purged from the system.

4.2 The Formal Model

We present the formal model of the main routine `OLSR` of OLSRv2 in detail; it is depicted in Process 1.

The main OLSR routine performs a number of different roles. The most basic of these is receiving a packet from the input queue, which occurs in the block on lines 1-3. Packets of type `MSG` are simply lists of HELLO and TC messages, so we concatenate received packets with an existing queue of to-be-processed messages. When the protocol is ready to process a HELLO or TC message, the choice on line 8 is taken, with `msg` and `msgs` assigned to the head and tail of `mqueue` respectively by the guard. Note that this guard contains an `Updated` conjunct, which asserts that the information bases have already been updated by the block on lines 5-6. Once `msg` is assigned to the element at the head of the queue, the ensuing assignment statement assigns `mqueue` to its tail `msgs`. The guards on lines 9 and 12 then ensure that `msg` is processed according to its type, be that a HELLO message or a TC message.

Aside from processing messages, a router must generate its own HELLO and TC messages periodically. The guard on line 15 asserts that a new HELLO message is ready to be added to `pkt`, a list which accumulates all messages generated during a time tick in order to circumvent broadcasting delays. The guard is true whenever the local clock `now` enters the jitter period before the message's preparation deadline `hello_time`. The condition $\texttt{now} \leq \texttt{hello\_time}$ is redundant, as one can prove that it is always met even when left out; it reminds us that we have not yet exceeded the deadline `hello_time` when line 16 is executed. A new HELLO message is generated by line 16 and appended to `pkt`. Then, `hello_time` is increased to `hello_interval` time units away from `now`, and `send_time` is set to $\texttt{now} + 1$ so that the packet will be broadcast during the next tick. Sending a TC message in the block starting at line 21 follows a similar process, except that a sequence number `sqn` is included in the message and subsequently incremented. Once all messages have been accumulated and the guard on line 28 becomes

Process 1: The main OLSR process

```
   OLSR(σ,Γ) =def
   /* Receive a packet (i.e. a list of messages) from the queue process */
1  receive(msgs).
2      [[mqueue := concat(mqueue,msgs)]]
3      OLSR(σ,Γ)
4  +
   /* Execute pending updates to relevant information bases */
5  [¬Updated]
6      UPDATE_INFO(σ,Γ)
7  +
   /* Process a received message */
8  [Updated ∧ send_time ≠ now ∧ mqueue = (msg : msgs)] [[mqueue := msgs]]
       /* Process a received HELLO message */
9      [isHELLO(msg)]
10         PROCESS_HELLO(σ,Γ,msg)
11     +
       /* Process a received TC message */
12     [isTC(msg)]
13         PROCESS_TC(σ,Γ,msg)
14 +
   /* Time to generate a HELLO message */
15 [Updated ∧ send_time ≠ now ∧ hello_time − hp_maxjitter ≤ now ≤ hello_time]
       /* Add the message to the current packet */
16     [[pkt := append(pkt,newHELLO(ip,h_hold_time,ls,now))]]
       /* Set relevant timers */
17     [[hello_time := now + hello_interval]]
18     [[send_time := now + 1]]
19     OLSR(σ,Γ)
20 +
   /* Time to generate a TC message */
21 [Updated ∧ send_time ≠ now ∧ tc_time − tp_maxjitter ≤ now ≤ tc_time]
       /* Add the message to the current packet */
22     [[pkt := append(pkt,newTC(ip,t_hold_time,sqn,ansn,ls,now))]]
       /* Increment the sequence number */
23     [[sqn := sqn + 1]]
       /* Set relevant timers */
24     [[tc_time := now + tc_interval]]
25     [[send_time := now + 1]]
26     OLSR(σ,Γ)
27 +
   /* Broadcast the accumulated packet */
28 [Updated ∧ send_time = now]
29     [[send_time := ∞]]
30     broadcast(pkt).
31     [[pkt := [ ]]]
32     OLSR(σ,Γ)
```

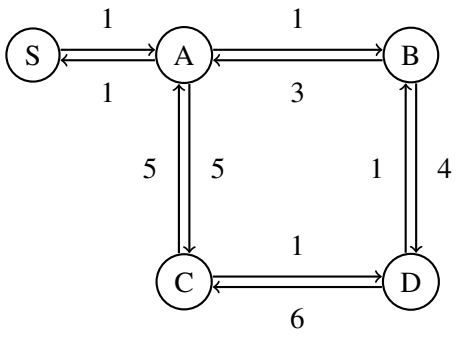


Figure 3: A counterexample to route optimality

true, the accumulated packet is broadcast, and both `pkt` and `send_time` are reset to indicate no pending messages.

5 Correcting the Specification

As currently specified, the OLSRv2 model does not guarantee that optimal routes will be established to all reachable destinations. This is in contrast to the intention of the RFC. The reason is that incorrect directional link metrics are being used in section 18.5 of RFC 7181 [8] during routing MPR selection. Rather than using the incoming metric between 1-hop and 2-hop neighbours, the outgoing metric from the 1-hop neighbour to the 2-hop neighbour is used instead:

> "For each element x in N1, define N2(x) as the set of elements y in N2 whose corresponding address is the N2_2hop_addr of an allowed 2-Hop Tuple that has N2_neighbor_iface_addr_list contained in N_neighbor_addr_list of the Neighbor Tuple corresponding to x. For all such x and y, define d2(x,y) := N2_out_metric of that 2-Hop Tuple."

From this definition, we can construct a straightforward counterexample. Consider the topology in Figure 3 and assume that each node is aware of the links to its 1-hop neighbours and 2-hop neighbours. To ensure that S has a shortest path to D, D must ensure that its sole 2-hop neighbour, A, has a shortest path to D. When selecting a routing MPR to advertise an appropriate link, D compares the cost of the paths from A via C and from A via B. However, it uses the outgoing metric when calculating the cost of a link from A to C and from A to B. The path via C, which has a cost of $5+1=6$, is "shorter" than the path via B, which has a cost of $3+4=7$. Therefore, D will require the path from A to take the "shorter" route via C. When D sends its next HELLO message, it tells C to advertise this link, which C subsequently includes in its TC messages. Eventually, S learns of the links from A to C, A to B and C to D via TC messages. However, it never learns of the link from B to D since D does not tell B to advertise on its behalf. Therefore, S will not be able to construct the true shortest path to D via B.

This error is straightforward to correct, simply by replacing `N2_out_metric` with `N2_in_metric`. The "bug" is already corrected in the model presented in the appendix.

Note that this bug does not affect the `OLSR.org` open source implementation of OLSR version 2. The OLSR daemon version 2 [21], included in the OLSR.org Network Framework, correctly uses the incoming metric.

6 Related Work, Discussion & Future Work

This paper provides a full and detailed model of the routing protocol OLSRv2, written in the process algebra T-AWN [6]. Thanks to the formal semantics of T-AWN, this model is completely unambiguous and forms a good basis for the verification of useful protocol properties. Moreover, it is considerably shorter than the 203 pages of English prose present in the RFCs for OLSRv2 [7] and its base NHDP [8]. As with all formal models, our model represents our interpretation of the English prose; a formal proof that a model or an implementation is compliant with a textual specification is impossible. However, the existence of a detailed, unambiguous model, such as ours, can be the base for a formal refinement proof that an implementation is compliant to that model. The fact that our model is written in "pseudo-code" would make such a proof easier compared to other modelling languages.

To the best of our knowledge, our model is by far the most detailed model of the Optimised Link State Routing protocol found in the literature—we believe that it is the very first model of version 2. There have only been a few other efforts to formally model OLSR.

Baras et al. [1] present a component-based methodology for modelling and designing wireless routing protocols, and illustrate it on OLSR. Although many crucial aspects of the protocol, such as MPR selection, are captured this way, the result cannot be seen as a full rendering of the core functionality of the protocol. Consequently, it could not be used as basis for the verification of properties like route optimality.

Steele and Andel [22] provide a model of OLSR for analysis with the model checker SPIN. The model abstracts, among others, from timing aspects, and thus cannot detect possible shortcomings of the protocol resulting, for example, from premature route expiry.

Kamali et al. [18] model OLSR in the input language of the model checker UPPAAL. To facilitate model checking, the model abstracts from large parts of the information bases of OLSR. To avoid the computation of shortest paths they update the routing table whenever a TC message is received. Although this reduces the state space drastically, it changes the protocol behaviour in such a way that optimal routes cannot be guaranteed. This problem is a shortcoming of the abstract model rather than the protocol itself.

In [19], Kamali and Petre model OLSR in the state-based formal method *Event-B*. This model abstracts from timing, but is otherwise largely consistent with the above UPPAAL model.

Currently, there is a limited amount of tool support available for (T-)AWN. The untimed fragment has been integrated into the interactive proof assistant Isabelle/HOL [3, 2]. Specifications formalised in (T-)AWN can also be a base for model checking: in [14] it is shown how to analyse AWN-specifications using *mCRL2*; a translation from (T-)AWN into UPPAAL is sketched in [11] and illustrated in [10] for the routing protocol OSPF.

The one feature our model of OLSRv2 abstracts from is that routers may have multiple communication interfaces. Adding support for multiple interfaces is a possible topic for future work; it might, however, necessitate extending the expressiveness of T-AWN.

Future work additionally involves employing the model provided here for the verification of requirements a routing protocol like OLSRv2 should satisfy: *route correctness*, saying that all routes found by the protocol actually exist, *route discovery*, saying that when a route between two nodes exists, a route will be found by the protocol, and *route optimality*, saying that the protocol finds the best routes possible.

References

[1] J.S. Baras, V. Tabatabaee, P. Purkayastha & K. Somasundaram (2009): *Component Based Performance Modelling of Wireless Routing Protocols*. In: *International Conference on Communications* (ICC'09), IEEE, pp. 1–6, doi:10.1109/ICC.2009.5198840.

[2] T. Bourke (2014): *Mechanization of the Algebra for Wireless Networks (AWN)*. *Archive of Formal Proofs*. `http://isa-afp.org/entries/AWN.html`, Formal proof development.

[3] T. Bourke, R. J. van Glabbeek & P. Höfner (2016): *Mechanizing a Process Algebra for Network Protocols*. *Journal of Automated Reasoning* 56(3), pp. 309–341, doi:10.1007/s10817-015-9358-9.

[4] T. Bourke, R.J. van Glabbeek & P. Höfner (2014): *A mechanized proof of loop freedom of the (untimed) AODV routing protocol*. In F. Cassez & J.-F. Raskin, editors: *Automated Technology for Verification and Analysis* (ATVA'14), LNCS 8837, Springer, pp. 47–63, doi:10.1007/978-3-319-11936-6_5. Available at `http://arxiv.org/abs/1505.05646`.

[5] E. Bres, R.J. van Glabbeek & P. Höfner (2016): *A Timed Process Algebra for Wireless Networks with an Application in Routing*. Technical Report 9145, NICTA. Available at `http://arxiv.org/abs/1606.03663`.

[6] E. Bres, R.J. van Glabbeek & P. Höfner (2016): *A Timed Process Algebra for Wireless Networks with an Application in Routing (Extended Abstract)*. In P. Thiemann, editor: *Programming Languages and Systems* (ESOP'16), LNCS 9632, Springer, pp. 95–122, doi:10.1007/978-3-662-49498-1_5.

[7] T.H. Clausen, C. Dearlove & J. Dean (2011): *Mobile Ad Hoc Network (MANET) Neighborhood Discovery Protocol (NHDP)*. RFC 6130 (Proposed Standard), doi:10.17487/RFC6130.

[8] T.H. Clausen, C. Dearlove, P. Jacquet & U. Herberg (2014): *The Optimized Link State Routing Protocol Version 2*. RFC 7181 (Proposed Standard), doi:10.17487/RFC7181.

[9] T.H. Clausen & P. Jacquet (2003): *Optimized Link State Routing Protocol (OLSR)*. RFC 3626 (Experimental), doi:10.17487/RFC3626.

[10] J. Drury, P. Höfner & W. Wang (2020): *Formal Models of the OSPF Routing Protocol*. In A. Fehnker & H. Garavel, editors: *Models for Formal Analysis of Real Systems (*MARS'20*)*, EPTCS 316, Open Publishing Association, pp. 72–120, doi:10.4204/EPTCS.316.4.

[11] A. Fehnker, R. J. van Glabbeek, P Höfner, A. K. McIver, M. Portmann & W. L. Tan (2012): *Automated Analysis of AODV using UPPAAL*. In C. Flanagan & B. König, editors: *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'12), LNCS 7214, Springer, pp. 173–187, doi:10.1007/978-3-642-28756-5_13.

[12] A. Fehnker, R.J. van Glabbeek, P. Höfner, A.K. McIver, M. Portmann & W.L. Tan (2012): *A Process Algebra for Wireless Mesh Networks*. In H. Seidl, editor: *Programming Languages and Systems* (ESOP '12), LNCS 7211, Springer, pp. 295–315, doi:10.1007/978-3-642-28869-2_15.

[13] A. Fehnker, R.J. van Glabbeek, P. Höfner, A.K. McIver, M. Portmann & W.L. Tan (2013): *A Process Algebra for Wireless Mesh Networks used for Modelling, Verifying and Analysing AODV*. Technical Report 5513, NICTA. Available at `http://arxiv.org/abs/1312.7645`.

[14] R. J. van Glabbeek, P. Höfner & D. van der Wal (2018): *Analysing AWN-Specifications Using mCRL2 (Extended Abstract)*. In C. A. Furia & K. Winter, editors: *Integrated Formal Methods* (iFM'18), LNCS 11023, Springer, pp. 398–418, doi:10.1007/978-3-319-98938-9_23.

[15] R.J. van Glabbeek, P. Höfner, M. Portmann & W.L. Tan (2016): *Modelling and Verifying the AODV Routing Protocol*. *Distributed Computing* 29(4), pp. 279–315, doi:10.1007/s00446-015-0262-7.

[16] R.J. van Glabbeek, P. Höfner, W.L. Tan & M. Portmann (2013): *Sequence Numbers Do Not Guarantee Loop Freedom —AODV Can Yield Routing Loops—*. In: *Modeling, Analysis and Simulation of Wireless and Mobile Systems* (MSWiM'13), ACM, pp. 91–100, doi:10.1145/2507924.2507943.

[17] ISO/IEC/IEEE 8802-11 (2018): *Information Technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Specific requirements — Part 11: Wireless LAN*

Medium Access Control (MAC) and Physical Layer (PHY) specifications. Available at `https://www.iso.org/standard/73367.html`.

[18] M. Kamali, P. Höfner, M. Kamali & L. Petre (2015): *Formal Analysis of Proactive, Distributed Routing*. In R. Calinescu & B. Rumpe, editors: *Software Engineering and Formal Methods* (SEFM'15), LNCS 9276, Springer, pp. 175–189, doi:10.1007/978-3-319-22969-0_13.

[19] M. Kamali & L. Petre (2016): *Modelling Link State Routing in Event-B*. In H. Wang & M. Mokhtari, editors: *International Conference on Engineering of Complex Computer Systems* (ICECCS'16), IEEE Computer Society, pp. 207–210, doi:10.1109/ICECCS.2016.035.

[20] C. E. Perkins, E. M. Belding-Royer & S. Das (2003): *Ad hoc On-Demand Distance Vector (AODV) Routing*. RFC 3561 (Experimental), Network Working Group. Available at `http://www.ietf.org/rfc/rfc3561.txt`.

[21] H. Rogge (2018): *OLSR.org Network Framework - olsrd v2 / DLEP*. `https://github.com/OLSR/OONF`.

[22] M.F. Steele & T.R. Andel (2012): *Modeling the optimized link-state routing protocol for verification*. In G.A. Wainer & P.J. Mosterman, editors: *Theory of Modeling and Simulation* (TMS/DEVS'12), 35, SCS/ACM. Available at `http://dl.acm.org/citation.cfm?id=2346651`.

A Full and Detailed Model of OLSRv2

In this appendix we provide a full specification of the routing protocol OLSRv2 [8], written in the formal language T-AWN [6]. In Section A.1 we present the full data structure (types, variables, functions, etc.) needed to formally specify the protocol in Section A.2.

A.1 Data Structure

In this section, we describe the data types and functions used in modelling OLSRv2. We define some basic types in addition to those required by T-AWN, provide functions that operate on these types, and finally list the variables which form the protocol's state.

In the remainder, we use the notation $\mathtt{x}_{1..n}$ as shorthand for the tuple $(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$. Moreover, we use $e[\mathtt{x} := t]$ for syntactic substitution. Here, all instances of $\mathtt{x}$ in e are replaced with t.

Basic Types

We begin by defining a few basic data types and type synonyms.

```
data STATUS = SYMMETRIC | HEARD | LOST
data MPR = FLOODING | ROUTING | FLOOD_ROUTE
type SQN = ℤ
type METRIC = ℕ>0 ∪ {∞}
```

Moreover, we assume the existence of a polymorphic data type for lists.

```
data [a] = [] | a:[a]
```

We enable the concatenation of two lists and appending to a list via the functions `concat`, and `append`, respectively.

Messages

The `MESSAGE` data type encompasses both HELLO messages and TC messages, the two communication primitives used by the protocol.

```
data MESSAGE =
  HELLO IP TIME 𝒫(IP × STATUS) 𝒫(IP × MPR) 𝒫(IP × METRIC) 𝒫(IP × METRIC)
| TC IP IP TIME SQN SQN 𝒫(IP × METRIC)
```

In our implementation, HELLO messages contain six elements: (*i*) an originator address uniquely identifying the router that generated the message; (*ii*) a validity time detailing how long the message's contents should be considered valid for once the message is processed; (*iii*) a set of tuples assigning link statuses to each of the originating router's neighbours; (*iv*) a set of tuples indicating which neighbours the originating router has selected as flooding and routing MPRs; (*v*) a set of tuples representing the incoming metrics to the originating router from each of its neighbouring routers, and (*vi*) a set of tuples representing the outgoing metrics from the originating router to its neighbours.

TC messages also contain six elements: (*i*) an originator address uniquely identifying the router that generated the message; (*ii*) a sender address uniquely identifying the router that last broadcast the message; (*iii*) a validity time detailing how long the message's contents should be considered valid for once the message is processed; (*iv*) a sequence number which, when used in tandem with the originator address, uniquely identifies the message; (*v*) an advertising neighbour sequence number indicating how fresh the advertised information in the message is, and (*vi*) a set of advertised links between the originating router and its routing MPR selectors.

Routers combine these messages into packets when broadcasting them. We do not care about the packet header, and so take the `MSG` type of T-AWN to represent a list of `MESSAGE` values.

```
type MSG = [MESSAGE]
```

We determine whether a message is a HELLO message or a TC message by invoking the `isHELLO` and `isTC` functions.

$$\begin{aligned}
&\texttt{isHELLO} :: \texttt{MESSAGE} \to \mathbb{B}\\
&\texttt{isHELLO(HELLO _ _ _ _ _ _)} \equiv \mathit{True}\\
&\texttt{isHELLO(TC _ _ _ _ _ _)} \equiv \mathit{False}
\end{aligned}$$

$$\begin{aligned}
&\texttt{isTC} :: \texttt{MESSAGE} \to \mathbb{B}\\
&\texttt{isTC(HELLO _ _ _ _ _ _)} \equiv \mathit{False}\\
&\texttt{isTC(TC _ _ _ _ _ _)} \equiv \mathit{True}
\end{aligned}$$

We also use functions to extract the contents of these messages. Both HELLO and TC messages contain an originator address and validity time, which we access via the `oip` and `vtime` functions.

$$\begin{aligned}
&\texttt{oip} :: \texttt{MESSAGE} \to \texttt{IP}\\
&\texttt{oip(HELLO } x_1 \texttt{ _ _ _ _ _)} \equiv x_1\\
&\texttt{oip(TC } x_1 \texttt{ _ _ _ _ _)} \equiv x_1
\end{aligned}$$

$$\begin{aligned}
&\texttt{vtime} :: \texttt{MESSAGE} \to \texttt{TIME}\\
&\texttt{vtime(HELLO _ } x_2 \texttt{ _ _ _ _)} \equiv x_2\\
&\texttt{vtime(TC _ _ } x_3 \texttt{ _ _ _)} \equiv x_3
\end{aligned}$$

HELLO messages contain some unique elements that are not present in TC messages. We define a number of partial functions for extracting these elements.

$$\begin{aligned}
&\texttt{statuses} :: \texttt{MESSAGE} \rightharpoonup \mathscr{P}(\texttt{IP} \times \texttt{STATUS})\\
&\texttt{statuses(HELLO _ _ } x_3 \texttt{ _ _ _)} \equiv x_3
\end{aligned}$$

$$\begin{aligned}
&\texttt{mprs} :: \texttt{MESSAGE} \rightharpoonup \mathscr{P}(\texttt{IP} \times \texttt{MPR})\\
&\texttt{mprs(HELLO _ _ _ } x_4 \texttt{ _ _)} \equiv x_4
\end{aligned}$$

$$\begin{aligned}
&\texttt{inMetrics} :: \texttt{MESSAGE} \rightharpoonup \mathscr{P}(\texttt{IP} \times \texttt{METRIC})\\
&\texttt{inMetrics(HELLO _ _ _ _ } x_5 \texttt{ _)} \equiv x_5
\end{aligned}$$

$$\texttt{outMetrics} :: \texttt{MESSAGE} \rightharpoonup \mathscr{P}(\texttt{IP} \times \texttt{METRIC})$$
$$\texttt{outMetrics}(\texttt{HELLO}\ \_\ \_\ \_\ \_\ \_\ x_6) \equiv x_6$$

Similarly, we define some partial functions for extracting the unique elements of TC messages.

$$\texttt{sip} :: \texttt{MESSAGE} \rightharpoonup \texttt{IP}$$
$$\texttt{sip}(\texttt{TC}\ \_\ x_2\ \_\ \_\ \_\ \_) \equiv x_2$$

$$\texttt{sqn} :: \texttt{MESSAGE} \rightharpoonup \texttt{SQN}$$
$$\texttt{sqn}(\texttt{TC}\ \_\ \_\ \_\ x_4\ \_\ \_) \equiv x_4$$

$$\texttt{ansn} :: \texttt{MESSAGE} \rightharpoonup \texttt{SQN}$$
$$\texttt{ansn}(\texttt{TC}\ \_\ \_\ \_\ \_\ x_5\ \_) \equiv x_5$$

$$\texttt{dests} :: \texttt{MESSAGE} \rightharpoonup \mathscr{P}(\texttt{IP} \times \texttt{METRIC})$$
$$\texttt{dests}(\texttt{TC}\ \_\ \_\ \_\ \_\ \_\ x_6) \equiv x_6$$

In addition to manipulating existing messages, routers must be able to generate new messages periodically. To this end, the functions `newHELLO` and `newTC` are used to generate new HELLO and TC messages respectively. Both make use of a set `ls` containing values of type `L` (defined on Page 57).

$$\begin{aligned}
&\texttt{newHELLO} :: \texttt{IP} \times \texttt{TIME} \times \mathscr{P}(\texttt{L}) \times \texttt{TIME} \to \texttt{MESSAGE}\\
&\texttt{newHELLO(ip,vtime,ls,now)} \equiv\\
&\quad \textbf{let}\ \ \texttt{statuses} = \{(\texttt{L_oip}(\texttt{lt}), \texttt{L_status}(\texttt{lt},\texttt{now})) \mid \texttt{lt} \in \texttt{ls}\}\\
&\quad \textbf{and}\ \texttt{mprs} =\\
&\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{FLOODING}) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_fmpr}(\texttt{lt}) \wedge \neg\texttt{L_rmpr}(\texttt{lt})\} \cup\\
&\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{ROUTING}) \mid \texttt{lt} \in \texttt{ls} \wedge \neg\texttt{L_fmpr}(\texttt{lt}) \wedge \texttt{L_rmpr}(\texttt{lt})\} \cup\\
&\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{FLOOD_ROUTE}) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_fmpr}(\texttt{lt}) \wedge \texttt{L_rmpr}(\texttt{lt})\}\\
&\quad \textbf{and}\ \texttt{in_metrics} =\\
&\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{L_in_metric}(\texttt{lt})) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_status}(\texttt{lt},\texttt{now}) \neq \texttt{LOST}\}\\
&\quad \textbf{and}\ \texttt{out_metrics} =\\
&\qquad \{(\texttt{L_oip}(\texttt{lt}), \texttt{L_out_metric}(\texttt{lt})) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}\}\\
&\quad \textbf{in}\ \ \texttt{HELLO ip vtime statuses mprs in_metrics out_metrics}
\end{aligned}$$

$$\begin{aligned}
&\texttt{newTC} :: \texttt{IP} \times \texttt{TIME} \times \texttt{SQN} \times \texttt{SQN} \times \mathscr{P}(\texttt{L}) \times \texttt{TIME} \to \texttt{MESSAGE}\\
&\texttt{newTC(ip,vtime,sqn,ansn,ls,now)} \equiv\\
&\quad \textbf{let}\ \texttt{dests} = \{(\texttt{L_oip}(\texttt{lt}), \texttt{L_out_metric}(\texttt{lt})) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_rmpr_selector}(\texttt{lt}) \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}\ \}\\
&\quad \textbf{in}\ \ \texttt{TC ip ip vtime sqn ansn dests}
\end{aligned}$$

Outside of periodic generation, a TC message may also be forwarded by a router if received from one of that router's flooding MPR selectors. In this scenario, we use the function `forward` to update the sender IP of the TC message to the router's own address before broadcasting it.

$$\begin{aligned}
&\texttt{forward} :: \texttt{IP} \times \texttt{MSG} \rightharpoonup \texttt{MSG}\\
&\texttt{forward(ip,TC oip sip vtime sqn ansn dests)} \equiv\\
&\quad \texttt{TC oip ip vtime sqn ansn dests}
\end{aligned}$$

Interface Information Base

The *interface information base* records information about the links between a router and its 1-hop/2-hop neighbours.

$$\texttt{type L} = \texttt{IP} \times \texttt{TIME} \times \texttt{TIME} \times \texttt{TIME} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \texttt{METRIC} \times \texttt{METRIC}$$

$$\texttt{type N2} = \texttt{IP} \times \texttt{IP} \times \texttt{TIME} \times \texttt{METRIC} \times \texttt{METRIC}$$

A link tuple of type L contained within a router's link set consists of: (*i*) an originator address uniquely identifying the neighbouring router; (*ii*) a symmetric time until which the neighbour should be considered symmetric; (*iii*) a heard time until which the neighbour should be considered heard; (*iv*) a validity time after which the tuple should be removed; (*v*) a Boolean denoting whether the neighbour is a flooding MPR; (*vi*) a Boolean denoting whether the neighbour is a routing MPR; (*vii*) a Boolean denoting whether the neighbour is a flooding MPR selector; (*viii*) a Boolean denoting whether the neighbour is a routing MPR selector; (*ix*) an incoming link metric from the neighbour to this router, and (*x*) an outgoing link metric to the neighbour from this router.

A 2-hop tuple of type N2 contained within a router's 2-hop set consists of: (*i*) an originator address uniquely identifying the 1-hop neighbour; (*ii*) an originator address uniquely identifying the 2-hop neighbour; (*iii*) a validity time after which the tuple should be removed; (*iv*) an incoming link metric from the 2-hop neighbour to the 1-hop neighbour, and (*v*) an outgoing link metric from the 1-hop neighbour to the 2-hop neighbour.

Elements of the link set and 2-hop set are accessed through functions of the form L_* and N2_*.

$$\texttt{L\_oip} :: \texttt{L} \to \texttt{IP} \qquad \texttt{L\_oip}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_1$$

$$\texttt{L\_symmetric\_time} :: \texttt{L} \to \texttt{TIME} \qquad \texttt{L\_symmetric\_time}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_2$$

$$\texttt{L\_heard\_time} :: \texttt{L} \to \texttt{TIME} \qquad \texttt{L\_heard\_time}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_3$$

$$\texttt{L\_time} :: \texttt{L} \to \texttt{TIME} \qquad \texttt{L\_time}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_4$$

$$\texttt{L\_fmpr} :: \texttt{L} \to \mathbb{B} \qquad \texttt{L\_fmpr}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_5$$

$$\texttt{L\_rmpr} :: \texttt{L} \to \mathbb{B} \qquad \texttt{L\_rmpr}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_6$$

$$\texttt{L\_fmpr\_selector} :: \texttt{L} \to \mathbb{B} \qquad \texttt{L\_fmpr\_selector}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_7$$

$$\texttt{L\_rmpr\_selector} :: \texttt{L} \to \mathbb{B} \qquad \texttt{L\_rmpr\_selector}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_8$$

$$\texttt{L\_in\_metric} :: \texttt{L} \to \texttt{METRIC} \qquad \texttt{L\_in\_metric}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_9$$

$$\texttt{L\_out\_metric} :: \texttt{L} \to \texttt{METRIC} \qquad \texttt{L\_out\_metric}(\texttt{lt}_{1..10}) \equiv \texttt{lt}_{10}$$

$$\texttt{N2\_1h\_oip} :: \texttt{N2} \to \texttt{IP} \qquad \texttt{N2\_1h\_oip}(\texttt{n2}_{1..5}) \equiv \texttt{n2}_1$$

$$\texttt{N2\_2h\_oip} :: \texttt{N2} \to \texttt{IP} \qquad \texttt{N2\_2h\_oip}(\texttt{n2}_{1..5}) \equiv \texttt{n2}_2$$

$$\texttt{N2\_time} :: \texttt{N2} \to \texttt{TIME} \qquad \texttt{N2\_time}(\texttt{n2}_{1..5}) \equiv \texttt{n2}_3$$

$$\texttt{N2\_in\_metric} :: \texttt{N2} \to \texttt{METRIC} \qquad \texttt{N2\_in\_metric}(\texttt{n2}_{1..5}) \equiv \texttt{n2}_4$$

$$\texttt{N2\_out\_metric} :: \texttt{N2} \to \texttt{METRIC} \qquad \texttt{N2\_out\_metric}(\texttt{n2}_{1..5}) \equiv \texttt{n2}_5$$

The status of link tuples is determined as per section 7.1 of RFC 6130 [7], although we do not make use of link hysteresis. If the symmetric time of the tuple is greater than the current time, then the link is considered symmetric. Else, if the heard time of the tuple is greater than the current time, then the link is considered heard. Otherwise, the link is considered lost.

$$\begin{array}{l}
\texttt{L_status} :: \texttt{L} \times \texttt{TIME} \to \texttt{STATUS} \\
\texttt{L_status}(\texttt{lt}_{1..10}, \texttt{now}) \equiv \\
\quad \textbf{if} \quad \texttt{lt}_2 > \texttt{now} \ \textbf{then} \ \texttt{SYMMETRIC} \\
\quad \textbf{elsif} \ \texttt{lt}_3 > \texttt{now} \ \textbf{then} \ \texttt{HEARD} \\
\quad \textbf{else} \ \texttt{LOST}
\end{array}$$

When a new HELLO message is received, both the link set and 2-hop set are updated. The functions `addLinkTuple` and `add2HopTuples` create new link tuples and 2-hop tuples if they do not already exist. In the latter case, there is an additional check to ensure that a symmetric link tuple for the neighbour exists before considering 2-hop neighbours. We also include functions to update the metrics, timing values and MPR selection statuses of each of the tuples, existing or newly added, based on the message contents.

$$\begin{array}{l}
\texttt{addLinkTuple} :: \mathscr{P}(\texttt{L}) \times \texttt{IP} \times \texttt{TIME} \times \texttt{METRIC} \times \texttt{TIME} \to \mathscr{P}(\texttt{L}) \\
\texttt{addLinkTuple}(\texttt{ls},\texttt{moip},\texttt{vtime},\texttt{in_metric},\texttt{now}) \equiv \\
\quad \textbf{if} \quad \forall \texttt{lt} \in \texttt{ls}.\ \texttt{L_oip}(\texttt{lt}) \neq \texttt{moip} \\
\quad \textbf{then} \ \texttt{ls} \cup \{(\texttt{moip}, -\infty, -\infty, \texttt{now} + \texttt{vtime}, \mathit{False}, \mathit{False}, \mathit{False}, \mathit{False}, \texttt{in_metric}, \infty)\} \\
\quad \textbf{else} \ \texttt{ls}
\end{array}$$

$$\begin{array}{l}
\texttt{updateLinkOutMetrics} :: \texttt{IP} \times \mathscr{P}(\texttt{L}) \times \texttt{IP} \times \mathscr{P}(\texttt{IP} \times \texttt{METRIC}) \to \mathscr{P}(\texttt{L}) \\
\texttt{updateLinkOutMetrics}(\texttt{ip},\texttt{ls},\texttt{moip},\texttt{in_metrics}) \equiv \\
\quad \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_oip}(\texttt{lt}) \neq \texttt{moip} \vee \forall \texttt{m}_{1..2} \in \texttt{in_metrics}.\ \texttt{m}_1 \neq \texttt{ip}\} \cup \\
\quad \{\texttt{lt}_{1..10}[\texttt{lt}_{10} := \texttt{m}_2] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L_oip}(\texttt{lt}_{1..10}) = \texttt{moip} \ \wedge \texttt{m}_{1..2} \in \texttt{in_metrics} \wedge \texttt{m}_1 = \texttt{ip}\}
\end{array}$$

$$\begin{array}{l}
\texttt{updateSymmetricTime} :: \texttt{IP} \times \mathscr{P}(\texttt{L}) \times \texttt{IP} \times \texttt{TIME} \times \mathscr{P}(\texttt{IP} \times \texttt{STATUS}) \times \texttt{TIME} \times \texttt{TIME} \to \mathscr{P}(\texttt{L}) \\
\texttt{updateSymmetricTime}(\texttt{ip},\texttt{ls},\texttt{moip},\texttt{vtime},\texttt{statuses},\texttt{htime},\texttt{now}) \equiv \\
\quad \textbf{if} \quad \exists \texttt{x}_{1..2} \in \texttt{statuses}.\ \texttt{x}_1 = \texttt{ip} \wedge \texttt{x}_2 \neq \texttt{LOST} \\
\quad \textbf{then} \ \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_oip}(\texttt{lt}) \neq \texttt{moip}\} \cup \\
\qquad \{\texttt{lt}_{1..10}[\texttt{lt}_2 := \texttt{now} + \texttt{vtime}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L_oip}(\texttt{lt}_{1..10}) = \texttt{moip}\} \\
\quad \textbf{elsif} \ \exists \texttt{x}_{1..2} \in \texttt{statuses}.\ \texttt{x}_1 = \texttt{ip} \wedge \texttt{x}_2 = \texttt{LOST} \wedge \\
\qquad \exists \texttt{lt} \in \texttt{ls}.\ \texttt{L_oip}(\texttt{lt}) = \texttt{moip} \wedge \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC} \\
\quad \textbf{then} \ \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_oip}(\texttt{lt}) \neq \texttt{moip}\} \cup \\
\qquad \{\texttt{lt}_{1..10}[\texttt{lt}_2 := -\infty, \texttt{lt}_4 := \texttt{now} + \texttt{htime}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L_oip}(\texttt{lt}_{1..10}) = \texttt{moip}\} \\
\quad \textbf{else} \ \texttt{ls}
\end{array}$$

$$\begin{array}{l}
\texttt{updateHeardTime} :: \mathscr{P}(\texttt{L}) \times \texttt{IP} \times \texttt{TIME} \times \texttt{TIME} \to \mathscr{P}(\texttt{L}) \\
\texttt{updateHeardTime}(\texttt{ls},\texttt{moip},\texttt{vtime},\texttt{now}) \equiv \\
\quad \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_oip}(\texttt{lt}) \neq \texttt{moip}\} \cup \\
\quad \{\texttt{lt}_{1..10}[\texttt{lt}_3 := \max(\texttt{now} + \texttt{vtime}, \texttt{lt}_2)] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L_oip}(\texttt{lt}_{1..10}) = \texttt{moip}\}
\end{array}$$

$$\begin{array}{l}
\texttt{updateValidityTime} :: \mathscr{P}(\texttt{L}) \times \texttt{IP} \times \texttt{TIME} \times \texttt{TIME} \to \mathscr{P}(\texttt{L}) \\
\texttt{updateValidityTime}(\texttt{ls},\texttt{moip},\texttt{htime},\texttt{now}) \equiv \\
\quad \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_oip}(\texttt{lt}) \neq \texttt{moip}\} \cup \\
\quad \{\texttt{lt}_{1..10}[\texttt{lt}_4 := \max(\texttt{lt}_3 + \texttt{htime}, \texttt{lt}_4)] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L_oip}(\texttt{lt}_{1..10}) = \texttt{moip}\}
\end{array}$$

$\texttt{updateFMPRSelectors} :: \texttt{IP} \times \mathscr{P}(\texttt{L}) \times \texttt{IP} \times \mathscr{P}(\texttt{IP} \times \texttt{METRIC}) \times \mathscr{P}(\texttt{IP} \times \texttt{MPR}) \times \texttt{TIME} \to \mathscr{P}(\texttt{L})$

$\texttt{updateFMPRSelectors(ip,ls,moip,statuses,mprs,now)} \equiv$

if $\exists \texttt{x}_{1..2} \in \texttt{mprs}.\ \texttt{x}_1 = \texttt{ip} \wedge \texttt{x}_2 \in \{\texttt{FLOODING}, \texttt{FLOOD\_ROUTE}\}$

then $\{\texttt{lt} \in \texttt{ls} \mid \texttt{L\_oip}(\texttt{lt}) \neq \texttt{moip}\} \cup$
$\{\texttt{lt}_{1..10}[\texttt{lt}_7 := \mathit{True}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L\_oip}(\texttt{lt}_{1..10}) = \texttt{moip}\}$

elsif $\exists \texttt{x}_{1..2} \in \texttt{statuses}.\ \texttt{x}_1 = \texttt{ip} \wedge \texttt{x}_2 = \texttt{SYMMETRIC}$

then $\{\texttt{lt} \in \texttt{ls} \mid \texttt{L\_oip}(\texttt{lt}) \neq \texttt{moip}\} \cup$
$\{\texttt{lt}_{1..10}[\texttt{lt}_7 := \mathit{False}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L\_oip}(\texttt{lt}_{1..10}) = \texttt{moip}\}$

else $\texttt{ls}$

$\texttt{updateRMPRSelectors} :: \texttt{IP} \times \mathscr{P}(\texttt{L}) \times \texttt{IP} \times \mathscr{P}(\texttt{IP} \times \texttt{METRIC}) \times \mathscr{P}(\texttt{IP} \times \texttt{MPR}) \times \texttt{TIME} \to \mathscr{P}(\texttt{L})$

$\texttt{updateRMPRSelectors(ip,ls,moip,statuses,mprs,now)} \equiv$

if $\exists \texttt{x}_{1..2} \in \texttt{mprs}.\ \texttt{x}_1 = \texttt{ip} \wedge \texttt{x}_2 \in \{\texttt{ROUTING}, \texttt{FLOOD\_ROUTE}\}$

then $\{\texttt{lt} \in \texttt{ls} \mid \texttt{L\_oip}(\texttt{lt}) \neq \texttt{moip}\} \cup$
$\{\texttt{lt}_{1..10}[\texttt{lt}_8 := \mathit{True}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L\_oip}(\texttt{lt}_{1..10}) = \texttt{moip}\}$

elsif $\exists \texttt{x}_{1..2} \in \texttt{statuses}.\ \texttt{x}_1 = \texttt{ip} \wedge \texttt{x}_2 = \texttt{SYMMETRIC}$

then $\{\texttt{lt} \in \texttt{ls} \mid \texttt{L\_oip}(\texttt{lt}) \neq \texttt{moip}\} \cup$
$\{\texttt{lt}_{1..10}[\texttt{lt}_8 := \mathit{False}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L\_oip}(\texttt{lt}_{1..10}) = \texttt{moip}\}$

else $\texttt{ls}$

$\texttt{add2HopTuples} :: \texttt{IP} \times \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{IP} \times \mathscr{P}(\texttt{IP} \times \texttt{STATUS}) \times \texttt{TIME} \to \mathscr{P}(\texttt{N2})$

$\texttt{add2HopTuples(ip,ls,2hs,moip,statuses,now)} \equiv$

if $\exists \texttt{lt} \in \texttt{ls}.\ \texttt{L\_oip}(\texttt{lt}) = \texttt{moip} \wedge \texttt{L\_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}$

then $\texttt{2hs} \cup \{(\texttt{moip}, \texttt{x}_1, -\infty, \infty, \infty) \mid \texttt{x}_{1..2} \in \texttt{statuses} \wedge \texttt{x}_1 \neq \texttt{ip} \wedge \texttt{x}_2 = \texttt{SYMMETRIC} \wedge$
$\forall \texttt{n2} \in \texttt{2hs}.\ \texttt{N2\_1h\_oip}(\texttt{n2}) \neq \texttt{moip} \vee \texttt{N2\_2h\_oip}(\texttt{n2}) \neq \texttt{x}_1\}$

else $\texttt{2hs}$

$\texttt{update2HopInMetrics} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{IP} \times \mathscr{P}(\texttt{IP} \times \texttt{METRIC}) \times \texttt{TIME} \to \mathscr{P}(\texttt{N2})$

$\texttt{update2HopInMetrics(ls,2hs,moip,in\_metrics,now)} \equiv$

if $\exists \texttt{lt} \in \texttt{ls}.\ \texttt{L\_oip}(\texttt{lt}) = \texttt{moip} \wedge \texttt{L\_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}$

then $\{\texttt{n2} \in \texttt{2hs} \mid \texttt{N2\_1h\_oip}(\texttt{n2}) \neq \texttt{moip} \vee \forall \texttt{x}_{1..2} \in \texttt{in\_metrics}.\ \texttt{x}_1 \neq \texttt{N2\_2h\_oip}(\texttt{n2})\} \cup$
$\{\texttt{n2}_{1..5}[\texttt{n2}_4 := \texttt{x}_2] \mid \texttt{n2}_{1..5} \in \texttt{2hs} \wedge \texttt{N2\_1h\_oip}(\texttt{n2}_{1..5}) = \texttt{moip} \wedge$
$\texttt{x}_{1..2} \in \texttt{in\_metrics} \wedge \texttt{x}_1 = \texttt{N2\_2h\_oip}(\texttt{n2}_{1..5})\}$

else $\texttt{2hs}$

$\texttt{update2HopOutMetrics} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{IP} \times \mathscr{P}(\texttt{IP} \times \texttt{METRIC}) \times \texttt{TIME} \to \mathscr{P}(\texttt{N2})$

$\texttt{update2HopOutMetrics(ls,2hs,moip,out\_metrics,now)} \equiv$

if $\exists \texttt{lt} \in \texttt{ls}.\ \texttt{L\_oip}(\texttt{lt}) = \texttt{moip} \wedge \texttt{L\_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}$

then $\{\texttt{n2} \in \texttt{2hs} \mid \texttt{N2\_1h\_oip}(\texttt{n2}) \neq \texttt{moip} \vee \forall \texttt{x}_{1..2} \in \texttt{out\_metrics}.\ \texttt{x}_1 \neq \texttt{N2\_2h\_oip}(\texttt{n2})\} \cup$
$\{\texttt{n2}_{1..5}[\texttt{n2}_5 := \texttt{x}_2] \mid \texttt{n2}_{1..5} \in \texttt{2hs} \wedge \texttt{N2\_1h\_oip}(\texttt{n2}_{1..5}) = \texttt{moip} \wedge$
$\texttt{x}_{1..2} \in \texttt{out\_metrics} \wedge \texttt{x}_1 = \texttt{N2\_2h\_oip}(\texttt{n2}_{1..5})\}$

else $\texttt{2hs}$

$$\begin{aligned}
&\texttt{update2HopTime} :: \texttt{IP} \times \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{IP} \times \texttt{TIME} \times \mathscr{P}(\texttt{IP} \times \texttt{STATUS}) \times \texttt{TIME} \to \mathscr{P}(\texttt{N2}) \\
&\texttt{update2HopTime(ip,ls,2hs,moip,vtime,statuses,now)} \equiv \\
&\quad \textbf{if} \quad \exists \texttt{lt} \in \texttt{ls}.\ \texttt{L_oip}(\texttt{lt}) = \texttt{moip} \wedge \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC} \\
&\quad \textbf{then} \ \{\texttt{n2} \in \texttt{2hs} \mid \texttt{N2_1h_oip}(\texttt{n2}) \neq \texttt{moip} \vee \forall \texttt{x}_{1..2} \in \texttt{statuses}.\ \texttt{x}_1 \neq \texttt{N2_2h_oip}(\texttt{n2})\} \cup \\
&\qquad \{\texttt{n2}_{1..5}[\texttt{n2}_3 := \texttt{now} + \texttt{vtime}] \mid \texttt{n2}_{1..5} \in \texttt{2hs} \wedge \texttt{N2_1h_oip}(\texttt{n2}_{1..5}) = \texttt{moip} \wedge \\
&\qquad\qquad \exists \texttt{x}_{1..2} \in \texttt{statuses}.\ \texttt{N2_2h_oip}(\texttt{n2}_{1..5}) = \texttt{x}_1 \wedge \\
&\qquad\qquad\qquad \texttt{x}_1 \neq \texttt{ip} \wedge \texttt{x}_2 = \texttt{SYMMETRIC}\} \\
&\quad \textbf{else} \ \texttt{2hs}
\end{aligned}$$

Updates to the interface information base may also be performed to preserve its consistency. For one, expired tuples should be purged from the link set and 2-hop set, which we achieve using the functions `purgeLinkSet` and `purge2HopSet`. Moreover, we must recalculate and update our sets of MPRs when they no longer satisfy the properties required of them. First, we non-deterministically pick sets of flooding and routing mprs based on sections 18.3, 18.4 and 18.5 of RFC 7181 [8]. We then update the link set in `updateFMPRs` and `updateRMPRs` if the condition for recalculation holds.

$$\begin{aligned}
&\texttt{purgeLinkSet} :: \mathscr{P}(\texttt{L}) \times \texttt{TIME} \to \mathscr{P}(\texttt{L}) \\
&\texttt{purgeLinkSet(ls,now)} \equiv \\
&\quad \{\texttt{lt} \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L_time}(\texttt{lt}) > \texttt{now} \wedge \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}\} \cup \\
&\quad \{\texttt{lt}_{1..10}[\texttt{lt}_5,\texttt{lt}_6,\texttt{lt}_7,\texttt{lt}_8 := \mathit{False}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{L_time}(\texttt{lt}_{1..10}) > \texttt{now} \wedge \\
&\qquad\qquad \texttt{L_status}(\texttt{lt}_{1..10},\texttt{now}) \neq \texttt{SYMMETRIC}\}
\end{aligned}$$

$$\begin{aligned}
&\texttt{purge2HopSet} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{TIME} \to \mathscr{P}(\texttt{N2}) \\
&\texttt{purge2HopSet(ls,2hs,now)} \equiv \\
&\quad \{\texttt{n2} \in \texttt{2hs} \mid \texttt{N2_time}(\texttt{n2}) > \texttt{now} \wedge \exists \texttt{lt} \in \texttt{ls}.\ \texttt{L_oip}(\texttt{lt}) = \texttt{N2_1h_oip}(\texttt{n2}) \wedge \\
&\qquad\qquad \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}\}
\end{aligned}$$

$$\begin{aligned}
&\texttt{validFMPRs} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{TIME} \to \mathscr{P}(\mathscr{P}(\texttt{L})) \\
&\texttt{validFMPRs(ls,2hs,now)} \equiv \\
&\quad \textbf{let} \quad \texttt{N1} = \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_status}(\texttt{lt},\texttt{now}) = \texttt{SYMMETRIC}\} \\
&\quad \textbf{and} \quad \texttt{N2} = \{\texttt{n2} \in \texttt{2hs} \mid \exists \texttt{lt} \in \texttt{N1}.\ \texttt{L_oip}(\texttt{lt}) = \texttt{N2_1h_oip}(\texttt{n2})\} \\
&\quad \textbf{and} \quad \texttt{d}(\texttt{ip},\texttt{S}) = \texttt{min}\big(\{\infty\} \cup \{1 \mid \exists \texttt{x} \in \texttt{S}.\ \texttt{L_oip}(\texttt{x}) = \texttt{ip}\} \cup \\
&\qquad\qquad \{2 \mid \exists \texttt{x} \in \texttt{S},\ \texttt{y} \in \texttt{N2}.\ \texttt{N2_2h_oip}(\texttt{y}) = \texttt{ip} \wedge \\
&\qquad\qquad\qquad \texttt{L_oip}(\texttt{x}) = \texttt{N2_1h_oip}(\texttt{y})\}\big) \\
&\quad \textbf{in} \quad \{\texttt{M} \subseteq \texttt{N1} \mid \forall \texttt{y} \in \texttt{N2}.\ \texttt{d}(\texttt{N2_2h_oip}(\texttt{y}),\texttt{M}) = \texttt{d}(\texttt{N2_2h_oip}(\texttt{y}),\texttt{N1})\}
\end{aligned}$$

$$\begin{aligned}
&\texttt{updateFMPRs} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{TIME} \times \mathscr{P}(\texttt{L}) \to \mathscr{P}(\texttt{L}) \\
&\texttt{updateFMPRs(ls,2hs,now,fmprs)} \equiv \\
&\quad \textbf{if} \quad \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_fmpr}(\texttt{lt})\} \notin \texttt{validFMPRs}(\texttt{ls},\texttt{2hs},\texttt{now}) \\
&\quad \textbf{then} \ \{\texttt{lt}_{1..10}[\texttt{lt}_5 := \mathit{False}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{lt}_{1..10} \notin \texttt{fmprs}\} \cup \\
&\qquad \{\texttt{lt}_{1..10}[\texttt{lt}_5 := \mathit{True}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{lt}_{1..10} \in \texttt{fmprs}\} \cup \\
&\quad \textbf{else} \ \texttt{ls}
\end{aligned}$$

$$\begin{array}{l}
\texttt{validRMPRs} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{TIME} \to \mathscr{P}(\mathscr{P}(\texttt{L})) \\
\texttt{validRMPRs(ls,2hs,now)} \equiv \\
\quad \textbf{let} \quad \texttt{N1} = \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_status(lt,now)} = \texttt{SYMMETRIC}\} \\
\quad \textbf{and} \quad \texttt{N2} = \{\texttt{n2} \in \texttt{2hs} \mid \exists \texttt{lt} \in \texttt{N1}.\ \texttt{L_oip(lt)} = \texttt{N2_1h_oip(n2)}\} \\
\quad \textbf{and} \quad \texttt{d1(x)} = \texttt{L_in_metric(x)} \\
\quad \textbf{and} \quad \texttt{d2(y)} = \cancel{\texttt{N2_out_metric(y)}}\ \texttt{N2_in_metric(y)}\ ^{3} \\
\quad \textbf{and} \quad \texttt{d(ip,S)} = \texttt{min}\big(\{\infty\} \cup \{\texttt{d1(x)} \mid \texttt{x} \in \texttt{S} \wedge \texttt{L_oip(x)} = \texttt{ip}\} \cup \\
\qquad\qquad\qquad \{\texttt{d1(x)} + \texttt{d2(y)} \mid \texttt{x} \in \texttt{S} \wedge \texttt{y} \in \texttt{N2} \wedge \texttt{N2_2h_oip(y)} = \texttt{ip} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \texttt{L_oip(x)} = \texttt{N2_1h_oip(y)}\}\big) \\
\quad \textbf{in} \quad \{\texttt{M} \subseteq \texttt{N1} \mid \forall \texttt{y} \in \texttt{N2}.\ \texttt{d(N2_2h_oip(y),M)} = \texttt{d(N2_2h_oip(y),N1)}\}
\end{array}$$

$$\begin{array}{l}
\texttt{updateRMPRs} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \texttt{TIME} \times \mathscr{P}(\texttt{L}) \to \mathscr{P}(\texttt{L}) \\
\texttt{updateRMPRs(ls,2hs,now,rmprs)} \equiv \\
\quad \textbf{if} \quad \{\texttt{lt} \in \texttt{ls} \mid \texttt{L_rmpr(lt)}\} \notin \texttt{validRMPRs(ls,2hs,now)} \\
\quad \textbf{then} \{\texttt{lt}_{1..10}[\texttt{lt}_6 := \mathit{False}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{lt}_{1..10} \notin \texttt{rmprs}\} \cup \\
\qquad \{\texttt{lt}_{1..10}[\texttt{lt}_6 := \mathit{True}] \mid \texttt{lt}_{1..10} \in \texttt{ls} \wedge \texttt{lt}_{1..10} \in \texttt{rmprs}\} \cup \\
\quad \textbf{else} \quad \texttt{ls}
\end{array}$$

Topology Information Base

The *topology information base* records information received via TC messages, and in particular the links advertised in such messages. It also maintains a routing set that consists of shortest routes to reachable destinations.

$$\begin{array}{l}
\texttt{type AR = IP} \times \texttt{SQN} \times \texttt{TIME} \\
\texttt{type TR = IP} \times \texttt{IP} \times \texttt{TIME} \times \texttt{METRIC} \\
\texttt{type R = IP} \times \texttt{IP} \times \texttt{METRIC}
\end{array}$$

An entry of type AR in the advertising remote router set consists of: (*i*) an originator address identifying a router from which a TC message was recently received; (*ii*) an advertising neighbour sequence number identifying the most recent advertised information received from the originating router, and (*iii*) a validity time after which the tuple should be removed. Elements of the advertising router tuples are accessed via functions of the form AR_*.

$$\begin{array}{lll}
\texttt{AR_oip} :: \texttt{AR} \to \texttt{IP} & \texttt{AR_sqn} :: \texttt{AR} \to \texttt{SQN} & \texttt{AR_time} :: \texttt{AR} \to \texttt{TIME} \\
\texttt{AR_oip(ar}_{1..3}\texttt{)} \equiv \texttt{ar}_1 & \texttt{AR_sqn(ar}_{1..3}\texttt{)} \equiv \texttt{ar}_2 & \texttt{AR_time(ar}_{1..3}\texttt{)} \equiv \texttt{ar}_3
\end{array}$$

An entry of type TR in the router topology set consists of: (*i*) the originator address of an advertising router; (*ii*) the originator address of a destination router which can be reached directly via the advertising router; (*iii*) a validity time after which the tuple should be removed, and (*iv*) a link metric from the advertising router to the destination router.

[3] See Section 5.

An entry of type R in the routing set consists of: (*i*) the destination address of a reachable router; (*ii*) the neighbour via which the router can be reached, and (*iii*) the cost of the path to the destination via the neighbour.

When a new TC message is processed, we store new tuples of type AR for the advertising router and type TR for the links advertised in the message.

$$\begin{array}{l}\texttt{updateAdvertisingRouters} :: \mathscr{P}(\texttt{AR}) \times \texttt{IP} \times \texttt{SQN} \times \texttt{TIME} \times \texttt{TIME} \to \mathscr{P}(\texttt{AR}) \\ \texttt{updateAdvertisingRouters(arrs,moip,mansn,vtime,now)} \equiv \\ \quad \{\texttt{ar} \in \texttt{arrs} \mid \texttt{AR\_oip(ar)} \neq \texttt{moip}\} \cup \{(\texttt{moip},\texttt{mansn},\texttt{now}+\texttt{vtime})\}\end{array}$$

$$\begin{array}{l}\texttt{updateRouterTopology} :: \texttt{IP} \times \mathscr{P}(\texttt{TR}) \times \texttt{IP} \times \texttt{TIME} \times \mathscr{P}(\texttt{IP} \times \texttt{METRIC}) \times \texttt{TIME} \to \mathscr{P}(\texttt{TR}) \\ \texttt{updateRouterTopology(ip,rts,moip,vtime,dests,now)} \equiv \\ \quad \{\texttt{tr}_{1..4} \in \texttt{rts} \mid \texttt{tr}_1 \neq \texttt{moip}\} \cup \{(\texttt{moip},\texttt{d}_1,\texttt{now}+\texttt{vtime},\texttt{d}_2) \mid \texttt{d}_{1..2} \in \texttt{dests} \wedge \texttt{d}_1 \neq \texttt{ip}\}\end{array}$$

As with the interface information base, the topology information base must be kept consistent. We have two functions which remove expired tuples from the advertised remote router set and router topology set respectively. We also have a function called incrementANSN which records changes to the set of advertised links by incrementing the router's advertising neighbour sequence number.

$$\begin{array}{l}\texttt{purgeAdvertisingRouters} :: \mathscr{P}(\texttt{AR}) \times \texttt{TIME} \to \mathscr{P}(\texttt{AR}) \\ \texttt{purgeAdvertisingRouters(arrs,now)} \equiv \{\texttt{ar} \in \texttt{arrs} \mid \texttt{AR\_time(ar)} > \texttt{now}\}\end{array}$$

$$\begin{array}{l}\texttt{purgeRouterTopology} :: \mathscr{P}(\texttt{TR}) \times \texttt{TIME} \to \mathscr{P}(\texttt{TR}) \\ \texttt{purgeRouterTopology(rts,now)} \equiv \{\texttt{tr}_{1..4} \in \texttt{rts} \mid \texttt{tr}_3 > \texttt{now}\}\end{array}$$

$$\begin{array}{l}\texttt{incrementANSN} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{L}) \times \texttt{SQN} \to \texttt{SQN} \\ \texttt{incrementANSN(ls,prev\_ls,ansn)} \equiv \\ \quad \textbf{if} \quad \{\texttt{L\_oip(lt)} \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{L\_rmpr\_selector(lt)}\} \neq \\ \qquad \{\texttt{L\_oip(lt)} \mid \texttt{lt} \in \texttt{prev\_ls} \wedge \texttt{L\_rmpr\_selector(lt)}\} \\ \quad \textbf{then } \texttt{ansn}+1 \textbf{ else } \texttt{ansn}\end{array}$$

Finally, we have a function which calculates a router's optimal routing sets and another which updates the routing set to an optimal set if the current set is not optimal.

$$\begin{array}{l}\texttt{optimalRoutingSets} :: \texttt{IP} \times \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{TR}) \times \texttt{TIME} \to \mathscr{P}(\mathscr{P}(\texttt{R})) \\ \texttt{optimalRoutingSets(ip,ls,rts,now)} \equiv \\ \quad \textbf{let} \quad \texttt{links} = \{(\texttt{tr}_1,\texttt{tr}_2,\texttt{tr}_4) \mid \texttt{tr}_{1..4} \in \texttt{rts}\} \cup \\ \qquad \{(\texttt{ip},\texttt{L\_oip(lt)},\texttt{L\_out\_metric(lt)}) \mid \texttt{lt} \in \texttt{ls} \wedge \texttt{status(lt,now)} = \texttt{SYMMETRIC}\} \\ \quad \textbf{and } \texttt{routes} = \\ \qquad \{(\texttt{d}_\texttt{n},\texttt{d}_1,\texttt{m}) \mid (\texttt{s}_1,\texttt{d}_1,\texttt{m}_1)...(\texttt{s}_\texttt{n},\texttt{d}_\texttt{n},\texttt{m}_\texttt{n}) \in \texttt{links}^+ \wedge \texttt{s}_1 = \texttt{ip} \wedge \bigwedge_{i=1}^{n-1} \texttt{d}_\texttt{i} = \texttt{s}_{\texttt{i}+1} \wedge \texttt{m} = \sum_{i=1}^{n} \texttt{m}_\texttt{i}\} \\ \quad \textbf{and } \texttt{shortest\_routes} = \{(\texttt{d},\texttt{s},\texttt{m}) \in \texttt{routes} \mid \forall(\texttt{d}',\texttt{s}',\texttt{m}') \in \texttt{routes}.\ \texttt{d} = \texttt{d}' \Longrightarrow \texttt{m} \leq \texttt{m}'\} \\ \quad \textbf{in} \quad \{\texttt{rs}' \subseteq \texttt{shortest\_routes} \mid (\forall(\texttt{d},\texttt{s},\texttt{m}) \in \texttt{routes}.\ \exists(\texttt{d}',\texttt{s}',\texttt{m}') \in \texttt{rs}'.\ \texttt{d} = \texttt{d}') \wedge \\ \qquad\qquad (\forall(\texttt{d},\texttt{s},\texttt{m}),(\texttt{d}',\texttt{s}',\texttt{m}') \in \texttt{rs}'.\ \texttt{d} = \texttt{d}' \Longrightarrow \texttt{s} = \texttt{s}' \wedge \texttt{m} = \texttt{m}')\}\end{array}$$

$$\begin{aligned}&\texttt{updateRoutingSet} :: \texttt{IP} \times \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{TR}) \times \texttt{TIME} \times \mathscr{P}(\texttt{R}) \times \mathscr{P}(\texttt{R}) \to \mathscr{P}(\texttt{R})\\&\texttt{updateRoutingSet}(\texttt{ip},\texttt{ls},\texttt{rts},\texttt{now},\texttt{rs},\texttt{rs}') \equiv\\&\quad \textbf{if } \texttt{rs} \notin \texttt{optimalRoutingSets}(\texttt{ip},\texttt{ls},\texttt{rts},\texttt{now}) \textbf{ then } \texttt{rs}' \textbf{ else } \texttt{rs}\end{aligned}$$

Received Message Information Base

The *received message information base* records all TC messages that have been processed and received by the router. Messages are uniquely identified by their originator address and sequence number, so we store both pieces of information for when we make future processing and forwarding decisions.

$$\begin{aligned}\texttt{type P} &= \texttt{IP} \times \texttt{SQN}\\\texttt{type RX} &= \texttt{IP} \times \texttt{SQN}\end{aligned}$$

As with the other information bases, we define some basic functions for extracting information from tuples.

$$\begin{array}{llll}\texttt{P\_oip} :: \texttt{P} \to \texttt{IP} & \texttt{P\_sqn} :: \texttt{P} \to \texttt{SQN} & \texttt{RX\_oip} :: \texttt{RX} \to \texttt{IP} & \texttt{RX\_sqn} :: \texttt{RX} \to \texttt{SQN}\\\texttt{P\_oip}(\texttt{p}_{1..2}) \equiv \texttt{p}_1 & \texttt{P\_sqn}(\texttt{p}_{1..2}) \equiv \texttt{p}_2 & \texttt{RX\_oip}(\texttt{rx}_{1..2}) \equiv \texttt{rx}_1 & \texttt{RX\_sqn}(\texttt{rx}_{1..2}) \equiv \texttt{rx}_2\end{array}$$

We also provide two functions for storing new tuples in either a processed set or a received set.

$$\begin{aligned}&\texttt{addProcessedTuple} :: \mathscr{P}(\texttt{P}) \times \texttt{IP} \times \texttt{SQN} \to \mathscr{P}(\texttt{P})\\&\texttt{addProcessedTuple}(\texttt{ps},\texttt{moip},\texttt{msqn}) \equiv \texttt{ps} \cup \{(\texttt{moip},\texttt{msqn})\}\\&\\&\texttt{addReceivedTuple} :: \mathscr{P}(\texttt{RX}) \times \texttt{IP} \times \texttt{SQN} \to \mathscr{P}(\texttt{RX})\\&\texttt{addReceivedTuple}(\texttt{rxs},\texttt{moip},\texttt{msqn}) \equiv \texttt{rxs} \cup \{(\texttt{moip},\texttt{msqn})\}\end{aligned}$$

Information Base Changes

In our model, we localise all relevant information base updates to a single process. The condition to call the process and trigger these updates, given by the `updatesPending` function, asserts that the updates would modify the protocol's information bases and implies that they are currently inconsistent.

$$\begin{aligned}&\texttt{updatesPending} :: \mathscr{P}(\texttt{L}) \times \mathscr{P}(\texttt{N2}) \times \mathscr{P}(\texttt{AR}) \times \mathscr{P}(\texttt{TR}) \times \mathscr{P}(\texttt{R}) \times \texttt{SQN} \times \mathscr{P}(\texttt{L}) \times \texttt{IP} \times \texttt{TIME} \to \mathbb{B}\\&\texttt{updatesPending}(\texttt{ls},\texttt{2hs},\texttt{arrs},\texttt{rts},\texttt{rs},\texttt{ansn},\texttt{prev\_ls},\texttt{ip},\texttt{now}) \equiv\\&\quad \texttt{ls} \neq \texttt{purgeLinkSet}(\texttt{ls},\texttt{now}) \vee\\&\quad \texttt{2hs} \neq \texttt{purge2HopSet}(\texttt{ls},\texttt{2hs},\texttt{now}) \vee\\&\quad \texttt{arrs} \neq \texttt{purgeAdvertisingRouters}(\texttt{arrs},\texttt{now}) \vee\\&\quad \texttt{rts} \neq \texttt{purgeRouterTopology}(\texttt{rts},\texttt{now}) \vee\\&\quad \{\texttt{lt} \in \texttt{ls} \mid \texttt{L\_fmpr}(\texttt{lt})\} \notin \texttt{validFMPRs}(\texttt{ls},\texttt{2hs},\texttt{now}) \vee\\&\quad \{\texttt{lt} \in \texttt{ls} \mid \texttt{L\_rmpr}(\texttt{lt})\} \notin \texttt{validRMPRs}(\texttt{ls},\texttt{2hs},\texttt{now}) \vee\\&\quad \texttt{ansn} \neq \texttt{incrementANSN}(\texttt{ls},\texttt{prev\_ls},\texttt{ansn}) \vee\\&\quad \texttt{rs} \notin \texttt{optimalRoutingSets}(\texttt{ip},\texttt{ls},\texttt{rts},\texttt{now})\end{aligned}$$

Process State

Our protocol maintains its state in a list of variables. The main variables are summarised in Table 2. We make a distinction between those variables that are modified during the protocol's execution and those that are not. For the former, we use the shorthand σ to represent a comma-separated list of these variables. Similarly, we use Γ for the latter. We also maintain a variable `queue` of type `[MSG]` in our input queue process and a variable `msg` of type `MESSAGE` when processing or forwarding a message.

Table 2: The variables constituting the protocol's state

	Name	Type	Description
σ	`ls`	$\mathscr{P}$(`L`)	Link set maintaining information about 1-hop neighbours and their statuses
	`2hs`	$\mathscr{P}$(`N2`)	2-hop set maintaining information about 2-hop neighbours
	`arrs`	$\mathscr{P}$(`AR`)	Advertising remote router set containing information about routers which have advertised links
	`rts`	$\mathscr{P}$(`TR`)	Router topology set containing advertised links
	`rs`	$\mathscr{P}$(`R`)	Routing set containing shortest known routes
	`ps`	$\mathscr{P}$(`P`)	Processed set identifying processed TC messages
	`rxs`	$\mathscr{P}$(`RX`)	Received set identifying TC messages received and considered for forwarding
	`pkt`	`[MESSAGE]`	List of messages requiring sending, such as those generated by the router or forwarded by it
	`hello_time`	`TIME`	Time when next HELLO message must be added to `pkt`
	`tc_time`	`TIME`	Time when next TC message must be added to `pkt`
	`send_time`	`TIME`	Time when `pkt` must be broadcast
	`mqueue`	`[MESSAGE]`	Queue of to-be-processed messages
	`sqn`	`SQN`	Sequence number identifying a TC message
	`ansn`	`SQN`	Advertising neighbour sequence number included in TC messages to indicate how recent the advertised contents are
	`prev_ls`	$\mathscr{P}$(`L`)	Previous link set used to check for updates
Γ	`ip`	`IP`	Address of the router
	`hp_maxjitter`	`TIME`	Maximum jitter time for HELLO messages
	`tp_maxjitter`	`TIME`	Maximum jitter time for TC messages
	`h_hold_time`	`TIME`	Validity time for generated HELLO messages
	`t_hold_time`	`TIME`	Validity time for generated TC messages
	`l_hold_time`	`TIME`	Length that lost links should be kept for
	`hello_interval`	`TIME`	Period between HELLO message transmissions
	`tc_interval`	`TIME`	Period between TC message transmissions

A.2 T-AWN-Specification of OLSRv2

In this section, we present our T-AWN model of OLSRv2. The model consists of five main processes implementing the OLSRv2 specification and a queue process to receive packets from other routers:

`OLSR` **(Process 1):** The `OLSR` process constitutes the main protocol loop. It is responsible for receiving packets from the input queue and processing these packets according to their type. It it also responsible for periodically generating new HELLO and TC messages.

`UPDATE_INFO` **(Process 2):** The `UPDATE_INFO` process ensures that the protocol's information bases remain consistent with certain constraints.

`PROCESS_HELLO` **(Process 3):** The `PROCESS_HELLO` process is responsible for recording information obtained through HELLO messages in the relevant information bases.

`PROCESS_TC`**, (Process 4):** The `PROCESS_TC` process is responsible for recording information obtained through TC messages in the relevant information bases.

`FORWARD_TC`**, (Process 5):** The `FORWARD_TC` process forwards TC messages received from the router's flooding MPR selectors, subject to a few side conditions.

`QUEUE`**, (Process 6):** The `QUEUE` process receives packets from other routers in the network and delivers them to the `OLSR` process.

For clarity, we use three pieces of syntactic sugar in our guards. The first, `otherwise`, stands for the negation of the previous guard. The second, `Let`, denotes a guard used as a non-deterministic assignment but has no meaning otherwise. The third, `Updated`, is shorthand for the expression

$$\neg \texttt{updatesPending}(\texttt{ls}, \texttt{2hs}, \texttt{arrs}, \texttt{rts}, \texttt{rs}, \texttt{ansn}, \texttt{prev\_ls}, \texttt{ip}, \texttt{now}) \ .$$

A.2.1 The Main Routine

The main OLSR routine performs a number of different roles. The most basic of these is receiving a packet from the input queue, which occurs in the block on lines 1-3. Packets of type `MSG` are simply lists of HELLO and TC messages, so we concatenate received packets with an existing queue of to-be-processed messages. When the protocol is ready to process a HELLO or TC message, the choice on line 8 is taken, with `msg` and `msgs` assigned to the head and tail of `mqueue` respectively by the guard. Note that this guard contains an `Updated` conjunct, which asserts that the information bases have already been updated by the block on lines 5-6. Once `msg` is assigned to the element at the head of the queue, the ensuing assignment statement assigns `mqueue` to its tail `msgs`. The guards on lines 9 and 12 then ensure that `msg` is processed according to its type, be that a HELLO message or a TC message.

Aside from processing messages, a router must generate its own HELLO and TC messages periodically. The guard on line 15 asserts that a new HELLO message is ready to be added to `pkt`, a list which accumulates all messages generated during a time tick in order to circumvent broadcasting delays. The guard is true whenever the local clock `now` enters the jitter period before the message's preparation deadline `hello_time`. The condition $\texttt{now} \leq \texttt{hello\_time}$ is redundant, as one can prove that it is always met even when left out; it reminds us that we have not yet exceeded the deadline `hello_time` when line 16 is executed. A new HELLO message is generated by line 16 and appended to `pkt`. Then, `hello_time` is increased to `hello_interval` time units away from `now`, and `send_time` is set to $\texttt{now} + 1$ so that the packet will be broadcast during the next tick. Sending a TC message in the block starting at line 21 follows a similar process, except that a sequence number `sqn` is included in the message and subsequently incremented. Once all messages have been accumulated and the guard on line 28 becomes true, the accumulated packet is broadcast, and both `pkt` and `send_time` are reset to indicate no pending messages.

Process 1: The main OLSR process

```
   OLSR(σ,Γ) ≝
   /* Receive a packet (i.e. a list of messages) from the queue process */
1  receive(msgs).
2      [[mqueue := concat(mqueue,msgs)]]
3      OLSR(σ,Γ)
4  +
   /* Execute pending updates to relevant information bases */
5  [¬Updated]
6      UPDATE_INFO(σ,Γ)
7  +
   /* Process a received message */
8  [Updated ∧ send_time ≠ now ∧ mqueue = (msg : msgs)] [[mqueue := msgs]]
       /* Process a received HELLO message */
9      [isHELLO(msg)]
10         PROCESS_HELLO(σ,Γ,msg)
11     +
       /* Process a received TC message */
12     [isTC(msg)]
13         PROCESS_TC(σ,Γ,msg)
14 +
   /* Time to generate a HELLO message */
15 [Updated ∧ send_time ≠ now ∧ hello_time − hp_maxjitter ≤ now ≤ hello_time]
       /* Add the message to the current packet */
16     [[pkt := append(pkt,newHELLO(ip,h_hold_time,ls,now))]]
       /* Set relevant timers */
17     [[hello_time := now + hello_interval]]
18     [[send_time := now + 1]]
19     OLSR(σ,Γ)
20 +
   /* Time to generate a TC message */
21 [Updated ∧ send_time ≠ now ∧ tc_time − tp_maxjitter ≤ now ≤ tc_time]
       /* Add the message to the current packet */
22     [[pkt := append(pkt,newTC(ip,t_hold_time,sqn,ansn,ls,now))]]
       /* Increment the sequence number */
23     [[sqn := sqn + 1]]
       /* Set relevant timers */
24     [[tc_time := now + tc_interval]]
25     [[send_time := now + 1]]
26     OLSR(σ,Γ)
27 +
   /* Broadcast the accumulated packet */
28 [Updated ∧ send_time = now]
29     [[send_time := ∞]]
30     broadcast(pkt).
31     [[pkt := [ ]]]
32     OLSR(σ,Γ)
```

A.2.2 Updating the Information Bases

Changes to the router's information bases may trigger updates such as those defined in section 13 of RFC 6130 [7]. Whenever an update is triggered, the UPDATE_INFO process is called to restore the consistency of these information bases. On lines 1-4, expired tuples are removed from the link set, 2-hop set, advertising remove router set and router topology set. The functions purgeLinkSet and purge2HopSet perform additional steps to maintain consistency, with purgeLinkSet resetting the MPR statuses of non-symmetric tuples and purge2HopSet removing all tuples without a symmetric neighbour. Next, a set of valid flooding MPRs and a set of valid routing MPRs are non-deterministically chosen by lines 5 and 7. The link set is updated to use these MPR sets on lines 6 and 8 iff the currently chosen MPR sets are not valid. On line 9, we increment the advertising neighbour sequence number to indicate a change in advertised information iff the set of routing MPR selectors has changed. We then assign prev_ls to the current link set so that we can test for changes in the future. Finally, the routing set is updated on lines 11 and 12 iff it does not use optimal routes to all known destinations.

Process 2: Update the information bases to maintain consistency

```
   UPDATE_INFO(σ,Γ) ≝
   /* Remove expired tuples, perform additional consistency checks */
1  ⟦ ls := purgeLinkSet(ls,now) ⟧
2  ⟦ 2hs := purge2HopSet(ls,2hs,now) ⟧
3  ⟦ arrs := purgeAdvertisingRouters(arrs,now) ⟧
4  ⟦ rts := purgeRouterTopology(rts,now) ⟧
   /* Update the router's flooding and routing MPRs if necessary */
5  [Let fmprs ∈ validFMPRs(ls,2hs,now)]
6  ⟦ ls := updateFMPRs(ls,2hs,now,fmprs) ⟧
7  [Let rmprs ∈ validRMPRs(ls,2hs,now)]
8  ⟦ ls := updateRMPRs(ls,2hs,now,rmprs) ⟧
   /* Increment the advertising neighbour sequence number if the routing MPR selectors have changed */
9  ⟦ ansn := incrementANSN(ls,prev_ls,ansn) ⟧
10 ⟦ prev_ls := ls ⟧
   /* If the current routing set is invalid, update it */
11 [Let rs′ ∈ optimalRoutingSets(ip,ls,rts,now)]
12 ⟦ rs := updateRoutingSet(ip,ls,rts,now,rs,rs′) ⟧
13 OLSR(σ,Γ)
```

A.2.3 Processing a HELLO Message

When a new HELLO message is received by a router, its link set and 2-hop set must be updated. We perform the updates to the link set in the first half of PROCESS_HELLO. First, we non-deterministically choose an incoming link metric from the router sending the message to the receiving router. In fact, this link metric is usually determined by the quality of the link, as measured by the receiving router. However, the RFC does not specify how this is done, and therefore we simply model this as a nondeterministic choice. A new tuple using this incoming metric is then created with an originator address equal to the sending router iff such a tuple did not already exist in the link set. The link tuple for the sending router,

existing or new, is then updated based on the contents of the message. First, line 3 updates the outgoing metric of the tuple to the incoming metric measured at the sending router iff this element was advertised in the message. Next, the symmetric, heard and expiry times of the tuple are modified on lines 4, 5 and 6. Both the heard and expiry times will be made valid for at least the validity time of the message, whereas the symmetric time will be updated based on whether the router found itself advertised as a neighbour in the HELLO message. The router also determines whether it has been selected as a flooding MPR or a routing MPR from the contents of the message.

In the second half of the process, the 2-hop set is updated to ensure that 2-hop tuples exist for all neighbours of the sending router. On line 9, new 2-hop tuples are created for neighbours besides the receiving router if they do not already exist. Next, the metrics and validity time of these tuples are updated. All of these functions require there to exist a symmetric link tuple for the sending router. If not, then the assignments have no effect.

Process 3: Process a HELLO message

$\texttt{PROCESS\_HELLO}(\sigma,\Gamma,\texttt{msg}) \stackrel{\text{def}}{=}$

/* Update the link set */

1 [Let $\texttt{in\_metric} \neq \infty$]

2 $[\![\texttt{ls} := \texttt{addLinkTuple}(\texttt{ls},\texttt{oip}(\texttt{msg}),\texttt{vtime}(\texttt{msg}),\texttt{in\_metric},\texttt{now})]\!]$

3 $[\![\texttt{ls} := \texttt{updateLinkOutMetrics}(\texttt{ip},\texttt{ls},\texttt{oip}(\texttt{msg}),\texttt{inMetrics}(\texttt{msg}))]\!]$

4 $[\![\texttt{ls} := \texttt{updateSymmetricTime}(\texttt{ip},\texttt{ls},\texttt{oip}(\texttt{msg}),\texttt{vtime}(\texttt{msg}),\texttt{statuses}(\texttt{msg}),\texttt{l\_hold\_time},\texttt{now})]\!]$

5 $[\![\texttt{ls} := \texttt{updateHeardTime}(\texttt{ls},\texttt{oip}(\texttt{msg}),\texttt{vtime}(\texttt{msg}),\texttt{now})]\!]$

6 $[\![\texttt{ls} := \texttt{updateValidityTime}(\texttt{ls},\texttt{oip}(\texttt{msg}),\texttt{l\_hold\_time},\texttt{now})]\!]$

7 $[\![\texttt{ls} := \texttt{updateFMPRSelectors}(\texttt{ip},\texttt{ls},\texttt{oip}(\texttt{msg}),\texttt{statuses}(\texttt{msg}),\texttt{mprs}(\texttt{msg}),\texttt{now})]\!]$

8 $[\![\texttt{ls} := \texttt{updateRMPRSelectors}(\texttt{ip},\texttt{ls},\texttt{oip}(\texttt{msg}),\texttt{statuses}(\texttt{msg}),\texttt{mprs}(\texttt{msg}),\texttt{now})]\!]$

/* Update the 2-hop set */

9 $[\![\texttt{2hs} := \texttt{add2HopTuples}(\texttt{ip},\texttt{ls},\texttt{2hs},\texttt{oip}(\texttt{msg}),\texttt{statuses}(\texttt{msg}),\texttt{now})]\!]$

10 $[\![\texttt{2hs} := \texttt{update2HopInMetrics}(\texttt{ls},\texttt{2hs},\texttt{oip}(\texttt{msg}),\texttt{inMetrics}(\texttt{msg}),\texttt{now})]\!]$

11 $[\![\texttt{2hs} := \texttt{update2HopOutMetrics}(\texttt{ls},\texttt{2hs},\texttt{oip}(\texttt{msg}),\texttt{outMetrics}(\texttt{msg}),\texttt{now})]\!]$

12 $[\![\texttt{2hs} := \texttt{update2HopTime}(\texttt{ip},\texttt{ls},\texttt{2hs},\texttt{oip}(\texttt{msg}),\texttt{vtime}(\texttt{msg}),\texttt{statuses}(\texttt{msg}),\texttt{now})]\!]$

13 $\texttt{OLSR}(\sigma,\Gamma)$

A.2.4 Processing a TC Message

TC messages must also be processed once received, subject to a few additional checks. Firstly, the message must be discarded if it was originated by the receiving router. This check is required because TC messages, unlike HELLO message, are flooded through the network and may eventually reach the router that generated them. Processing is also optional if the message was not received from a known symmetric neighbour of the router, i.e. there is no symmetric link tuple for the sending router.

If the guard on line 7 is true, then two additional checks must be performed prior to processing. Firstly, we check whether or not the message has been processed before. This is determined by the guard on line 8, which asserts that there is a processed tuple with the same originator address and message sequence number as the received message. If there is no such tuple, we create a new one for the message and add it to the processed set. The last guard, located on line 13, checks the advertising neighbour

sequence number of the message and determines whether a message from the same router with a greater advertising neighbour sequence number has already been received. If not, we create an advertising remote router tuple and record the advertised links in the router topology set.

Process 4: Process a TC message

```
   PROCESS_TC(σ,Γ,msg) ≝
   /* If the message was originated by this router, then discard it */
1  [oip(msg) = ip]
2      OLSR(σ,Γ)
3  +
   /* If the message was not received from a known symmetric neighbour, then processing is optional */
4  [oip(msg) ≠ ip ∧ ∀lt ∈ ls. L_oip(lt) ≠ sip(msg) ∨ L_status(lt,now) ≠ SYMMETRIC]
5      FORWARD_TC(σ,Γ,msg)
6  +
   /* The message was not originated by this router */
7  [oip(msg) ≠ ip]
       /* If a message with the same originating router and sequence number was received previously, then
          do not process the message */
8      [∃p ∈ ps. P_oip(p) = oip(msg) ∧ P_sqn(p) = sqn(msg)]
9          FORWARD_TC(σ,Γ,msg)
10     +
11     [otherwise]
           /* Mark the message as processed */
12         ⟦ps := addProcessedTuple(ps,oip(msg),sqn(msg))⟧
               /* If the advertising neighbour sequence number included in the message is out of date, then
                  discard the message */
13             [∃ar ∈ arrs. AR_oip(ar) = oip(msg) ∧ AR_sqn(ar) > ansn(msg)]
14                 FORWARD_TC(σ,Γ,msg)
15             +
               /* Process the advertised information */
16             [otherwise]
17                 ⟦arrs :=
                    updateAdvertisingRouters(arrs,oip(msg),ansn(msg),vtime(msg),now)⟧
18                 ⟦rts :=
                    updateRouterTopology(ip,rts,oip(msg),vtime(msg),dests(msg),now)⟧
19                 FORWARD_TC(σ,Γ,msg)
```

A.2.5 Forwarding a TC Message

If a router receives a TC message, then that message will be considered for forwarding iff the message was not originated by the receiving router. The guard on line 1 discards messages that were not received from a known symmetric neighbour. The message is also discarded if the guard on line 5 is true, indicating that the message was received previously. Once these guards are passed, a new received tuple for the message is created to prevent it from being forwarded again in the future. The final guard on line 10 asserts that the router which last forwarded the message is a flooding MPR of this router. In this case, the message should be forwarded. The router then modifies the message's sender IP address to its own IP address and appends it to `pkt` on line 11. Finally, before returning to the main protocol loop, `send_time` is assigned to `now` + 1 to trigger an eventual broadcast of the packet.

Process 5: Forward a TC message

```
   FORWARD_TC(σ,Γ,msg) ≝
   /* If the message was not received from a known symmetric neighbour, then discard it */
1  [∀lt ∈ ls. L_oip(lt) ≠ sip(msg) ∨ L_status(lt,now) ≠ SYMMETRIC]
2      OLSR(σ,Γ)
3  +
4  [otherwise]
       /* If a message with the same originator address and sequence number was received previously, do
          not consider the message for forwarding */
5      [∃rx ∈ rxs. RX_oip(rx) = oip(msg) ∧ RX_sqn(rx) = sqn(msg)]
6          OLSR(σ,Γ)
7      +
8      [otherwise]
           /* Create a received tuple for the message */
9          ⟦rxs := addReceivedTuple(rxs,oip(msg),sqn(msg))⟧
               /* If the message was received from a flooding MPR selector, forward the message */
10             [∃lt ∈ ls. L_oip(lt) = sip(msg) ∧ L_fmpr_selector(lt)]
11                 ⟦pkt := append(pkt,forward(ip,msg))⟧
12                 ⟦send_time := now + 1⟧
13                 OLSR(σ,Γ)
14             +
               /* If the message was not received from a flooding MPR selector, do not forward it */
15             [otherwise]
16                 OLSR(σ,Γ)
```

A.2.6 The Message Queue

The QUEUE process can either receive a message and append it to the queue variable, as on lines 1 and 6, or send a message to the OLSR process, as on line 4. The guard on line 3 asserts that the queue is not empty and assigns the free variables q and qs to its head and tail respectively. The additional receive on line 6 is needed to prevent blocking, such as when the OLSR process cannot currently receive a packet but the QUEUE process can.

Process 6: Message Queue

```
   QUEUE(queue) ≝
   /* Receive a packet from another router and append it to the queue */
1  receive(pkt). QUEUE(append(queue,pkt))
2  +
   /* Packet queue is not empty */
3  [queue = (q : qs)]
       /* Dequeue a packet and send it to the main OLSR process */
4      send(q). QUEUE(qs)
5      +
       /* Receive a packet from another router and append it to the queue */
6      receive(pkt). QUEUE(append(queue,pkt))
```

A.2.7 Initial State and Constraints

The initial state of the system is expressed as an encapsulated parallel composition of a finite number of nodes, each of the form

$$ip : (\xi, \texttt{OLSR}(\sigma,\Gamma) \mathbin{\langle\langle} \zeta, \texttt{QUEUE(queue)}) : R\ .$$

In practice, we restrict the addresses ip of nodes to a finite set **IP** and ensure that each address in **IP** corresponds to exactly one node. Then, $R \subseteq \mathbf{IP}$ represents the nodes currently within range of a router. In a network of $|\mathbf{IP}|$ nodes with minimum broadcast duration $\texttt{LB}$ and maximum broadcast duration $\texttt{LB}+\Delta\texttt{B}$, we use the conjunction of the following three formulas to constrain the initial values of ξ and ζ.

$$\begin{gathered}
\xi(\texttt{ls}) = \emptyset \ \wedge\ \xi(\texttt{2hs}) = \emptyset \ \wedge\ \xi(\texttt{arrs}) = \emptyset \ \wedge\ \xi(\texttt{rts}) = \emptyset \ \wedge \\
\xi(\texttt{rs}) = \emptyset \ \wedge\ \xi(\texttt{ps}) = \emptyset \ \wedge\ \xi(\texttt{rxs}) = \emptyset \ \wedge \xi(\texttt{prev_ls}) = \emptyset \wedge \\
\xi(\texttt{sqn}) = 0 \ \wedge \xi(\texttt{ansn}) = 0 \ \wedge \xi(\texttt{ip}) = ip \ \wedge\ \xi(\texttt{send_time}) = \infty \wedge \\
\xi(\texttt{mqueue}) = [\,] \ \wedge\ \xi(\texttt{pkt}) = [\,] \ \wedge\ \zeta(\texttt{queue}) = [\,]
\end{gathered}$$

$$\begin{gathered}
0 < \texttt{LB} < \texttt{LB}+\Delta\texttt{B} < \xi(\texttt{hp_maxjitter}) < \xi(\texttt{hello_interval}) < \infty \wedge \\
\texttt{LB}+2\Delta\texttt{B}+\xi(\texttt{hello_interval}) < \xi(\texttt{h_hold_time}) < \infty \wedge \\
\texttt{LB}+\Delta\texttt{B} < \xi(\texttt{tp_maxjitter}) < \xi(\texttt{tc_interval}) < \infty \wedge \\
(2(\texttt{LB}+\Delta\texttt{B})+1)(|\mathbf{IP}|-1)-(\texttt{LB}+1)+\xi(\texttt{tc_interval}) < \xi(\texttt{t_hold_time}) < \infty \wedge \\
0 \le \xi(\texttt{l_hold_time}) < \infty
\end{gathered}$$

$$\begin{gathered}
-\infty < \xi(\texttt{now}) < \infty \wedge \\
\xi(\texttt{now}) \le \xi(\texttt{hello_time}) \le \xi(\texttt{now})+\xi(\texttt{hello_interval}) \wedge \\
\xi(\texttt{now}) \le \xi(\texttt{tc_time}) \le \xi(\texttt{now})+\xi(\texttt{tc_interval})
\end{gathered}$$

To prevent information from expiring prematurely, we require messages to be valid until another message of the same type and from the same originating router is received. Under the assumption that links do not change during transmission, a worst-case execution time analysis yields a maximum elapsed time of $\texttt{LB}+2\Delta\texttt{B}$ between HELLO messages and $(2(\texttt{LB}+\Delta\texttt{B})+1)(|\mathbf{IP}|-1)-(\texttt{LB}+1)$ between TC messages. This explains the conditions on the constants `h_hold_time` and `t_hold_time`.

All other variables used in Processes 1–5 (`msgs`, `msg`, `fmprs`, `rmprs`, `rs`$'$, `in_metric` and `x`), as well as the variables `pkt`, `q` and `qs` used in Process 6, are initially undefined.