A Timed Process Algebra for Wireless Networks with an Application in Routing[⋆]

Emile Bres[1,3], Rob van Glabbeek[1,2], and Peter Höfner[1,2]

[1] NICTA, Australia
[2] Computer Science and Engineering, University of New South Wales, Australia
[3] École Polytechnique, Paris, France

Abstract. This paper proposes a timed process algebra for wireless networks, an extension of the Algebra for Wireless Networks. It combines treatments of local broadcast, conditional unicast and data structures, which are essential features for the modelling of network protocols. In this framework we model and analyse the Ad hoc On-Demand Distance Vector routing protocol, and show that, contrary to claims in the literature, it fails to be loop free. We also present boundary conditions for a fix ensuring that the resulting protocol is indeed loop free.

1 Introduction

In 2011 we developed the *Algebra for Wireless Networks* (AWN) [10], a process algebra particularly tailored for Wireless Mesh Networks (WMNs) and Mobile Ad Hoc Networks (MANETs). Such networks are currently being used in a wide range of application areas, such as public safety and mining. They are self-organising wireless multi-hop networks that provide network communication without relying on a wired backhaul infrastructure. A significant characteristic of such networks is that they allow highly dynamic network topologies, meaning that network nodes can join, leave, or move within the network at any moment. As a consequence routing protocols have constantly to check for broken links, and to replace invalid routes by better ones.

To capture the typical characteristics of WMNs and MANETs, AWN offers a unique set of features: *conditional unicast* (a message transmission attempt with different follow-up behaviour depending on its success), *groupcast* (communication to a specific set of nodes), *local broadcast* (messages are received only by nodes within transmission range of the sender), and *data structure*. We are not aware of any other process algebra that provides all these features, and hence could not use any other algebra to model certain protocols for WMNs or MANETs in a straightforward fashion.[1] Case studies [10,11,15,9] have shown that AWN provides the right level of abstraction to model full IETF protocols, such as the Ad hoc On-Demand Distance Vector (AODV) routing protocol [29]. AWN has been employed to formally model this protocol—thereby eliminating ambiguities and contradictions from the official specification, written in English

[⋆] An extended abstract of this paper—everything but the appendices—appeared as [5].

[1] A comparison between AWN and other process algebras can be found in [11, Sect. 11].

Prose—and to reason about protocol behaviour and provide rigorous proofs of key protocol properties such as loop freedom and route correctness.

However, AWN abstracts from time. Analysing routing protocols without considering timing issues is useful in its own right; for AODV it has revealed many shortcomings in drafts as well as in the standard (e.g., [3,19,16]). Including time in a formal analysis, however, will pave the way to analyse protocols that repeat some procedures every couple of time units; examples are OLSR [7] and B.A.T.M.A.N. [26]. Even for a reactive protocol such as AODV, which does not schedule tasks regularly, it has been shown that timing aspects are important: if timing parameters are chosen poorly, some routes are not established since data that is stored locally at network nodes expires too soon and is erased [6]. Besides such shortcomings in "performance", also fundamental correctness properties like loop freedom can be affected by the treatment of time—as we will illustrate.

To enable time analyses of WMNs and MANETs, this paper proposes a *Timed (process) Algebra for Wireless Networks* (T-AWN), an extension of AWN. It combines AWN's unique set of features, such as local broadcast, with time.

In this framework we model and analyse the AODV routing protocol, and show that, contrary to claims in the literature, e.g., [30], it fails to be loop free, as data required for routing can expire. We also present boundary conditions for a fix ensuring that the resulting protocol is loop free.

Design Decisions

Prior to the development of T-AWN we had to make a couple of decisions.

Intranode computations. In wireless networks sending a packet from one node to another takes multiple microseconds. Compared to these "slow" actions, time spent for internal (intranode) computations, such as variable assignments or evaluations of expressions, is negligible. We therefore postulate that only transmissions from one node to another take time.

This decision is debatable for processes that can perform infinite sequences of intranode computations without ever performing a durational action. In this paper (and in all applications), we restrict ourselves to *well-timed* processes in the spirit of [27], i.e., to processes where any infinite sequence of actions contains infinitely many time steps or infinitely many input actions, such as receiving an incoming packet.

But, in the same spirit as T-AWN assigns time to internode communications, it is more or less straightforward to assign times to other operations as well.

Guaranteed Message Receipt and Input Enabledness. A fundamental assumption underlying the semantics of (T-)AWN is that any broadcast message *is* received by all nodes within transmission range [11, §1].[2] This abstraction enables us to

[2] In reality, communication is only half-duplex: a single-interface network node cannot receive messages while sending and hence messages can be lost. However, the CSMA protocol used at the link layer—not modelled by (T-)AWN—keeps the probability of packet loss due to two nodes (within range) sending at the same time rather low.

interpret a failure of route discovery (as documented for AODV in [11, §9]) as an imperfection in the protocol, rather than as a result of a chosen formalism not ensuring guaranteed receipt.

A consequence of this design decision is that in the operational semantics of (T-)AWN a broadcast of one node in a network needs to synchronise with some (in)activity of all other nodes in the network [11, §11]. If another node is within transmission range of the broadcast, the broadcast synchronises with a receive action of that node, and otherwise with a non-arrive transition, which signals that the node is out of range for this broadcast [11, §4.3].

A further consequence is that we need to specify our nodes in such a way that they are *input-enabled*, meaning that in any state they are able to receive messages from any other node within transmission range.

Since a transmission (broadcast, groupcast, or unicast) takes multiple units of time, we postulate that another node can only receive a message if it remains within transmission range during the whole period of sending.[3] A possible way to model the receive action that synchronises with a transmission such as a broadcast is to let it take the same amount of time as the broadcast action. However, a process that is busy executing a durational receive action would fail to be input-enabled, for it would not be able to start receiving another message before the ongoing message receipt is finished. For this reason, we model the receipt of a message as an instantaneous action that synchronises with the very end of a broadcast action.[4]

T-AWN Syntax. 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, we use the syntax of AWN also for T-AWN; “extended” by a local timer `now`. Hence we can perform a timed analysis of any specification written in AWN, since they are also T-AWN specifications.

2 A Timed Process Algebra for Wireless Networks

In this section we propose T-AWN (Timed Algebra for Wireless Networks), an extension of the process algebra AWN [10,11] with time. AWN itself is a variant of standard process algebras [23,18,2,4], tailored to protocols in wireless mesh networks, such as the Ad-hoc on Demand Distance Vector (AODV) routing protocol. In (T-)AWN, a WMN is modelled as an encapsulated parallel composition

[3] To be precise, we forgive very short interruptions in the connection between two nodes—those that begin and end within the same unit of time.

[4] Another solution would be to assume that a broadcast-receiving process can receive multiple messages in parallel. In case the process is meant to add incoming messages to a message queue (as happens in our application to AODV), one can assume that a message that is being received in parallel is added to that queue as soon as its receipt is complete. However, such a model is equivalent to one in which only the very last stage of the receipt action is modelled.

of network nodes. On each node several sequential processes may be running in parallel. Network nodes communicate with their direct neighbours—those nodes that are in transmission range—using either broadcast, groupcast or unicast. Our formalism maintains for each node the set of nodes that are currently in transmission range. Due to mobility of nodes and variability of wireless links, nodes can move in or out of transmission range. The encapsulation of the entire network inhibits communications between network nodes and the outside world, with the exception of the receipt and delivery of data packets from or to clients[5] of the modelled protocol that may be hooked up to various nodes.

In T-AWN we apply a discrete model of time, where each sequential process maintains a local variable `now` holding its local clock value—an integer. We employ only one clock for each sequential process. All sequential processes in a network synchronise in taking time steps, and at each time step all local clocks advance by one 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-dependant, and any value can be assigned to it, thereby resetting the local clock.

In our model of a sequential process p running on a node, time can elapse only when p is transmitting a message to another node, or when p currently has no way to proceed—for instance, when waiting on input, or for its local clock to reach a specified value. All other actions of p, such as assigning values to variables, evaluating guards, communicating with other processes running on the same node, or communicating with clients of the modelled protocol hooked up at that node, are assumed to be an order of magnitude faster, and in our model take no time at all. Thus they are executed in preference to time steps.

2.1 The Syntax of T-AWN

The syntax of T-AWN is the same as the syntax of AWN [10,11], except for the presence of the variable `now` of the new type `TIME`. This brings the advantage that any specification written in AWN can be interpreted and analysed in a timed setting. The rest of this Section 2.1 is almost copied verbatim from the original articles about AWN [10,11].

A Language for Sequential Processes. The internal state of a process is determined, in part, by the values of certain data variables that are maintained by that process. To this end, we assume a data structure 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.[6] Our data structure always contains the types `TIME`, `DATA`, `MSG`, `IP` and $\mathscr{P}$(`IP`) of *time values*, which we take to be integers (together with the special value ∞), *application layer data*, *messages*, *IP addresses*—or

[5] The application layer that initiates packet sending and/or awaits receipt of a packet.

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

any other node identifiers—and *sets of IP addresses.* We further assume that there is a variable $\texttt{now}$ of type $\texttt{TIME}$ and a function $\texttt{newpkt} : \texttt{DATA} \times \texttt{IP} \rightarrow \texttt{MSG}$ that generates a message with new application layer data for a particular destination. The purpose of this function is to inject data into the protocol; details will be given later.

In addition, we assume a type $\texttt{SPROC}$ of *sequential processes*, and a collection of *process names*, each being an operator of type $\texttt{TYPE}_1 \times \cdots \times \texttt{TYPE}_n \rightarrow \texttt{SPROC}$ for certain data types $\texttt{TYPE}_i$. Each process name X comes with a *defining equation*

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

in which, for each $i = 1, \ldots, n$, $\texttt{var}_i$ is a variable of type $\texttt{TYPE}_i$ and p a *guarded*[7] *sequential process expression* defined by the grammar below. The expression p may contain the variables $\texttt{var}_i$ as well as X; however, all occurrences of data variables in p have to be *bound.* The choice of the underlying data structure and the process names with their defining equations can be tailored to any particular application of our language; our decisions made for modelling AODV are presented in Section 3. The process names are used to denote the processes that feature in this application, with their arguments $\texttt{var}_i$ binding the current values of the data variables maintained by these processes.

The *sequential process expressions* are given by the following grammar:

$$\begin{aligned} \mathit{SP} \ &::= X(\mathit{exp}_1, \ldots, \mathit{exp}_n) \ \mid\ [\varphi]\mathit{SP} \ \mid\ [\![\texttt{var} := \mathit{exp}]\!]\mathit{SP} \ \mid\ \mathit{SP} + \mathit{SP} \ \mid \\ &\quad\ \ \alpha.\mathit{SP} \ \mid\ \mathbf{unicast}(\mathit{dest}, \mathit{ms}).\mathit{SP} \blacktriangleright \mathit{SP} \\ \alpha \ &::= \mathbf{broadcast}(\mathit{ms}) \ \mid\ \mathbf{groupcast}(\mathit{dests}, \mathit{ms}) \ \mid\ \mathbf{send}(\mathit{ms}) \ \mid \\ &\quad\ \ \mathbf{deliver}(\mathit{data}) \ \mid\ \mathbf{receive}(\texttt{msg}) \end{aligned}$$

Here X is a process name, exp_i a data expression of the same type as $\texttt{var}_i$, φ a data formula, $\texttt{var} := \mathit{exp}$ an assignment of a data expression exp to a variable $\texttt{var}$ of the same type, *dest*, *dests*, *data* and *ms* data expressions of types $\texttt{IP}$, $\mathscr{P}(\texttt{IP})$, $\texttt{DATA}$ and $\texttt{MSG}$, respectively, and $\texttt{msg}$ a data variable of type $\texttt{MSG}$.

The internal state of a sequential process described by an expression p in this language is determined by p, together with a *valuation* ξ associating data values $\xi(\texttt{var})$ to the data variables $\texttt{var}$ maintained by this process. Valuations naturally extend to *ξ-closed* data expressions—those in which all variables are either bound or in the domain of ξ.

Given a valuation of the data variables by concrete data values, the sequential process $[\varphi]p$ acts as p if φ evaluates to $\texttt{true}$, and deadlocks if φ evaluates to $\texttt{false}$. 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 sequential process $[\![\texttt{var} := \mathit{exp}]\!]p$ acts as p, but under an updated valuation of the data variable $\texttt{var}$. The sequential process $p + q$ may act either as p or as q, depending on which of the two processes is able to act at all. In a context where both are able to act, it is not specified how the choice is made. The sequential process $\alpha.p$ first performs the action α and subsequently

[7] An expression p is *guarded* if each call of a process name $X(\mathit{exp}_1, \ldots, \mathit{exp}_n)$ occurs with a subexpression $[\varphi]q$, $[\![\texttt{var} := \mathit{exp}]\!]q$, $\alpha.q$ or $\mathbf{unicast}(\mathit{dest}, \mathit{ms}).q \blacktriangleright r$ of p.

acts as p. The action **broadcast**(ms) broadcasts (the data value bound to the expression) ms to the other network nodes within transmission range, whereas **unicast**$(dest, ms).p \blacktriangleright q$ is a sequential process that tries to unicast the message ms to the destination $dest$; if successful it continues to act as p and otherwise as q. In other words, **unicast**$(dest, ms).p$ is prioritised over q; only if the action **unicast**$(dest, ms)$ is not possible, the alternative q will happen. It 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 relevant wireless standards such as IEEE 802.11 [20]. 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. Unlike **unicast** and **broadcast**, the expression **groupcast** does not have a unique counterpart in networking. Depending on the protocol and the implementation it can be an iterative unicast, a broadcast, or a multicast; thus **groupcast** abstracts from implementation details. The action **send**(ms) synchronously transmits a message to another process running on the same network node; this action can occur only when this other sequential process is able to receive the message. The sequential process **receive**$(\mathtt{msg}).p$ receives any message m (a data value of type $\mathtt{MSG}$) either from another node, from another sequential process running on the same node or from the client hooked up to the local node. It then proceeds as p, but with the data variable $\mathtt{msg}$ bound to the value m. The submission of data from a client is modelled by the receipt of a message $\mathtt{newpkt}(d, dip)$, where the function $\mathtt{newpkt}$ generates a message containing the data d and the intended destination dip. Data is delivered to the client by **deliver**$(data)$.

A Language for Parallel Processes. *Parallel process expressions* are given by the grammar

$$PP ::= \xi, SP \mid PP \langle\!\langle PP ,$$

where SP is a sequential process expression and ξ a valuation. An expression ξ, p denotes a sequential process expression equipped with a valuation of the variables it maintains. The process $P \langle\!\langle Q$ is a parallel composition of P and Q, running on the same network node. An action **receive**(m) of P synchronises with an action **send**(m) of Q into an internal action τ, as formalised in Table 2. These receive actions of P and send actions of Q cannot happen separately. All other actions of P and Q, except time steps, including receive actions of Q and send actions of P, occur interleaved in $P \langle\!\langle Q$. Therefore, a parallel process expression denotes a parallel composition of sequential processes ξ, P with information flowing from right to left. The variables of different sequential processes running on the same node are maintained separately, and thus cannot be shared.

Though $\langle\!\langle$ only allows information flow in one direction, it reflects reality of WMNs. Usually two sequential processes run on the same node: $P \langle\!\langle Q$. The main process P deals with all protocol details of the node, e.g., message handling and maintaining the data such as routing tables. The process Q manages the queueing of messages as they arrive; it is always able to receive a message even if P is busy. The use of message queueing in combination with $\langle\!\langle$ is crucial in order to create input-enabled nodes (cf. Section 1).

A Language for Networks. We model network nodes in the context of a wireless mesh network by *node expressions* of the form $ip : PP : R$. Here $ip \in \mathtt{IP}$ is the *address* of the node, PP is a parallel process expression, and $R \subseteq \mathtt{IP}$ is the *range* of the node—the set of nodes that are currently within transmission range of ip.

A *partial network* is then modelled by a *parallel composition* $\|$ of node expressions, one for every node in the network, and a *complete network* is a partial network within an *encapsulation operator* $[\_]$ that limits the communication of network nodes and the outside world to the receipt and the delivery of data packets to and from the application layer attached to the modelled protocol in the network nodes. This yields the following grammar for network expressions:

$$N ::= [M] \qquad\qquad M ::= \;\; ip : PP : R \;\;\mid\;\; M \| M \;.$$

2.2 The Semantics of T-AWN

As mentioned in the introduction, the transmission of a message takes time. Since our main application assumes wireless links and node mobility, the packet delivery time varies. Hence we assume a minimum time that is required to send a message, as well as an optional extra transmission time. In T-AWN the values of these parameters are given for each type of sending separately: $\mathtt{LB}$, $\mathtt{LG}$, and $\mathtt{LU}$, satisfying $\mathtt{LB}, \mathtt{LG}, \mathtt{LU} > 0$, specify the minimum bound, in units of time, on the duration of a broadcast, groupcast and unicast transmission; the optional additional transmission times are denoted by $\Delta\mathtt{B}$, $\Delta\mathtt{G}$ and $\Delta\mathtt{U}$, satisfying $\Delta\mathtt{B}, \Delta\mathtt{G}, \Delta\mathtt{U} \geq 0$. Adding up these parameters (e.g. $\mathtt{LB}$ and $\Delta\mathtt{B}$) yields maximum transmission times. We allow any execution consistent with these parameters. For all other actions our processes can take we postulate execution times of 0.

Sequential Processes. The structural operational semantics of T-AWN, given in Tables 1–4, is in the style of Plotkin [31] and describes how one internal state can evolve into another by performing an *action.*

A difference with AWN is that some of the transitions are time steps. On the level of node and network expressions they are labelled "tick" and the parallel composition of multiple nodes can perform such a transition iff each of those nodes can—see the third rule in Table 4. On the level of sequential and parallel process expressions, time-consuming transitions are labelled with *wait actions* from $\mathcal{W} = \{\mathrm{w}, \mathrm{ws}, \mathrm{wr}, \mathrm{wrs}\} \subseteq \mathrm{Act}$ and *transmission actions* from $\mathcal{R}\!:\!\mathcal{W} = \{R\!:\!w_1 \mid w_1 {\in} \mathcal{W} \wedge R \subseteq \mathtt{IP}\} \subseteq \mathrm{Act}$. Wait actions $w_1 {\in} \mathcal{W}$ indicate that the system is waiting, possibly only as long as it fails to synchronise on a **receive** action (wr), a **send** action (ws) or both of those (wrs); actions $R\!:\!w_1$ indicate that the system is transmitting a message while the current transmission range of the node is $R \subseteq \mathtt{IP}$. In the operational rule for choice (+) we combine any two wait actions $w_1, w_2 \in \mathcal{W}$ with the operator $\wedge$, which joins the conditions under which these wait actions can occur.

$\wedge$	w	wr	ws	wrs
w	w	wr	ws	wrs
wr	wr	wr	wrs	wrs
ws	ws	wrs	ws	wrs
wrs	wrs	wrs	wrs	wrs

Table 1. Structural operational semantics for sequential process expressions

(bc)	$\xi, \textbf{broadcast}(\mathit{ms}).p \xrightarrow{\tau} \xi, \texttt{IP}:\textbf{*cast}(\xi(\mathit{ms}))[\texttt{LB},\Delta\texttt{B}].p \blacktriangleright p$	(if $\xi(\mathit{ms})\downarrow$)
(gc)	$\xi, \textbf{groupcast}(\mathit{dests}, \mathit{ms}).p \xrightarrow{\tau} \xi, \xi(\mathit{dests}):\textbf{*cast}(\xi(\mathit{ms}))[\texttt{LG},\Delta\texttt{G}].p \blacktriangleright p$	(if $\xi(\mathit{dests})\downarrow$ and $\xi(\mathit{ms})\downarrow$)
(uc)	$\xi, \textbf{unicast}(\mathit{dest}, \mathit{ms}).p \blacktriangleright q \xrightarrow{\tau} \xi, \{\xi(\mathit{dest})\}:\textbf{*cast}(\xi(\mathit{ms}))[\texttt{LU},\Delta\texttt{U}].p \blacktriangleright q$	(if $\xi(\mathit{dest})\downarrow$ and $\xi(\mathit{ms})\downarrow$)
(tr)	$\xi, \mathit{dsts}:\textbf{*cast}(m)[n{+}1,o].p \blacktriangleright q \xrightarrow{R:\mathrm{w}} \xi[\texttt{now++}], (\mathit{dsts}\cap R):\textbf{*cast}(m)[n,o].p \blacktriangleright q$	$(\forall R \subseteq \texttt{IP})$
(tr-o)	$\xi, \mathit{dsts}:\textbf{*cast}(m)[n{+}1,o{+}1].p \blacktriangleright q \xrightarrow{R:\mathrm{w}} \xi[\texttt{now++}], (\mathit{dsts}\cap R):\textbf{*cast}(m)[n{+}1,o].p \blacktriangleright q$	$(\forall R \subseteq \texttt{IP})$
(sc)	$\xi, \mathit{dsts}:\textbf{*cast}(m)[0,o].p \blacktriangleright q \xrightarrow{\mathit{dsts}:\textbf{*cast}(m)} \xi, p$	(if $\mathit{dsts} \neq \emptyset$)
(¬sc)	$\xi, \mathit{dsts}:\textbf{*cast}(m)[0,o].p \blacktriangleright q \xrightarrow{\mathit{dsts}:\textbf{*cast}(m)} \xi, q$	(if $\mathit{dsts} = \emptyset$)
(snd)	$\xi, \textbf{send}(\mathit{ms}).p \xrightarrow{\textbf{send}(\xi(\mathit{ms}))} \xi, p$	(if $\xi(\mathit{ms})\downarrow$)
(ws)	$\xi, \textbf{send}(\mathit{ms}).p \xrightarrow{\mathrm{ws}} \xi[\texttt{now++}], \textbf{send}(\mathit{ms}).p$	(if $\xi(\mathit{ms})\downarrow$)
(del)	$\xi, \textbf{deliver}(\mathit{data}).p \xrightarrow{\textbf{deliver}(\xi(\mathit{data}))} \xi, p$	(if $\xi(\mathit{data})\downarrow$)
(rcv)	$\xi, \textbf{receive}(\texttt{msg}).p \xrightarrow{\textbf{receive}(m)} \xi[\texttt{msg} := m], p$	$(\forall m \in \texttt{MSG})$
(wr)	$\xi, \textbf{receive}(\texttt{msg}).p \xrightarrow{\mathrm{wr}} \xi[\texttt{now++}], \textbf{receive}(\texttt{msg}).p$	
(ass)	$\xi, [\![\texttt{var} := \mathit{exp}]\!]p \xrightarrow{\tau} \xi[\texttt{var} := \xi(\mathit{exp})], p$	(if $\xi(\mathit{exp})\downarrow$)
(w)	$\xi, p \xrightarrow{\mathrm{w}} \xi[\texttt{now++}], p$	(if $\xi(p)\uparrow$)
(rec)	$\dfrac{\emptyset[\texttt{var}_i := \xi(\mathit{exp}_i)]_{i=1}^{n}, p \xrightarrow{a} \zeta, p'}{\xi, X(\mathit{exp}_1,\dots,\mathit{exp}_n) \xrightarrow{a} \zeta, p'}$	$(X(\texttt{var}_1,\dots,\texttt{var}_n) \stackrel{\mathit{def}}{=} p)$ ($\forall a \in \mathrm{Act} - \mathcal{W}$, if $\xi(\mathit{exp}_i)\downarrow$)
(rec-w)	$\dfrac{\emptyset[\texttt{var}_i := \xi(\mathit{exp}_i)]_{i=1}^{n}, p \xrightarrow{w_1} \zeta, p'}{\xi, X(\mathit{exp}_1,\dots,\mathit{exp}_n) \xrightarrow{w_1} \xi[\texttt{now++}], X(\mathit{exp}_1,\dots,\mathit{exp}_n)}$	$(X(\texttt{var}_1,\dots,\texttt{var}_n) \stackrel{\mathit{def}}{=} p)$ ($\forall w_1 {\in} \mathcal{W}$, if $\xi(\mathit{exp}_i)\downarrow$)
(grd)	$\dfrac{\xi \xrightarrow{\varphi} \zeta}{\xi, [\varphi]p \xrightarrow{\tau} \zeta, p}$	
(¬grd)	$\dfrac{\xi \stackrel{\varphi}{\not\to}}{\xi, [\varphi]p \xrightarrow{\mathrm{w}} \xi[\texttt{now++}], [\varphi]p}$	
(alt-l)	$\dfrac{\xi, p \xrightarrow{a} \zeta, p'}{\xi, p+q \xrightarrow{a} \zeta, p'}$	$(\forall a \in \mathrm{Act} - \mathcal{W})$
(alt-r)	$\dfrac{\xi, q \xrightarrow{a} \zeta, q'}{\xi, p+q \xrightarrow{a} \zeta, q'}$	
(alt-w)	$\dfrac{\xi, p \xrightarrow{w_1} \zeta, p' \quad \xi, q \xrightarrow{w_2} \zeta, q'}{\xi, p+q \xrightarrow{w_1 \wedge w_2} \zeta, p'+q'}$	$(\forall w_1, w_2 \in \mathcal{W})$

In Table 1, which gives the semantics of sequential process expressions, a state is given as a pair ξ, p of a sequential process expression p and a valuation ξ of the data variables maintained by p. The set Act of actions that can be executed by sequential and parallel process expressions, and thus occurs as transition labels, consists of $R\!:\!\textbf{*cast}(m)$, $\textbf{send}(m)$, $\textbf{deliver}(d)$, $\textbf{receive}(m)$, durational actions w_1 and $R\!:\!w_1$, and internal actions τ, for each choice of $R \subseteq \texttt{IP}$, $m \in \texttt{MSG}$, $d \in \texttt{DATA}$ and $w_1 \in \mathcal{W}$. Here $R\!:\!\textbf{*cast}(m)$ is the action of transmitting the message m, to be received by the set of nodes R, which is the intersection of the set of intended destinations with the nodes that are within transmission range throughout the transmission. We do not distinguish whether this message has been broadcast, groupcast or unicast—the differences show up merely in the value of R.

In Table 1 $\xi[\texttt{var} := v]$ denotes the valuation that assigns the value v to the variable var, and agrees with ξ on all other variables. We use $\xi[\texttt{now}{+}{+}]$ as an abbreviation for $\xi[\texttt{now} := \xi(\texttt{now}){+}1]$, the valuation ξ in which the variable now is incremented by 1. This describes the state of data variables after 1 unit of time elapses, while no other changes in data occurred. The empty valuation $\emptyset$ assigns values to no variables. Hence $\emptyset[\texttt{var}_i := v_i]_{i=1}^{n}$ is the valuation that *only* assigns the values v_i to the variables $\texttt{var}_i$ for $i = 1, \ldots, n$. Moreover, $\xi(\mathit{exp}){\downarrow}$, with *exp* a data expression, is the statement that $\xi(\mathit{exp})$ is defined; this might fail because *exp* contains a variable that is not in the domain of ξ or because *exp* contains a partial function that is given an argument for which it is not defined.

A state ξ, r is *unvalued*, denoted by $\xi(r){\uparrow}$, if r has the form $\textbf{broadcast}(\mathit{ms}).p$, $\textbf{groupcast}(\mathit{dests}, \mathit{ms}).p$, $\textbf{unicast}(\mathit{dest}, \mathit{ms}).p$, $\textbf{send}(\mathit{ms}).p$, $\textbf{deliver}(\mathit{data}).p$, $[\![\texttt{var} := \mathit{exp}]\!]p$ or $X(\mathit{exp}_1, \ldots, \mathit{exp}_n)$ with either $\xi(\mathit{ms})$ or $\xi(\mathit{dests})$ or $\xi(\mathit{dest})$ or $\xi(\mathit{data})$ or $\xi(\mathit{exp})$ or some $\xi(\mathit{exp}_i)$ undefined. From such a state no progress is possible. However, Rule (w) in Table 1 does allow time to progress. We use $\xi(r){\downarrow}$ to denote that a state is not unvalued.

Rule (rec) for process names in Table 1 is motivated and explained in [11, §4.1]. The variant (rec-w) of this rule for wait actions $w_1 \in \mathcal{W}$ has been modified such that the recursion is not yet unfolded while waiting. This simulates the behaviour of AWN where a process is only unwound if the first action of the process can be performed.

In the subsequent rules (grd) and ($\neg$grd) for variable-binding guards $[\varphi]$, the notation $\xi \xrightarrow{\varphi} \zeta$ says that ζ is an extension of ξ that satisfies φ: a valuation that agrees with ξ on all variables on which ξ is defined, and valuates the other variables occurring free in φ, such that the formula φ holds under ζ. All variables not free in φ and not evaluated by ξ are also not evaluated by ζ. Its negation $\xi \stackrel{\varphi}{\not\rightarrow}$ says that no such extension exists, and thus that φ is false in the current state, no matter how we interpret the variables whose values are still undefined. If that is the case, the process $[\varphi]p$ will idle by performing the action w (of waiting) without changing its state, except that the variable now will be incremented.

Example 1. The process $[\![\texttt{timeout} := \texttt{now} + 2]\!][\texttt{now} = \texttt{timeout}]p$ first sets the variable timeout to 2 units after the current time. Then it encounters a guard that evaluates to false, and therefore takes a w-transition, twice. After two time units, the guard evaluates to true and the process proceeds as p.

The process **receive**$(\texttt{msg}).p$ can receive any message m from the environment in which this process is running. As long as the environment does not provide a message, this process will wait. This is indicated by the transition labelled wr in Table 1. The difference between a wr-and a w-transition is that the former can be taken only when the environment does not synchronise with the **receive**-transition. In our semantics any state with an outgoing wr-transition also has an outgoing **receive**-transition (see Theorem 1), which conceptually has priority over the wr-transition. Likewise the transition labelled ws is only enabled in states that also admit a **send**-transition, and is taken only in a context where the **send**-transition cannot be taken.

Rules (alt-l) and (alt-r), defining the behaviour of the choice operator for non-wait actions are standard. Rule (alt-w) for wait actions says that a process $p+q$ can wait only if both p and q can wait; if one of the two arguments can make real progress, the choice process $p+q$ always chooses this progress over waiting. This is a direct generalisation of the law $p + \mathbf{0} = p$ of CCS [23]. As a consequence, a condition on the possibility of p or q to wait is inherited by $p+q$. This gives rise to the transition label wrs, that makes waiting conditional on the environment failing to synchronising with a **receive** as well as a **send**-transition. In understanding the target $\zeta, p'{+}q'$ of this rule, it is helpful to realise that whenever $\xi, p \xrightarrow{w_1} \zeta, q$, then $q = p$ and $\zeta = \xi[\texttt{now}{+}{+}]$; see Proposition 1.

In order to give semantics to the transmission constructs (broadcast, groupcast, unicast), the language of sequential processes is extended with the auxiliary construct

$$\mathit{dsts}: {}^{*}\mathbf{cast}(m)[n,o].\mathit{SP} \blacktriangleright \mathit{SP}\ ,$$

with $m \in \texttt{MSG}$, $n, o \in \mathbb{N}$ and $\mathit{dsts} \subseteq \texttt{IP}$. This is a variant of the **broadcast**-, **groupcast**- and **unicast**-constructs, describing intermediate states of the transmission of message m. The argument dsts of ***cast** denotes those intended destinations that were not out of transmission range during the part of the transmission that already took place.

In a state $\mathit{dsts}: {}^{*}\mathbf{cast}(m)[n,o].p \blacktriangleright q$ with $n > 0$ the transmission still needs between n and $n{+}o$ time units to complete. If $n = 0$ the actual ***cast**-transition will take place; resulting in state p if the message is delivered to at least one node in the network (dsts is non-empty), and q otherwise.

Rule (gc) says that once a process commits to a **groupcast**-transmission, it is going to behave as $\mathit{dsts}: {}^{*}\mathbf{cast}(m)[n,o]$ with time parameters $n := \texttt{LG}$ and $o := \Delta\texttt{G}$. The transmitted message m is calculated by evaluating the argument ms, and the transmission range dsts of this ***cast** is initialised by evaluating the argument dests, indicating the intended destinations of the **groupcast**. Rules (bc) and (uc) for **broadcast** and **unicast** are the same, except that in the case of **broadcast** the intended destinations are given by the set IP of *all* possible destinations, whereas a **unicast** has only one intended destination. Moreover, only **unicast** exploits the difference in the continuation process depending on whether an intended destination is within transmission range. Subsequently, Rules (tr) and (tr-o) come into force; they allow time-consuming transmission steps to take place, each decrementing one of the time parameters n or o. Each time step of a transmission corresponds to a transition labelled $R\!:\!\mathrm{w}$, where R records the

Table 2. Structural operational semantics for parallel process expressions

$$\textsf{(p-al)}\ \frac{P \xrightarrow{a} P'}{P \langle\!\langle Q \xrightarrow{a} P' \langle\!\langle Q} \left(\begin{array}{l}\forall a \neq \textbf{receive}(m),\\ a \notin \mathcal{W}, a \notin \mathcal{R}:\mathcal{W}\end{array}\right) \qquad \textsf{(p-ar)}\ \frac{Q \xrightarrow{a} Q'}{P \langle\!\langle Q \xrightarrow{a} P \langle\!\langle Q'} \left(\begin{array}{l}\forall a \neq \textbf{send}(m),\\ a \notin \mathcal{W}, a \notin \mathcal{R}:\mathcal{W}\end{array}\right)$$

$$\textsf{(p-a)}\ \frac{P \xrightarrow{\textbf{receive}(m)} P' \quad Q \xrightarrow{\textbf{send}(m)} Q'}{P \langle\!\langle Q \xrightarrow{\tau} P' \langle\!\langle Q'} \ (\forall m \in \texttt{MSG}) \qquad \textsf{(p-w)}\ \frac{P \xrightarrow{w_1} P' \quad Q \xrightarrow{w_2} Q'}{P \langle\!\langle Q \xrightarrow{w_3} P' \langle\!\langle Q'}$$

$$\textsf{(p-tl)}\ \frac{P \xrightarrow{R:w_1} P' \quad Q \xrightarrow{w_2} Q'}{P \langle\!\langle Q \xrightarrow{R:w_3} P' \langle\!\langle Q'} \qquad \textsf{(p-tr)}\ \frac{P \xrightarrow{w_1} P' \quad Q \xrightarrow{R:w_2} Q'}{P \langle\!\langle Q \xrightarrow{R:w_3} P' \langle\!\langle Q'} \qquad \textsf{(p-t)}\ \frac{P \xrightarrow{R:w_1} P' \quad Q \xrightarrow{R:w_2} Q'}{P \langle\!\langle Q \xrightarrow{R:w_3} P' \langle\!\langle Q'}$$

$$(\forall w_1, w_2, w_3 \in \mathcal{W}, w_3 = w_1 \langle\!\langle w_2)$$

current transmission range. Since sequential processes store no information on transmission ranges—this information is added only when moving from process expressions to node expressions—at this stage of the description all possibilities for the transmission range need to be left open, and hence there is a transition labelled $R\!:\!\mathrm{w}$ for each choice of R.[8] When transitions for process expressions are inherited by node expressions, only one of the transitions labelled $R\!:\!\mathrm{w}$ is going to survive, namely the one where R equals the transmission range given by the node expression (cf. Rule (n-t) in Table 3). Upon doing a transition $R\!:\!\mathrm{w}$, the range *dsts* of the ***cast** is restricted to R. As soon as $n = 0$, regardless of the value of o, the transmission is completed by the execution of the action $\mathit{dsts}\!:\!\textbf{*cast}(m)$ (Rules (sc) and (¬sc)). Here the actual message m is passed on for synchronisation with **receive**-transitions of all nodes $\mathit{ip} \in \mathit{dsts}$.

This treatment of message transmission is somewhat different from the one in AWN. There, the rule $\xi, \textbf{groupcast}(\mathit{dests}, \mathit{ms}).p \xrightarrow{\textbf{groupcast}(\xi(\mathit{dests}),\xi(\mathit{ms}))} \xi, p$ describes the behaviour of the **groupcast** construct for sequential processes, and the rule

$$\frac{P \xrightarrow{\textbf{groupcast}(D,m)} P'}{\mathit{ip} : P : R \xrightarrow{R\cap D\,:\,\textbf{*cast}(m)} \mathit{ip} : P' : R}$$

lifts this behaviour from processes to nodes. In this last stage the **groupcast**-action is unified with the **broadcast**- and **unicast**-action into a ***cast**, at which occasion the range of the ***cast** is calculated as the intersection of the intended destinations D of the **groupcast** and the ones in transmission range R. In T-AWN, on the other hand, the conversion of **groupcast** to ***cast** happens already at the level of sequential processes.

Parallel Processes. Rules (p-al), (p-ar) and (p-a) of Table 2 are taken from AWN, and formalise the description of the operator $\langle\!\langle$ given in Section 2.1. Rule (p-w) stipulates under which conditions a process $P \langle\!\langle Q$ can do a wait action, and of which kind. Here $\langle\!\langle$ is also a partial binary function on the set $\mathcal{W}$, specified by the table on the right. The process $P \langle\!\langle Q$ can do a wait action only if both P and Q can do so. In case P can do a wr or a wrs-action, P can also do a **receive** and in case Q can do a ws or a wrs, Q can also

$\langle\!\langle$	w	wr	ws	wrs
w	w	wr	w	wr
wr	w	wr	–	–
ws	ws	wrs	ws	wrs
wrs	ws	wrs	–	–

[8] Similar to $\textbf{receive}(\texttt{msg}).p$ having a transition for each possible incoming message m.

do a **send**. When both these possibilities apply, the **receive** of P synchronises with the **send** of Q into a τ-step, which has priority over waiting. In the other 12 cases no synchronisation between P and Q is possible, and we do obtain a wait action. Since a **receive**-action of P that does not synchronise with Q is dropped, so is the corresponding side condition of a wait action of P. Hence (within the remaining 12 cases) a wr of P is treated as a w, and a wrs as a ws. Likewise a ws of Q is treated as a w, and a wrs as a wr. This leaves 4 cases to be decided. In all four, we have $\mathrm{w}_1 \langle\!\langle\, \mathrm{w}_2 = \mathrm{w}_1 \wedge \mathrm{w}_2$.

Time steps $R\!:\!w_1$ are treated exactly like wait actions from $\mathcal{W}$ (cf. Rules (p-tl), (p-tr) and (p-t)). If for instance P can do a $R\!:\!\mathrm{w}$, meaning that it spends a unit of time on a transmission, while Q can do a wr, meaning that it waits a unit of time only when it does not receive anything from another source, the result is that $P \langle\!\langle\, Q$ can spend a unit of time transmitting something, but only as long as $P \langle\!\langle\, Q$ does not receive any message; if it does, the receive action of Q happens with priority over the wait action of Q, and thus occurs before P spends a unit of time transmitting.

Node and Network Expressions. The operational semantics of node and network expressions of Tables 3 and 4 uses transition labels tick, $R\!:\!\mathbf{*cast}(m)$, $H\neg K\!:\!\mathbf{arrive}(m)$, $\mathit{ip}\!:\!\mathbf{deliver}(d)$, $\mathbf{connect}(\mathit{ip},\mathit{ip}')$, $\mathbf{disconnect}(\mathit{ip},\mathit{ip}')$, τ and $\mathit{ip}\!:\!\mathbf{newpkt}(d,\mathit{dip})$. As before, $m \in \mathtt{MSG}$, $d \in \mathtt{DATA}$, $R \subseteq \mathtt{IP}$, and $\mathit{ip},\mathit{ip}' \in \mathtt{IP}$. Moreover, $H, K \subseteq \mathtt{IP}$ are sets of IP addresses.

The actions $R\!:\!\mathbf{*cast}(m)$ are inherited by nodes from the processes that run on these nodes (cf. Rule (n-sc)). The action $H\neg K\!:\!\mathbf{arrive}(m)$ states that the message m simultaneously arrives at all addresses $\mathit{ip} \in H$, and fails to arrive at all addresses $\mathit{ip} \in K$. The rules of Table 4 let a $R\!:\!\mathbf{*cast}(m)$-action of one node synchronise with an $\mathbf{arrive}(m)$ of all other nodes, where this $\mathbf{arrive}(m)$ amalgamates the arrival of message m at the nodes in the transmission range R of the $\mathbf{*cast}(m)$, and the non-arrival at the other nodes. Rules (n-rcv) and (n-dis) state that arrival of a message at a node happens if and only if the node receives it, whereas non-arrival can happen at any time. This embodies our assumption that, at any time, any message that is transmitted to a node within range of the sender is actually received by that node. (Rule (n-dis) may appear to say that any node ip has the option to disregard any message at any time. However, the encapsulation operator (below) prunes away all such disregard transitions that do not synchronise with a cast action for which ip is out of range.)

The action $\mathbf{send}(m)$ of a process does not give rise to any action of the corresponding node—this action of a sequential process cannot occur without communicating with a receive action of another sequential process running on the same node. Time-consuming actions w_1 and $R\!:\!w_1$, with $w_1 \in \mathcal{W}$, of a process are renamed into tick on the level of node expressions.[9] All we need to remember of these actions is that they take one unit of time. Since on node expressions the actions $\mathbf{send}(m)$ have been dropped, the side condition making the wait actions

[9] Rule (n-t) ensures that only those $R\!:\!w_1$-transitions survive for which R is the current transmission range of the node.

Table 3. Structural operational semantics for node expressions

(n-sc) $$\frac{P \xrightarrow{dsts:*\textbf{cast}(m)} P'}{ip:P{:}R \xrightarrow{dsts:*\textbf{cast}(m)} ip:P'{:}R}$$

(n-rcv) $$\frac{P \xrightarrow{\textbf{receive}(m)} P'}{ip:P{:}R \xrightarrow{\{ip\}\neg\emptyset:\textbf{arrive}(m)} ip:P'{:}R}$$

(n-del) $$\frac{P \xrightarrow{\textbf{deliver}(d)} P'}{ip:P{:}R \xrightarrow{ip:\textbf{deliver}(d)} ip:P'{:}R}$$

(n-dis) $$ip:P{:}R \xrightarrow{\emptyset\neg\{ip\}:\textbf{arrive}(m)} ip:P{:}R$$

(n-τ) $$\frac{P \xrightarrow{\tau} P'}{ip:P{:}R \xrightarrow{\tau} ip:P'{:}R}$$

(n-w) $$\frac{P \xrightarrow{w_1} P'}{ip:P{:}R \xrightarrow{\text{tick}} ip:P'{:}R}$$

(n-t) $$\frac{P \xrightarrow{R:w_1} P'}{ip:P{:}R \xrightarrow{\text{tick}} ip:P'{:}R}$$

$(\forall w_1 \in \mathcal{W})$

(con) $$ip{:}P{:}R \xrightarrow{\textbf{connect}(ip,ip')} ip{:}P{:}R\cup\{ip'\}$$

(dis) $$ip{:}P{:}R \xrightarrow{\textbf{disconnect}(ip,ip')} ip{:}P{:}R-\{ip'\}$$

Table 4. Structural operational semantics for network expressions

(nw-tl/nw-tr) $$\frac{M \xrightarrow{R:*\textbf{cast}(m)} M' \quad N \xrightarrow{H\neg K:\textbf{arrive}(m)} N'}{M\|N \xrightarrow{R:*\textbf{cast}(m)} M'\|N' \qquad N\|M \xrightarrow{R:*\textbf{cast}(m)} N'\|M'} \quad \left(\begin{array}{c}H \subseteq R,\\ K\cap R = \emptyset\end{array}\right)$$

(arr) $$\frac{M \xrightarrow{H\neg K:\textbf{arrive}(m)} M' \quad N \xrightarrow{H'\neg K':\textbf{arrive}(m)} N'}{M\|N \xrightarrow{(H\cup H')\neg(K\cup K'):\textbf{arrive}(m)} M'\|N'}$$

(tck) $$\frac{M \xrightarrow{\text{tick}} M' \quad N \xrightarrow{\text{tick}} N'}{M\|N \xrightarrow{\text{tick}} M'\|N'}$$

(nw-al) $$\frac{M \xrightarrow{a} M'}{M\|N \xrightarrow{a} M'\|N}$$

(nw-ar) $$\frac{N \xrightarrow{a} N'}{M\|N \xrightarrow{a} M\|N'}$$

(e-a) $$\frac{M \xrightarrow{a} M'}{[M] \xrightarrow{a} [M']}$$

$(\forall a \in \{ip{:}\textbf{deliver}(d), \tau, \textbf{connect}(ip,ip'), \textbf{disconnect}(ip,ip')\})$

(e-tck) $$\frac{M \xrightarrow{\text{tick}} M'}{[M] \xrightarrow{\text{tick}} [M']}$$

(e-sc) $$\frac{M \xrightarrow{R:*\textbf{cast}(m)} M'}{[M] \xrightarrow{\tau} [M']}$$

(e-np) $$\frac{M \xrightarrow{\{ip\}\neg K:\textbf{arrive}(\texttt{newpkt}(d,dip))} M'}{[M] \xrightarrow{ip:\textbf{newpkt}(d,dip)} [M']}$$

ws and wrs conditional on the absence of a **send**-action can be dropped as well. The priority of **receive**-actions over the wait action wr can now also be dropped, for in the absence of **send**-actions, **receive**-actions are entirely reactive. A node can do a **receive**-action only when another node, or the application layer, casts a message, and in this case that other node is not available to synchronise with a tick-transition.

Internal actions τ and the action $ip{:}\textbf{deliver}(d)$ are simply inherited by node expressions from the processes that run on these nodes (Rules (n-τ) and (n-del)), and are interleaved in the parallel composition of nodes that makes up a network. Finally, we allow actions $\textbf{connect}(ip, ip')$ and $\textbf{disconnect}(ip, ip')$ for $ip, ip' \in \texttt{IP}$ modelling a change in network topology. In this formalisation node ip' may be in the range of node ip, meaning that ip can send to ip', even when the reverse does not hold. For some applications, in particular the one to AODV in Section 3, it is useful to assume that ip' is in the range of ip if and only if ip is in the range of ip'. This symmetry can be enforced by adding the following rules to Table 3:

$$ip\!:\!P\!:\!R \xrightarrow{\textbf{connect}(ip',ip)} ip\!:\!P\!:\!R \cup \{ip'\} \qquad ip\!:\!P\!:\!R \xrightarrow{\textbf{disconnect}(ip',ip)} ip\!:\!P\!:\!R - \{ip'\}$$

$$\frac{ip \notin \{ip', ip''\}}{ip\!:\!P\!:\!R \xrightarrow{\textbf{connect}(ip',ip'')} ip\!:\!P\!:\!R} \qquad \frac{ip \notin \{ip', ip''\}}{ip\!:\!P\!:\!R \xrightarrow{\textbf{disconnect}(ip',ip'')} ip\!:\!P\!:\!R}$$

and replacing the rules in the third line of Table 4 for (dis)connect actions by

$$\frac{M \xrightarrow{a} M' \quad N \xrightarrow{a} N'}{M \| N \xrightarrow{a} M' \| N'} \qquad \frac{M \xrightarrow{a} M'}{[M] \xrightarrow{a} [M']} \qquad \left(\forall a \in \left\{ \begin{array}{c} \textbf{connect}(ip, ip'), \\ \textbf{disconnect}(ip, ip') \end{array} \right\}\right).$$

The main purpose of the encapsulation operator is to ensure that no messages will be received that have never been sent. In a parallel composition of network nodes, any action **receive**(m) of one of the nodes ip manifests itself as an action $H\neg K$: **arrive**(m) of the parallel composition, with $ip \in H$. Such actions can happen (even) if within the parallel composition they do not communicate with an action ***cast**(m) of another component, because they might communicate with a ***cast**(m) of a node that is yet to be added to the parallel composition. However, once all nodes of the network are accounted for, we need to inhibit unmatched arrive actions, as otherwise our formalism would allow any node at any time to receive any message. One exception however are those arrive actions that stem from an action **receive**($\texttt{newpkt}(d, dip)$) of a sequential process running on a node, as those actions represent communication with the environment. Here, we use the function `newpkt`, which we assumed to exist.[10] It models the injection of new data d for destination `dip`.

The encapsulation operator passes through internal actions, as well as delivery of data to destination nodes, this being an interaction with the outside world (Rule (e-a)). ***cast**(m)-actions are declared internal actions at this level (Rule (e-sc)); they cannot be steered by the outside world. The connect and disconnect actions are passed through in Table 4 (Rule (e-a)), thereby placing them under control of the environment; to make them nondeterministic, their rules should have a τ-label in the conclusion, or alternatively **connect**(ip, ip') and **disconnect**(ip, ip') should be thought of as internal actions. Finally, actions **arrive**(m) are simply blocked by the encapsulation—they cannot occur without synchronising with a ***cast**(m)—except for $\{ip\}\neg K$: **arrive**($\texttt{newpkt}(d, dip)$) with $d \in \texttt{DATA}$ and $dip \in \texttt{IP}$ (Rule (e-np)). This action represents new data d that is submitted by a client of the modelled protocol to node ip, for delivery at destination dip.

Optional Augmentations to Ensure Non-Blocking Broadcast. Our process algebra, as presented above, is intended for networks in which each node is *input enabled* [21], meaning that it is always ready to receive any message, i.e., able to engage in the transition **receive**(m) for any $m \in \texttt{MSG}$—in the default version of T-AWN, network expressions are required to have this property. In our model of AODV (Section 3) we will ensure this by equipping each node with

[10] To avoid the function `newpkt` we could have introduced a new primitive **newpkt**, which is dual to **deliver**.

a message queue that is always able to accept messages for later handling—even when the main sequential process is currently busy. This makes our model input enabled and hence *non-blocking*, meaning that no sender can be delayed in transmitting a message simply because one of the potential recipients is not ready to receive it.

In [10,11] we additionally presented two versions of AWN without the requirement that all nodes need to be input enabled: one in which we kept the same operational semantics and simply accept blocking, and one were we added operational rules to avoid blocking, thereby giving up on the requirement that any broadcast message is received by all nodes within transmission range.

The first solution does not work for T-AWN, as it would give rise to *time deadlocks*, reachable states where time is unable to progress further.

The second solution is therefore our only alternative to requiring input enabledness for T-AWN. As in [10,11], it is implemented by the addition of the rule

$$\frac{P \overset{\textbf{receive}(m)}{\not\longrightarrow}}{ip : P : R \xrightarrow{\{ip\}\neg\emptyset : \textbf{arrive}(m)} ip : P : R}\ .$$

It states that a message may arrive at a node *ip* regardless whether the node is ready to receive it or not; if it is not ready, the message is simply ignored, and the process running on the node remains in the same state.

In [11, §4.5] also a variant of this idea is presented that avoids negative premises, yet leads to the same transition system. The same can be done to T-AWN in the same way, we skip the details and refer to [11, §4.5].

2.3 Results on the Process Algebra

In this section we list a couple of useful properties of our timed process algebra. In particular, we show that wait actions do not change the data state, except for the value of `now`. Moreover, we show the absence of *time deadlocks*: a complete network N described by T-AWN always admits a transition, independently of the outside environment. More precisely, either $N \xrightarrow{\text{tick}}$, or $N \xrightarrow{ip : \textbf{deliver}(d)}$ or $N \xrightarrow{\tau}$. We also show that our process algebra admits a translation into one without data structure. The operational rules of the translated process algebra are in the de Simone format [33], which immediately implies that strong bisimilarity is a congruence, and yields the associativity of our parallel operators. Last, we show that T-AWN and AWN are related by a simulation relation. Due to lack of space, most of the proofs are omitted; they are deferred to Appendix A.1.

Proposition 1. *On the level of sequential processes, wait actions change only the value of the variable* `now`*, i.e.,* $\xi, p \xrightarrow{w_1} \zeta, q \Rightarrow (p = q \wedge \zeta = \xi[\texttt{now++}])$.

Proof Sketch. One inspects all rules of Table 1 that can generate w-steps, and then reasons inductively on the derivation of these steps.

Similarly, it can be observed that for transmission actions (actions from the set $\mathcal{R}\!:\!\mathcal{W}$) the data state does not change either; the process, however, changes.

That means $\xi, p \xrightarrow{rw} \zeta, q \Rightarrow \zeta = \xi[\texttt{now++}]$ for all $rw \in \mathcal{R}\!:\!\mathcal{W}$. Furthermore, this result can easily be lifted to all other layers of our process algebra (with minor adaptations: for example on node expressions one has to consider tick actions).

To shorten the forthcoming definitions and properties we use the following abbreviations:

1. $P \xrightarrow{\textbf{rcv.}}$ iff $P \xrightarrow{\textbf{receive}(m)}$ for some $m \in \texttt{MSG}$,
2. $P \xrightarrow{\textbf{send}}$ iff $P \xrightarrow{\textbf{send}(m)}$ for some $m \in \texttt{MSG}$,
3. $P \xrightarrow{\textbf{wait}}$ iff $P \xrightarrow{w_1}$ for some $w_1 \in \mathcal{W}$,
4. $P \xrightarrow{\textbf{other}}$ iff $P \xrightarrow{a}$ for some $a \in \mathrm{Act}$ not of the forms above,

where P is a parallel process expression—possibly incorporating the construct $\mathit{dsts}\!:\!\textbf{*cast}(m)[n,o].p$, but never in a +-context. Note that the last line covers also transmission actions $rw \in \mathcal{R}\!:\!\mathcal{W}$. The following result shows that the wait actions of a sequential process (with data evaluation) P are completely determined by the other actions P offers.

Theorem 1. *Let P be a state of a sequential process.*

1. $P \xrightarrow{\mathrm{w}}$ *iff* $P \not\xrightarrow{\textbf{rcv.}} \wedge P \not\xrightarrow{\textbf{send}} \wedge P \not\xrightarrow{\textbf{other}}$.
2. $P \xrightarrow{\mathrm{wr}}$ *iff* $P \xrightarrow{\textbf{rcv.}} \wedge P \not\xrightarrow{\textbf{send}} \wedge P \not\xrightarrow{\textbf{other}}$.
3. $P \xrightarrow{\mathrm{ws}}$ *iff* $P \not\xrightarrow{\textbf{rcv.}} \wedge P \xrightarrow{\textbf{send}} \wedge P \not\xrightarrow{\textbf{other}}$.
4. $P \xrightarrow{\mathrm{wrs}}$ *iff* $P \xrightarrow{\textbf{rcv.}} \wedge P \xrightarrow{\textbf{send}} \wedge P \not\xrightarrow{\textbf{other}}$.

Proof Sketch. The proof is by structural induction. It requires, however, a distinction between guarded terms (as defined in Footnote 7) and unguarded ones.

We could equivalently have omitted all transition rules involving wait actions from Table 1, and defined the wait transitions for sequential processes as described by Theorem 1 and Proposition 1. That our transition rules give the same result constitutes a sanity check of our operational semantics.

Theorem 1 does not hold in the presence of unguarded recursion. A counterexample is given by the expression $X()$ with $X() \overset{def}{=} X()$, for which we would have $X() \not\xrightarrow{\textbf{rcv.}} \wedge X() \not\xrightarrow{\textbf{send}} \wedge X() \not\xrightarrow{\textbf{other}} \wedge X() \not\xrightarrow{\textbf{wait}}$.

Lemma 1. *Let P be a state of a sequential or parallel process. If $P \xrightarrow{R:w_1}$ for some $R \subseteq \texttt{IP}$ and $w_1 \in \mathcal{W}$ then $P \xrightarrow{R':w_1}$ for any $R' \subseteq \texttt{IP}$.*

Observation 1. *Let P be a state of a sequential process. If $P \xrightarrow{R:w_1}$ for some $w_1 \in \mathcal{W}$ then w_1 must be* w *and all outgoing transitions of P are labelled $R'\!:\!\mathrm{w}$.*

For N a (partial) network expression, or a parallel process expression, write $N \xrightarrow{\textbf{inb}}$ iff $N \xrightarrow{a}$ with a of the form $R\!:\!\textbf{*cast}(m)$, $ip\!:\!\textbf{deliver}(d)$ (or $\textbf{deliver}(d)$) or τ—an *instantaneous non-blocking action.* Hence, for a parallel process expression P, $P \xrightarrow{\textbf{other}}$ iff $P \xrightarrow{\textbf{inb}}$ or $P \xrightarrow{R:w_1}$ for $w_1 \in \mathcal{W}$. Furthermore, write $P \xrightarrow{\textbf{time}}$ iff $P \xrightarrow{w_1}$ or $P \xrightarrow{R:w_1}$ for some $w_1 \in \mathcal{W}$. We now lift Theorem 1 to the level of parallel processes.

Theorem 2. *Let P be a state of a parallel process.*

1. $P\xrightarrow{\mathrm{w}} \vee P\xrightarrow{R:\mathrm{w}}$ *iff* $P\not\xrightarrow{\textbf{rcv.}} \wedge P\not\xrightarrow{\textbf{send}} \wedge P\not\xrightarrow{\textbf{inb}}$.
2. $P\xrightarrow{\mathrm{wr}} \vee P\xrightarrow{R:\mathrm{wr}}$ *iff* $P\xrightarrow{\textbf{rcv.}} \wedge P\not\xrightarrow{\textbf{send}} \wedge P\not\xrightarrow{\textbf{inb}}$.
3. $P\xrightarrow{\mathrm{ws}} \vee P\xrightarrow{R:\mathrm{ws}}$ *iff* $P\not\xrightarrow{\textbf{rcv.}} \wedge P\xrightarrow{\textbf{send}} \wedge P\not\xrightarrow{\textbf{inb}}$.
4. $P\xrightarrow{\mathrm{wrs}} \vee P\xrightarrow{R:\mathrm{wrs}}$ *iff* $P\xrightarrow{\textbf{rcv.}} \wedge P\xrightarrow{\textbf{send}} \wedge P\not\xrightarrow{\textbf{inb}}$.

Corollary 1. *Let P be a state of a parallel process. Then $P\xrightarrow{\textbf{time}}$ iff $P\not\xrightarrow{\textbf{inb}}$.* □

Lemma 2. *Let N be a partial network expression with L the set of addresses of the nodes of N. Then $N\xrightarrow{H\neg K:\textbf{\textit{arrive}}(m)}$, for any partition $L = H \uplus K$ of L into sets H and K, and any $m \in \mathtt{MSG}$.*

Using this lemma, we can finally show one of our main results: an (encapsulated) network expression can perform a time-consuming action iff an instantaneous non-blocking action is not possible.

Theorem 3. *Let N be a partial or complete network expression.*
Then $N\xrightarrow{\mathrm{tick}}$ iff $N\not\xrightarrow{\textbf{inb}}$.

Proof. We apply structural induction on N. First suppose N is a node expression $ip\!:\!P\!:\!R$. Then $N\xrightarrow{\mathrm{tick}}$ iff $P\xrightarrow{w_1} \vee P\xrightarrow{R:w_1}$ for some $w_1 \in \mathcal{W}$. By Lemma 1 this is the case iff $P\xrightarrow{w_1} \vee P\xrightarrow{R':w_1}$ for some $R' \subseteq \mathtt{IP}$ and $w_1 \in \mathcal{W}$, i.e., iff $P\xrightarrow{\textbf{time}}$. Moreover $N\xrightarrow{\textbf{inb}}$ iff $P\xrightarrow{\textbf{inb}}$. Hence the claim follows from Corollary 1.

Now suppose N is a partial network expression $M_1\|M_2$. In case $M_i\not\xrightarrow{\textbf{inb}}$ for $i = 1, 2$ then $N\not\xrightarrow{\textbf{inb}}$. By induction $M_i\xrightarrow{\mathrm{tick}}$ for $i = 1, 2$, and hence $N\xrightarrow{\mathrm{tick}}$. Otherwise, $M_i\xrightarrow{\textbf{inb}}$ for $i = 1$ or 2. Now $N\xrightarrow{\textbf{inb}}$. In case $M_i\xrightarrow{\tau}$ or $M_i\xrightarrow{ip:\textbf{deliver}(d)}$ this follows from the third line of Table 4; if $M_i\xrightarrow{R:\textbf{*cast}(m)}$ it follows from the first line, in combination with Lemma 2. By induction $M_i\not\xrightarrow{\mathrm{tick}}$, and thus $N\not\xrightarrow{\mathrm{tick}}$.

Finally suppose that N is a complete network expression $[M]$. By the rules of Table 4 $N\xrightarrow{\mathrm{tick}}$ iff $M\xrightarrow{\mathrm{tick}}$, and $N\xrightarrow{\textbf{inb}}$ iff $M\xrightarrow{\textbf{inb}}$, so the claim follows from the case for partial network expressions. □

Corollary 2. *A complete network N described by T-AWN always admits a transition, independently of the outside environment, i.e., $\forall N, \exists a$ such that $N \xrightarrow{a}$ and $a \notin \{\textbf{\textit{connect}}(ip, ip'), \textbf{\textit{disconnect}}(ip, ip'), \mathtt{newpkt}(d, dip)\}$.*
More precisely, either $N \xrightarrow{\mathrm{tick}}$ or $N \xrightarrow{ip:\textbf{\textit{deliver}}(d)}$ or $N \xrightarrow{\tau}$. □

Our process algebra admits a translation into one without data structures (although we cannot *describe* the target algebra without using data structures). The idea is to replace any variable by all possible values it can take. The target algebra differs from the original only on the level of sequential processes; the subsequent layers are unchanged. A formal definition can be found in Appendix A.2. The resulting process algebra has a structural operational semantics in the (infinitary) *de Simone* format, generating the same transition system—up to strong bisimilarity, $\mathrel{\underline{\leftrightarrow}}$—as the original, which provides some results 'for free'. For example, it follows that $\mathrel{\underline{\leftrightarrow}}$, and many other semantic equivalences, are congruences on our language.

Theorem 4. *Strong bisimilarity is a congruence for all operators of T-AWN.*

This is a deep result that usually takes many pages to establish (e.g., [34]). Here we get it directly from the existing theory on structural operational semantics, as a result of carefully designing our language within the disciplined framework described by de Simone [33].

Theorem 5. $\langle\!\langle$ *is associative, and* $\|$ *is associative and commutative, up to* $\leftrightarroweq$*.*

Proof. The operational rules for these operators fit a format presented in [8], guaranteeing associativity up to $\leftrightarroweq$. The details are similar to the case for AWN, as elaborated in [10,11]; the only extra complication is the associativity of the operator $\langle\!\langle$ on $\mathcal{W}$, as defined on Page 11, which we checked automatically by means of the theorem prover Prover9 [22]. Commutativity of $\|$ follows by symmetry of the rules. □

Theorem 6. *Each AWN process P, seen as a T-AWN process, can be simulated by the AWN process P. Likewise, each AWN network N, seen as a T-AWN network, can be simulated by the AWN network N.*

Here a *simulation* refers to a *weak simulation* as defined in [14], but treating **(dis)connect**-actions as τ, and with the extra requirement that the data states maintained by related expressions are identical—except of course for the variables `now`, that are missing in AWN. Details can be found in Appendix A.3.

Thanks to Theorem 6, we can prove that all invariants on the data structure of a process expressed in AWN are still preserved when the process is interpreted as a T-AWN expression. As an application of this, an untimed version of AODV, formalised as an AWN process, has been proven loop free in [11,15]; the same system, seen as a T-AWN expression—and thus with specific execution times associated to uni-, group-, and broadcast actions—is still loop free when given the operational semantics of T-AWN.

3 Case Study: The AODV Routing Protocol

Routing protocols are crucial to the dissemination of data packets between nodes in WMNs and MANETs. Highly dynamic topologies are a key feature of WMNs and MANETs, due to mobility of nodes and/or the variability of wireless links. This makes the design and implementation of robust and efficient routing protocols for these networks a challenging task. In this section we present a formal specification of the Ad hoc On-Demand Distance Vector (AODV) routing protocol. AODV [29] is a widely-used routing protocol designed for MANETs, and is one of the four protocols currently standardised by the IETF MANET working group[11]. It also forms the basis of new WMN routing protocols, including HWMP in the IEEE 802.11s wireless mesh network standard [20].

[11] `http://datatracker.ietf.org/wg/manet/charter/`

Our formalisation is based on an untimed formalisation of AODV [11,15], written in AWN, and models the exact details of the core functionality of AODV as standardised in IETF RFC 3561 [29]; e.g., route discovery, route maintenance and error handling. We demonstrate how T-AWN can be used to reason about critical protocol properties. As major outcome we demonstrate that AODV is *not* loop free, which is in contrast to common belief. Loop freedom is a critical property for any routing protocol, but it is particularly relevant and challenging for WMNs and MANETs. We close the section by discussing a fix to the protocol and prove that the resulting protocol is indeed loop free.

3.1 Brief Overview

AODV is a reactive protocol, which means that routes are established only on demand. If a node S wants to send a data packet to a node D, but currently does not know a route, it temporarily buffers the packet and initiates a route discovery process by broadcasting a route request (RREQ) message in the network. An intermediate node A that receives the RREQ message creates a routing table entry for a route towards node S referred to as a *reverse route*, and re-broadcasts the RREQ. This is repeated until the RREQ reaches the destination node D, or alternatively a node that knows a route to D. In both cases, the node replies by unicasting a corresponding route reply (RREP) message back to the source S, via a previously established reverse route. When forwarding RREP messages, nodes create a routing table entry for node D, called the *forward route*. When the RREP reaches the originating node S, a route from S to D is established and data packets can start to flow. Both forward and reverse routes are maintained in a routing table at every node—details are given below. In the event of link and route breaks, AODV uses route error (RERR) messages to notify the affected nodes: if a link break is detected by a node, it first invalidates all routes stored in the node's own routing table that actually use the broken link. Then it sends a RERR message containing the unreachable destinations to all (direct) neighbours using this route.

In AODV, a routing table consists of a list of entries—at most one for each destination—each containing the following information: (i) the destination IP address; (ii) the *destination sequence number*; (iii) the sequence-number-status flag—tagging whether the recorded sequence number can be trusted; (iv) a flag tagging the route as being valid or invalid—this flag is set to invalid when a link break is detected or the route's lifetime is reached; (v) the hop count, a metric to indicate the distance to the destination; (vi) the next hop, an IP address that identifies the next (intermediate) node on the route to the destination; (vii) a list of precursors, a set of IP addresses of those 1-hop neighbours that use this particular route; and (viii) the lifetime (expiration or deletion time) of the route. The destination sequence number constitutes a measure approximating the relative freshness of the information held—a higher number denotes newer information. The routing table is updated whenever a node receives an AODV control message (RREQ, RREP or RERR) or detects a link break.

During the lifetime of the network, each node not only maintains its routing table, it also stores its *own sequence number*. This number is used as a local "timer" and is incremented whenever a new route request is initiated. It is the source of the destination sequence numbers in routing tables of other nodes.

Full details of the protocol are outlined in the request for comments (RFC) [29].

3.2 Route Request Handling Handled Formally

Our formal model consists of seven processes: `AODV` reads a message from the message queue (modelled in process `QMSG`, see below) and, depending on the type of the message, calls other processes. Each time a message has been handled the process has the choice between handling another message, initiating the transmission of queued data packets or generating a new route request. `NEWPKT` and `PKT` describe all actions performed by a node when a data packet is received. The former process handles a newly injected packet. The latter describes all actions performed when a node receives data from another node via the protocol. `RREQ` models all events that might occur after a route request message has been received. Similarly, `RREP` describes the reaction of the protocol to an incoming route reply. `RERR` models the part of AODV that handles error messages. The last process `QMSG` queues incoming messages. Whenever a message is received, it is first stored in a message queue. When the corresponding node is able to handle a message, it pops the oldest message from the queue and handles it. An AODV network is an encapsulated parallel composition of node expressions, each with a different node address (identifier), and all initialised with the parallel composition $\texttt{AODV}(\dots) \langle\langle \texttt{QMSG}(\dots)$.

Here we only present parts of the `RREQ` process, depicted in Process 4; the full formal specification of the entire protocol can be found in Appendix B.1. There, we also discuss all differences between the untimed version of AODV, as formalised in [11,15], and the newly developed timed version. These differences mostly consist of setting expiration times for routing table entries and other data maintained by AODV, and handling the expiration of this data.

A route discovery in AODV is initiated by a source node broadcasting a RREQ message; this message is subsequently re-broadcast by other nodes. Process 4 shows major parts of our process algebra specification for handling a RREQ message received by a node *ip*. The incoming message carries eight parameters, including *hops*, indicating how far the RREQ had travelled so far, *rreqid*, an identifier for this request, *dip*, the destination IP address, and *sip*, the sender of the incoming message; the parameters *ip*, *sn* and *rt*, storing the node's address, sequence number and routing table, as well as *rreqs* and *store*, are maintained by the process RREQ itself.

Before handling the incoming message, the process first updates *rreqs* (Line 1), a list of (unique) pairs containing the originator IP address *oip* and a route request identifier *rreqid* received within the last `PATH_DISCOVERY_TIME`: the update

Process 4 Parts of the RREQ handling

```
RREQ(hops, rreqid, dip, dsn, dsk, oip, osn, sip , ip, sn, rt, rreqs, store) $\stackrel{def}{=}$
 1. $\llbracket$exp_rreqs(rreqs, now)$\rrbracket$
 2. (
 3.    [ (oip, rreqid, $*$) $\in$ rreqs ]      /* the RREQ has been received previously */
 4.       AODV(ip, sn, rt, rreqs, store)      /* silently ignore RREQ, i.e., do nothing */
 5.    + [ (oip, rreqid, $*$) $\notin$ rreqs ]      /* the RREQ is new to this node */
 6.       $\llbracket$rt := update(rt, (oip, osn, kno, val, hops + 1, sip, $\emptyset$, now + ACTIVE_ROUTE_TIMEOUT))$\rrbracket$
 7.       $\llbracket$rt := setTime_rt(rt, oip, now + 2 $\cdot$ NET_TRAVERSAL_TIME $-$ 2 $\cdot$ (hops + 1) $\cdot$ NODE_TRAVERSAL_TIME)$\rrbracket$
 8.       $\llbracket$rreqs := rreqs $\cup$ {(oip, rreqid, now + PATH_DISCOVERY_TIME)}$\rrbracket$      /* update rreqs */
 9.       (
10.          [ dip = ip ]      /* this node is the destination node */
               [. . .]

23.          + [ dip $\neq$ ip ]      /* this node is not the destination node */
24.             (
25.                /* valid route to dip that is fresh enough */
26.                [dip $\in$ vD(rt) $\wedge$ dsn $\leq$ sqn(rt,dip) $\wedge$ sqnf(rt,dip) = kno]
27.                   /* update rt by adding precursors */
28.                   $\llbracket$rt := addpreRT(rt, dip, {sip})$\rrbracket$
29.                   $\llbracket$rt := addpreRT(rt, oip, {nhop(rt, dip)})$\rrbracket$
30.                   /* unicast a RREP towards the oip of the RREQ */
31.                   unicast(nhop(rt, oip),
                              rrep(dhops(rt, dip), dip, sqn(rt, dip), oip, $\sigma_{time}$(rt, dip) $-$ now, ip) .
32.                      AODV(ip, sn, rt, rreqs, store)
33.                   $\blacktriangleright$ /* If the transmission is unsuccessful, a RERR message is generated */
                         [. . .]      /* update local data structure */
40.                      groupcast(pre, rerr(dests, ip)) . AODV(ip, sn, rt, rreqs, store)
41.                + [ dip $\notin$ vD(rt) $\vee$ sqn(rt,dip) < dsn $\vee$ sqnf(rt,dip) = unk ]      /* no fresh route */
42.                   /* no further update of rt */
43.                   broadcast(rreq(hops+1, rreqid, dip, max(sqn(rt, dip), dsn), dsk, oip, osn, ip)) .
44.                   AODV(ip, sn, rt, rreqs, store)
45.             )
46.       )
47. )
```

removes identifiers that are too old. Based on this list, the node then checks whether it has recently received a RREQ with the same *oip* and *rreqid*.

If this is the case, the RREQ message is ignored, and the protocol continues to execute the main AODV process (Lines 3–4). If the RREQ is new (Line 5), the process updates the routing table by adding a "reverse route" entry to *oip*, the originator of the RREQ, via node *sip*, with distance *hops*+1 (Line 6). If there already is a route to *oip* in the node's routing table *rt*, it is only updated with the new route if the new route is "better", i.e., fresher and/or shorter and/or replacing an invalid route. The lifetime of this reverse route is updated as well (Line 7): it is set to the maximum of the currently stored lifetime and the minimal lifetime, which is determined by $\texttt{now} + 2 \cdot \texttt{NET\_TRAVERSAL\_TIME} - 2 \cdot (hops + 1) \cdot \texttt{NODE\_TRAVERSAL\_TIME}$ [29, Page 17]. The process also adds the message to the list of known RREQs (Line 8).

Lines 10–22 (only shown in Appendix B.1.2) deal with the case where the node receiving the RREQ is the intended destination, i.e., *dip*=*ip* (Line 10).

Lines 23–45 deal with the case where the node receiving the RREQ is not the destination, i.e., $dip \neq ip$ (Line 23). The node can respond to the RREQ with a corresponding RREP on behalf of the destination node *dip*, if its route to *dip* is "fresh enough" (Line 26). This means that (a) the node has a valid route to *dip*, (b) the destination sequence number in the node's current routing table entry (sqn(*rt*, *dip*)) is greater than or equal to the requested sequence number to *dip* in the RREQ message, and (c) the node's destination sequence number is trustworthy (sqnf(*rt*,*dip*)=kno). If these three conditions are met (Line 26), the node generates a RREP message, and unicasts it back to the originator node *oip* via the reverse route. Before unicasting the RREP message, the intermediate node updates the forward routing table entry to *dip* by placing the last hop node (*sip*) into the precursor list for that entry (Line 28). Likewise, it updates the reverse routing table entry to *oip* by placing the first hop nhop(*rt*, *dip*) towards *dip* in the precursor list for that entry (Line 29). To generate the RREP message, the process copies the sequence number for the destination *dip* from the routing table *rt* into the destination sequence number field of the RREP message and it places its distance in hops from the destination (dhops(*rt*, *dip*)) in the corresponding field of the new reply (Line 31). The RREP message is unicast to the next hop along the reverse route back to the originator of the corresponding RREQ message. If this unicast is successful, the process goes back to the AODV routine (Line 32). If the unicast of the RREP fails, we proceed with Lines 33–40, in which a route error (RERR) message is generated and sent. This conditional unicast is implemented in our model with the (T-)AWN construct **unicast**$(dest, ms).P \blacktriangleright Q$. In the latter case, the node sends a RERR message to all nodes that rely on the broken link for one of their routes. For this, the process first determines which destination nodes are affected by the broken link, i.e., the nodes that have this unreachable node listed as a next hop in the routing table (not shown in the shortened specification). Then, it invalidates any affected routing table entries, and determines the list of *precursors*, which are the neighbouring nodes that have a route to one of the affected destination nodes via the broken link. Finally, a RERR message is sent via groupcast to all these precursors (Line 40).

If the node is not the destination and there is either no route to the destination *dip* inside the routing table or the route is not fresh enough, the route request received has to be forwarded. This happens in Line 43. The information inside the forwarded request is mostly copied from the request received. Only the hop count is increased by 1 and the destination sequence number is set to the maximum of the destination sequence number in the RREQ packet and the current sequence number for *dip* in the routing table. In case *dip* is an unknown destination, sqn(*rt*, *dip*) returns the unknown sequence number 0.

To ensure that our time-free model from [11,15] accurately captures the intended behaviour of AODV [29], we spent a long time reading and interpreting the RFC, inspecting open-source implementations, and consulting network engineers. We now prove that our timed version of AODV behaves similar to our original formal specification, and hence (still) captures the intended behaviour.

Theorem 7. *The timed version of AODV (as presented in this paper) is a proper extension of the untimed version (as presented in [11]). By this we mean that if all timing constants, such as* ACTIVE_ROUTE_TIMEOUT*, are set to* ∞*, and the maximal number of pending route request retries* RREQ_RETRIES *is set to* 1*, then the (T-AWN) transition systems of both versions of AODV are weakly bisimilar.*

Proof Sketch. First, one shows that the newly introduced functions, such as `exp_rreqs` and `setTime_rt` do not change the data state in case the time parameters equal ∞; and hence lead to transitions of the form $\xi, p \xrightarrow{\tau} \xi, p'$. This kind of transitions are the ones that make the bisimulation weak, since they do not occur in the formal specification of [11]. Subsequently, one proves that all other transitions are basically identical.

3.3 Loop Freedom

Loop freedom is a critical property for any routing protocol, but it is particularly relevant and challenging for WMNs and MANETs. "A routing-table loop is a path specified in the nodes' routing tables at a particular point in time that visits the same node more than once before reaching the intended destination" [12]. Packets caught in a routing loop can quickly saturate the links and have a detrimental impact on network performance.

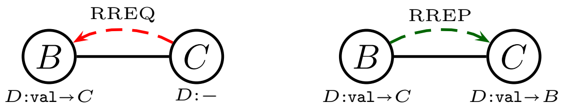
For AODV and many other protocols sequence numbers are used to guarantee loop freedom. Such protocols usually claim to be loop free due to the use of monotonically increasing sequence numbers. For example, AODV "uses destination sequence numbers to ensure loop freedom at all times (even in the face of anomalous delivery of routing control messages), ..." [29]. It has been shown that sequence numbers do not a priori guarantee loop freedom [16]; for some plausible interpretations[12] of different versions of AODV, however, loop freedom has been proven [30,3,35,34,19,11,15,25][13]. With the exception of [3], all these papers consider only untimed versions of AODV. As mentioned in Section 1 untimed analyses revealed many shortcomings of AODV; hence they are necessary. At the same time, a timed analysis is required as well. [3] shows that the premature deletion of invalid routes, and a too quick restart of a node after a reboot, can yield routing loops. Since then, AODV has changed to such a degree that the examples of [3] do not apply any longer.

In [13], "it is shown that the use of a `DELETE_PERIOD` in the current AODV specification can result in loops". However, the loop constructed therein at any time passes through at least one invalid routing table entry. As such, it is not a routing loop in the sense of [11,15]—we only consider loops consisting of valid routing table entries, since invalid ones do not forward data packets. In a loop as in [13] data packets cannot be sent in circles forever.

It turns out that AODV as standardised in the RFC (and carefully formalised in Section 3.2 and Appendix B.1) is *not* loop free. A potential cause of routing loops, sketched in Figure 1, is a situation where a node B has a `valid`

[12] By a plausible interpretation of a protocol standard written in English prose we mean an interpretation that fills the missing bits, and resolves ambiguities and contradictions occurring in the standard in a sensible and meaningful way.

[13] The proofs in [30] and [3] are incorrect; the model of [34] does not capture the full behaviour of the routing protocol; and [35] is based on a subset of AODV that does not cover the "intermediate route reply" feature, a source of loops. In [25] a draft of a new version of AODV is modelled, without intermediate route reply. For a more detailed discussion see [15].



a) C's entry for D has expired; B's has not

b) B sends old data; C establishes loop

Fig. 1. Premature Route Expiration

routing table entry for a destination D (in Figure 1 denoted D:val→C), but the next hop C no longer has a routing table entry for D (D:−), valid or invalid. In such a case, C might search for a new route to D and create a new routing table entry pointing to B as next hop, or to a node A upstream from B. We refer to this scenario as a case of *premature route expiration*.

A related scenario, which we also call premature route expiration, is when a node C sends a RREP message with destination D or a RREQ messages with originator D to a node B, but looses its route to D before that message arrives. This scenario can easily give rise to the scenario above.

Premature route expiration can be avoided by setting DELETE_PERIOD to ∞, which is essentially the case in the untimed version of AODV (cf. Theorem 7). In that case, no routing table entry expires or is erased. Hence, the situation where C no longer has a routing table entry for D is prevented.

In [11] we studied 5184 possible interpretations of the AODV RFC [29], a proliferation due to ambiguities, contradictions and cases of underspecification that could be resolved in multiple ways. In 5006 of these readings of the standard, including some rather plausible ones, we found routing loops, even when excluding all loops that are due to timing issues [16,11]. In [19,11,15] we have chosen a default reading of the RFC that avoids these loops, formalised it in AWN, and formally proved loop freedom, still assuming (implicitly) DELETE_PERIOD $= \infty$.

After taking this hurdle, the present paper continues the investigation by allowing arbitrary values for time parameters and for RREQ_RETRIES; hence dropping the simplifying assumption that DELETE_PERIOD $= \infty$.

One of our key results is that for the formalisation of AODV presented here, premature route expiration is the *only* potential source of routing loops. Under the assumption that premature route expiration does not occur, it turns out that, with minor modifications, the loop freedom proof of [11,15] applies to our timed model of AODV as well. A proof of this result is presented in Appendix B.2. There, we revisit all the invariants from [11] that contribute to the loop-freedom proof, and determine which of them are still valid in the timed setting, and how others need to be modified.

It is trivial to find an example where premature route expiration does occur in AODV, and a routing loop ensues. This can happen when a message spends an inordinate amount of time in the queue of incoming messages of a node. However, this situation tends not to occur in realistic scenarios. To capture this, we now make the assumption that the transmission time of a message plus the period it spends in the queue of incoming messages of the receiving node is bounded by NODE_TRAVERSAL_TIME. We also assume that the period a route request travels through the network is bounded by NET_TRAVERSAL_TIME.

These assumptions eliminate the "trivial" counterexample mentioned above. As we show in Appendix B.3, we now *almost* can prove an invariant that essentially says that premature route expiration does not occur. Following the methodology from [19,11,15], we establish our invariants by showing that they hold in all possible initial states of AODV, and are preserved under the transitions of our operation semantics, which correspond to the line numbers in our process algebraic specification.

We said “almost”, because, as indicated in Appendix B.3, our main invariant is not preserved by five lines of our AODV specification. Additionally, we need to make the assumption that when a RREQ message is forwarded, the forwarding node has a valid routing table entry to the originator of the route request. This does not hold for our formalisation of AODV: in Process 4 no check is performed on `oip`, only the routing table to the destination node `dip` has to satisfy certain conditions (Lines 23 and 41).

It turns out that for each of these failures we can construct an example of premature route expiration, and, by that, a counterexample to loop freedom.

However, if we skip all five offending lines (or adapt them in appropriate ways) and make a small change to process RREQ that makes the above assumption valid,[14] we obtain a proof of loop freedom for the resulting version of AODV. This follows immediately from the invariants established in Appendix B.3.

4 Conclusion

In this paper we have proposed T-AWN, a timed process algebra for wireless networks. We are aware that there are many other timed process algebras, such as timed CCS [24], timed CSP [32,28], timed ACP [1], ATP [27] and TPL [17]. However, none of these algebras provides the unique set of features needed for modelling and analysing protocols for wireless networks (e.g. a conditional unicast).[15] These features are provided by (T-)AWN, though. Our treatment of time is based on design decisions that appear rather different from the ones in [24,32,28,1,27]. Our approach appears to be closest to [17], but avoiding the negative premises that play a crucial role in the operational semantics of [17].

We have illustrated the usefulness of T-AWN by analysing the Ad hoc On-Demand Distance Vector routing protocol, and have shown that, contrary to claims in the literature and to common belief, it fails to be loop free. We have also discussed boundary conditions for a fix ensuring that the resulting protocol is loop free.

Acknowledgement. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

[14] The change basically introduces the test “$\texttt{oip} \in \texttt{vD}(\texttt{rt})$” in Line 41 or 9 of Process 4.

[15] This is similar to the untimed situation. A detailed comparison between AWN and other process calculi for wireless networks is given in [11, Section 11.1]; this discussion can directly be transferred to the timed case.

References

1. J.C.M. Baeten & J.A. Bergstra (1996): *Discrete Time Process Algebra. Formal Aspects of Computing* 8(2), pp. 188–208, doi:10.1007/BF01214556.
2. J.A. Bergstra & J.W. Klop (1986): *Algebra of Communicating Processes.* In J.W. de Bakker, M. Hazewinkel & J.K. Lenstra, eds.: *Mathematics and Computer Science*, CWI Monograph 1, North-Holland, pp. 89–138.
3. K. Bhargavan, D. Obradovic & C.A. Gunter (2002): *Formal Verification of Standards for Distance Vector Routing Protocols. Journal of the ACM* 49(4), pp. 538–576, doi:10.1145/581771.581775.
4. T. Bolognesi & E. Brinksma (1987): *Introduction to the ISO Specification Language LOTOS. Computer Networks* 14, pp. 25–59, doi:10.1016/0169-7552(87)90085-7.
5. 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, ed.: *Programming Languages and Systems (ESOP'16)*, *LNCS* 9632, Springer, pp. 95–122, doi:10.1007/978-3-662-49498-1_5.
6. S. Chiyangwa & M. Kwiatkowska (2005): *A Timing Analysis of AODV.* In: *Formal Methods for Open Object-based Distributed Systems (FMOODS'05)*, *LNCS* 3535, Springer, pp. 306–322, doi:10.1007/11494881_20.
7. T. Clausen & P. Jacquet (2003): *Optimized Link State Routing Protocol (OLSR).* RFC 3626 (Experimental), Network Working Group. Available at http://www.ietf.org/rfc/rfc3626.txt.
8. S. Cranen, M.R. Mousavi & M.A. Reniers (2008): *A Rule Format for Associativity.* In F. van Breugel & M. Chechik, eds.: *Concurrency Theory (CONCUR '08)*, *LNCS* 5201, Springer, pp. 447–461, doi:10.1007/978-3-540-85361-9_35.
9. S. Edenhofer & P. Höfner (2012): *Towards a Rigorous Analysis of AODVv2 (DYMO).* In: *Rigorous Protocol Engineering (WRiPE '12)*, IEEE, doi:10.1109/ICNP.2012.6459942.
10. 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, ed.: *ESOP'12*, *LNCS* 7211, Springer, pp. 295–315, doi:10.1007/978-3-642-28869-2_15.
11. 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.
12. J.J. Garcia-Luna-Aceves (1989): *A Unified Approach to Loop-free Routing using Distance Vectors or Link States.* In: *SIGCOMM'89*, *SIGCOMM Computer Communication Review* 19(4), ACM Press, pp. 212–223, doi:10.1145/75246.75268.
13. J.J. Garcia-Luna-Aceves & H. Rangarajan (2004): *A New Framework for Loop-free On-demand Routing using Destination Sequence Numbers.* In: *MASS'04*, IEEE, pp. 426–435, doi:10.1109/MAHSS.2004.1392182.
14. R.J. van Glabbeek (1993): *The Linear Time – Branching Time Spectrum II; The semantics of sequential systems with silent moves.* In E. Best, ed.: *CONCUR'93*, *LNCS* 715, Springer, pp. 66–81, doi:10.1007/3-540-57208-2_6.
15. R.J. van Glabbeek, P. Höfner, M. Portmann & W.L. Tan (2016): *Modelling and Verifying the AODV Routing Protocol. Distributed Computing.* To appear.
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: *MSWiM '13*, ACM Press, pp. 91–100, doi:10.1145/2507924.2507943.
17. M. Hennessy & T. Regan (1995): *A Process Algebra for Timed Systems. Information and Computation* 117(2), pp. 221–239, doi:10.1006/inco.1995.1041.

18. C.A.R. Hoare (1985): *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs.
19. P. Höfner, R.J. van Glabbeek, W.L. Tan, M. Portmann, A.K. McIver & A. Fehnker (2012): *A Rigorous Analysis of AODV and its Variants.* In: *MSWiM'12*, ACM Press, pp. 203–212, doi:`10.1145/2387238.2387274`.
20. IEEE (2011): *IEEE Standard for 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 Amendment 10: Mesh Networking*, doi:`10.1109/IEEESTD.2011.6018236`.
21. N. Lynch & M. Tuttle (1989): *An Introduction to Input/Output Automata. CWI-Quarterly* 2(3), pp. 219–246. Centrum voor Wiskunde en Informatica, Amsterdam.
22. W.W. McCune: *Prover9 and Mace4.* `http://www.cs.unm.edu/~mccune/prover9`. (accessed 10 October 2015).
23. R. Milner (1989): *Communication and Concurrency.* Prentice Hall.
24. F. Moller & C. Tofts (1990): *A Temporal Calculus of Communicating Systems.* In: *CONCUR '90*, *LNCS* 458, Springer, pp. 401–415, doi:`10.1007/BFb0039073`.
25. K.S. Namjoshi & R.J. Trefler (2015): *Loop Freedom in AODVv2.* In S. Graf & M. Viswanathan, eds.: *Formal Techniques for Distributed Objects, Components, and Systems (FORTE '15)*, *LNCS* 9039, Springer, pp. 98–112, doi:`10.1007/978-3-319-19195-9_7`.
26. A. Neumann, M. Aichele, C. Lindner & S. Wunderlich (2008): *Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.).* Internet-Draft (Experimental), Network Working Group. Available at `http://tools.ietf.org/html/draft-openmesh-b-a-t-m-a-n-00`.
27. X. Nicollin & J. Sifakis (1994): *The Algebra of Timed Processes, ATP: Theory and Application. Information and Computation* 114(1), pp. 131–178, doi:`10.1006/inco.1994.1083`.
28. J. Ouaknine & S. Schneider (2006): *Timed CSP: A Retrospective. Electronic Notes in Theoretical Computer Science* 162, pp. 273–276, doi:`10.1016/j.entcs.2005.12.093`.
29. 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`.
30. C.E. Perkins & E.M. Royer (1999): *Ad-hoc On-Demand Distance Vector Routing.* In: *Mobile Computing Systems and Applications (WMCSA '99)*, IEEE, pp. 90–100, doi:`10.1109/MCSA.1999.749281`.
31. G.D. Plotkin (2004): *A Structural Approach to Operational Semantics. Journal of Logic and Algebraic Programming* 60–61, pp. 17–139, doi:`10.1016/j.jlap.2004.05.001`. Originally appeared in 1981.
32. G.M. Reed & A.W. Roscoe (1986): *A Timed Model for Communicating Sequential Processes.* In L. Kott, ed.: *Automata, Languages and Programming (ICALP '86)*, *LNCS* 226, Springer, pp. 314–323, doi:`10.1007/3-540-16761-7_81`.
33. R. de Simone (1985): *Higher-Level Synchronising Devices in* MEIJE-*SCCS. Theoretical Computer Science* 37, pp. 245–267, doi:`10.1016/0304-3975(85)90093-3`.
34. A. Singh, C.R. Ramakrishnan & S.A. Smolka (2010): *A process calculus for Mobile Ad Hoc Networks. Science of Computer Programming* 75, pp. 440–469, doi:`10.1016/j.scico.2009.07.008`.
35. M. Zhou, H. Yang, X. Zhang & J. Wang (2009): *The Proof of AODV Loop Freedom.* In: *Wireless Communications & Signal Processing (WCSP '09)*, IEEE, doi:`10.1109/WCSP.2009.5371479`.

Appendices

A Results on the Process Algebra

A.1 Deferred Proofs and Auxiliary Lemmas

Proof of Proposition 1 . Only the six rules below generate a w_1-step (under certain conditions).

1. $\xi, \textbf{send}(\mathit{ms}).p \xrightarrow{\text{ws}} \xi[\texttt{now++}], \textbf{send}(\mathit{ms}).p$
2. $\xi, \textbf{receive}(\texttt{msg}).p \xrightarrow{\text{wr}} \xi[\texttt{now++}], \textbf{receive}(\texttt{msg}).p$
3. $\xi, p \xrightarrow{\text{w}} \xi[\texttt{now++}], p$
4. $\dfrac{\emptyset[\texttt{var}_i := \xi(\mathit{exp}_i)]_{i=1}^{n}, p \xrightarrow{w_1} \zeta, p'}{\xi, X(\mathit{exp}_1, ..., \mathit{exp}_n) \xrightarrow{w_1} \xi[\texttt{now++}], X(\mathit{exp}_1, ..., \mathit{exp}_n)} \ (X(\texttt{var}_1, ..., \texttt{var}_n) \overset{\mathit{def}}{=} p)$
5. $\dfrac{\xi \not\xrightarrow{\varphi}}{\xi, [\varphi]p \xrightarrow{\text{w}} \xi[\texttt{now++}], [\varphi]p}$
6. $\dfrac{\xi, p \xrightarrow{w_1} \zeta, p' \quad \xi, q \xrightarrow{w_2} \zeta, q'}{\xi, p+q \xrightarrow{w_1 \wedge w_2} \zeta, p' + q'}$

We reason inductively on the derivation of the w_1-step. If one of the Rules 1–5 is applied then the result follows by the form of the rule. For Rule 6, by the induction hypothesis, $p = p'$, $q = q'$ and hence $p + q = p' + q'$. Moreover, $\zeta = \xi[\texttt{now++}]$. □

Lemma A.1. *Let* $X(\texttt{var}_1, \ldots, \texttt{var}_n) \overset{\mathit{def}}{=} p$. *Then*

1. $\xi, X(\mathit{exp}_1, \ldots, \mathit{exp}_n) \xrightarrow{\textbf{rcv.}}$ *iff* $\emptyset[\texttt{var}_i := \xi(\mathit{exp}_i)]_{i=1}^{n}, p \xrightarrow{\textbf{rcv.}}$,
2. $\xi, X(\mathit{exp}_1, \ldots, \mathit{exp}_n) \xrightarrow{\textbf{send}}$ *iff* $\emptyset[\texttt{var}_i := \xi(\mathit{exp}_i)]_{i=1}^{n}, p \xrightarrow{\textbf{send}}$,
3. $\xi, X(\mathit{exp}_1, \ldots, \mathit{exp}_n) \xrightarrow{\textbf{other}}$ *iff* $\emptyset[\texttt{var}_i := \xi(\mathit{exp}_i)]_{i=1}^{n}, p \xrightarrow{\textbf{other}}$,
4. $\xi, X(\mathit{exp}_1, \ldots, \mathit{exp}_n) \xrightarrow{w_1}$ *iff* $\emptyset[\texttt{var}_i := \xi(\mathit{exp}_i)]_{i=1}^{n}, p \xrightarrow{w_1}$, $\forall w_1 \in \mathcal{W}$.

Proof. The first three claims follow immediately from Rule (rec), and the last claim from (rec-w). □

Proof of Theorem 1 . Let $P = \xi, s$. Let us first show the result for guarded terms s (as defined in Footnote 7). We reason inductively on the structure of s.

- $s = \textbf{unicast}(\mathit{dest}, \mathit{ms})p \blacktriangleright q$ or $s = \alpha.p$ with $\alpha \in \{\textbf{groupcast}(\mathit{dests}, \mathit{ms}), \textbf{broadcast}(\mathit{ms}), \textbf{deliver}(\mathit{data}), [\![\texttt{var}{:=}\mathit{exp}]\!]\}$. In case $\xi(s){\downarrow}$ we have $P\xrightarrow{\textbf{other}}$ and $P\not\xrightarrow{\textbf{wait}}$, using the rules of Table 1. In case $\xi(s){\uparrow}$ we have $P\xrightarrow{\text{w}}$ and $P\not\xrightarrow{\textbf{rcv.}} \wedge P\not\xrightarrow{\textbf{send}} \wedge P\not\xrightarrow{\textbf{other}}$.
- $s = \textbf{receive}(\texttt{msg}).p$. In this case $P\xrightarrow{\text{wr}}$ and $P\xrightarrow{\textbf{rcv.}} \wedge P\not\xrightarrow{\textbf{send}} \wedge P\not\xrightarrow{\textbf{other}}$.
- $s = \textbf{send}(\mathit{ms}).p$. In case $\xi(s){\downarrow}$ we have $P\xrightarrow{\text{ws}}$ and $P\not\xrightarrow{\textbf{rcv.}}\wedge P\xrightarrow{\textbf{send}}\wedge P\not\xrightarrow{\textbf{other}}$. In case $\xi(s){\uparrow}$ we have $P\xrightarrow{\text{w}}$ and $P\not\xrightarrow{\textbf{rcv.}} \wedge P\not\xrightarrow{\textbf{send}} \wedge P\not\xrightarrow{\textbf{other}}$.
- $s = \mathit{dsts}\colon \textbf{*cast}(m)[n, o].p$. In this case $P\xrightarrow{\textbf{other}}$ and $P\not\xrightarrow{\textbf{wait}}$.

- $s = [\varphi]p$. In case $\xi \xrightarrow{\varphi} \zeta$ for some ζ we have $P\xrightarrow{\tau}$, hence $P\xrightarrow{\textbf{other}}$, and $P\not\xrightarrow{\textbf{wait}}$. In case $\xi \not\xrightarrow{\varphi}$ we have $P\xrightarrow{\mathrm{w}}$ and $P\not\xrightarrow{\textbf{rcv.}} \wedge P\not\xrightarrow{\textbf{send}} \wedge P\not\xrightarrow{\textbf{other}}$.
- $s = s_1 + s_2$. Since s is a guarded term, also s_1 and s_2 must be guarded terms.
  - Assume $\xi, s_i\xrightarrow{\textbf{other}}$ for $i = 1$ or 2. By induction, $\xi, s_i\not\xrightarrow{\textbf{wait}}$, and hence $\xi, s\not\xrightarrow{\textbf{wait}}$. Moreover, by Rules (alt-l) and (alt-r) of Table 1, $\xi, s\xrightarrow{\textbf{other}}$. For the remaining cases assume that $\xi, s_i\not\xrightarrow{\textbf{other}}$ for $i = 1$ and 2.
  - Depending on whether $\xi, s_i\xrightarrow{\textbf{rcv.}}$ and $\xi, s_i\xrightarrow{\textbf{send}}$ for $i = 1, 2$ there are 2^4=16 cases left. As they all proceed in the same way, we show only one. Assume $\xi, s_1\not\xrightarrow{\textbf{rcv.}} \wedge \xi, s_1\xrightarrow{\textbf{send}} \wedge \xi, s_2\xrightarrow{\textbf{rcv.}} \wedge \xi, s_2\not\xrightarrow{\textbf{send}}$. By induction $\xi, s_1\xrightarrow{\mathrm{ws}}$ and $\xi, s_2\xrightarrow{\mathrm{wr}}$. By Rule (alt-w) of Table 1 $\xi, s\xrightarrow{\mathrm{wrs}}$, and by Rules (alt-l) and (alt-r) $\xi, s\xrightarrow{\textbf{rcv.}}$ and $\xi, s\xrightarrow{\textbf{send}}$.
- $s = X(\mathit{exp}_1, \ldots, \mathit{exp}_n)$. This case cannot occur, as s is not a guarded term.

Let us now show the result for all terms, again using structural induction on s. All cases proceed exactly as above (but skipping the guardedness check in the case for +), except for the case $s = X(\mathit{exp}_1, \ldots, \mathit{exp}_n)$.

- $s = X(\mathit{exp}_1, \ldots, \mathit{exp}_n)$. In this case $X(\mathtt{var}_1, \ldots, \mathtt{var}_n) \overset{\mathit{def}}{=} p$ for a guarded term p. Now the claim is an immediate corollary of Lemma A.1 and the result for guarded terms p obtained above. □

It is tempting to integrate the two parts of the above proof into one treatment of guarded and unguarded terms alike. A problem with that approach would be that in the very last case p is a bigger term than s, so that structural induction on s does not work. It is not a priori clear which inductive argument would take its place. In fact, there is no straightforward solution to this, because if there were, the result would hold without the restriction of T-AWN to guarded recursion, considering that that this restriction is not needed for Lemma A.1 and is not used anywhere else in the above proof than in the topmost case distinction.

Proof of Lemma 1 . For sequential processes, this follows directly from Rules (tr) and (tr-o) of Table 1. For parallel processes, it is a trivial structural induction. □

Proof of Theorem 2 . We apply structural induction on P. First suppose P has the form ξ, s. In case $P\xrightarrow{R:w_1}$ with $w_1 \in \mathcal{W}$, the claim follows from Observation 1. In case $P\not\xrightarrow{R:w_1}$ for all $w_1 \in \mathcal{W}$, the claim follows from Theorem 1.

Now consider an expression $P \langle\!\langle Q$. In case $P\xrightarrow{\textbf{inb}}$ or $Q\xrightarrow{\textbf{inb}}$ then also $P \langle\!\langle Q\xrightarrow{\textbf{inb}}$ by Rules (p-al) and (p-ar) of Table 2. By induction, $P\not\xrightarrow{\textbf{time}}$ or $Q\not\xrightarrow{\textbf{time}}$, so $P \langle\!\langle Q\not\xrightarrow{\textbf{time}}$. For the remaining cases assume that $P\not\xrightarrow{\textbf{inb}}$ and $Q\not\xrightarrow{\textbf{inb}}$.

In case $P\xrightarrow{\textbf{rcv.}}$ and $Q\xrightarrow{\textbf{send}}$ we have $P \langle\!\langle Q\xrightarrow{\tau}$ by the third rule of Table 2. Moreover, $P \langle\!\langle Q\not\xrightarrow{\textbf{time}}$. For the remaining cases assume that the combination $P\xrightarrow{\textbf{rcv.}}$ and $Q\xrightarrow{\textbf{send}}$ does not apply, so that $P \langle\!\langle Q\not\xrightarrow{\textbf{inb}}$.

In case $P\not\xrightarrow{\textbf{send}}$ and $Q\not\xrightarrow{\textbf{rcv.}}$ we have $P \langle\!\langle Q\not\xrightarrow{\textbf{send}}$ and $P \langle\!\langle Q\not\xrightarrow{\textbf{rec}}$. By induction, $P\xrightarrow{\mathrm{w}} \vee P\xrightarrow{R\,:\,\mathrm{w}} \vee P\xrightarrow{\mathrm{wr}} \vee P\xrightarrow{R\,:\,\mathrm{wr}}$ and $Q\xrightarrow{\mathrm{w}} \vee Q\xrightarrow{R\,:\,\mathrm{w}} \vee Q\xrightarrow{\mathrm{ws}} \vee Q\xrightarrow{R\,:\,\mathrm{ws}}$, so that $P \langle\!\langle Q\xrightarrow{\mathrm{w}} \vee P \langle\!\langle Q\xrightarrow{R\,:\,\mathrm{w}}$.

In case $P\not\xrightarrow{\textbf{send}}$ and $Q\xrightarrow{\textbf{rcv.}}$ we have $P\langle\!\langle Q\not\xrightarrow{\textbf{send}}$ and $P\langle\!\langle Q\xrightarrow{\textbf{rec}}$. By induction, $P\xrightarrow{\text{w}}\vee P\xrightarrow{R\,:\,\text{w}}\vee P\xrightarrow{\text{wr}}\vee P\xrightarrow{R\,:\,\text{wr}}$ and $Q\xrightarrow{\text{wr}}\vee Q\xrightarrow{R\,:\,\text{wr}}\vee Q\xrightarrow{\text{wrs}}\vee Q\xrightarrow{R\,:\,\text{wrs}}$, so that $P\langle\!\langle Q\xrightarrow{\text{wr}}\vee P\langle\!\langle Q\xrightarrow{R\,:\,\text{wr}}$.

In case $P\xrightarrow{\textbf{send}}$ and $Q\not\xrightarrow{\textbf{rcv.}}$ we have $P\langle\!\langle Q\xrightarrow{\textbf{send}}$ and $P\langle\!\langle Q\not\xrightarrow{\textbf{rec}}$. By induction, $P\xrightarrow{\text{ws}}\vee P\xrightarrow{R\,:\,\text{ws}}\vee P\xrightarrow{\text{wrs}}\vee P\xrightarrow{R\,:\,\text{wrs}}$ and $Q\xrightarrow{\text{w}}\vee Q\xrightarrow{R\,:\,\text{w}}\vee Q\xrightarrow{\text{ws}}\vee Q\xrightarrow{R\,:\,\text{ws}}$, so that $P\langle\!\langle Q\xrightarrow{\text{ws}}\vee P\langle\!\langle Q\xrightarrow{R\,:\,\text{ws}}$.

In case $P\xrightarrow{\textbf{send}}$ and $Q\xrightarrow{\textbf{rcv.}}$ we have $P\langle\!\langle Q\xrightarrow{\textbf{send}}$ and $P\langle\!\langle Q\xrightarrow{\textbf{rec}}$. By induction, $P\xrightarrow{\text{ws}}\vee P\xrightarrow{R\,:\,\text{ws}}\vee P\xrightarrow{\text{wrs}}\vee P\xrightarrow{R\,:\,\text{wrs}}$ and $Q\xrightarrow{\text{wr}}\vee Q\xrightarrow{R\,:\,\text{wr}}\vee Q\xrightarrow{\text{wrs}}\vee Q\xrightarrow{R\,:\,\text{wrs}}$, so that $P\langle\!\langle Q\xrightarrow{\text{wrs}}\vee P\langle\!\langle Q\xrightarrow{R\,:\,\text{wrs}}$. □

Lemma A.2. *$ip:P:R\xrightarrow{\{ip\}\neg\emptyset\,:\,\textbf{\textit{arrive}}(m)}$ for any $m\in\texttt{MSG}$, and any ip, P and R.*

Proof. This is our only proof in which the selected version of T-AWN matters—see Pages 14–15.

In the default version, we require for any node expression $ip:P:R$ that $P\xrightarrow{\textbf{receive}(m)}$ for any m—this is the definition of *input enabledness.* The claim then follows from Rule (n-rcv) of Table 3.

In the alternative version, the claim follows from that rule, in combination with the rule with a negative premise on Page 15. □

Proof of Lemma 2 . We apply structural induction on N. If N is a node expression $ip:P:R$, we have to show that $ip:P:R\xrightarrow{\{ip\}\neg\emptyset\,:\,\textbf{arrive}(m)}$ and also that $ip:P:R\xrightarrow{\emptyset\neg\{ip\}\,:\,\textbf{arrive}(m)}$. The former follows by Lemma A.2, and the latter by Rule (n-dis).

In case $N=M_1\|M_2$ the result is obtained using Rule (arr) of Table 4. □

A.2 Eliminating Data Structures

Our process algebra admits a translation into one without data structures (although we cannot *describe* the target algebra without using data structures). The target algebra differs from the original only on the level of sequential processes; the subsequent layers are unchanged. The syntax of the target language of sequential processes is given by the following grammar:

$$\begin{aligned} P \ &::= \mathbf{0} \ \mid \ P+P \ \mid \ \alpha.P \ \mid \ \mathit{dsts}\!:\!{}^{*}\mathbf{cast}(m).P \blacktriangleright P \ \mid \ X \ \mid \\ &\qquad \sum_{i\in I} P_i \ \mid \ \Delta^i P \ \mid \ \overset{\infty}{\underset{i=k}{\triangle}} P_i \ \mid \ {}^{n}\overset{o}{\underset{i=k}{\triangle_{*}}} P_i \\ \alpha \ &::= \tau \ \mid \ \mathbf{send}(m) \ \mid \ \mathbf{receive}(m) \ \mid \ \mathbf{deliver}(d) \end{aligned}$$

Its structural operational semantics displayed in Table 5.

Here $\mathbf{0}$ denotes the inactive process (that can only wait), $+$ is a binary choice (as before) and $\sum_{i\in I}$ denotes a choice with one argument P_i for each index i from a possibly infinite set I—the chosen process cannot start with a wait action, however. The process $\tau.P$ performs an internal action τ and proceeds as P. The actions $\mathbf{send}(m)$ and $\mathbf{receive}(m)$ are as before, but now there is one such action

Table 5. Structural operational semantics for sequential process expressions after elimination of data structures

$$\mathbf{0} \xrightarrow{\mathrm{w}} \mathbf{0} \qquad \frac{P \xrightarrow{w_1} P' \quad Q \xrightarrow{w_2} Q'}{P + Q \xrightarrow{w_1 \wedge w_2} P' + Q'} \qquad (\forall w_1, w_2 \in \mathcal{W})$$

$$\frac{P_j \xrightarrow{a} P'}{\sum_{i\in I} P_i \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'} \qquad (\forall j \in I,\ a \in \mathrm{Act}{-}\mathcal{W})$$

$$\tau.P \xrightarrow{\tau} P \qquad \mathbf{send}(m).P \xrightarrow{\mathbf{send}(m)} P$$

$$\mathbf{receive}(m).P \xrightarrow{\mathbf{receive}(m)} P \qquad \mathbf{deliver}(d).P \xrightarrow{\mathbf{deliver}(d)} P$$

$$\mathit{dsts}\colon{}^{*}\mathbf{cast}(m).P \blacktriangleright Q \xrightarrow{R:\mathrm{w}} (\mathit{dsts} \cap R)\colon{}^{*}\mathbf{cast}(m).P \blacktriangleright Q \qquad (\forall R \subseteq \mathtt{IP})$$

$$\emptyset\colon{}^{*}\mathbf{cast}(m).P \blacktriangleright Q \xrightarrow{\emptyset\,:\,{}^{*}\mathbf{cast}(m)} Q \qquad R\colon{}^{*}\mathbf{cast}(m).P \blacktriangleright Q \xrightarrow{R\,:\,{}^{*}\mathbf{cast}(m)} P \quad (\forall R \neq \emptyset)$$

$$\frac{P \xrightarrow{a} P'}{X \xrightarrow{a} P'}\ (X \overset{\mathit{def}}{=} P) \qquad \Delta^{i+1}P \xrightarrow{\mathrm{w}} \Delta^{i}P \qquad \frac{P \xrightarrow{a} P'}{\Delta^{0}P \xrightarrow{a} P'} \qquad (\forall i \geq 0,\ a \in \mathrm{Act})$$

$$\frac{P_k \xrightarrow{w_1} P'}{\mathop{\triangle}\limits_{i=k}^{\infty} P_i \xrightarrow{w_1} \mathop{\triangle}\limits_{i=k+1}^{\infty} P_i} \qquad \frac{P_k \xrightarrow{a} P'}{\mathop{\triangle}\limits_{i=k}^{\infty} P_i \xrightarrow{a} P'} \qquad (\forall w_1 \in \mathcal{W},\ a \in \mathrm{Act} - \mathcal{W},\ k \geq 0)$$

$$\frac{P_k \xrightarrow{\mathbf{send}(m)} P'}{\mathop{\triangle}\limits_{i=k}^{\infty} P_i \xrightarrow{\mathrm{ws}} \mathop{\triangle}\limits_{i=k+1}^{\infty} P_i} \qquad \frac{P_k \xrightarrow{\mathbf{receive}(m)} P'}{\mathop{\triangle}\limits_{i=k}^{\infty} P_i \xrightarrow{\mathrm{wr}} \mathop{\triangle}\limits_{i=k+1}^{\infty} P_i}$$

$$\frac{P_i \xrightarrow{R:\mathrm{w}} P_i'\ (\forall i \in [k..o])}{{}^{n}\mathop{\triangle\!\!\!\!\!{\scriptstyle *}}\limits_{i=k}^{o} P_i \xrightarrow{R:\mathrm{w}} {}^{n-1}\mathop{\triangle\!\!\!\!\!{\scriptstyle *}}\limits_{i=k}^{o} P_i'} \qquad \frac{P_i \xrightarrow{R:\mathrm{w}} P_i'\ (\forall i \in [k..o{+}1])}{{}^{n}\mathop{\triangle\!\!\!\!\!{\scriptstyle *}}\limits_{i=k}^{o+1} P_i \xrightarrow{R:\mathrm{w}} {}^{n}\mathop{\triangle\!\!\!\!\!{\scriptstyle *}}\limits_{i=k+1}^{o+1} P_i'} \qquad \begin{pmatrix}\forall R \subseteq \mathtt{IP}, \\ \forall n{>}0 \text{ and } o{\geq}k{\geq}0\end{pmatrix}$$

$$\frac{P_k \xrightarrow{R\,:\,{}^{*}\mathbf{cast}(m)} P'}{{}^{0}\mathop{\triangle\!\!\!\!\!{\scriptstyle *}}\limits_{i=k}^{o} P_k \xrightarrow{R\,:\,{}^{*}\mathbf{cast}(m)} P'} \qquad (\forall R \subseteq \mathtt{IP} \text{ and } o{\geq}k{\geq}0)$$

for each message $m \in \mathtt{MSG}$ (not an expression that evaluates to a message). Likewise, there is one action $\mathbf{deliver}(d)$ for each $d \in \mathtt{DATA}$. The process $\mathit{dsts}\colon{}^{*}\mathbf{cast}(m).P \blacktriangleright Q$ can cast the message $m \in \mathtt{MSG}$ to the destinations $\mathit{dsts} \subseteq \mathtt{IP}$ and then proceeds as P or Q, depending on whether $\mathit{dsts} = \emptyset$ or not. Alternatively, $\mathit{dsts}\colon{}^{*}\mathbf{cast}(m).P \blacktriangleright Q$ can perform an action R:w and restrict its set of destinations dsts to R. The language features process names X with defining equations $X \overset{\mathit{def}}{=} P$, as usual [23].

The unary operator Δ^i, parametrised with a natural number i, performs exactly i wait actions w before proceeding as its argument P. The operator $\mathop{\triangle}\limits_{i=k}^{\infty}$, with a countably infinite sequence of arguments P_i, performs a number of wait actions $w_1 \in \mathcal{W}$ (either w, wr, ws, or wrs)—possibly 0 or ∞; if a finite amount of wait actions is taken it proceeds as one of its arguments. In any state during its initial sequence of wait actions it has a choice between proceeding as its first argument P_k, provided P_k starts with a non-wait action, or doing another wait action w_1 and dropping P_k from the list of arguments. The latter is possible if and only if (1) P_k can do a **send**-action, in which case $w_1 := \text{ws}$, (2) P_k can do a **receive**-action, in which case $w_1 := \text{wr}$, or (3) P_k itself can do $w_1 \in \mathcal{W}$. The operator ${}^{n}\mathop{\triangle\!\!\!\!\!\!{\scriptstyle *}\;\,}\limits_{i=k}^{o}$ has parameters $k, n \geq 0$ and $o \geq k$, and $o{-}k{+}1$ arguments. As long as $n > 0$ all its arguments can synchronously perform an $R{:}\text{w}$-action, thereby either decrementing n or incrementing k, in the latter case loosing its first argument. When $n = 0$ the process ${}^{n}\mathop{\triangle\!\!\!\!\!\!{\scriptstyle *}\;\,}\limits_{i=k}^{o} P_i$ behaves as its first argument P_k, provided it starts with a ***cast**-action; otherwise the process deadlocks.

The idea behind the translation is to replace any variable by all possible values it can take. Formally, processes ξ, p are replaced by $\mathscr{T}_\xi(p)$, where $\mathscr{T}_\xi$ is defined inductively by

$$\mathscr{T}_\xi(p) = \begin{cases} \mathbf{0} & \text{if } \xi[\mathtt{now} := \mathtt{now} + i](p)\uparrow \quad \forall i \\ \Delta^{i_0}\, \mathscr{T}_{\xi[\mathtt{now}:=\mathtt{now}+i_0]}(p) & \text{with } i_0 = \min\limits_{i \in \mathbb{N}}(\xi[\mathtt{now} := \mathtt{now} + i](p)\downarrow), \end{cases} \quad \text{if } \xi(p)\uparrow;$$

Otherwise ($\xi(p)\downarrow$):

$$\mathscr{T}_\xi(\mathbf{broadcast}(\mathit{ms}).p) = \tau.\mathscr{T}_\xi(\mathtt{IP} : {*}\mathbf{cast}(\xi(\mathit{ms}))[\mathtt{LB}, \Delta\mathtt{B}].p \blacktriangleright p),$$

$$\mathscr{T}_\xi(\mathbf{groupcast}(\mathit{dests}, \mathit{ms}).p) = \tau.\mathscr{T}_\xi(\xi(\mathit{dests}) : {*}\mathbf{cast}(\xi(\mathit{ms}))[\mathtt{LG}, \Delta\mathtt{G}].p \blacktriangleright p),$$

$$\mathscr{T}_\xi(\mathbf{unicast}(\mathit{dest}, \mathit{ms}).p \blacktriangleright q) = \tau.\mathscr{T}_\xi(\{\xi(\mathit{dest})\} : {*}\mathbf{cast}(\xi(\mathit{ms}))[\mathtt{LU}, \Delta\mathtt{U}].p \blacktriangleright q),$$

$$\mathscr{T}_\xi(\mathit{dsts} : {*}\mathbf{cast}(m)[n, o].p \blacktriangleright q) =$$

$$\qquad {}^{n}\mathop{\triangle\!\!\!\!\!\!{\scriptstyle *}\;\,}\limits_{i=0}^{o} \mathit{dsts} : {*}\mathbf{cast}(m).\mathscr{T}_{\xi[\mathtt{now}:=\mathtt{now}+i+n]}(p) \blacktriangleright \mathscr{T}_{\xi[\mathtt{now}:=\mathtt{now}+i+n]}(q),$$

$$\mathscr{T}_\xi(\mathbf{send}(\mathit{ms}).p) = \mathop{\triangle}\limits_{i=0}^{\infty} P_i, \text{ with}$$

$$P_i = \begin{cases} \mathbf{send}(\xi[\mathtt{now} := \mathtt{now} + i](\mathit{ms})).\mathscr{T}_{\xi[\mathtt{now}:=\mathtt{now}+i]}(p) \\ \qquad\qquad \text{if } \xi[\mathtt{now} := \mathtt{now}{+}i](\mathit{ms})\downarrow \\ \mathbf{0} \qquad\qquad \text{otherwise,} \end{cases}$$

$$\mathscr{T}_\xi(\mathbf{receive}(\mathtt{msg}).p) = \mathop{\triangle}\limits_{i=0}^{\infty} \textstyle\sum_{m \in \mathtt{MSG}} \mathbf{receive}(m).\mathscr{T}_{\xi[\mathtt{msg}:=m;\ \mathtt{now}:=\mathtt{now}+i]}(p),$$

$$\mathscr{T}_\xi(\mathbf{deliver}(\mathit{data}).p) = \mathbf{deliver}(\xi(\mathit{data})).\mathscr{T}_\xi(p),$$

$$\mathscr{T}_\xi([\![\mathtt{var} := \mathit{exp}]\!]p) = \tau.\mathscr{T}_{\xi[\mathtt{var}:=\xi(\mathit{exp})]}(p),$$

$$\mathscr{T}_\xi([\varphi]p) = \begin{cases} \mathbf{0} & \text{if } \ \xi[\mathtt{now} := \mathtt{now} + i] \overset{\varphi}{\not\rightarrow} \quad \forall i \\ \Delta^{i_0} \sum_{\{\zeta \mid \xi[\mathtt{now}:=\mathtt{now}+i_0] \overset{\varphi}{\rightarrow} \zeta\}} \tau.\mathscr{T}_\zeta(p) & \begin{array}{l}\text{with} \\ i_0 = \min\limits_{i \in \mathbb{N}}(\xi[\mathtt{now} := \mathtt{now}{+}i] \overset{\varphi}{\rightarrow}),\end{array} \end{cases}$$

$$\mathscr{T}_\xi(p + q) = \mathscr{T}_\xi(p) + \mathscr{T}_\xi(q),$$

$$\mathscr{T}_\xi(X(\mathit{exp}_1, \ldots, \mathit{exp}_n)) = \mathop{\triangle}\limits_{i=0}^{\infty} X_{\xi[\mathtt{now}:=\mathtt{now}+i](\mathit{exp}_1),\ldots,\xi[\mathtt{now}:=\mathtt{now}+i](\mathit{exp}_n)}.$$

The last equation requires the introduction of a process name $X_{\vec{v}}$ for every name $X : \mathtt{TYPE}_1 \times \cdots \times \mathtt{TYPE}_n \rightarrow \mathtt{SPROC}$ (with defining equation $X(\overrightarrow{\mathtt{var}}) \overset{def}{=} p$) in the source language and every vector $\vec{v} \in \mathtt{TYPE}_1 \times \cdots \times \mathtt{TYPE}_n$ of data values of the appropriate types the for arguments of X. Its defining equation in the data-free target language is

$$X_{\vec{v}} \overset{def}{=} \mathscr{T}_{\emptyset[\overrightarrow{\mathtt{var}}:=\vec{v}]}(p) \ .$$

The resulting process algebra has a structural operational semantics in the (infinitary) *de Simone* format [33], generating the same transition system—up to strong bisimilarity, $\leftrightarroweq$—as the original.

Theorem A.1. *There exists a relation $\mathcal{B}$, a* bisimulation, *between states ξ, p of sequential processes in the source algebra, and sequential processes P in the target algebra, such that*

- $(\xi, p)\ \mathcal{B}\ \mathscr{T}_\xi(p)$ *for all sequential process expressions p and valuations ξ,*
- *if* $(\xi, p)\ \mathcal{B}\ P$ *and* $\xi, p \xrightarrow{a} \xi', p'$ *then* $\exists P'$ *such that* $P \xrightarrow{a} P'$ *and* $(\xi', p')\ \mathcal{B}\ P'$,
- *if* $(\xi, p)\ \mathcal{B}\ P$ *and* $P \xrightarrow{a} P'$ *then* $\exists \xi', p'$ *such that* $\xi, p \xrightarrow{a} \xi', p'$ *and* $(\xi', p')\ \mathcal{B}\ P'$.

Proof. We call ${}^{n}\overset{o}{\underset{i=k}{\triangle\!\!\!\!*}}\, P_i$ a *variant* of ${}^{n}\overset{o-k}{\underset{i=0}{\triangle\!\!\!\!*}}\, P_{i+k}$. Likewise, $\overset{\infty}{\underset{i=k}{\triangle}}\, P_i$ is a variant of $\overset{\infty}{\underset{i=0}{\triangle}}\, P_{i+k}$.

The relation $\mathcal{B}$ relates any state ξ, p to $\mathscr{T}_\xi(p)$ and to all variants of $\mathscr{T}_\xi(p)$. It therefore automatically satisfies the first requirement of Theorem A.1. That it satisfies the second requirement follows by a straightforward induction on the derivation of $\xi, p \xrightarrow{a} \xi', p'$ from the rules of Table 1. That it satisfies the last requirement follows by a straightforward induction on the derivation of $P \xrightarrow{a} P'$ from the rules of Table 5. □

A.3 Simulation Results

A *doubly labelled transition system* (L^{2}TS) (over sets Act and Σ) is a triple $(\mathbb{P}, \rightarrow, \ell)$, where $\mathbb{P}$ is as a set of *processes* or *states*, $\rightarrow\ \subseteq \mathbb{P} \times \mathrm{Act} \times \mathbb{P}$ is a *transition relation* and $\ell : \mathbb{P} \rightarrow \Sigma$ a *state labelling.*

A *simulation* is a binary relation between the states of two L^{2}TSs satisfying the *transfer property*: any transition from a state in the source L^{2}TS can be mimicked by a "similar" transition from a related state in the target L^{2}TS, such that the end states of both transitions are again related. Usually a similar transition is taken to be one with the same label, but here we generalise this by parametrising a simulation with an explicit similarity relation $\mathcal{U}^{\mathrm{Act}}$ between the transition labels of the two L^{2}TSs. We additionally require related states to have a similar state label, a notion parametrised by an explicit similarity relation $\mathcal{U}^{\Sigma}$ between the state labels of the two L^{2}TSs. A *weak* simulation [14] allows, in satisfying the transfer property, internal actions τ to precede and follow the mimicking transition—moreover, it allows internal transitions τ to be mimicked by doing no transition.

For $P, Q \in \mathbb{P}$, write $P \xrightarrow{a} Q$ for $(P, a, Q) \in\ \rightarrow$. Suppose that Act contains the *internal action* τ. Then $P \Longrightarrow Q$ for $P, Q \in \mathbb{P}$ denotes an arbitrary (possibly empty) sequence of τ-transitions, i.e., there are states $P_0, \ldots, P_n$ with $P = P_0 \xrightarrow{\tau} P_1 \cdots P_{n-1} \xrightarrow{\tau} P_n = Q$. Moreover, $P \overset{a}{\Longrightarrow} Q$, with $a \in \mathrm{Act}$, denotes $P \Longrightarrow\xrightarrow{a}\Longrightarrow Q$, and $P \overset{\hat{a}}{\Longrightarrow} Q$ denotes $P \Longrightarrow Q$ if $a = \tau$ and $P \overset{a}{\Longrightarrow} Q$ otherwise.

Definition A.1. Let $(\mathbb{P}_i, \rightarrow_i, \ell_i)$ for $i = 1, 2$ be two L^{2}TSs, labelled over sets Act_i and Σ_{i}, respectively. Furthermore, let $\mathcal{U}^{\mathrm{Act}} \subseteq \mathrm{Act}_1 \times \mathrm{Act}_2$ and $\mathcal{U}^{\Sigma} \subseteq \Sigma_1 \times \Sigma_2$. A *weak simulation* w.r.t. $\mathcal{U}^{\mathrm{Act}}$ and $\mathcal{U}^{\Sigma}$ is a binary relation $\mathcal{S} \subseteq \mathbb{P}_1 \times \mathbb{P}_2$ between the states of the two L^{2}TSs, such that

- if $P \mathrel{\mathcal{S}} Q$ and $P \xrightarrow{a} P'$ then $\exists Q', b$ such that $Q \overset{\hat{b}}{\Longrightarrow} Q'$, $a \mathrel{\mathcal{U}^{\mathrm{Act}}} b$ and $P' \mathrel{\mathcal{S}} Q'$,[16]
- if $P \mathrel{\mathcal{S}} Q$ then $\ell_1(P) \mathrel{\mathcal{U}^{\Sigma}} \ell_2(Q)$.

When $P \mathrel{\mathcal{S}} Q$, we speak of a weak simulation of P *by* Q.

In (T-)AWN the sequential processes ξ, p, equipped with the transition relation generated by the rules of Table 1, form a double labelled transition system by taking $\ell(\xi, p) := \xi$, the *data state* of ξ, p. In generalising this idea to parallel processes and network expressions we postulate two requirements on applications of (T-)AWN:

1. In a parallel process expression $\xi_1, p_1 \langle\!\langle \dots \langle\!\langle\, \xi_n, p_n$ the variables maintained by the p_i (i.e., the domains of the partial functions ξ_i) are pairwise disjoint.
2. Each node expression $\mathit{ip} : P : R$ occurring in a (partial) network expression has a different address *ip*.

The first requirement is not a restriction at all, since it can easily be achieved by renaming.[17] The second requirement is satisfied for all applications we encountered so far. Dropping one of the requirements would merely increase the bookkeeping effort of defining the notion of a global data state. Requirement 1 allows us to define the data state $\ell(P)$ of a parallel process expression $P = \xi_1, p_1 \langle\!\langle \dots \langle\!\langle\, \xi_n, p_n$ as $\bigcup_{i=1}^{n} \xi_n$, whereas Requirement 2 enables a definition of the global data state $\ell(N)$ of a (partial) network expression as a partial function σ that associates with each address *ip* of a node expression $\mathit{ip} : P : R$ occurring in N the data state of P.

The above definitions yield L^{2}TSs for the processes and network expressions of (T-)AWN. To compare the behaviour of AWN and T-AWN, we construct weak simulations between their L^{2}TSs. These will show that each AWN network expression N, seen as a T-AWN network expression, is weakly simulated by the AWN-expression N, and likewise for AWN process expressions.

To this end, we first define similarity relations between the state and transition labels that occur in the semantics of T-AWN and AWN. The only difference in their data states is that T-AWN processes maintain the variable `now`, which is absent in AWN. Consequently, the similarity relation between the state labels of processes is given by $\xi \mathrel{\mathcal{U}^{\Sigma}} \xi_{\backslash\mathtt{now}}$ for any T-AWN-valuation ξ. Here $\xi_{\backslash\mathtt{now}}$ is the AWN valuation obtained by omitting the value of `now` from ξ. For networks, this generalises to $\sigma \mathrel{\mathcal{U}^{\Sigma}} \sigma_{\backslash\mathtt{now}}$, where $\sigma_{\backslash\mathtt{now}}(\mathit{ip}) := \sigma(\mathit{ip})_{\backslash\mathtt{now}}$ for all $\mathit{ip} \in \mathrm{dom}(\sigma)$.

The translation labels of AWN processes include the actions **broadcast**(m), **groupcast**(D, m), **unicast**(dip, m) and $\neg$**unicast**(dip, m) for $m \in \mathtt{MSG}$, $D \subseteq \mathtt{IP}$ and $\mathit{dip} \in \mathtt{IP}$; in T-AWN these are replaced by dsts:***cast**(m). Furthermore,

[16] In case $b = \tau$, no action needs to be taken, that means, $Q = Q'$ is allowed if $P' \mathrel{\mathcal{S}} Q$.

[17] If the variable `now` is renamed, the SOS rules of Section 2.2 have to be adapted accordingly.

T-AWN processes have transition labels w_1 and $R{:}w_1$ for $w_1 \in \mathcal{W}$ and $R \subseteq \mathtt{IP}$, which are absent in AWN. Consequently, the similarity relation between the transition labels of processes is given by

- the identity relation on the (T-)AWN transition labels $\mathbf{send}(m)$, $\mathbf{deliver}(d)$, $\mathbf{receive}(m)$ and internal actions τ, for all $m \in \mathtt{MSG}$, $d \in \mathtt{DATA}$,
- $\mathit{dsts}\,{:}\,{*}\mathbf{cast}(m)\ \mathcal{U}^{\mathrm{Act}}\ b$, where b is either $\mathbf{broadcast}(m)$, $\mathbf{groupcast}(D,m)$ with $\mathit{dsts} \subseteq D$, $\mathbf{unicast}(\mathit{dip},m)$ with $\mathit{dsts} = \{\mathit{dip}\}$ or $\neg\mathbf{unicast}(\mathit{dip},m)$ with $\mathit{dsts} = \emptyset$,
- $w_1\ \mathcal{U}^{\mathrm{Act}}\ \tau$ and $R{:}w_1\ \mathcal{U}^{\mathrm{Act}}\ \tau$ for $w_1 \in \mathcal{W}$ and $R \subseteq \mathtt{IP}$.

On the level of network expressions, the only difference is the T-AWN transition label tick, which is absent in AWN. The similarity relation between the transition labels of network expressions is given by

- the identity relation on AWN transition labels,[18]
- $\emptyset\,{:}\,{*}\mathbf{cast}(m)\ \mathcal{U}^{\mathrm{Act}}\ \tau$,
- $\mathrm{tick}\ \mathcal{U}^{\mathrm{Act}}\ \tau$.

In order for our envisioned simulation to exist, we make one more abstraction: we read all **(dis)connect**-actions as τs. It is with this modification of the L^{2}TSs of AWN and T-AWN in mind that we speak of weak simulations below.

Theorem A.2. *Given a common underlying data structure modulo the variables* $\mathtt{now}$[19] *there exists a weak simulation $\mathcal{S}$ w.r.t. $\mathcal{U}^{Act}$ and $\mathcal{U}^{\Sigma}$ of the sequential processes of T-AWN by the ones of AWN, such that each AWN process simulates its interpretation as a T-AWN process.*

Proof. Define $\mathcal{S}$ as $\mathcal{S}_0 \cup \mathcal{S}_2$, where $\mathcal{S}_0$ consists of all pairs $((\xi,p),(\xi_{\backslash\mathtt{now}},p))$ for arbitrary sequential AWN expressions p—which also are sequential T-AWN expressions—and T-AWN valuations ξ.

Let $\mathcal{S}_1$ be the relation containing the following pairs:

- $((\xi, \mathit{dsts}\,{:}\,{*}\mathbf{cast}(\xi(\mathit{ms}))[n,o].p \blacktriangleright p), (\xi_{\backslash\mathtt{now}}, \mathbf{broadcast}(\mathit{ms}).p))$ if $\xi(\mathit{ms}){\downarrow}$,
- $((\xi, \mathit{dsts}\,{:}\,{*}\mathbf{cast}(\xi(\mathit{ms}))[n,o].p \blacktriangleright p), (\xi_{\backslash\mathtt{now}}, \mathbf{groupcast}(\mathit{dests},\mathit{ms}).p))$ if $\xi(\mathit{ms}){\downarrow}$, $\xi(\mathit{dests}){\downarrow}$ and $\mathit{dsts} \subseteq \xi(\mathit{dests})$,
- $((\xi, \mathit{dsts}\,{:}\,{*}\mathbf{cast}(\xi(\mathit{ms}))[n,o].p \blacktriangleright q), (\xi_{\backslash\mathtt{now}}, \mathbf{unicast}(\mathit{dest},\mathit{ms}).p \blacktriangleright q))$ if $\xi(\mathit{ms}){\downarrow}$, $\xi(\mathit{dest}){\downarrow}$ and $\mathit{dsts} \subseteq \{\xi(\mathit{dest})\}$,

for T-AWN valuations ξ, $\mathit{dsts} \subseteq \mathtt{IP}$, $n,o \in \mathbb{N}$, sequential AWN processes p, q (in the source algebra of $\mathcal{S}_1$ again interpreted as T-AWN expressions), and data expressions ms, dest and dests of type $\mathtt{MSG}$, $\mathtt{IP}$ and $\mathscr{P}(\mathtt{IP})$, respectively. Then $\mathcal{S}_2$ is the smallest relation containing $\mathcal{S}_1$, such that $((\xi,p),(\zeta,q)) \in \mathcal{S}_2 \Rightarrow ((\xi,p),(\zeta,r+q)),((\xi,p),(\zeta,q+r)) \in \mathcal{S}_2$, for all AWN-processes r.

To show that $\mathcal{S}$ is a weak simulation, we need to demonstrate that it satisfies the two requirements of Definition A.1. The second requirement is satisfied by construction. Moreover, we obtain a stronger version of the first requirement:

[18] The labels are $R\,{:}\,{*}\mathbf{cast}(m)$, $H\neg K\,{:}\,\mathbf{arrive}(m)$, $\mathit{ip}\,{:}\,\mathbf{deliver}(d)$, $\mathbf{connect}(\mathit{ip},\mathit{ip}')$, $\mathbf{disconnect}(\mathit{ip},\mathit{ip}')$, $\mathit{ip}\,{:}\,\mathbf{newpkt}(d,\mathit{dip})$ and τ.

[19] The variables $\mathtt{now}$ only occur in the data structure of T-AWN.

- if $P\;\mathcal{S}_0\;Q$ and $P \xrightarrow{w_1} P'$ with $w_1 \in \mathcal{W}$ then $P'\;\mathcal{S}_1\;Q$,
- if $P\;\mathcal{S}_0\;Q$ and $P \xrightarrow{\tau} P'$ then either $P'\,\mathcal{S}_2\,Q$ or $\exists Q'$ with $Q \xrightarrow{\tau} Q'$ and $P'\,\mathcal{S}_1\,Q'$,
- if $P\;\mathcal{S}_0\;Q$ and $P \xrightarrow{a} P'$ with $a \,{\notin}\, \mathcal{W}{\cup}\{\tau\}$ then $\exists Q'$ with $Q \xrightarrow{a} Q'$ and $P'\,\mathcal{S}_1\,Q'$,
- if $P\;\mathcal{S}_2\;Q$ and $P \xrightarrow{R:\mathrm{w}} P'$ then $P'\;\mathcal{S}_2\;Q$,
- if $P\;\mathcal{S}_2\;Q$ and $P \xrightarrow{dsts:\,{}^{*}\mathbf{cast}(m)} P'$ then $\exists Q'$ such that $Q \xrightarrow{b} Q'$ and $P'\;\mathcal{S}_1\;Q'$, where b is either $\mathbf{broadcast}(m)$, or $\mathbf{groupcast}(D,m)$ with $dsts \subseteq D$, or $\mathbf{unicast}(dip,m)$ with $dsts = \{dip\}$ or $\neg\mathbf{unicast}(dip,m)$ with $dsts = \emptyset$,

considering that other combinations of $P\;\mathcal{S}\;Q$ and $P \xrightarrow{a} P'$ cannot occur. The first of these properties follows from Proposition 1. The fourth follows from Rules (tr) and (tr-o) of Table 1 and a trivial induction on the definition of $\mathcal{S}_2$. Demonstrating the others proceeds by a straightforward induction on the derivation of T-AWN transitions from the rules of Table 1. □

We have shown that each sequential AWN process P, seen as a T-AWN process, can be simulated by the AWN process P. We will now lift this result to parallel processes, node expressions, partial network expressions and finally complete networks. This constitutes the proof of Theorem 6.

Proof of Theorem 6. Given weak simulations $\mathcal{S}_1$ and $\mathcal{S}_2$ of parallel T-AWN processes by parallel AWN processes, define the simulation $\mathcal{S}_1 \langle\!\langle \mathcal{S}_2$ of parallel T-AWN processes by parallel AWN processes by

$$P_1 \langle\!\langle\, P_2\; (\mathcal{S}_1 \langle\!\langle\, \mathcal{S}_2)\; Q_1 \langle\!\langle\, Q_2 \quad :\Leftrightarrow \quad P_1\;\mathcal{S}_1\;Q_1 \wedge P_2\;\mathcal{S}_2\;Q_2\;.$$

By construction, this relation satisfies the second requirement of Definition A.1. A straightforward induction on the derivation of T-AWN transitions from the rules of Table 2 shows that $\mathcal{S}_1 \langle\!\langle\, \mathcal{S}_2$ also satisfies the first requirement, and thus is a weak simulation indeed. Now a trivial induction on the number of sequential processes occurring in a parallel process, with Theorem A.2 as base case and the above observation as induction step, lifts Theorem A.2 to parallel processes.

Given a weak simulation $\mathcal{S}$ of parallel T-AWN processes by parallel AWN processes, define the simulation $\mathcal{S}'$ of T-AWN node expressions by AWN node expressions by

$$ip{:}P{:}R\;\;\mathcal{S}\;\;ip'{:}Q{:}R' \quad :\Leftrightarrow \quad ip = ip' \wedge P\;\mathcal{S}\;Q \wedge R = R'\;.$$

By construction, this relation satisfies the second requirement of Definition A.1. By induction on the derivation of T-AWN transitions from the rules of Table 3 we show that $\mathcal{S}_1 \langle\!\langle\, \mathcal{S}_2$ also satisfies the first requirement, and thus is a weak simulation. The only non-trivial cases are when $ip{:}P{:}R \xrightarrow{dsts:\,{}^{*}\mathbf{cast}(m)} ip{:}P'{:}R$ and $ip{:}P{:}R \xrightarrow{\text{tick}} ip{:}P'{:}R$. In the former case $P \xrightarrow{dsts:\,{}^{*}\mathbf{cast}(m)} P'$, so by induction $Q \overset{b}{\Longrightarrow} Q'$ for some Q' with $P'\;\mathcal{S}\;Q'$, where b is either $\mathbf{broadcast}(m)$, $\mathbf{groupcast}(D,m)$ with $dsts \subseteq D$, $\mathbf{unicast}(dip,m)$ with $dsts = \{dip\}$ or $\neg\mathbf{unicast}(dip,m)$ with $dsts = \emptyset$. By the rules of [10, Table 3],

$$ip{:}Q{:}R \Longrightarrow ip{:}Q{:}dsts \xRightarrow{dsts:\,{}^{*}\mathbf{cast}(m)} ip{:}Q'{:}dsts \Longrightarrow ip{:}Q'{:}R\;,$$

except in the case that $b = \neg\textbf{unicast}(\mathit{dip}, m)$, when we obtain

$$\mathit{ip}{:}Q{:}R \Longrightarrow \mathit{ip}{:}Q{:}\emptyset \overset{\tau}{\Longrightarrow} \mathit{ip}{:}Q'{:}\emptyset \Longrightarrow \mathit{ip}{:}Q'{:}R\ .$$

Here the derivations $\mathit{ip}{:}Q{:}R \Longrightarrow \mathit{ip}{:}Q{:}\mathit{dsts}$ and $\mathit{ip}{:}Q'{:}\mathit{dsts} \Longrightarrow \mathit{ip}{:}Q'{:}R$ consists of **(dis)connect**-actions—this is the reason these are seen as τ's here. In case $\mathit{ip}{:}P{:}R \xrightarrow{\text{tick}} \mathit{ip}{:}P'{:}R$, we have $P \xrightarrow{w_1} P'$ or $P \xrightarrow{R:w_1} P'$, so by induction there is a Q' with $Q \Longrightarrow Q'$ and $P'\ \mathcal{S}\ Q'$. By the rules of [10, Table 3] we have $\mathit{ip}{:}Q{:}R \Longrightarrow \mathit{ip}{:}Q'{:}R$.

Given weak simulations $\mathcal{S}_1$ and $\mathcal{S}_2$ of T-AWN partial network expressions by AWN partial network expressions, define the simulation $\mathcal{S}_1\|\mathcal{S}_2$ of T-AWN partial network expressions by AWN partial network expressions by

$$N_1\|N_2\ (\mathcal{S}_1\|\mathcal{S}_2)\ M_1\|M_2 \quad :\Leftrightarrow \quad N_1\ \mathcal{S}_1\ M_1 \wedge N_2\ \mathcal{S}_2\ M_2\ .$$

By construction, this relation satisfies the second requirement of Definition A.1. A straightforward induction on the derivation of T-AWN transitions from the rules of Table 4 shows that $\mathcal{S}_1\|\mathcal{S}_2$ also satisfies the first requirement, and thus is a weak simulation indeed. Now a trivial induction on the number of node expressions occurring in a partial network expression lifts Theorem A.2 to partial network expressions.

Given a weak simulation $\mathcal{S}$ of T-AWN partial network expressions by AWN partial network expressions, define the simulation $\mathcal{S}'$ of complete T-AWN networks by complete AWN networks by

$$[N]\ \mathcal{S}'\ [M] \quad :\Leftrightarrow \quad N\ \mathcal{S}\ M\ .$$

By construction, this relation satisfies the second requirement of Definition A.1. A straightforward induction on the derivation of T-AWN transitions from the rules of Table 4 shows that $\mathcal{S}'$ also satisfies the first requirement, and thus is a weak simulation indeed. □

An immediate corollary of this result is that for each T-AWN network expression N', reachable from an (initial) AWN network expression N, seen as T-AWN network expression, there exists an AWN network expression N'', reachable from N, such that $\ell(N')_{\backslash\texttt{now}} = \ell(N'')$, i.e., having the same global data state as N', except of course for the variables `now`.

This in turn implies that any invariant for the AWN network N—a property that holds for all global data states of network expressions reachable from N—is also an invariant for N seen as a T-AWN network.

B Case Study: The AODV Routing Protocol

B.1 Formal Specification of AODV

This appendix provides a complete and accurate formal specification of the AODV routing protocol, as defined in IETF RFC 3561 [29]. The presented formalisation is based on an untimed model formalised in AWN [11,15], and includes all core components of the protocol, but abstracts from optional protocol features.[20] The only difference with our previous formalisation of AODV in [11,15] is the inclusion of timing issues.

To keep this appendix 'short', we focus on the difference between the two models; an interested reader can either study the formal model on her own, or have a look at [11,15], where we explain each and every line of the specification.

B.1.1 Data Structure

In [11, Section 5] we define the basic data structure needed for the detailed formal specification of AODV, when abstracting from timing issues. Here we merely list the changes in this data structure needed to deal with time.

Constants and Basic Types. The new type TIME and the variable now have been introduced in Section 2. Additionally, we use the following constants of type TIME. They all follow the RFC; their default values are found in [29, Section 10].

DELETE_PERIOD: the lifetime of an invalid routing table entry;
ACTIVE_ROUTE_TIMEOUT: the time after which a valid entry is invalidated;
MY_ROUTE_TIMEOUT: amount of time entered as last parameter of a route reply issued by the destination node of a route request—to be used as the time during which the route to the destination created by that route reply remains valid;
NODE_TRAVERSAL_TIME: a conservative estimate of the average one hop traversal time for packets—it should include queueing delays and transfer times;
NET_TRAVERSAL_TIME: a conservative estimate on the time it takes for a message to travel from one end of the network to the other and back—calculated as $2 \cdot \mathtt{NODE\_TRAVERSAL\_TIME} \cdot \mathtt{NET\_DIAMETER}$;
PATH_DISCOVERY_TIME: time during which identifiers of handled route requests are kept.

The type P, which, in the original specification, was a Boolean flag indicating whether a new route request needs to be initiated, is now of type $\mathbb{N}$; it tells the number of pending route request. The constants no-req and req do not exist any more, but there is a new constant of type P, discussed in [29, Sections 6.3 and 10]:

RREQ_RETRIES: maximal number of retries for a route discovery process.

Routing table entries are of type R and have an additional parameter, their expiration time, which in the RFC is called Lifetime. An entry is now an eight-tuple of type

$$\mathtt{IP} \times \mathtt{SQN} \times \mathtt{K} \times \mathtt{F} \times \mathbb{N} \times \mathtt{IP} \times \mathscr{P}(\mathtt{IP}) \times \mathtt{TIME}$$

A tuple $(\mathit{dip}, \mathit{dsn}, \mathit{dsk}, \mathit{flag}, \mathit{hops}, \mathit{nhip}, \mathit{pre}, \mathit{ltime})$ describes a route to *dip* of length *hops* and validity *flag*; the very next node on this route is *nhip*; the last time the entry was updated the destination sequence number was *dsn*; *dsk*

[20] A list and discussion of all omitted features occurs in [11, Section 3].

denotes whether the sequence number is “outdated” or can be used to reason about the freshness of the route. *pre* is a set of all neighbours who are “interested” in the route to *dip*. Finally, *ltime* states the expiration time of the route; if this time is reached, a valid route will be set to invalid, and an invalid route will be deleted from the routing table. We use projections $\pi_1, \dots \pi_8$ to select the corresponding component from the 8-tuple: for example, $\pi_6 : \mathtt{R} \to \mathtt{IP}$ determines the next hop, and $\pi_8 : \mathtt{R} \to \mathtt{IP}$ distills the expiration time. A routing table is an element of type $\mathtt{RT}$ and defined as a set of entries, with the restriction that each has a different destination $\mathtt{dip}$, i.e., the first component of each entry in a routing table is unique.

The data type $\mathtt{STORE}$, which models a set of data queues for injected data packets, is equipped with timers as well. Each queue now contains a timer that indicates when a new/the next route request should be sent. The special value 0 means “to be sent immediately”.

$$\mathtt{STORE} := \left\{ \mathit{store} \;\middle|\; \begin{array}{l} \mathit{store} \in \mathscr{P}(\mathtt{IP} \times \mathtt{P} \times \mathtt{TIME} \times [\mathtt{DATA}]) \ \wedge \\ \big((\mathit{dip}, p, t, q), (\mathit{dip}, p', t', q') \in \mathit{store} \Rightarrow \\ \hfill p = p' \wedge t = t' \wedge q = q'\big) \end{array} \right\}$$

Here $[\mathtt{DATA}]$ denotes a queue of elements from $\mathtt{DATA}$.

The last type that needs to be changed is the set of pairs $(\mathtt{oip}, \mathtt{rreqid}) \in \mathtt{IP} \times \mathtt{RREQID}$, which uniquely identify route requests. (For our specification we set $\mathtt{RREQID} = \mathbb{N}$.) As such pairs are stored by nodes for a limited amount of time, we add a third component to indicate when the pair can be dropped. In our model, each node maintains a variable $\mathtt{rreqs}$ of type

$$\mathscr{P}(\mathtt{IP} \times \mathtt{RREQID} \times \mathtt{TIME})$$

of sets of such triples to store the set of route requests seen by the node so far.

Functions. A brief overview of all functions used in our specifications can be found in Table 6. Here, we only list changes w.r.t. untimed formal specification of AODV.

First, we need to change/update a couple of functions. This is mainly due to the new and changed data types. For example, the function $\mathtt{qD} : \mathtt{STORE} \to \mathscr{P}(\mathtt{IP})$, which extracts the destinations for which there are unsent packets is changed from $\{\mathtt{dip} \mid (\mathtt{dip}, *, *) \in \mathtt{store}\}$ to $\{\mathtt{dip} \mid (\mathtt{dip}, *, *, *) \in \mathtt{store}\}$. In a similar (straightforward) manner the functions $\mathtt{add}$, $\mathtt{drop}$, σ_{queue}, $\mathtt{vD}$, $\mathtt{iD}$, $\mathtt{kD}$ $\mathtt{addpre}$, $\mathtt{addpreRT}$, and $\mathtt{nrreqid}$ are adapted.

In [11,15] the functions $\mathtt{sqn}$, $\mathtt{sqnf}$, $\mathtt{flag}$, $\mathtt{dhops}$, $\mathtt{nhop}$, $\mathtt{precs}$, and $\mathtt{precs}$ distill particular information for a specified route in the routing table (if it exists). Since routing table entries are extended with an additional field, we define a new function $\mathtt{ltime}$ that selects the newly introduced expiration time:

$$\begin{aligned} \mathtt{ltime} :\ & \mathtt{RT} \times \mathtt{IP} \rightharpoonup \mathtt{TIME} \\ \mathtt{ltime}(\mathit{rt}, \mathit{dip}) \ :=\ & \begin{cases} \pi_8(r) & \text{if } r \in \mathit{rt} \wedge \pi_1(r) = \mathit{dip} \\ \text{undefined} & \text{otherwise .} \end{cases} \end{aligned}$$

Next to these changes, we now discuss changes in a few functions that either are non-trivial or of particular interest for the timed version of AODV.

Invalidating routes is a main feature of AODV; if a route is not valid any longer, its validity flag has to be set to invalid. By doing this, the stored information about the route, such as the sequence number and the hop count, remains accessible. The function for invalidating routing table entries takes as arguments a routing table, a set of destinations $\mathit{dests} \in \mathscr{P}(\mathtt{IP} \times \mathtt{SQN})$, and the expiration time for the newly invalidated routes. Elements of *dests* are $(\mathit{rip}, \mathit{rsn})$-pairs that not only identify an unreachable destination *rip*, but also a sequence number that describes the freshness of the faulty route. We restrict ourselves to sets that have at most one entry for each destination—formally we define *dests* as a *partial function* from IP to SQN, i.e., a subset of IP × SQN satisfying $(\mathit{rip}, \mathit{rsn}), (\mathit{rip}, \mathit{rsn}') \in \mathit{dests} \Rightarrow \mathit{rsn} = \mathit{rsn}'$.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{invalidate} : \mathtt{RT} \times (\mathtt{IP} \rightharpoonup \mathtt{SQN}) \times \mathtt{TIME} \to \mathtt{RT}} \\
\mathtt{invalidate}(\mathit{rt}, \mathit{dests}, t) & := & \{r \,|\, r \in \mathtt{rt} \ \wedge\ (\pi_1(r), *) \notin \mathit{dests}\} \\
 & \cup & \{(\pi_1(r), \mathit{rsn}, \pi_3(r), \mathtt{inv}, \pi_5(r), \pi_6(r), \pi_7(r), t) \mid \\
 & & \quad r \in \mathtt{rt} \ \wedge\ (\pi_1(r), \mathit{rsn}) \in \mathit{dests}\}
\end{array}$$

Similar to invalidate, *updating a routing table* must take the expiration time of a route into account. The update function now works on 8-tuples as routing table entries, and the new expiration time of a route is taken as the maximum of the one from the routing table (if any) and the one from the incoming route, but only if the route is actually updated with new important information. This is in line with the RFC, which updates a route's expiration time to the maximum of the `ExistingLifetime` and the `MinimalLifetime`. In AODV the minimal expiration time is often set to $\mathtt{now} + \mathtt{ACTIVE\_ROUTE\_TIMEOUT}$. As in [11,15] we define an update function $\mathtt{update}(\mathit{rt}, r)$ of a routing table *rt* with an entry *r* only when *r* is valid, i.e., $\pi_4(r) = \mathtt{val}$, $\pi_2(r) = 0 \Leftrightarrow \pi_3(r) = \mathtt{unk}$, and $\pi_3(r) = \mathtt{unk} \Rightarrow \pi_5(r) = 1$. Formally the function $\mathtt{update} : \mathtt{RT} \times \mathtt{R} \rightharpoonup \mathtt{RT}$ is given by

$$\mathtt{update}(\mathit{rt}, r) := \begin{cases}
\mathit{rt} \cup \{r\} & \text{if } \pi_1(r) \notin \mathtt{kD}(\mathit{rt}) \\
\mathit{nrt} \cup \{\mathit{nr}\} & \text{if } \pi_1(r) \in \mathtt{kD}(\mathit{rt}) \wedge \mathtt{sqn}(\mathit{rt}, \pi_1(r)) < \pi_2(r) \\
\mathit{nrt} \cup \{\mathit{nr}\} & \text{if } \pi_1(r) \in \mathtt{kD}(\mathit{rt}) \wedge \mathtt{sqn}(\mathit{rt}, \pi_1(r)) = \pi_2(r) \\
 & \qquad\qquad\qquad\quad \wedge\ \mathtt{dhops}(\mathit{rt}, \pi_1(r)) > \pi_5(r) \\
\mathit{nrt} \cup \{\mathit{nr}\} & \text{if } \pi_1(r) \in \mathtt{kD}(\mathit{rt}) \wedge \mathtt{sqn}(\mathit{rt}, \pi_1(r)) = \pi_2(r) \\
 & \qquad\qquad\qquad\quad \wedge\ \mathtt{flag}(\mathit{rt}, \pi_1(r)) = \mathtt{inv} \\
\mathit{nrt} \cup \{\mathit{nr}'\} & \text{if } \pi_1(r) \in \mathtt{kD}(\mathit{rt}) \wedge \pi_3(r) = \mathtt{unk} \\
\mathit{nrt} \cup \{\mathit{ns}\} & \text{otherwise} ,
\end{cases}$$

where $s := \sigma_{\mathit{route}}(\mathit{rt}, \pi_1(r))$ is the current entry in the routing table for the destination of r (if it exists), and $\mathit{nrt} := \mathit{rt} - \{s\}$ is the routing table without that entry. The entry

$$\mathit{nr} := (\pi_1(r), \pi_2(r), \pi_3(r), \pi_4(r), \pi_5(r), \pi_6(r), \pi_7(r) \cup \pi_7(s), \max(\pi_8(r), \pi_8(s)))$$

is identical to r except that the precursors from s are added and the lifetime is set to the maximum of the routes r and s. Similarly, $\mathit{ns} := \texttt{addpre}(s, \pi_7(r)) = (\pi_1(s), \pi_2(s), \pi_3(s), \pi_4(s), \pi_5(s), \pi_6(s), \pi_7(s) \cup \pi_7(r), \pi_8(s))$ is generated from s by adding the precursors from r; the lifetime, however, is *not* updated.[21] Lastly, $\mathit{nr}' := (\pi_1(r), \pi_2(s), \pi_3(r), \pi_4(r), \pi_5(r), \pi_6(r), \pi_7(r) \cup \pi_7(s), \max(\pi_8(r), \pi_8(s)))$ is identical to nr except that the sequence number is replaced by the one from the route s.

One of the AODV control messages needs to be modified as well: the route reply. It is the only message type that carries, according to the RFC, a time parameter. It specifies the time for which nodes receiving the RREP message consider the route to be valid. The function that generates a RREP message has the form $\texttt{rrep} : \mathbb{N} \times \texttt{IP} \times \texttt{SQN} \times \texttt{IP} \times \texttt{TIME} \times \texttt{IP} \to \texttt{MSG}$.

Since $\texttt{P}$ is not a Boolean flag anymore, but of type $\mathbb{N}$, the functions `unsetRRF` and `setRRF` (for updating the request-required flag) are replaced by the functions `incRetries` and `resetRetries`, respectively.

The function `incRetries` increments the number of pending requests:

$$\texttt{incRetries} : \texttt{STORE} \times \texttt{IP} \to \texttt{STORE}$$

$$\texttt{incRetries}(\mathit{store}, \mathit{dip}) := \begin{cases} \mathit{store} - \{(\mathit{dip}, n, t, q)\} \cup \{(\mathit{dip}, n+1, t, q)\} & \\ & \text{if } (\mathit{dip}, n, t, q) \in \mathit{store} \\ \mathit{store} & \text{otherwise .} \end{cases}$$

The function `resetRetries` resets the number of pending requests (to 0):

$$\begin{aligned} &\texttt{resetRetries} : \texttt{STORE} \times (\texttt{IP} \rightharpoonup \texttt{SQN}) \to \texttt{STORE} \\ &\texttt{resetRetries}(\mathit{store}, \mathit{dests}) := \{\mathit{st} \mid \mathit{st} \in \mathit{store} \;\wedge\; (\pi_1(\mathit{st}), *) \notin \mathit{dests}\} \\ &\qquad\qquad \cup \{(\pi_1(\mathit{st}), 0, 0, \pi_4(\mathit{st})) \mid \\ &\qquad\qquad\qquad \mathit{st} \in \mathit{store} \;\wedge\; (\pi_1(\mathit{st}), *) \in \mathit{dests}\} \;. \end{aligned}$$

It also resets the waiting time before a new route request may be scheduled.

We define two new (partial) functions that extract the number of route requests already initiated for a particular destination, and the time one has to wait before a new route request may be scheduled, respectively:

$$\sigma_{\mathit{retries}} : \texttt{STORE} \times \texttt{IP} \rightharpoonup \texttt{P}$$

$$\sigma_{\mathit{retries}}(\mathit{store}, \mathit{dip}) := \begin{cases} p & \text{if } (\mathit{dip}, p, *, *) \in \mathit{store} \\ \text{undefined} & \text{otherwise ,} \end{cases}$$

$$\sigma_{\mathit{time}} : \texttt{STORE} \times \texttt{IP} \rightharpoonup \texttt{TIME}$$

$$\sigma_{\mathit{time}}(\mathit{store}, \mathit{dip}) := \begin{cases} t & \text{if } (\mathit{dip}, *, t, *) \in \mathit{store} \\ \text{undefined} & \text{otherwise .} \end{cases}$$

Finally, to cope with the newly introduced expiration times (lifetimes), we define new functions for modifying the routing tables and other data structures.

A (valid) route that has expired, has to be marked as invalid; and an expired invalid route has to be removed from the routing table. The function `exp_rt`

[21] We could have updated the expiration time to $\max(\pi_8(r), \pi_8(s))$; our results on loop freedom are not affected by this choice.

models this behaviour:

$$\begin{aligned}
&\texttt{exp\_rt} : \texttt{RT} \times \texttt{TIME} \times \texttt{TIME} \to \texttt{RT}\\
&\texttt{exp\_rt}(\mathit{rt}, t, t') := \{r \mid r \in \texttt{rt} \wedge \pi_8(r) > t \wedge \texttt{1hoplife}(\pi_6(r), t)\} \cup\\
&\qquad\{(\mathit{dip}, \texttt{inc}(\mathit{dsn}), \mathit{dsk}, \texttt{inv}, \mathit{hops}, \mathit{nhip}, \mathit{pre}, \mathit{lifetime} + t') \mid\\
&\qquad\quad(\mathit{dip}, \mathit{dsn}, \mathit{dsk}, \texttt{val}, \mathit{hops}, \mathit{nhip}, \mathit{pre}, \mathit{lifetime}) \in \texttt{rt}\\
&\qquad\quad\wedge (\mathit{lifetime} \leq t \vee \neg\texttt{1hoplife}(\mathit{nhip}, t))\\
&\qquad\quad\wedge \mathit{lifetime} + t' > t\}\ .
\end{aligned}$$

Here $\texttt{1hoplife}(\mathit{nhip}, t)$ is a shorthand for

$$(\mathit{nhip}, *, *, \texttt{val}, 1, *, *, \mathit{ltime}) \in \texttt{rt} \Rightarrow \mathit{ltime} > t\ ;$$

it says that if there is a valid routing table entry for node nhip with hop count 1 (in the routing table), then it is not yet expired. The first set keeps all routing table entries that have not expired at time t. Here we take into account two ways a routing table entry r can expire: when the (current) time t equals or exceeds its expiration time $\pi_8(r)$, or when the 1-hop routing table entry to its next hop expires [29, Section 6.1]. The second set selects all expired valid routes and marks them as invalid, thereby incrementing the destination sequence number; it also sets a new expiration time to indicate when the entry should be removed. In the (rare) case that even the new expiration time counts as expired, the entry is dropped. Expired invalid routes are not added to the created set, and are hence erased.

In applications we take $t = \texttt{now}$ and $t' = \texttt{DELETE\_PERIOD}$. In case an entry is invalidated, the new expiration time is set to be DELETE_PERIOD after the previous expiration time. So valid entries with $\mathit{lifetime} + \texttt{DELETE\_PERIOD} \leq \texttt{now}$ skip the phase of being invalid and are erased right away.

Similar to exp_rt we define a function that modifies the set of route request identifiers by expunging the expired ones.

$$\begin{aligned}
&\texttt{exp\_rreqs} : \mathscr{P}(\texttt{IP} \times \texttt{RREQID} \times \texttt{TIME}) \times \texttt{TIME} \to \mathscr{P}(\texttt{IP} \times \texttt{RREQID} \times \texttt{TIME})\\
&\texttt{exp\_rreqs}(\mathit{rreqs}, t) := \{\mathit{rq} \mid \mathit{rq} \in \mathit{rreqs} \wedge \pi_3(\mathit{rq}) > t\}^{22}\ .
\end{aligned}$$

In the same vain, we introduce a function that drops all packets enqueued for destinations that have RREQ_RETRIES pending route requests, and for which the waiting period has expired. This means that no further route request will be sent, and hence the packets will not be delivered.

$$\begin{aligned}
&\texttt{exp\_store} : \texttt{STORE} \times \texttt{TIME} \to \texttt{STORE}\\
&\texttt{exp\_store}(\mathit{store}, t') = \{(\mathit{dip}, p, t, *) \in \mathit{store} \mid p < \texttt{RREQ\_RETRIES} \vee t > t'\}
\end{aligned}$$

Last, but not least, we introduce two functions to update the expiration times in routing tables and in stores, respectively.

$$\begin{aligned}
&\texttt{setTime\_rt} : \texttt{RT} \times \texttt{IP} \times \texttt{TIME} \to \texttt{RT}\\
&\texttt{setTime\_rt}(\mathit{rt}, \mathit{dip}, t) := \begin{cases} \mathit{rt} - \{r\} \cup \{\mathit{nr}\} & \text{if } \mathit{dip} \in \texttt{kD}(\mathit{rt})\\ \mathit{rt} & \text{otherwise}\ , \end{cases}
\end{aligned}$$

[22] Projections on route requests identifiers are defined as usual. Here this means that $\pi_3 : \texttt{IP} \times \texttt{RREQID} \times \texttt{TIME} \to \texttt{TIME}$ determines the expiration time of the triple.

where $r := \sigma_{route}(rt, dip) = (dip, dsn, dsk, flag, hops, nhip, pre, ltime)$ is the current entry in the routing table for *dip* and $nr := (dip, dsn, dsk, flag, hops, nhip, pre, \max(ltime, t))$ is identical to r except for the expiration time, which is updated.

$$\texttt{setTime\_store} : \texttt{STORE} \times \texttt{IP} \times \texttt{TIME} \to \texttt{STORE}$$

$$\texttt{setTime\_store}(store, dip, t) := \begin{cases} store - \{(dip, p, *, q)\} \cup \{(dip, p, t, q)\} & \\ \qquad\qquad \text{if } (dip, p, *, q) \in store & \\ store \qquad \text{otherwise} & \end{cases}$$

Summary

Table 6 shows AODV's data structure; detailed explanations can be found in [11].

B.1.2 Modelling AODV

Our model of AODV consists of 7 processes, named AODV, NEWPKT, PKT, RREQ, RREP, RERR and QMSG; their formal specifications are displayed as Processes 1–7. The red-coloured parts are those bits that differ from the specification given in [11,15]. In this paper we only explain those parts, and refer to [11, Section 6] for a detailed explanation of all other parts.

The Basic Routine. The basic process AODV, depicted in Process 1, either handles a message from the corresponding queue, sends a queued data packet if a route to the destination has been established, or initiates a new route discovery process in case of queued data packets with invalid or unknown routes. This process maintains five data variables, ip, sn, rt, rreqs and store, in which it stores its own identity, its own sequence number, its current routing table, the list of route requests seen so far, and its current store of queued data packets that await transmission.

With timers in place, the routing table needs regular updates. In particular, valid routing table entries have to be invalidated, and invalid ones need to be erased when the expiration time of an entry has been reached. Hence each time before we use information from the routing table rt maintained by a node, we prune expired routes from the routing table, and invalidate routes that have not been used for a long time; this happens for instance in Line 2 of Process 1, so that the updated routing table is used when we evaluate the guard of Line 27, checking that there is a valid route to dip.

We again prune rt in Line 7, prior to for instance evaluating the guard in Line 5 of Process 3—repeated pruning is needed because time may have passed upon receiving the message in Line 6 of Process 1. A similar argument applies to Lines 30 and 36.

Likewise, before we consult the store of queued data packets (e.g. in Lines 27 and 43) we drop all packets from those queues for which RREQ_RETRIES unsuccessful attempts have been made to find a route to the destination (Line 3).

Each time a routing table entry is updated (Lines 16, 20 and 24) the lifetime of the entry is set to ACTIVE_ROUTE_TIMEOUT (so that the expiration time becomes

Table 6. Data structure of AODV

Type	**Variables**	**Description**
$\mathtt{IP}$	$\mathtt{ip}, \mathtt{dip}, \mathtt{oip}, \mathtt{rip}, \mathtt{sip}, \mathtt{nhip}$	node identifiers
$\mathtt{SQN}$	$\mathtt{dsn}, \mathtt{osn}, \mathtt{rsn}, \mathtt{sn}$	sequence numbers
$\mathtt{K}$	$\mathtt{dsk}$	sequence-number-status flag
$\mathtt{F}$	$\mathtt{flag}$	route validity
$\mathbb{N}$	$\mathtt{hops}$	hop counts
$\mathtt{R}$		routing table entries
$\mathtt{RT}$	$\mathtt{rt}$	routing tables
$\mathtt{RREQID}$	$\mathtt{rreqid}$	request identifiers
$\mathtt{P}$		pending-request counter
$\mathtt{STORE}$	$\mathtt{store}$	store of queued data packets
$\mathtt{MSG}$	$\mathtt{msg}$	messages
$[\mathtt{TYPE}]$		queues with elements of type $\mathtt{TYPE}$
$[\mathtt{MSG}]$	$\mathtt{msgs}$	message queues
$\mathtt{IP} \rightharpoonup \mathtt{SQN}$	$\mathtt{dests}$	sets of destinations with sequence numbers
$\mathscr{P}(\mathtt{IP})$	$\mathtt{pre}$	sets of identifiers (precursors, destinations, ...)
$\mathscr{P}(\mathtt{IP}\times\mathtt{RREQID}\times\mathtt{TIME})$	$\mathtt{rreqs}$	sets of request identifiers with originator IP

Constant/Predicate	**Description**
$\mathtt{kno}, \mathtt{unk} : \mathtt{K}$	constants to distinguish known and unknown sqns
$\mathtt{val}, \mathtt{inv} : \mathtt{F}$	constants to distinguish valid and invalid routes
$\mathtt{RREQ\_RETRIES} : \mathtt{P}$	maximal number of RREQ attempts
$\mathtt{DELETE\_PERIOD}, \mathtt{ACTIVE\_ROUTE\_TIMEOUT}, \mathtt{MY\_ROUTE\_TIMEOUT}, \mathtt{NODE\_TRAVERSAL\_TIME}, \mathtt{NET\_TRAVERSAL\_TIME}, \mathtt{PATH\_DISCOVERY\_TIME} : \mathtt{TIME}$	time constants

Operator	**Description**
$\mathtt{head} : [\mathtt{TYPE}] \rightharpoonup \mathtt{TYPE}$	returns the "oldest" element in the queue
$\mathtt{tail} : [\mathtt{TYPE}] \rightharpoonup [\mathtt{TYPE}]$	removes the "oldest" element in the queue
$\mathtt{append} : \mathtt{TYPE}\times[\mathtt{TYPE}] \rightarrow [\mathtt{TYPE}]$	inserts a new element into the queue
$\mathtt{drop} : \mathtt{IP}\times\mathtt{STORE} \rightharpoonup \mathtt{STORE}$	deletes a packet from the queued data packets
$\mathtt{add} : \mathtt{DATA}\times\mathtt{IP}\times\mathtt{STORE} \rightarrow \mathtt{STORE}$	adds a packet to the queued data packets
$\mathtt{incRetries} : \mathtt{STORE}\times\mathtt{IP} \rightarrow \mathtt{STORE}$	increments the number of pending requests
$\mathtt{resetRetries} : \mathtt{STORE}\times(\mathtt{IP} \rightharpoonup \mathtt{SQN}) \rightarrow \mathtt{STORE}$	resets the number of pending requests
$\sigma_{queue} : \mathtt{STORE}\times\mathtt{IP} \rightarrow [\mathtt{DATA}]$	selects the data queue for a particular destination
$\sigma_{retries} : \mathtt{STORE}\times\mathtt{IP} \rightharpoonup \mathtt{P}$	returns the number of route requests initiated
$\sigma_{time} : \mathtt{STORE}\times\mathtt{IP} \rightharpoonup \mathtt{TIME}$	tells when the next route request should be sent
$\sigma_{route} : \mathtt{RT}\times\mathtt{IP} \rightharpoonup \mathtt{R}$	selects the route for a particular destination
$(\_,\_,\_,\_,\_,\_,\_,\_) :$ $\mathtt{IP}\times\mathtt{SQN}\times\mathtt{K}\times\mathtt{F}\times\mathbb{N}\times\mathtt{IP}\times\mathscr{P}(\mathtt{IP})\times\mathtt{TIME} \rightarrow \mathtt{R}$	generates a routing table entry
$\mathtt{inc} : \mathtt{SQN} \rightarrow \mathtt{SQN}$	increments the sequence number
$\max : \mathtt{SQN}\times\mathtt{SQN} \rightarrow \mathtt{SQN}$	returns the larger sequence number
$\mathtt{sqn} : \mathtt{RT}\times\mathtt{IP} \rightarrow \mathtt{SQN}$	returns the sequence number of a particular route
$\mathtt{sqnf} : \mathtt{RT}\times\mathtt{IP} \rightarrow \mathtt{K}$	determines whether the sequence number is known
$\mathtt{flag} : \mathtt{RT}\times\mathtt{IP} \rightharpoonup \mathtt{F}$	returns the validity of a particular route
$\mathtt{dhops} : \mathtt{RT}\times\mathtt{IP} \rightharpoonup \mathbb{N}$	returns the hop count of a particular route
$\mathtt{nhop} : \mathtt{RT}\times\mathtt{IP} \rightharpoonup \mathtt{IP}$	returns the next hop of a particular route
$\mathtt{precs} : \mathtt{RT}\times\mathtt{IP} \rightharpoonup \mathscr{P}(\mathtt{IP})$	returns the set of precursors of a particular route
$\mathtt{ltime} : \mathtt{RT}\times\mathtt{IP} \rightharpoonup \mathtt{TIME}$	returns the expiration time of a particular route
$\mathtt{vD}, \mathtt{iD}, \mathtt{kD} : \mathtt{RT} \rightarrow \mathscr{P}(\mathtt{IP})$	returns the set of valid/invalid/known destinations
$\mathtt{qD} : \mathtt{STORE} \rightarrow \mathscr{P}(\mathtt{IP})$	returns the set of destinations with unsent packets
$\mathtt{addpre} : \mathtt{R}\times\mathscr{P}(\mathtt{IP}) \rightarrow \mathtt{R}$	adds a set of precursors to a routing table entry
$\mathtt{addpreRT} : \mathtt{RT}\times\mathtt{IP}\times\mathscr{P}(\mathtt{IP}) \rightharpoonup \mathtt{RT}$	adds a set of precursors to an entry inside a table
$\mathtt{update} : \mathtt{RT}\times\mathtt{R} \rightharpoonup \mathtt{RT}$	updates a routing table with a route
$\mathtt{invalidate} : \mathtt{RT}\times(\mathtt{IP} \rightharpoonup \mathtt{SQN}) \rightarrow \mathtt{RT}$	invalidates a set of routes within a routing table
$\mathtt{nrreqid} : \mathscr{P}(\mathtt{IP}\times\mathtt{RREQID}\times\mathtt{TIME})\times\mathtt{IP} \rightarrow \mathtt{RREQID}$	generates a new route request identifier
$\mathtt{newpkt} : \mathtt{DATA}\times\mathtt{IP} \rightarrow \mathtt{MSG}$	generates a message with new appl. layer data
$\mathtt{pkt} : \mathtt{DATA}\times\mathtt{IP}\times\mathtt{IP} \rightarrow \mathtt{MSG}$	generates a message containing appl. layer data
$\mathtt{rreq}{:}\mathbb{N}\times\mathtt{RREQID}\times\mathtt{IP}\times\mathtt{SQN}\times\mathtt{K}\times\mathtt{IP}\times\mathtt{SQN}\times\mathtt{IP} \rightarrow \mathtt{MSG}$	generates a route request
$\mathtt{rrep}{:}\mathbb{N}\times\mathtt{IP}\times\mathtt{SQN}\times\mathtt{IP}\times\mathtt{TIME}\times\mathtt{IP} \rightarrow \mathtt{MSG}$	generates a route reply
$\mathtt{rerr} : (\mathtt{IP} \rightharpoonup \mathtt{SQN})\times\mathtt{IP} \rightarrow \mathtt{MSG}$	generates a route error message
$\mathtt{exp\_rt} : \mathtt{RT}\times\mathtt{TIME}\times\mathtt{TIME} \rightarrow \mathtt{RT}$	invalidates/removes expired routing table entries
$\mathtt{exp\_rreqs} : \mathscr{P}(\mathtt{IP}\times\mathtt{RREQID}\times\mathtt{TIME})\times\mathtt{TIME}$ $\rightarrow \mathscr{P}(\mathtt{IP}\times\mathtt{RREQID}\times\mathtt{TIME})$	removes expired route request identifiers
$\mathtt{exp\_store} : \mathtt{STORE}\times\mathtt{TIME} \rightarrow \mathtt{STORE}$	removes expired entries form a store
$\mathtt{setTime\_rt} : \mathtt{RT}\times\mathtt{IP}\times\mathtt{TIME} \rightarrow \mathtt{RT}$	updates expiration time for a routing table entry
$\mathtt{setTime\_store} : \mathtt{STORE}\times\mathtt{IP}\times\mathtt{TIME} \rightarrow \mathtt{STORE}$	updates expiration time for an entry in the store

Process 1 The basic routine

```
AODV(ip, sn, rt, rreqs, store) $\overset{def}{=}$
 1. /* clean up routing table, and data storage */
 2. ⟦rt := exp_rt(rt, now, DELETE_PERIOD)⟧
 3. ⟦store := exp_store(store, now)⟧
 4. (
 5.     /* node receives a message */
 6.         receive(msg) .
 7.         ⟦rt := exp_rt(rt, now, DELETE_PERIOD)⟧
 8.         /* depending on the message, the node calls different processes */
 9.         (
10.             [ msg = newpkt(data, dip) ]       /* new DATA packet */
11.                 NEWPKT(data, dip , ip, sn, rt, rreqs, store)
12.             + [ msg = pkt(data, dip, oip) ]       /* incoming DATA packet */
13.                 PKT(data, dip, oip , ip, sn, rt, rreqs, store)
14.             + [ msg = rreq(hops, rreqid, dip, dsn, dsk, oip, osn, sip) ]       /* RREQ */
15.                 /* update the route to sip in rt */
16.                 ⟦rt := update(rt, (sip, 0, unk, val, 1, sip, $\emptyset$, now + ACTIVE_ROUTE_TIMEOUT))⟧
17.                 RREQ(hops, rreqid, dip, dsn, dsk, oip, osn, sip , ip, sn, rt, rreqs, store)
18.             + [ msg = rrep(hops, dip, dsn, oip, ltime, sip) ]       /* RREP */
19.                 /* update the route to sip in rt */
20.                 ⟦rt := update(rt, (sip, 0, unk, val, 1, sip, $\emptyset$, now + ACTIVE_ROUTE_TIMEOUT))⟧
21.                 RREP(hops, dip, dsn, oip, ltime, sip , ip, sn, rt, rreqs, store)
22.             + [ msg = rerr(dests, sip) ]       /* RERR */
23.                 /* update the route to sip in rt */
24.                 ⟦rt := update(rt, (sip, 0, unk, val, 1, sip, $\emptyset$, now + ACTIVE_ROUTE_TIMEOUT))⟧
25.                 RERR(dests, sip , ip, sn, rt, rreqs, store)
26.         )
27.     + [ Let dip $\in$ qD(store) $\cap$ vD(rt) ]       /* send a queued packet if a valid route is known */
28.         ⟦data := head($\sigma_{queue}$(store, dip))⟧
29.         unicast(nhop(rt, dip), pkt(data, dip, ip)) .
30.             ⟦rt := exp_rt(rt, now, DELETE_PERIOD)⟧
31.             ⟦store := drop(dip, store)⟧       /* drop data from the store for dip */
32.             ⟦rt := setTime_rt(rt, dip, now + ACTIVE_ROUTE_TIMEOUT)⟧
33.             ⟦rt := setTime_rt(rt, nhop(rt, dip), now + ACTIVE_ROUTE_TIMEOUT)⟧
34.             AODV(ip, sn, rt, rreqs, store)
35.         ▶ /* an error is produced and the routing table is updated */
36.             ⟦rt := exp_rt(rt, now, DELETE_PERIOD)⟧
37.             ⟦dests := {(rip, inc(sqn(rt, rip))) | rip $\in$ vD(rt) $\wedge$ nhop(rt, rip) = nhop(rt, dip)}⟧
38.             ⟦rt := invalidate(rt, dests, now + DELETE_PERIOD)⟧
39.             ⟦store := resetRetries(store, dests)⟧
40.             ⟦pre := $\bigcup${precs(rt, rip) | (rip, $*$) $\in$ dests}⟧
41.             ⟦dests := {(rip, rsn) | (rip, rsn) $\in$ dests $\wedge$ precs(rt, rip) $\neq \emptyset$}⟧
42.             groupcast(pre, rerr(dests, ip)) . AODV(ip, sn, rt, rreqs, store)
43.     + [ Let dip $\in$ qD(store) $-$ vD(rt) $\wedge$ $\sigma_{retries}$(store, dip) $<$ RREQ_RETRIES $\wedge$ $\sigma_{time}$(store, dip) $\leq$ now ]
          /* a route discovery process is initiated */
44.         ⟦store := incRetries(store, dip)⟧
45.         ⟦store := setTime_store(store, dip, now + $2^{\sigma_{retries}(\mathtt{store},\mathtt{dip})}$ $\cdot$ NET_TRAVERSAL_TIME)⟧
46.         ⟦rt := setTime_rt(rt, dip, now + 2 $\cdot$ NET_TRAVERSAL_TIME)⟧
47.         ⟦sn := inc(sn)⟧       /* increment own sequence number */
48.         /* update rreqs by adding (ip, nrreqid(rreqs, ip)) */
49.         ⟦rreqid := nrreqid(rreqs, ip)⟧
50.         ⟦rreqs := rreqs $\cup$ {(ip, rreqid, now + PATH_DISCOVERY_TIME)}⟧
51.         broadcast(rreq(0, rreqid, dip, sqn(rt, dip), sqnf(rt, dip), ip, sn, ip)) .
52.         AODV(ip, sn, rt, rreqs, store)
53. )
```

now + ACTIVE_ROUTE_TIMEOUT), according to [29, Section 6.2]. Likewise, after a route is used to forward a data packet (Line 29), the lifetime of the routing table entries for the destination and for the next hop on the path to the destination are updated in the same way (Line 32 and 33), again according to [29, Section 6.2]. The lifetime parameter of route reply messages is simply passed on from the incoming message of Line 18 to the process RREP in Line 21.

When invalidating routing table entries in Line 38, the expiration time of the invalidated entries is set to $\texttt{now}+\texttt{DELETE\_PERIOD}$, according to [29, Section 6.11]. For each of the newly invalidated destinations, a fresh route discovery process needs to be initiated. To this end, the number of pending route request for that destination is set to 0, and the time after which the next route request can be made to now (Line 39).

If the guard of Line 43 evaluates to true, a route discovery process for a destination dip will be initiated. For this to happen, according to [29, Section 6.3], the number $\sigma_{retries}(\texttt{store}, \texttt{dip})$ of pending route requests for dip needs to be smaller than the parameter RREQ_RETRIES. Moreover, the time we were instructed to wait for has been reached ($\sigma_{time}(\texttt{store}, \texttt{dip}) \leq \texttt{now}$). When a new route request is being made, the recorded number of pending route requests for dip is incremented (Line 44), and, again according to [29, Section 6.3], an instruction is processed to wait until time $\texttt{now} + 2^{\sigma_{retries}(\texttt{store},\texttt{dip})} \cdot \texttt{NET\_TRAVERSAL\_TIME}$ before issuing a new route request for dip (Line 45). Furthermore, Line 46 says that a routing table entry waiting for a route reply should not be expunged before time $\texttt{now} + 2 \cdot \texttt{NET\_TRAVERSAL\_TIME}$ [29, Section 6.4]. Finally, Line 50 indicates that “before broadcasting the RREQ, the originating node buffers the RREQ ID and the Originator IP address (its own address) of the RREQ for PATH_DISCOVERY_TIME” ([29, Section 6.3]).

Data Packet Handling. The process NEWPKT (Process 2), describing all actions performed by a node when a data packet is injected by a client hooked up to the local node, is unchanged w.r.t. [11,15].

Process 2 Routine for handling a newly injected data packet

```
NEWPKT(data, dip , ip, sn, rt, rreqs, store) $\stackrel{def}{=}$
  [ dip = ip ]        /* the DATA packet is intended for this node */
      deliver(data) . AODV(ip, sn, rt, rreqs, store)
  + [ dip $\neq$ ip ]        /* the DATA packet is not intended for this node */
      $[\![$store := add(data, dip, store)$]\!]$ . AODV(ip, sn, rt, rreqs, store)
```

In the process PKT (Process 3), dealing with data packets received via the protocol, a data packet is forwarded to the next hop on the route to the destination in Line 7. According to [29, Section 6.2], the expiration times of the routing table entries for the destination, the next hop on the path to the destination, the source and the next hop on the path to the source of the message are all set to $\texttt{now} + \texttt{ACTIVE\_ROUTE\_TIMEOUT}$ (Lines 9–12). The handling of an unsuccessful transmission is exactly as in Process 1. Line 26 says that “if a data packet is received for an invalid route, the lifetime field is updated to current time plus DELETE_PERIOD” [29, Section 6.11].

Receiving Route Requests. The process RREQ (Process 4) models all events that may occur after a route request has been received. In case the node itself is the intended destination of the RREQ message, the node generates a route reply (RREP) message, which is then sent along the established reverse route. A RREP message is also generated in case an intermediate node (a node that is

Process 3 Routine for handling a received data packet

```
PKT(data, dip, oip , ip, sn, rt, rreqs, store) $\stackrel{def}{=}$
 1. [ dip = ip ]        /* the DATA packet is intended for this node */
 2.     deliver(data) . AODV(ip, sn, rt, rreqs, store)
 3. + [ dip $\neq$ ip ]       /* the DATA packet is not intended for this node */
 4.     (
 5.         [ dip $\in$ vD(rt) ]       /* valid route to dip */
 6.             /* forward packet */
 7.             unicast(nhop(rt, dip), pkt(data, dip, oip)) .
 8.                 $[\![$rt := exp_rt(rt, now, DELETE_PERIOD)$]\!]$
 9.                 $[\![$rt := setTime_rt(rt, dip, now + ACTIVE_ROUTE_TIMEOUT)$]\!]$
10.                 $[\![$rt := setTime_rt(rt, nhop(rt, dip), now + ACTIVE_ROUTE_TIMEOUT)$]\!]$
11.                 $[\![$rt := setTime_rt(rt, oip, now + ACTIVE_ROUTE_TIMEOUT)$]\!]$
12.                 $[\![$rt := setTime_rt(rt, nhop(rt, oip), now + ACTIVE_ROUTE_TIMEOUT)$]\!]$
13.                 AODV(ip, sn, rt, rreqs, store)
14.             $\blacktriangleright$ /* If the packet transmission is unsuccessful, a RERR message is generated */
15.                 $[\![$rt := exp_rt(rt, now, DELETE_PERIOD)$]\!]$
16.                 $[\![$dests := {(rip, inc(sqn(rt, rip))) | rip $\in$ vD(rt) $\wedge$ nhop(rt, rip) = nhop(rt, dip)}$]\!]$
17.                 $[\![$rt := invalidate(rt, dests, now + DELETE_PERIOD)$]\!]$
18.                 $[\![$store := resetRetries(store, dests)$]\!]$
19.                 $[\![$pre := $\bigcup${precs(rt, rip) | (rip, $*$) $\in$ dests}$]\!]$
20.                 $[\![$dests := {(rip, rsn) | (rip, rsn) $\in$ dests $\wedge$ precs(rt, rip) $\neq \emptyset$}$]\!]$
21.                 groupcast(pre, rerr(dests, ip)) . AODV(ip, sn, rt, rreqs, store)
22.         + [ dip $\notin$ vD(rt) ]       /* no valid route to dip */
23.             /* no local repair occurs; data is lost */
24.             (
25.                 [ dip $\in$ iD(rt) ]       /* invalid route to dip */
26.                     $[\![$rt := setTime_rt(rt, dip, now + DELETE_PERIOD)$]\!]$
27.                     /* if the route is invalid, a RERR is sent to the precursors */
28.                     groupcast(precs(rt, dip), rerr({(dip, sqn(rt, dip))}, ip)) .
29.                     AODV(ip, sn, rt, rreqs, store)
30.                 + [ dip $\notin$ iD(rt) ]       /* route not in rt */
31.                     AODV(ip, sn, rt, rreqs, store)
32.             )
33.     )
```

neither the destination nor the originator of the RREQ message) receives it and has knowledge about a valid and fresh enough route to the destination.

Just as in Process 1, the process prunes expired routes from the routing table before reading the routing table. This happens in Lines 16 and 34. Likewise, before consulting the list of already handled route requests in Line 3 the process expunges expired entries from this list in Line 1.

In Line 6, the routing table for the originator `oip` of the received route request is updated. According to [29, Section 6.2], the lifetime of the entry is “initialized to `ACTIVE_ROUTE_TIMEOUT`”, whereas according to [29, Section 6.5], the expiration time “is set to be the maximum of (ExistingLifetime, MinimalLifetime)”, where MinimalLifetime =

$$\mathtt{now} + 2 \cdot \mathtt{NET\_TRAVERSAL\_TIME} - 2 \cdot (\mathtt{hops} + 1) \cdot \mathtt{NODE\_TRAVERSAL\_TIME}.$$

We implement both instructions, in Lines 6 and 7, thereby taking the maximum lifetime resulting from both instructions.

In Line 8 we add the unique identifier (`oip`, `rreqid`) for the current route request as a new entry in the list of already handled route requests; its expiration time is set to `now` + `PATH_DISCOVERY_TIME`, according to [29, Section 6.5].

Process 4 RREQ handling

```
RREQ(hops, rreqid, dip, dsn, dsk, oip, osn, sip , ip, sn, rt, rreqs, store) $\stackrel{def}{=}$
 1. ⟦exp_rreqs(rreqs, now)⟧
 2. (
 3.    [ (oip, rreqid, $*$) $\in$ rreqs ]       /* the RREQ has been received previously */
 4.       AODV(ip, sn, rt, rreqs, store)       /* silently ignore RREQ, i.e., do nothing */
 5.    + [ (oip, rreqid, $*$) $\notin$ rreqs ]       /* the RREQ is new to this node */
 6.       ⟦rt := update(rt, (oip, osn, kno, val, hops + 1, sip, $\emptyset$, now + ACTIVE_ROUTE_TIMEOUT))⟧
 7.       ⟦rt := setTime_rt(rt, oip, now + 2 $\cdot$ NET_TRAVERSAL_TIME $-$ 2 $\cdot$ (hops + 1) $\cdot$ NODE_TRAVERSAL_TIME)⟧
 8.       ⟦rreqs := rreqs $\cup$ {(oip, rreqid, now + PATH_DISCOVERY_TIME)}⟧       /* update rreqs */
 9.       (
10.          [ dip = ip ]       /* this node is the destination node */
11.             ⟦sn := max(sn, dsn)⟧       /* update the sqn of ip */
12.             /* unicast a RREP towards oip of the RREQ */
13.             unicast(nhop(rt, oip), rrep(0, dip, sn, oip, MY_ROUTE_TIMEOUT, ip)) .
14.                AODV(ip, sn, rt, rreqs, store)
15.             ▶ /* If the transmission is unsuccessful, a RERR message is generated */
16.                ⟦rt := exp_rt(rt, now, DELETE_PERIOD)⟧
17.                ⟦dests := {(rip, inc(sqn(rt, rip))) | rip $\in$ vD(rt) $\wedge$ nhop(rt, rip) = nhop(rt, oip)}⟧
18.                ⟦rt := invalidate(rt, dests, now + DELETE_PERIOD)⟧
19.                ⟦store := resetRetries(store, dests)⟧
20.                ⟦pre := $\bigcup${precs(rt, rip) | (rip, $*$) $\in$ dests}⟧
21.                ⟦dests := {(rip, rsn) | (rip, rsn) $\in$ dests $\wedge$ precs(rt, rip) $\neq \emptyset$}⟧
22.                groupcast(pre, rerr(dests, ip)) . AODV(ip, sn, rt, rreqs, store)
23.          + [ dip $\neq$ ip ]       /* this node is not the destination node */
24.             (
25.                /* valid route to dip that is fresh enough */
26.                [ dip $\in$ vD(rt) $\wedge$ dsn $\leq$ sqn(rt,dip) $\wedge$ sqnf(rt,dip) = kno ]
27.                   /* update rt by adding precursors */
28.                   ⟦rt := addpreRT(rt, dip, {sip})⟧
29.                   ⟦rt := addpreRT(rt, oip, {nhop(rt, dip)})⟧
30.                   /* unicast a RREP towards the oip of the RREQ */
31.                   unicast(nhop(rt, oip),
                                rrep(dhops(rt, dip), dip, sqn(rt, dip), oip, $\sigma_{time}$(rt, dip) $-$ now, ip) .
32.                      AODV(ip, sn, rt, rreqs, store)
33.                   ▶ /* If the transmission is unsuccessful, a RERR message is generated */
34.                      ⟦rt := exp_rt(rt, now, DELETE_PERIOD)⟧
35.                      ⟦dests := {(rip, inc(sqn(rt, rip))) |
                                   rip $\in$ vD(rt) $\wedge$ nhop(rt, rip) = nhop(rt, oip)}⟧
36.                      ⟦rt := invalidate(rt, dests, now + DELETE_PERIOD)⟧
37.                      ⟦store := resetRetries(store, dests)⟧
38.                      ⟦pre := $\bigcup${precs(rt, rip) | (rip, $*$) $\in$ dests}⟧
39.                      ⟦dests := {(rip, rsn) | (rip, rsn) $\in$ dests $\wedge$ precs(rt, rip) $\neq \emptyset$}⟧
40.                      groupcast(pre, rerr(dests, ip)) . AODV(ip, sn, rt, rreqs, store)
41.                + [ dip $\notin$ vD(rt) $\vee$ sqn(rt,dip) < dsn $\vee$ sqnf(rt,dip) = unk ]       /* no fresh route */
42.                   /* no further update of rt */
43.                   broadcast(rreq(hops+1, rreqid, dip, max(sqn(rt, dip), dsn), dsk, oip, osn, ip)) .
44.                   AODV(ip, sn, rt, rreqs, store)
45.             )
46.       )
47. )
```

In Line 13, when sending a route reply in answer to the incoming route request because the current node *is* the destination of the request, "the destination node copies the value MY_ROUTE_TIMEOUT [. . .] into the Lifetime field of the RREP" [29, Section 6.6.1]. However, when sending a route reply "as an intermediate hop along the path from the originator to the destination" (Line 31), "the Lifetime field of the RREP is calculated by subtracting the current time from the expiration time in its route table entry" [29, Section 6.6.2].

The treatment of an unsuccessful unicast (Lines 15–22 and Lines 33–40) is exactly as in Process 1.

Receiving Route Replies. We handle a received route reply only if it would give rise to a genuine update to the routing table entry for the destination dip of the original route request (Lines 1 and 28), not counting updates to the lifetime of that entry. When we do update the routing table (Line 2), "the expiry time is set to the current time plus the value of the Lifetime in the RREP message" [29, Section 6.7].

Process 5 RREP handling

```
RREP(hops, dip, dsn, oip, ltime, sip , ip, sn, rt, rreqs, store) $\stackrel{def}{=}$
 1. [rt $\neq$ update(rt, (dip, dsn, kno, val, hops + 1, sip, $\emptyset$, 0))]      /*routing table has to be updated*/
 2.    $[\![$rt := update(rt, (dip, dsn, kno, val, hops + 1, sip, $\emptyset$, now + ltime))$]\!]$
 3.    (
 4.       [ oip = ip ]      /* this node is the originator of the corresponding RREQ */
 5.          /* a packet may now be sent; this is done in the process AODV */
 6.          AODV(ip, sn, rt, rreqs, store)
 7.       + [ oip $\neq$ ip ]      /* this node is not the originator; forward RREP */
 8.          (
 9.             [ oip $\in$ vD(rt) ]      /* valid route to oip */
10.                /* add next hop towards oip as precursor and forward the route reply */
11.                $[\![$rt := addpreRT(rt, dip, {nhop(rt, oip)})$]\!]$
12.                $[\![$rt := addpreRT(rt, nhop(rt, dip), {nhop(rt, oip)})$]\!]$
13.                $[\![$rt := setTime_rt(rt, oip, now + ACTIVE_ROUTE_TIMEOUT)$]\!]$
14.                unicast(nhop(rt, oip), rrep(hops+1, dip, dsn, oip, ltime, ip)) .
15.                   AODV(ip, sn, rt, rreqs, store)
16.                $\blacktriangleright$ /* If the transmission is unsuccessful, a RERR message is generated */
17.                   $[\![$rt := exp_rt(rt, now, DELETE_PERIOD)$]\!]$
18.                   $[\![$dests := {(rip, inc(sqn(rt, rip))) |
                               rip $\in$ vD(rt) $\wedge$ nhop(rt, rip) = nhop(rt, oip)}$]\!]$
19.                   $[\![$rt := invalidate(rt, dests, now + DELETE_PERIOD)$]\!]$
20.                   $[\![$store := resetRetries(store, dests)$]\!]$
21.                   $[\![$pre := $\bigcup$ {precs(rt, rip) | (rip, $*$) $\in$ dests}$]\!]$
22.                   $[\![$dests := {(rip, rsn) | (rip, rsn) $\in$ dests $\wedge$ precs(rt, rip) $\neq$ $\emptyset$}$]\!]$
23.                   groupcast(pre, rerr(dests, ip)) . AODV(ip, sn, rt, rreqs, store)
24.             + [ oip $\notin$ vD(rt) ]      /* no valid route to oip */
25.                AODV(ip, sn, rt, rreqs, store)
26.          )
27.    )
28. + [ rt = update(rt, (dip, dsn, kno, val, hops + 1, sip, $\emptyset$, 0)) ]      /*routing table is not updated*/
29.    AODV(ip, sn, rt, rreqs, store)
```

As implemented in Line 13, "the (reverse) route used to forward a RREP has its lifetime changed to be the maximum of (existing-lifetime, (current time + ACTIVE_ROUTE_TIMEOUT)") [29, Section 6.7]. This literal reading of the RFC seems a bit weird, since the route to oip is not updated otherwise. Although not specified in the RFC, it would make sense to also add a precursor to the reverse route by $[\![\mathtt{rt} := \mathtt{addpreRT}(\mathtt{rt}, \mathtt{oip}, \{\mathtt{nhop}(\mathtt{rt}, \mathtt{dip})\})]\!]$. Inserting this line, would not change the results and proofs presented in this paper.

Receiving Route Errors. The process RERR models the part of AODV that handles error messages. An error message consists of a set dests of pairs of an unreachable destination IP address rip and the corresponding unreachable destination sequence number rsn. The adaptations to this process are just as the ones discussed earlier.

Process 6 RERR handling

```
$\mathtt{RERR}(\mathtt{dests},\mathtt{sip}, \mathtt{ip},\mathtt{sn},\mathtt{rt},\mathtt{rreqs},\mathtt{store}) \stackrel{def}{=}$
/* invalidate broken routes */
$[\![\mathtt{dests} := \{(\mathtt{rip},\mathtt{rsn}) \mid (\mathtt{rip},\mathtt{rsn}) \in \mathtt{dests} \wedge \mathtt{rip} \in \mathtt{vD}(\mathtt{rt}) \wedge \mathtt{nhop}(\mathtt{rt},\mathtt{rip}) = \mathtt{sip} \wedge \mathtt{sqn}(\mathtt{rt},\mathtt{rip}) < \mathtt{rsn}\}]\!]$
$[\![\mathtt{rt} := \mathtt{invalidate}(\mathtt{rt},\mathtt{dests},\mathtt{now} + \mathtt{DELETE\_PERIOD})]\!]$
$[\![\mathtt{store} := \mathtt{resetRetries}(\mathtt{store},\mathtt{dests})]\!]$
/* forward the RERR to all precursors for rt entries for broken connections */
$[\![\mathtt{pre} := \bigcup\{\mathtt{precs}(\mathtt{rt},\mathtt{rip}) \mid (\mathtt{rip},*) \in \mathtt{dests}\}]\!]$
$[\![\mathtt{dests} := \{(\mathtt{rip},\mathtt{rsn}) \mid (\mathtt{rip},\mathtt{rsn}) \in \mathtt{dests} \wedge \mathtt{precs}(\mathtt{rt},\mathtt{rip}) \neq \emptyset\}]\!]$
**groupcast**$(\mathtt{pre},\mathtt{rerr}(\mathtt{dests},\mathtt{ip}))$ . $\mathtt{AODV}(\mathtt{ip},\mathtt{sn},\mathtt{rt},\mathtt{rreqs},\mathtt{store})$
```

The Message Queue. Since we have to guarantee input-enabledness of all network nodes, a node *ip* must always be able to perform a receive action, regardless of which state it is in. For this reason we introduce a process QMSG, modelling a message queue, that runs in parallel with AODV or any other process that might be called. This process is unchanged w.r.t. [11,15].

Process 7 Message queue

```
$\mathtt{QMSG}(\mathtt{msgs}) \stackrel{def}{=}$
    /* store incoming message at the end of msgs */
    **receive**$(\mathtt{msg})$ . $\mathtt{QMSG}(\mathtt{append}(\mathtt{msg},\mathtt{msgs}))$
  + [ $\mathtt{msgs} \neq [\,]$ ]        /* the queue is not empty */
    (
       /* pop top message and send it to another sequential process */
       **send**$(\mathtt{head}(\mathtt{msgs}))$ . $\mathtt{QMSG}(\mathtt{tail}(\mathtt{msgs}))$
       /* or receive and store an incoming message */
       + **receive**$(\mathtt{msg})$ . $\mathtt{QMSG}(\mathtt{append}(\mathtt{msg},\mathtt{msgs}))$
    )
```

B.2 Invariants

We now analyse our timed version of AODV. We will go through the propositions proved for AODV without time in [11,15]—up to the proof of loop freedom—and check whether they still hold. Most propositions still hold and the proofs are, mutatis mutandis, the same as the proofs for AODV without time. Changes mainly concern line numbers, and the changes triggered by the introduction of the new functions that can modify the routing table, namely setTime_rt and exp_rt, as well as the function that can modify the set of route request identifiers, namely exp_rreqs. The modification of the other functions and of data types are mainly to include the role of time; they do not modify their roles.

A transition $N \xrightarrow{\tau} N'$ between two network expressions may arise from a transition $R\!:\!{*}\textbf{cast}(m)$ performed by a network node *ip*, synchronising with receive actions of all nodes $dip \in R$ in transmission range. In this case, we write $N \xrightarrow{R:*\textbf{cast}(m)}_{ip} N'$. This means that $N = [M]$ and $N' = [M']$ are network expressions such that $M \xrightarrow{R:*\textbf{cast}(m)} M'$, and the cast action is performed by node *ip*. This transition stems ultimately from an action **broadcast**(*ms*), **groupcast**(*dests*, *ms*), or **unicast**(*dest*, *ms*) (cf. Section 2). Each such action can be identified by a line number in one of the processes of Appendix B.1.2.

With $\xi_N^{ip}(\texttt{var})$ we denote the evaluation $\xi(\texttt{var})$ of the variable var maintained by node *ip* when AODV is in state N—see [11, Section 7.2] or [15, Section 6.2] for further explanation.

In B.1.1 we have defined functions that work on evaluated routing tables $\xi_N^{ip}(\texttt{rt})$, such as nhop. To ease readability, we abbreviate $\texttt{nhop}(\xi_N^{ip}(\texttt{rt}),dip)$ by $\texttt{nhop}_N^{ip}(dip)$. Similarly, we use $\texttt{sqn}_N^{ip}(dip)$, $\texttt{dhops}_N^{ip}(dip)$, $\texttt{flag}_N^{ip}(dip)$, $\texttt{ltime}_N^{ip}(dip)$, $\texttt{kD}_N^{ip}$, $\texttt{vD}_N^{ip}$ and $\texttt{iD}_N^{ip}$ for $\texttt{sqn}(\xi_N^{ip}(\texttt{rt}), dip)$, $\texttt{dhops}(\xi_N^{ip}(\texttt{rt}), dip)$, $\texttt{flag}(\xi_N^{ip}(\texttt{rt}), dip)$, $\texttt{ltime}(\xi_N^{ip}(\texttt{rt}), ip)$, $\texttt{kD}(\xi_N^{ip}(\texttt{rt}))$, $\texttt{vD}(\xi_N^{ip}(\texttt{rt}))$ and $\texttt{iD}(\xi_N^{ip}(\texttt{rt}))$, respectively.

B.2.1 Basic Properties

Proposition B.1. [11, Proposition 7.1]

(a) With the exception of new packets that are submitted to a node by a client of AODV, every message received and handled by the main routine of AODV has to be sent by some node before. More formally, we consider an arbitrary path $N_0 \xrightarrow{\ell_1} N_1 \xrightarrow{\ell_2} \dots \xrightarrow{\ell_k} N_k$ with N_0 an initial state in our model of AODV. If the transition $N_{k-1} \xrightarrow{\ell_k} N_k$ results from a synchronisation involving the action **receive**(msg) from Line 6 of Pro. 1—performed by the node *ip*—, where the variable msg is assigned the value m, then either $m = \texttt{newpkt}(d, dip)$ or one of the ℓ_i with $i < k$ stems from an action ***cast**(m) of a node ip' of the network.

(b) No node can receive a message directly from itself. Using the formalisation above, we must have $ip \neq ip'$.

Proof. Exactly as in [11]. (The process QMSG has not been changed.) □

Proposition B.2. [11, Proposition 7.2] The sequence number of any given node *ip* increases monotonically, i.e., never decreases, and is never unknown. That is, for $ip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $1 \leq \xi_N^{ip}(\texttt{sn}) \leq \xi_{N'}^{ip}(\texttt{sn})$.

Proof. Exactly as in [11]. There are no additional methods to modify a node's own sequence number. □

Remark B.1. Most of the forthcoming proofs can be done by showing the statement for each initial state and then checking all locations in the processes where the validity of the invariant is possibly changed. Note that routing table entries are only changed by the functions update, invalidate, addpreRT, setTime_rt or exp_rt.[23] Thus we have to show that an invariant dealing with routing tables is satisfied after the execution of these functions if it was valid before. In our proofs, we go through all occurrences of these functions. In case the invariant does not make statements about precursors, the function addpreRT need not be considered.

Proposition 7.4 in [11] says that the set of known destinations of a node increases monotonically. That is, for $ip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $\texttt{kD}_N^{ip} \subseteq \texttt{kD}_{N'}^{ip}$. This proposition no longer holds since exp_rt can remove routing table entries.

[23] The functions setTime_rt or exp_rt are added w.r.t. [11, Remark 7.3].

Proposition 7.5 in [11] says that the set of already seen route requests of a node increases monotonically. That is, for $ip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $\xi_N^{ip}(\texttt{rreqs}) \subseteq \xi_{N'}^{ip}(\texttt{rreqs})$. This proposition no longer holds since the function `exp_rreqs` prunes the list of route request seen by a node.

Proposition 7.6 in [11] says that in each node's routing table, the sequence number for any given destination increases monotonically, i.e., never decreases. This proposition no longer holds since routing table entries can be removed and recreated with an inferior sequence number. However, we have the following weakening.

Proposition B.3. In each node's routing table, the sequence number for any given destination, as long as it is not deleted, increases monotonically. That is, for $ip, dip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ and $dip \in \texttt{kD}_N^{ip} \cap \texttt{kD}_{N'}^{ip}$ then $\texttt{sqn}_N^{ip}(dip) \leq \texttt{sqn}_{N'}^{ip}(dip)$.

Proof. Identical to the proof of Proposition 7.6 in [11]. □

The next invariant tells that each node is correctly informed about its own identity.

Proposition B.4. [11, Proposition 7.7] For each $ip \in \mathbf{IP}$ and each reachable state N we have $\xi_N^{ip}(\texttt{ip}) = ip$.

Proof. Exactly as in [11]; there are no modifications of the variable `ip`. □

Proposition B.5. [11, Proposition 7.8] If an AODV control message is sent by node $ip \in \mathbf{IP}$, the node sending this message identifies itself correctly:

$$N \xrightarrow{R:\textbf{*cast}(m)}_{ip} N' \ \Rightarrow\ ip = ip_c\ ,$$

where the message m is either $\texttt{rreq}(*,*,*,*,*,*,*,ip_c)$, $\texttt{rrep}(*,*,*,*,*,ip_c)$, or $\texttt{rerr}(*,ip_c)$.

Proof. Exactly as in [11], using Proposition B.4. □

Corollary B.1. [11, Corollary 7.9] At no point will the variable `sip` maintained by node ip have the value ip.

$$\xi_N^{ip}(\texttt{sip}) \neq ip$$

Proof. The same proof as in [11], mutatis mutandis (different line numbers).

Proposition B.6. [11, Proposition 7.10] All routing table entries have a hop count greater than or equal to 1.

$$(*,*,*,*,hops,*,*,*) \in \xi_N^{ip}(\texttt{rt}) \ \Rightarrow\ hops \geq 1 \qquad (1)$$

Proof. Essentially the same proof as in [11], following Remark B.1. We have to consider the new functions `setTime_rt` and `exp_rt` and change the line numbers. `setTime_rt` does not modify the hop count. `exp_rt` either removes the entry or leaves the hop count unchanged. In both cases, the invariant is preserved. □

Proposition B.7. [11, Proposition 7.11]

(a) If a route request with hop count 0 is sent by a node $ip_c \in \mathbf{IP}$, the sender must be the originator.

$$N \xrightarrow{R{:}\textbf{*cast}(\texttt{rreq}(0,*,*,*,*,oip_c,*,ip_c))}_{ip} N' \;\Rightarrow\; oip_c = ip_c (= ip) \tag{2}$$

(b) If a route reply with hop count 0 is sent by a node $ip_c \in \mathbf{IP}$, the sender must be the destination.

$$N \xrightarrow{R{:}\textbf{*cast}(\texttt{rrep}(0,dip_c,*,*,*,ip_c))}_{ip} N' \;\Rightarrow\; dip_c = ip_c (= ip) \tag{3}$$

Proof. The same proof as in [11], mutatis mutandis; it uses Proposition B.6. □

Proposition B.8. [11, Proposition 7.12]

(a) Each routing table entry with 0 as its destination sequence number has a sequence-number-status flag valued unknown.

$$(dip, 0, f, *, *, *, *, *) \in \xi_N^{ip}(\texttt{rt}) \;\Rightarrow\; f = \texttt{unk} \tag{4}$$

(b) Unknown sequence numbers can only occur at 1-hop connections.

$$(*, *, \texttt{unk}, *, hops, *, *, *) \in \xi_N^{ip}(\texttt{rt}) \;\Rightarrow\; hops = 1 \tag{5}$$

(c) 1-hop connections must contain the destination as next hop.

$$(dip, *, *, *, 1, nhip, *, *) \in \xi_N^{ip}(\texttt{rt}) \;\Rightarrow\; dip = nhip \tag{6}$$

(d) If the sequence number 0 occurs within a routing table entry, the hop count as well as the next hop can be determined.

$$(dip, 0, f, *, hops, nhip, *, *) \in \xi_N^{ip}(\texttt{rt}) \;\Rightarrow\; f{=}\texttt{unk} \wedge hops{=}1 \wedge dip{=}nhip \tag{7}$$

Proof. At the initial states all routing tables are empty. Since `setTime_rt`, `exp_rt` and `addpreRT` neither decrease the sequence number nor change the sequence-number-status flag, the next hop or the hop count of a routing table entry, they cannot invalidate any of the above invariants. The function `invalidate` changes neither the sequence-number-status flag, nor the next hop or the hop count, but could decrease the sequence number of an entry. The proof in [11] points to [11, Proposition 7.6] to show that this cannot happen. Here this follows from Proposition B.3. For this reason, we still can disregard applications of `invalidate`. Hence, as in [11], one only has to look at the application calls of `update`. The remainder of the proof follows [11], mutatis mutandis. It uses Proposition B.7. □

Proposition B.9. [11, Proposition 7.13]

(a) Whenever an originator sequence number is sent as part of a route request message, it is known, i.e., it is greater than or equal to 1.

$$N \xrightarrow{R{:}\textbf{*cast}(\texttt{rreq}(*,*,*,*,*,*,osn_c,*))}_{ip} N' \;\Rightarrow\; osn_c \geq 1 \tag{8}$$

(b) Whenever a destination sequence number is sent as part of a route reply message, it is known, i.e., it is greater than or equal to 1.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rrep}(*,*,\mathit{dsn}_c,*,*,*))}_{ip} N' \ \Rightarrow\ \mathit{dsn}_c \geq 1 \tag{9}$$

Proof. Just as in [11], mutatis mutandis, using Propositions B.1, B.2 and B.8. □

Proposition B.10. [11, Proposition 7.14]

(a) If a route request is sent (forwarded) by a node ip_c different from the originator of the request then the content of ip_c's routing table must be fresher or at least as good as the information inside the message.

$$\begin{aligned} &N \xrightarrow{R:\textbf{*cast}(\texttt{rreq}(\mathit{hops}_c,*,*,*,*,\mathit{oip}_c,\mathit{osn}_c,\mathit{ip}_c))}_{ip} N' \ \wedge\ \mathit{ip}_c \neq \mathit{oip}_c \\ &\Rightarrow \mathit{oip}_c \in \texttt{kD}_N^{\mathit{ip}_c} \ \wedge\ \big(\texttt{sqn}_N^{\mathit{ip}_c}(\mathit{oip}_c) > \mathit{osn}_c \ \vee\ (\texttt{sqn}_N^{\mathit{ip}_c}(\mathit{oip}_c) = \mathit{osn}_c \\ &\quad \wedge\ \texttt{dhops}_N^{\mathit{ip}_c}(\mathit{oip}_c) \leq \mathit{hops}_c \ \wedge\ \texttt{flag}_N^{\mathit{ip}_c}(\mathit{oip}_c) = \texttt{val})\big) \end{aligned} \tag{10}$$

(b) If a route reply is sent by a node ip_c, different from the destination of the route, then the content of ip_c's routing table must be consistent with the information inside the message.

$$\begin{aligned} &N \xrightarrow{R:\textbf{*cast}(\texttt{rrep}(\mathit{hops}_c,\mathit{dip}_c,\mathit{dsn}_c,*,*,\mathit{ip}_c))}_{ip} N' \ \wedge\ \mathit{ip}_c \neq \mathit{dip}_c \\ &\Rightarrow \mathit{dip}_c \in \texttt{kD}_N^{\mathit{ip}_c} \ \wedge\ \texttt{sqn}_N^{\mathit{ip}_c}(\mathit{dip}_c) = \mathit{dsn}_c \\ &\quad \wedge\ \texttt{dhops}_N^{\mathit{ip}_c}(\mathit{dip}_c) = \mathit{hops}_c \ \wedge\ \texttt{flag}_N^{\mathit{ip}_c}(\mathit{dip}_c) = \texttt{val} \end{aligned} \tag{11}$$

Proof. The same proof as in [11], mutatis mutandis, using Proposition B.4.

Proposition B.10 states facts about the network state at the end of a ***cast**-action. Since the evaluation function of a node does not change while transmitting a message (taking a τ-action stemming from rules (bc), (gc) and (uc) of Table 1, an R: w-action or an R : ***cast**-action), a similar proposition can be shown for all such actions. If we consider the start of a transmission (the τ-action), we can even strengthen the proposition. We show this only for the case of unicasting a RREP message (the strengthened version of Proposition B.10(b)).

Proposition B.11. If the sending of a route reply is initiated by a node ip_c, different from the destination of the route, then the content of ip_c's routing table must be consistent with the information inside the message, including the lifetime field. Moreover the sequence number is known (the sequence-number-status flag is set to `kno`).

$$\begin{aligned} &N \xrightarrow{\textbf{unicast}(*,\texttt{rrep}(\mathit{hops}_c,\,\mathit{dip}_c,\,\mathit{dsn}_c,\,*,\,*,\,\mathit{ip}_c))}_{ip} N' \ \wedge\ \mathit{ip}_c \neq \mathit{dip}_c \\ &\Rightarrow \mathit{dip}_c \in \texttt{kD}_N^{\mathit{ip}_c} \ \wedge\ \texttt{sqn}_N^{\mathit{ip}_c}(\mathit{dip}_c) = \mathit{dsn}_c \\ &\quad \wedge\ \texttt{dhops}_N^{\mathit{ip}_c}(\mathit{dip}_c) = \mathit{hops}_c \ \wedge\ \texttt{flag}_N^{\mathit{ip}_c}(\mathit{dip}_c) = \texttt{val} \\ &\quad \wedge\ \texttt{sqnf}(\xi_N^{\mathit{ip}_c}(\texttt{rt}),\mathit{dip}_c) = \texttt{kno} \ \wedge\ \texttt{ltime}_N^{\mathit{ip}_c}(\mathit{dip}_c) > \xi_N^{\mathit{ip}_c}(\texttt{now})\ , \end{aligned} \tag{12}$$

where the label is a new notation indicating that a transition stemming from rule (uc) with $\mathit{ms} = \texttt{rrep}(\mathit{hops}_c, \mathit{dip}_c, \mathit{dsn}_c, *, *, \mathit{ip}_c)$ is taken by node ip.

Proof. The same proof as for Proposition 7.14(b) in [11], mutatis mutandis. The last line, however, did not occur in [11]. We extend the proof to justify this addition.

Pro. 4, Line 13: As in [11] a new route reply with $ip_c := \xi(\mathtt{ip}) = ip$ is initiated. Moreover, by Line 10, $dip_c := \xi(\mathtt{dip}) = \xi(\mathtt{ip}) = ip$ and thus $ip_c = dip_c$. Hence, the antecedent of (12) is not satisfied.

Pro. 4, Line 31: That $\mathtt{sqnf}(\xi_N^{ip_c}(\mathtt{rt}), dip_c) = \mathtt{kno}$ follows from Line 26; that $\mathtt{ltime}_N^{ip_c}(dip_c) > \xi_N^{ip_c}(\mathtt{now})$ follows from Line 7 of Pro. 1, which is always executed prior to Line 31 or Pro. 4, in the same time slice.

Pro. 5, Line 14: That $\mathtt{sqnf}(\xi_N^{ip_c}(\mathtt{rt}), dip_c) = \mathtt{kno}$ and $\mathtt{ltime}_N^{ip_c}(dip_c) > \xi_N^{ip_c}(\mathtt{now})$ follows from the `update` done in Line 2, with Line 1 ensuring its effect. □

Proposition B.12. [11, Proposition 7.15] Any sequence number appearing in a route error message stems from an invalid destination and is equal to the sequence number for that destination in the sender's routing table at the time of sending.

$$\begin{aligned} &N \xrightarrow{R:\mathbf{*cast}(\mathtt{rerr}(dests_c, ip_c))}_{ip} N' \ \wedge\ (rip_c, rsn_c) \in dests_c \\ \Rightarrow\ & rip_c \in \mathtt{iD}_N^{ip} \ \wedge\ rsn_c = \mathtt{sqn}_N^{ip}(rip_c) \end{aligned} \tag{13}$$

Proof. Same proof as in [11], mutatis mutandis. □

Propositions 7.16–7.25 in [11] show that all partial functions used in the specification of AODV are always defined when they occur outside of an atomic formula (when an undefined function call occurs in an atomic formula, that formula evaluates to `false`—cf. Footnote 6). The proofs, which use Propositions B.1 and B.9, apply to our timed model of AODV as well. Moreover, the arguments for the new partial functions $\sigma_{retries}$ and σ_{time} are identical to the argument for $\sigma_{p\text{-}flag}$ in [11, Proposition 7.25].

B.2.2 The Quality of Routing Table Entries

In [11, Section 7.5] the *net sequence number* of a route to a destination *dip* in a routing table *rt* is defined by

$$\begin{aligned} &\mathtt{nsqn} : \mathtt{RT} \times \mathtt{IP} \ \rightarrow\ \mathtt{SQN} \\ &\mathtt{nsqn}(rt, dip) \ := \begin{cases} \mathtt{sqn}(rt, dip) & \text{if } \mathtt{flag}(rt, dip) = \mathtt{val} \ \vee\ \mathtt{sqn}(rt, dip) = 0 \\ \mathtt{sqn}(rt, dip) - 1 & \text{otherwise .} \end{cases} \end{aligned}$$

If two routing tables *rt* and *rt′* have a routing table entry to destination *dip*, i.e., $dip \in \mathtt{kD}(rt) \cap \mathtt{kD}(rt')$, they can be compared w.r.t. their *quality* for that destination [11]:

$$\begin{aligned} rt \sqsubseteq_{dip} rt' \ :\Leftrightarrow\ & \mathtt{nsqn}(rt, dip) < \mathtt{nsqn}(rt', dip) \ \vee \\ & \big(\mathtt{nsqn}(rt, dip) = \mathtt{nsqn}(rt', dip) \wedge \mathtt{dhops}(rt, dip) \geq \mathtt{dhops}(rt', dip)\big) \end{aligned}$$

For all destinations $dip \in \mathbf{IP}$, the relation $\sqsubseteq_{dip}$ on routing tables with an entry for *dip* is a total preorder. The equivalence relation induced by $\sqsubseteq_{dip}$ is denoted by $\approx_{dip}$.

Proposition B.13. [11, Proposition 7.26] Assume a routing table $\mathit{rt} \in \mathtt{RT}$ with $\mathit{dip} \in \mathtt{kD}(\mathit{rt})$.

(a) An $\mathtt{update}$ of rt can only increase the quality of the routing table. That is, for all routes r such that $\mathtt{update}(\mathit{rt}, r)$ is defined (i.e., $\pi_4(r) = \mathtt{val}$, $\pi_2(r) = 0 \Leftrightarrow \pi_3(r) = \mathtt{unk}$ and $\pi_3(r) = \mathtt{unk} \Rightarrow \pi_5(r) = 1$) we have

$$\mathit{rt} \sqsubseteq_{\mathit{dip}} \mathtt{update}(\mathit{rt}, r) \ . \tag{14}$$

(b) An $\mathtt{invalidate}$ on rt does not change the quality of the routing table if, for each $(\mathit{rip}, \mathit{rsn}) \in \mathit{dests}$, rt has a valid entry for rip, and

- rsn is the by one incremented sequence number from the routing table, or
- both rsn and the sequence number in the routing table are 0.

That is, for all partial functions dests (subsets of $\mathtt{IP} \times \mathtt{SQN}$)

$$\begin{aligned}&\big((\mathit{rip}, \mathit{rsn}) \in \mathit{dests} \ \Rightarrow\ \mathit{rip} \in \mathtt{vD}(\mathit{rt}) \ \wedge\ \mathit{rsn} = \mathtt{inc}(\mathtt{sqn}(\mathit{rt}, \mathit{rip}))\big)\\ \Rightarrow\ &\mathit{rt} \approx_{\mathit{dip}} \mathtt{invalidate}(\mathit{rt}, \mathit{dests}, *) \ .\end{aligned} \tag{15}$$

(c) If precursors are added to an entry of rt, the quality of the routing table does not change. That is, for all $\mathit{dip} \in \mathbf{IP}$ and sets of precursors $\mathit{npre} \in \mathscr{P}(\mathbf{IP})$,

$$\mathit{rt} \approx_{\mathit{dip}} \mathtt{addpreRT}(\mathit{rt}, \mathit{dip}, \mathit{npre}) \ . \tag{16}$$

Proof. The same as in [11], using Proposition B.6. □

Further, we have to prove that the applications of $\mathtt{setTime\_rt}$ do not decrease the quality of a routing table entry, nor do applications of $\mathtt{exp\_rt}$ that do not delete the entry to dip. The first is straightforward since $\mathtt{setTime\_rt}$ only modifies time components. The second follows since such applications leave both $\mathtt{nsqn}(\mathit{rt}, \mathit{dip})$ and $\mathtt{dhops}(\mathit{rt}, \mathit{dip})$ invariant.

Theorem 7.27 of [11] says that the quality of routing table entries can never decrease. This result does not hold any longer, as the entry may expire and reemerge with a lower quality. However, we do have the following weakening of this result.

Proposition B.14. As long as a routing table entry is not deleted, its quality can only be increased, never decreased.

Assume $N \xrightarrow{\ell} N'$ and $\mathit{ip}, \mathit{dip} \in \mathbf{IP}$. If $\mathit{dip} \in \mathtt{kD}_N^{\mathit{ip}} \cap \mathtt{kD}_{N'}^{\mathit{ip}}$ then

$$\xi_N^{\mathit{ip}}(\mathtt{rt}) \sqsubseteq_{\mathit{dip}} \xi_{N'}^{\mathit{ip}}(\mathtt{rt}) \ .$$

Proof. By Proposition B.13, and the remark following it, the quality of routing table entries, as long as they are not deleted, cannot decrease due to applications of $\mathtt{update}$, $\mathtt{addpreRT}$, $\mathtt{setTime\_rt}$ and $\mathtt{exp\_rt}$. Hence we only need to check all applications of $\mathtt{invalidate}$. That proceeds exactly as the proof of Proposition 7.27 in [11]. □

Proposition B.14 states in particular that if $N \xrightarrow{\ell} N'$ and $dip \in \mathtt{kD}^{ip}_{N} \cap \mathtt{kD}^{ip}_{N'}$ then $\mathtt{nsqn}^{ip}_{N}(dip) \leq \mathtt{nsqn}^{ip}_{N'}(dip)$.

Proposition 7.28 and Theorem 7.30 of [11] state relations between the routing tables of different nodes. They are the key results in establishing loop freedom. Both results do not hold here, at least not unconditionally. However, a weakening of Proposition 7.28 and the full Theorem 7.30 hold if we assume that premature route expiration does not occur. We formalise this assumption in two parts as Assumptions 1 and 2 below.

For the value of the variable `now` in state N, we write now_N. Assuming that in an initial state of AODV the clocks of all nodes have the same value, this will continue to be the case throughout the life of the protocol, since AODV does not modify this variable. Hence we do not need a superscript *ip* to indicate which node's variable `now` is meant. Moreover, now_N increases monotonically.

A route to *dip* may be marked as *valid* in the routing table of a node *ip*, but if $\mathtt{ltime}^{ip}_{N}(dip) \leq now_N$ or $\neg\mathtt{1hoplife}(\mathtt{nhop}^{ip}_{N}(dip), now_N)$ its validity is questionable, and, following the RFC [29], the routing table entry ought to be marked as *invalid*. Hence, before the routing table is consulted, the function `exp_rt` is always applied, making all valid routing table entries invalid that are timed out themselves, or have a timed-out routing table entry to the next hop. We define the set of *intrinsically valid* routing table entries of node *ip* in state N as $\mathtt{VD}^{ip}_{N} := \{dip \in \mathtt{vD}^{ip}_{N} \mid \mathtt{ltime}^{ip}_{N}(dip) > now_N \wedge \mathtt{1hoplife}(\mathtt{nhop}^{ip}_{N}(dip), now_N)\} = \xi^{ip}_{N}(\mathtt{vD}(\mathtt{exp\_rt}(\mathtt{rt}, \mathtt{now}, \mathtt{DELETE\_PERIOD})))$.

Assumption 1. If a node has an intrinsically valid routing table entry to a destination *dip*, then the next hop, if not *dip* itself, has a known route to *dip*.

$$dip \in \mathtt{VD}^{ip}_{N} \;\wedge\; nhip := \mathtt{nhop}^{ip}_{N}(dip) \neq dip \;\Rightarrow\; dip \in \mathtt{kD}^{nhip}_{N} \tag{17}$$

To formalise the second part of the assumption that premature route expiration does not occur, we first define what we mean by a message being *underway*. A message starts being underway when its transmission is initiated. For a RREQ or RREP message this is between the states N and N' for which

$$N \xrightarrow{\textbf{broadcast}(\mathtt{rreq}(*,*,*,*,*,*,*,*))}_{sip} N' \quad \text{or} \quad N \xrightarrow{\textbf{unicast}(*,\mathtt{rrep}(*,*,*,*,*,*))}_{sip} N'$$

in the notation of Proposition B.11, with *sip* being the sending node. For a RREQ message, this indicates a transition stemming from Rule (bc) in Table 1, resulting from the execution of Process 1, Line 51 or Process 4, Line 43. When a message leaves the incoming message queue of the receiving node we still treat it as underway until the receiving node makes "sufficient" updates to its routing table triggered by the receipt of the message, or when it becomes clear that an update is not going to happen:[24] a PKT message is underway until Line 1, 5, 26 or 30 of Process 3 is executed; a RREQ message is underway until Line 3 or 6 of Process 4 is executed; a RREP message is underway until Line 2 or 28 of Process 5 is executed; and a RERR message until Line 3 of Process 6 is executed.[25] In each

[24] By sufficient we mean enough changes for the invariants presented later to hold.

[25] NEWPKT messages are not considered since they are not stored in the message queue.

case exactly one of these lines will in fact be executed, and this happens in the same time slice in which the message leaves the incoming message queue.

Assumption 2. If a RREP message with destination *dip* or a RREQ message with originator *dip*, sent by a node $sip \neq dip$, is underway to a node *ip*, then $dip \in \mathtt{kD}_N^{sip}$.

Proposition B.15. Assume that premature route expiration does not occur (Assumptions 1 and 2). If, in a reachable network expression N, a node $ip \in \mathbf{IP}$ has an intrinsically valid routing table entry to *dip*, then also the next hop *nhip* towards *dip*, if not *dip* itself, has a routing table entry to *dip*, and the net sequence number of the latter entry is at least as large as that of the former.

$$dip \in \mathtt{VD}_N^{ip} \ \wedge \ nhip \neq dip \ \Rightarrow \ dip \in \mathtt{kD}_N^{nhip} \wedge \mathtt{nsqn}_N^{ip}(dip) \leq \mathtt{nsqn}_N^{nhip}(dip) \ , \quad (18)$$

where $nhip := \mathtt{nhop}_N^{ip}(dip)$ is the IP address of the next hop.

Apart from its reliance on Assumptions 1 and 2, this proposition weakens [11, Proposition 7.28] by assuming $dip \in \mathtt{VD}_N^{ip}$ instead of $dip \in \mathtt{kD}_N^{ip}$.

Proof. We can forget about the conclusion $dip \in \mathtt{kD}_N^{nhip}$, since this follows by Assumption 1. For an initial network expression the invariant holds since all routing tables are empty. We need to make sure that the invariant is maintained under all modifications to $\xi_N^{ip}(\mathtt{rt})$ or $\xi_N^{nhip}(\mathtt{rt})$, and under progress of time.

Progress of time cannot invalidate the invariant; at most it can invalidate the antecedent.

A modification of $\xi_N^{nhip}(\mathtt{rt})$ is harmless, as it can only increase $\mathtt{nsqn}_N^{nhip}(dip)$ (cf. Proposition B.14). The antecedent of the proposition ($dip \in \mathtt{kD}_N^{ip} \cap \mathtt{kD}_{N'}^{ip}$) follows from Assumption 1 and the fact that $dip \in \mathtt{VD}_N^{ip}$ and $nhip \neq dip$ holds both before and after the modification of $\xi_N^{nhip}(\mathtt{rt})$.

Applications of addpreRT have no effect on the invariant. Applications of exp_rt have no effect on the invariant either, since exp_rt is idempotent, and net sequence numbers are not affected by exp_rt. Applications of invalidate cannot invalidate the invariant; at most they can invalidate the antecedent. Whenever setTime_rt is applied, exp_rt has been applied before in the same time slice. For this reason, we have $\mathtt{ltime}_N^{ip}(dip) > now_N \wedge \mathtt{1hoplife}(\mathtt{nhop}_N^{ip}(dip), now_N)$, so setTime_rt cannot chance the condition $dip \in \mathtt{VD}_N^{ip}$ from false to true. It also has no further effect on the invariant.

Hence, it suffices to check all applications of update that actually change a routing table entry, beyond its precursors. This proceeds as in the proof of [11, Proposition 7.28], using Propositions B.1, B.5 and B.10, but with one refinement.

In cases Pro. 4, Line 6 and Pro. 5, Line 2 we handle in a state N a RREQ message with originator *dip*, or a RREP message with destination *dip*, that was sent by a node $nhip \neq dip$ in a state $N^\dagger$. The proof of [11, Proposition 7.28] then calls [11, Theorem 7.27] to infer that $\mathtt{nsqn}_N^{nhip}(dip) \geq \mathtt{nsqn}_{N^\dagger}^{nhip}(dip)$. Here we can instead use Proposition B.14, but only under Assumption 2. □

To prove loop freedom we will show that on any route established by AODV the quality of routing table entries increases when going from one node to the next hop. Here, the preorder is not sufficient, since we need a strict increase in quality. Therefore, on routing tables rt and rt' that both have an entry to dip, i.e., $dip \in \mathtt{kD}(rt) \cap \mathtt{kD}(rt')$, we define a relation $\sqsubset_{dip}$ by

$$rt \sqsubset_{dip} rt' \;:\Leftrightarrow\; rt \sqsubseteq_{dip} rt' \;\wedge\; rt \not\approx_{dip} rt' \;.$$

Corollary B.2. [11, Corollary 7.29] The relation $\sqsubset_{dip}$ is irreflexive and transitive.

Proposition B.16. [11, Theorem 7.30] Assume that premature route expiration does not occur (Assumptions 1 and 2). The quality of the routing table entries for a destination dip is strictly increasing along a route towards dip, until it reaches either dip or a node with an invalid routing table entry to dip.

$$dip \in \mathtt{vD}^{ip}_{N} \cap \mathtt{vD}^{nhip}_{N} \;\wedge\; nhip \neq dip \;\Rightarrow\; \xi^{ip}_{N}(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N}(\mathtt{rt}) \;, \qquad (19)$$

where N is a reachable network expression and $nhip := \mathtt{nhop}^{ip}_{N}(dip)$ is the IP address of the next hop.

Proof. For an initial network expression the invariant holds since all routing tables are empty. We need to make sure that the invariant is maintained under all modifications to $\xi^{ip}_{N}(\mathtt{rt})$ or $\xi^{nhip}_{N}(\mathtt{rt})$. Applications of `addpreRT` and `setTime_rt` have no effect on the invariant. Applications of `invalidate` and `exp_rt` cannot invalidate the invariant; at most they can invalidate the antecedent. Hence, it suffices to check all applications of `update` that change a routing table entry, beyond its precursors, just as in the proof of [11, Theorem 7.30].

The argument that the invariant is maintained under updates of $\xi^{nhip}_{N}(\mathtt{rt})$ is unchanged w.r.t. the proof of [11, Theorem 7.30]. It uses Proposition B.8. At two occasions this proof refers to [11, Proposition 7.28] (addressed as "Invariant (20)"), and in both cases a reference to Proposition B.15 suffices as well. Here, we use that each `update` that is handled in a state N is preceded by an application of `exp_rt` in the same time slice. Hence $dip \in \mathtt{vD}^{ip}_{N}$ implies $dip \in \mathtt{VD}^{ip}_{N}$.

The argument that the invariant is maintained under updates of $\xi^{ip}_{N}(\mathtt{rt})$ is almost unchanged w.r.t. the proof of [11, Theorem 7.30]. It uses Propositions B.1 and B.5 and Invariants (10) and (11). However, there are two occasions where the argument needs to be refined.

- In the case Pro. 4, Line 6 we handle in a state N a RREQ message with originator dip that was sent by a node $nhip \neq dip$ in a state $N^{\dagger}$. The proof of [11, Theorem 7.30] then calls [11, Proposition 7.6] to infer that $\mathtt{sqn}^{nhip}_{N}(dip) \geq \mathtt{sqn}^{nhip}_{N^{\dagger}}(dip)$. Here we can use Proposition B.3, but only under Assumption 2.
- In cases Pro. 4, Line 6 and Pro. 5, Line 2 we handle in a state N a RREQ message with originator dip, or a RREP message with destination dip, that was sent by a node $nhip \neq dip$ in a state $N^{\dagger}$. The proof of [11, Theorem 7.30] then calls [11, Theorem 7.27] to infer that $\xi^{nhip}_{N^{\dagger}}(\mathtt{rt}) \sqsubseteq_{dip} \xi^{nhip}_{N}(\mathtt{rt})$. Here we can instead use Proposition B.14, but only under Assumption 2. □

Definition B.1. [11] The *routing graph* of network expression N with respect to $dip \in \texttt{IP}$ is $\mathcal{R}_N(dip) := (\mathbf{IP}, E)$, where all nodes of the network form the vertices and there is an arc $(ip, ip') \in E$ iff $ip \neq dip$ and $(dip, *, *, \texttt{val}, *, ip', *, *) \in \xi_N^{ip}(\texttt{rt})$.

We say that a network expression N is *loop free* if the corresponding routing graphs $\mathcal{R}_N(dip)$ are loop free, for all $dip \in \mathbf{IP}$. A routing protocol, such as AODV, is *loop free* iff all reachable network expressions are loop free.

An arc in a routing graph states that ip' is the next hop on a valid route to dip known by ip; a path in a routing graph describes a route towards dip discovered by AODV.

Using this definition of a routing graph, Proposition B.16 states that along a path towards a destination dip in the routing graph of a reachable network expression N, until it reaches either dip or a node with an invalided routing table entry to dip, the quality of the routing table entries for dip is strictly increasing. From this, we can immediately conclude

Theorem B.1. Assume that premature route expiration does not occur (Assumptions 1 and 2). Then the specification of AODV given in Appendix B.1 is loop free. □

B.3 Premature Route Expiration

By Theorem B.1, to establish loop freedom for AODV it suffices to show that premature route expiration cannot occur. In view of the counterexample to loop freedom sketched in Figure 1, this condition appears necessary as well. In this appendix we do an attempt to prove an invariant that implies that premature route expiration, and hence routing loops, do not occur in AODV. In this process we formalise postulates on the real-time behaviour of the protocol that need to be made in order to have any chance on success. Even when assuming these, our invariant turns out not to be preserved by 5 lines of the AODV specification. As documented in Appendix B.4, each of these violations gives rise to premature route expiration, and consequently to routing loops. Additionally, for our proof to go through, we need to make an assumption (Assumption 3 below) that does not hold for AODV. The key to modifying AODV into a loop free variant is (1) to make a small change that validates Assumption 3, and (2) to change the above-mentioned 5 lines in such a way that the intended invariant is maintained.

Assumption 3. When a RREQ message with originator *oip* is sent by a node $sip \neq oip$, the node *sip* has a valid routing table entry to *oip*.

We will mark results that depend on this assumption by (A3). When applying Assumption 3 we will also use that in the state $N^\dagger$ where the transmission of the above RREQ message commences (by the execution of transition (bc) of Table 1) the valid routing table entry to *oip* satisfies $\texttt{ltime}_{N^\dagger}^{sip}(oip) > now_{N^\dagger}$. This follows because the forwarding of the RREQ message is always preceded by an application of `exp_rt` in the same time slice (Line 7 of Process 1).

Proposition B.17. [11, Proposition 7.36c] The sequence number of an originator appearing in a route request can never be greater than the originator's own sequence number.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rreq}(*,*,*,*,*,\mathit{oip}_c,\mathit{osn}_c,*))}_{ip} N' \;\Rightarrow\; \mathit{osn}_c \leq \xi_N^{\mathit{oip}_c}(\texttt{sn}) \tag{20}$$

Proof. Exactly as in [11], using Propositions B.1, B.2 and B.4. □

Proposition B.18. [11, Proposition 7.37]

(a) The sequence number of a destination appearing in a route reply can never be greater than the destination's own sequence number.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rrep}(*,\mathit{dip}_c,\mathit{dsn}_c,*,*,))}_{ip} N' \;\Rightarrow\; \mathit{dsn}_c \leq \xi_N^{\mathit{dip}_c}(\texttt{sn}) \tag{21}$$

(b) A known destination sequence number of a valid routing table entry can never be greater than the destination's own sequence number.

$$(\mathit{dip},\mathit{dsn},\texttt{kno},\texttt{val},*,*,*) \in \xi_N^{ip}(\texttt{rt}) \;\Rightarrow\; \mathit{dsn} \leq \xi_N^{\mathit{dip}}(\texttt{sn}) \tag{22}$$

Proof. Exactly as in [11], using Propositions B.1, B.2, B.4 and B.17. □

Proposition B.19. (A3) Let $N^\ddagger$ be a state in which the own sequence number maintained by node *dip* is incremented to the value *dsn*, and let N be a state in which a node *ip* has a *valid* routing table entry to *dip* with next hop $\mathit{nhip} \neq \mathit{dip}$ and a destination sequence number $\mathit{dsn}' \geq \mathit{dsn}$. Then $\mathit{now}_N \geq \mathit{now}_{N^\ddagger}$ and

$$\mathit{dip} \in \texttt{vD}_N^{\mathit{nhip}} \;\Rightarrow\; \texttt{ltime}_N^{\mathit{nhip}}(\mathit{dip}) \geq \mathit{now}_{N^\ddagger}\ , \tag{23}$$

$$\mathit{dip} \in \texttt{iD}_N^{\mathit{nhip}} \;\Rightarrow\; \texttt{ltime}_N^{\mathit{nhip}}(\mathit{dip}) \geq \mathit{now}_{N^\ddagger} + \texttt{DELETE\_PERIOD}\ . \tag{24}$$

Proof. Using proof by contradiction, we show that the sequence number dsn' is known, i.e., $\texttt{sqnf}(\xi_N^{ip}(\texttt{rt}),\mathit{dip}) = \texttt{kno}$. If we were to assume $\texttt{sqnf}(\xi_N^{ip}(\texttt{rt}),\mathit{dip}) = \texttt{unk}$, then $\texttt{dhops}(\xi_N^{ip}(\texttt{rt}),\mathit{dip}) = 1$ and hence $\mathit{nhip} = \texttt{nhop}(\xi_N^{ip}(\texttt{rt}),\mathit{dip}) = \mathit{dip}$, both by Proposition B.8; a contradiction to the assumption $\mathit{nhip} \neq \mathit{dip}$. Hence

$$\xi_N^{\mathit{dip}}(\texttt{sn}) \geq \mathit{dsn}' \geq \mathit{dsn} = \xi_{N^\ddagger}^{\mathit{dip}}(\texttt{sn})\ ,$$

where the first step follows from (22). Since sequence numbers increase over time (Proposition B.2) and $N^\ddagger$ is the state where *dsn* is set, we get $\mathit{now}_N \geq \mathit{now}_{N^\ddagger}$.

The invariants hold in initial states, as all routing tables are empty. Applications of `addpreRT` and `setTime_rt` cannot invalidate the invariants. Neither can applications of `invalidate` or `exp_rt` to the routing table of *ip*, or an `update` to the routing table of *nhip*. An application of `exp_rt` to the routing table of *nhip* that invalidates the antecedent $\mathit{dip} \in \texttt{vD}_N^{\mathit{nhip}}$ but validates $\mathit{dip} \in \texttt{iD}_N^{\mathit{nhip}}$ always results in a state where the lifetime of the routing table entry is extended by `DELETE_PERIOD`. An application of `invalidate` to the routing table of *nhip* that invalidates the antecedent $\mathit{dip} \in \texttt{vD}_N^{\mathit{nhip}}$ always results in a state where $\texttt{ltime}_N^{\mathit{nhip}}(\mathit{dip}) = \mathit{now}_N + \texttt{DELETE\_PERIOD} \geq \mathit{now}_{N^\ddagger} + \texttt{DELETE\_PERIOD}$. It remains to examine all applications of `update` to the routing table of *ip*, restricting attention to updates that change more than precursors.

Pro. 1, Lines 16, 20, 24: After these updates the condition $\mathit{nhip} \neq \mathit{dip}$ is no longer met.

Pro. 4, Line 6: If this update results in a change to the routing table, beyond the addition of precursors, afterwards $\mathit{nhip} := \mathtt{nhop}^{\mathit{ip}}_{N}(\mathit{dip}) = \xi^{\mathit{ip}}_{N}(\mathtt{sip}) \neq \mathit{dip} := \xi^{\mathit{ip}}_{N}(\mathtt{oip})$ and $\mathit{dsn}' := \mathtt{sqn}^{\mathit{ip}}_{N}(\mathit{dip}) = \xi^{\mathit{ip}}_{N}(\mathtt{osn})$ are taken from the sender and sequence-number fields of the incoming RREQ message that is being processed here. (The inequation of nhip and dip is an assumption.) Let $N^{\#}$ be the state in which node dip initiated this route request, and thus incremented its own sequence number to the value $\mathit{dsn}' \geq \mathit{dsn}$. Then $\mathit{now}_{N^{\#}} \geq \mathit{now}_{N^{\ddagger}}$ by Proposition B.2. By Propositions B.1(a) and B.5, the RREQ message must have been forwarded by nhip. Let $N^{\dagger}$ be the state in which the transmission of the forwarded RREQ message by node nhip commenced. Obviously, $\mathit{now}_{N^{\dagger}} \geq \mathit{now}_{N^{\#}}$. Assumption 3 yields $\mathit{dip} \in \mathtt{vD}^{\mathit{nhip}}_{N^{\dagger}}$. Before node nhip forwarded the route request (by executing Line 43 of Pro. 4), and in the same time slice, it must have executed Line 7 of Pro. 1, so that $\mathtt{ltime}^{\mathit{nhip}}_{N^{\dagger}}(\mathit{dip}) > \mathit{now}_{N^{\dagger}}$. Hence $\mathtt{ltime}^{\mathit{nhip}}_{N^{\dagger}}(\mathit{dip}) > \mathit{now}_{N^{\ddagger}}$. Further modifications to the routing table of nhip (by addpreRT, setTime_rt, update, exp_rt and invalidate) between states $N^{\dagger}$ and N preserve the invariant in the ways surveyed above.

Pro 5, Line 2: If this update results in a change to the routing table, beyond the addition of precursors, afterwards $\mathit{nhip} := \mathtt{nhop}^{\mathit{ip}}_{N}(\mathit{dip}) = \xi^{\mathit{ip}}_{N}(\mathtt{sip}) \neq \mathit{dip} = \xi^{\mathit{ip}}_{N}(\mathtt{dip})$ and $\mathit{dsn}' := \mathtt{sqn}^{\mathit{ip}}_{N}(\mathit{dip}) = \xi^{\mathit{ip}}_{N}(\mathtt{dsn})$ are taken from the sender and sequence-number fields of the incoming RREP message that is being processed here. By Propositions B.1(a) and B.5, this RREP message must have been sent before by nhip; say its transmission started in state $N^{\dagger}$. Proposition B.11 yields $\mathit{dip} \in \mathtt{vD}^{\mathit{nhip}}_{N^{\dagger}}$ and

$$\mathtt{sqn}^{\mathit{nhip}}_{N^{\dagger}}(\mathit{dip}) = \mathit{dsn}' \wedge \mathtt{sqnf}(\xi^{\mathit{nhip}}_{N^{\dagger}}(\mathtt{rt}), \mathit{dip}) = \mathtt{kno} \wedge \mathtt{ltime}^{\mathit{nhip}}_{N^{\dagger}}(\mathit{dip}) > \mathit{now}_{N^{\dagger}}\,.$$

Hence, by Proposition B.18(b), $\mathit{now}_{N^{\dagger}} \geq \mathit{now}_{N^{\ddagger}}$. So $\mathtt{ltime}^{\mathit{nhip}}_{N^{\dagger}}(\mathit{dip}) > \mathit{now}_{N^{\ddagger}}$. Further modifications to the routing table of nhip (by addpreRT, setTime_rt, update, exp_rt and invalidate) between states $N^{\dagger}$ and N preserve the invariant in the ways surveyed above. □

We can only show the absence of premature route expiration under further assumptions. In particular, we postulate the following relations between time constants. Henceforth the timing parameters NODE_TRAVERSAL_TIME and NET_TRAVERSAL_TIME will be abbreviated by NODE_TT and NET_TT.

$$0 \leq \mathtt{NODE\_TT} \leq \mathtt{NET\_TT} \tag{25}$$

$$0 \leq \mathtt{ACTIVE\_ROUTE\_TIMEOUT} < \mathtt{DELETE\_PERIOD} - \mathtt{NODE\_TT} - \mathtt{NET\_TT} \tag{26}$$

$$0 \leq \mathtt{MY\_ROUTE\_TIMEOUT} < \mathtt{DELETE\_PERIOD} - \mathtt{NODE\_TT} - \mathtt{NET\_TT} \tag{27}$$

$$3 \cdot \mathtt{NET\_TT} < \mathtt{DELETE\_PERIOD} + \mathtt{NODE\_TT} \tag{28}$$

These conditions are in line with the RFC: [29, Section 10] recommends:

NODE_TRAVERSAL_TIME	$40\,ms$
NET_TRAVERSAL_TIME	$2 \cdot$ NODE_TRAVERSAL_TIME $\cdot$ NET_DIAMETER[26]
ACTIVE_ROUTE_TIMEOUT	$10.000\,ms$[27]
MY_ROUTE_TIMEOUT	$2 \cdot$ ACTIVE_ROUTE_TIMEOUT
DELETE_PERIOD	$5 \cdot$ ACTIVE_ROUTE_TIMEOUT

Proposition B.20.

(a) The expiration time of a valid route is always smaller than $\mathit{now}_N + \texttt{DELETE\_PERIOD} - \texttt{NODE\_TRAVERSAL\_TIME} - \texttt{NET\_TRAVERSAL\_TIME}$.

$$\begin{array}{rl} & \mathit{dip} \in \texttt{vD}_N^{ip} \\ \Rightarrow & \texttt{ltime}_N^{ip}(\mathit{dip}) < \mathit{now}_N + \texttt{DELETE\_PERIOD} - \texttt{NODE\_TT} - \texttt{NET\_TT} \end{array} \tag{29}$$

(b) The lifetime recorded in a route reply message is always smaller than $\texttt{DELETE\_PERIOD} - \texttt{NODE\_TRAVERSAL\_TIME} - \texttt{NET\_TRAVERSAL\_TIME}$.

$$\begin{array}{rl} & N \xrightarrow{R:\mathbf{*cast}(\mathtt{rrep}(*,*,*,*,\mathit{ltime},*))}_{ip} N' \\ \Rightarrow & \mathit{ltime} < \texttt{DELETE\_PERIOD} - \texttt{NODE\_TT} - \texttt{NET\_TT} \end{array} \tag{30}$$

Proof. We prove the two statements by simultaneous induction.

(a) The invariant holds in the initial states, as all routing tables are empty. The functions `invalidate`, `addpreRT`, and `exp_rt` cannot increase the lifetime of a valid routing table entry, without invalidating the entry. It therefore suffices to check whether the invariant is preserved under the applications of `update` and `setTime_rt` in Processes 1–6. (Process 7 does not use these functions.)

Pro. 1, Lines 16, 20, 24, 32, 33; Pro. 3, Lines 9, 10, 11, 12; Pro. 4, Line 6; Pro. 5, Line 13: If these potential changes to routing table entries increase the lifetime of a node at all, the expiration time is set to $\mathit{now}_N + \texttt{ACTIVE\_ROUTE\_TIMEOUT}$. That the invariant is preserved follows from (26).

Pro. 1, Line 46: As this affects an invalid route ($\mathit{dip} \in \texttt{qD}(\mathit{store}) - \texttt{vD}(\mathit{rt})$, by Line 43), the invariant is preserved.

Pro. 3, Line 26: As this affects an invalid route, the invariant is preserved.

Pro. 4, Line 7: Here $\texttt{ltime}_N^{ip}(\mathit{dip})$ is set to $\mathit{now}_N + 2 \cdot \texttt{NET\_TT} - 2 \cdot (\mathit{hops}+1) \cdot \texttt{NODE\_TT}$. Since $\mathit{hops} \in \mathbb{N}$, the result follows from (28).

Pro 5, Line 2: Here $\texttt{ltime}_N^{ip}(\mathit{dip})$ is set to $\mathit{now}_N + \mathit{ltime}$, where the value *ltime* stems from an incoming RREP message (Pro. 1, Line 6). By Proposition B.1(a), this RREP message must have be sent before by some node. By induction, using (30), the invariant holds.

(b) We check all occasions in Processes 1–7 where a route reply is sent.

Process 4, Line 13: Here *ltime* is set to `MY_ROUTE_TIMEOUT`, so the result follows from (27).

[26] The default value of `NET_DIAMETER` is 35, yielding a `NODE_TRAVERSAL_TIME` of 2800 ms.

[27] When link-layer indications are used to detect link breakages (rather than Hello messages) [29, Section 10], as we assume here; otherwise 3000 ms.

Pro. 4, Line 31: *ltime* is set to $\mathtt{ltime}^{ip}_{N}(dip) - now_N$. Hence the invariant holds by induction, using statement (a) of the lemma.

Pro. 5, Line 14: Here the value *ltime* is taken from an incoming RREP message. By Proposition B.1(a), this RREP message must have be sent before by some node. Hence the statement follows by induction. □

As indicated in Section 3.3, we now capture realistic network scenarios by assuming that the transmission time of a message plus the period it spends in the queue of incoming messages of the receiving node is bounded by NODE_TT. Since NODE_TT "is a conservative estimate of the average one hop traversal time for packets and should include queuing delays, interrupt processing times and transfer times" [29, Sect. 10], the following postulate makes sense.

Postulate 1. Let $N^\dagger$ be a state in which the transmission of a message to *ip* is initiated, and let N be the state in which the message leaves the queue of incoming messages of node *ip*. Then $now_N \leq now_{N^\dagger} + \mathtt{NODE\_TT}$.

Likewise, we assume that the period a route request travels through the network is bounded by NET_TT.

Postulate 2. Let $N^\ddagger$ be a state in which a route request is initiated, and N a state in which a corresponding RREQ message leaves the queue of incoming messages of an arbitrary node *ip*. Then $now_N \leq now_{N^\ddagger} + \mathtt{NET\_TT}$.

Together with Assumption 3, these postulates are strong enough to ensure the validity of Assumption 2.

A similar statement as Postulate 2 could be set up for route replies; it is, however, not needed for the current analysis.

Theorem B.2. (A3) Assumption 2 holds.

Proof. Suppose in state N a RREP message that establishes a route to *dip*, sent by a node $sip \neq dip$, is underway to a node *ip*. Let $N^\dagger$ be the state in which the transmission of the message was initiated. By Postulate 1, $now_N \leq now_{N^\dagger} + \mathtt{NODE\_TT}$. In state $N^\dagger$ node *sip* had a valid routing table entry to *dip*, with an expiration time larger than $now_{N^\dagger}$, by Proposition B.11, i.e., $dip \in \mathtt{vD}^{sip}_{N^\dagger}$ and $\mathtt{ltime}^{sip}_{N^\dagger}(dip) > now_{N^\dagger}$. Upon invalidation of an entry, the expiration time is always either set to $\mathtt{now} + \mathtt{DELETE\_PERIOD}$ or extended by DELETE_PERIOD. Since $\mathtt{NODE\_TT} < \mathtt{DELETE\_PERIOD}$, by (26), the routing table entry for *dip* cannot have expired in state N.

The case for a RREQ message proceeds likewise, but using Assumption 3 and the remark following it, instead of Proposition B.11. □

Write $\mathtt{pkt}^{nhip}_{N}(dip)$ if a data packet for destination *dip* is underway (from some node *sip*) to node *nhip* conform the definition given prior to Assumption 2. Moreover, if $\mathtt{pkt}^{nhip}_{N}(dip)$, write $\mathtt{atime}^{nhip}_{N}(dip)$ for the latest possible time the next data packet destined to *dip* will arrive at node *nhip* confirm the prediction of Postulate 1. It follows that

$$\mathtt{pkt}_N^{nhip}(dip) \;\Rightarrow\; now_N \leq \mathtt{atime}_N^{nhip}(dip) \leq now_N{+}\mathtt{NODE\_TT}\,. \tag{31}$$

The following "intended theorem" ensures that also Assumption 1 holds. This is a trivial corollary of the two invariants proposed below. Hence, if the intended theorem would hold, loop freedom follows. However, the invariant turns out not to be preserved under 5 lines of the AODV specification, as made clear by the last line in the following "intended proof".

Intended Theorem B.3. (A3)

(a) If a data packet destined for *dip* is underway to node *nhip*, then *nhip* has a routing table entry to *dip* that will not expire before (or upon) arrival of that (first) data packet.

$$\begin{aligned}&\mathtt{pkt}_N^{nhip}(dip) \wedge nhip \neq dip\\ \Rightarrow\ &(dip \in \mathtt{vD}_N^{nhip} \wedge \mathtt{ltime}_N^{nhip}(dip) > \mathtt{atime}_N^{nhip}(dip) - \mathtt{DELETE\_PERIOD})\\ &\vee (dip \in \mathtt{iD}_N^{nhip} \wedge \mathtt{ltime}_N^{nhip}(dip) > \mathtt{atime}_N^{nhip}(dip))\end{aligned} \tag{32}$$

(b) If a node *ip* has a valid routing table entry to a destination *dip* with expiration time $ltime > now_N$, and no data packet is underway to *nhip*—the next hop towards *dip*—then *nhip*, if not *dip* itself, has a valid entry to *dip* with expiration time $> ltime + \mathtt{NODE\_TT} - \mathtt{DELETE\_PERIOD}$, or an invalid one with expiration time $> ltime + \mathtt{NODE\_TT}$.

$$\begin{aligned}&dip \in \mathtt{vD}_N^{ip} \ \wedge\ \mathtt{ltime}_N^{ip}(dip) > now_N \ \wedge\ nhip \neq dip \ \wedge\ \neg\mathtt{pkt}_N^{nhip}(dip)\\ \Rightarrow\ &(dip \in \mathtt{vD}_N^{nhip} \wedge \mathtt{ltime}_N^{nhip}(dip) > \mathtt{ltime}_N^{ip}(dip){+}\mathtt{NODE\_TT}{-}\mathtt{DELETE\_PRD.})\\ &\vee (dip \in \mathtt{iD}_N^{nhip} \wedge \mathtt{ltime}_N^{nhip}(dip) > \mathtt{ltime}_N^{ip}(dip){+}\mathtt{NODE\_TT})\end{aligned} \tag{33}$$

where $nhip := \mathtt{nhop}_N^{ip}(dip)$ is the IP address of the next hop towards *dip*.

Proof. We prove the two statements by simultaneous induction. Both invariants hold in initial states, as no packet is underway and all routing tables are empty.

(a) We have to check that the invariant is preserved under (i) changes that validate the condition $\mathtt{pkt}_N^{nhip}(dip)$, (ii) changes to the routing table of node *nhip*, and (iii) changes that increase the value of $\mathtt{atime}_N^{nhip}(dip)$.

(i) Let $nhip \neq dip$. The only way the condition $\mathtt{pkt}_N^{nhip}(dip)$ can turn valid is when a node *ip* executes Line 29 of Pro. 1 or Line 7 of Pro. 3, with $\xi_N^{ip}(\mathtt{dip}) = dip$ and $\mathtt{nhop}_N^{ip}(dip) = nhip$, and $\mathtt{pkt}_N^{nhip}(dip)$ did not hold before. Right beforehand, node *ip* must have executed Line 27 of Pro. 1 or Line 5 of Pro. 3; hence $dip \in \mathtt{vD}_N^{ip}$. Before that, *ip* must have executed Line 2 of Pro. 1, so that $\mathtt{ltime}_N^{ip}(dip) > now_N$. By induction, invariant (33) yields

$$\begin{aligned}&(dip \in \mathtt{vD}_N^{nhip} \wedge \mathtt{ltime}_N^{nhip}(dip) > \mathtt{ltime}_N^{ip}(dip){+}\mathtt{NODE\_TT}{-}\mathtt{DELETE\_PRD.})\\ &\vee (dip \in \mathtt{iD}_N^{nhip} \wedge \mathtt{ltime}_N^{nhip}(dip) > \mathtt{ltime}_N^{ip}(dip){+}\mathtt{NODE\_TT}).\end{aligned}$$

This holds just before $\mathtt{pkt}_N^{nhip}(dip)$ turned valid, and hence also just after. By (31), $\mathtt{ltime}_N^{ip}(dip) + \mathtt{NODE\_TT} > now_N + \mathtt{NODE\_TT} \geq \mathtt{atime}_N^{nhip}(dip)$, so the invariant is maintained.

(ii) We now examine changes to the routing table of node *nhip*. These could be made by the functions `update`, `invalidate`, `addpreRT`, `setTime_rt` or `exp_rt`. An `update` cannot make a valid entry invalid, erase an invalid entry, or shorten the lifetime of an entry. For this reason, the invariant is maintained under applications of `update`. The same applies to applications of `setTime_rt`. Applications of `addpreRT` have no impact on the invariant.

If the routing table entry to *dip* is invalidated by `invalidate`, the expiration time of the entry is always set to $now_N + \texttt{DELETE\_PERIOD}$. Assuming that $\texttt{pkt}_N^{nhip}(dip) \wedge nhip \neq dip$, by (31,26), $\texttt{atime}_N^{nhip}(dip) \leq now_N + \texttt{NODE\_TT} < now_N + \texttt{DELETE\_PERIOD} = \texttt{ltime}_N^{nhip}(dip)$.

If the routing table entry to *dip* is invalidated by `exp_rt`, the expiration time of the entry is extended by `DELETE_PERIOD`. This preserves the invariant.

Finally consider the erasure of an entry by `exp_rt`. Suppose that right afterwards, and thus also right before, we have $\texttt{pkt}_N^{nhip}(dip) \wedge nhip \neq dip$. Then, by induction and (31), $\texttt{ltime}_N^{nhip}(dip) > \texttt{atime}_N^{nhip}(dip) \geq now_N$ holds when `exp_rt` is applied to an invalid route to *dip*, or

$$\begin{aligned}\texttt{ltime}_N^{nhip}(dip) &> \texttt{atime}_N^{nhip}(dip) - \texttt{DELETE\_PERIOD}\\ &\geq now_N - \texttt{DELETE\_PERIOD}\end{aligned}$$

when it is applied to an valid one, so the route is not deleted by `exp_rt`.

(iii) The only event that can increase the value of $\texttt{atime}_N^{nhip}(dip)$ is the arrival at node *nhip* of a data packet destined for *dip*, when another data packet for *dip* is already underway. When this happens, first Lines 2, 6 and 12 of Pro. 1 are executed, with $\xi(\texttt{dip}) = dip$. Then either $nhip = dip$, so that the invariant remains satisfied, or Line 3 of Pro. 3, with $\xi(\texttt{ip}) = nhip$, is executed in the same time slice. Assume the latter. In case *nhip* has a valid routing table entry for *dip*, since `exp_rt` has been executed in the same time slice, $\texttt{ltime}_N^{nhip}(dip) > now_N \geq \texttt{atime}_N^{nhip}(dip) - \texttt{NODE\_TT} > \texttt{atime}_N^{nhip}(dip) - \texttt{DELETE\_PRD}$ by (31,26), so the invariant remains satisfied.

Otherwise, Line 22 of Pro. 3 is executed in the same time slice. In that case, applying induction on the state just after this line, $dip \in \texttt{iD}_N^{nhip}$, so Lines 25 and 26 of Pro. 3 are executed in the same time slice. This results in a state where $\texttt{ltime}_N^{nhip}(dip)$ has at least the value $now_N + \texttt{DELETE\_PERIOD}$. The execution of Line 26 marks the arrival of the data packet, and thus the state chance where the value of $\texttt{atime}_N^{nhip}(dip)$ is increased. Right afterwards, $\texttt{ltime}_N^{nhip}(dip) \geq now_N + \texttt{DELETE\_PERIOD} > now_N + \texttt{NODE\_TT} \geq \texttt{atime}_N^{nhip}(dip)$, using (26) and (31). Hence the invariant is maintained.

(b) We have to check that the invariant is preserved under (i) changes that validate the condition $\neg\texttt{pkt}_N^{nhip}(dip)$, (ii) changes to the routing table of node *nhip*, and (iii) changes to the routing table of node *ip*. As now_N is monotonically increasing, changes to now_N cannot invalidate the invariant.

(i) Starting with $\texttt{pkt}^{nhip}_{N}(dip)$, suppose that $dip\in\texttt{vD}^{ip}_{N}\wedge\texttt{ltime}^{ip}_{N}(dip)>now_N$, and a data packet destined for *dip* is handled by node *nhip*, in the sense that Lines 6 and 12 of Pro. 1 are executed with $\xi(\texttt{dip})=dip$. Then either $nhip = dip$, so that the invariant remains satisfied, or Line 3 of Pro. 3, with $\xi(\texttt{ip}) = nhip$, is executed in the same time slice. Assuming the latter, in case *nhip* has a valid routing table entry for *dip*, since $\texttt{exp\_rt}$ has been executed in the same time slice, $\texttt{ltime}^{nhip}_{N}(dip) > now_N > \texttt{ltime}^{ip}_{N}(dip) + \texttt{NODE\_TT} - \texttt{DELETE\_PERIOD}$, by Proposition B.20, so the invariant remains satisfied.

Otherwise ($dip \notin \texttt{vD}^{nhip}_{N}$), Line 22 of Pro. 3 is executed in the same time slice. In that case, applying induction on the state just before the data packet arrived, invariant (32) yields $dip \in \texttt{iD}^{nhip}_{N}$, so Lines 25 and 26 of Pro. 3 are executed in the same time slice. This results in a state where $\texttt{ltime}^{nhip}_{N}(dip)$ has at least the value $now_N + \texttt{DELETE\_PERIOD}$. The execution of Line 26 marks the arrival of the data packet, and thus the state chance where the the condition $\neg\texttt{pkt}^{nhip}_{N}(dip)$ becomes valid. Right afterwards, $\texttt{ltime}^{ip}_{N}(dip)+\texttt{NODE\_TT} < now_N + \texttt{DELETE\_PERIOD}$, by Proposition B.20. Hence the invariant is maintained.

(ii) We now examine changes to the routing table of node *nhip*. These could be made by the functions `update`, `invalidate`, `addpreRT`, `setTime_rt` or `exp_rt`. An `update` cannot make a valid entry invalid, erase an invalid entry, or shorten the lifetime of an entry. For this reason, the invariant is maintained under applications of `update`. The same applies to applications of `setTime_rt`. Applications of `addpreRT` have no impact on the invariant.

If the routing table entry to *dip* is invalidated by `invalidate`, its expiration time $\texttt{ltime}^{nhip}_{N}(dip)$ is always set to $now_N + \texttt{DELETE\_PERIOD}$. Using the assumption $dip \in \texttt{vD}^{ip}_{N}$ and Equation (29), we get $\texttt{ltime}^{ip}_{N}(dip)+ \texttt{NODE\_TT} < now_N + \texttt{DELETE\_PERIOD}$. Hence the invariant is maintained.

If the routing table entry to *dip* is invalidated by `exp_rt`, the expiration time of the entry is extended by `DELETE_PERIOD`. This preserves the invariant.

Finally consider the erasure of an entry by `exp_rt`. Suppose that right afterwards, and thus also right before, the antecedent of the invariant holds. Then, by induction,

$$\begin{array}{ll} & (dip\in\texttt{vD}^{nhip}_{N}\wedge\texttt{ltime}^{nhip}_{N}(dip) > \texttt{ltime}^{ip}_{N}(dip) - \texttt{DELETE\_PERIOD}) \\ \vee & (dip\in\texttt{iD}^{nhip}_{N}\wedge\texttt{ltime}^{nhip}_{N}(dip) > \texttt{ltime}^{ip}_{N}(dip)). \end{array}$$

Using that $\texttt{ltime}^{ip}_{N}(dip) > now_N$, the route is not deleted by `exp_rt`.

(iii) We conclude with changes to the routing table of node *ip*. Clearly the invariant is maintained under applications of `invalidate`, `addpreRT` and `exp_rt`. We now go though all occurrences of `update` and `setTime_rt` in Processes 1–7.

Pro. 1, Lines 16, 20, 24: These entries create or update a routing table entry with $nhip = dip$, so the antecedent of the invariant is not met.

Pro 4, Line 6: If this update results in a change to the routing table, beyond the addition of precursors, afterwards $\mathit{oip} \in \mathtt{vD}^{\mathit{ip}}_N$ and $\mathit{nhip} := \mathtt{nhop}^{\mathit{ip}}_N(\mathit{oip}) = \xi^{\mathit{ip}}_N(\mathtt{sip})$ is the sender of the incoming RREQ message that is being processed here. We may assume that $\mathit{nhip} \neq \mathit{oip}$, as otherwise the invariant is maintained. By Proposition B.1(a), this RREQ message must have been sent before by nhip. Let $N^{\dagger}$ be the state in which the transmission of the message was initiated (by the execution of transition (bc) of Table 1). By Postulate 1 and (25) $\mathit{now}_{N^{\dagger}} \leq \mathit{now}_N \leq \mathit{now}_{N^{\dagger}} + \mathtt{NODE\_TT} \leq \mathit{now}_{N^{\dagger}} + \mathtt{NET\_TT}$. In state $N^{\dagger}$, node nhip had a valid routing table entry to oip, with a positive remaining lifetime, i.e., $\mathit{oip} \in \mathtt{vD}^{\mathit{nhip}}_{N^{\dagger}}(\mathit{oip})$ and $\mathtt{ltime}^{\mathit{nhip}}_{N^{\dagger}}(\mathit{oip}) > \mathit{now}_{N^{\dagger}}$, by Assumption 3 (A3) and the remark following it. By (29), using that $\mathit{oip} \in \mathtt{vD}^{\mathit{ip}}_N$ and $\mathit{now}_{N^{\dagger}} \geq \mathit{now}_N - \mathtt{NET\_TT}$, it follows that the condition

$$\begin{aligned}&(\mathit{oip} \in \mathtt{vD}^{\mathit{nhip}}_{N\#} \wedge \mathtt{ltime}^{\mathit{nhip}}_{N\#}(\mathit{oip}) > \mathtt{ltime}^{\mathit{ip}}_N(\mathit{oip}) + \mathtt{NODE\_TT} - \mathtt{DEL\_PRD.})\\ &\vee (\mathit{oip} \in \mathtt{iD}^{\mathit{nhip}}_{N\#} \wedge \mathtt{ltime}^{\mathit{nhip}}_{N\#}(\mathit{oip}) > \mathtt{ltime}^{\mathit{ip}}_N(\mathit{oip}) + \mathtt{NODE\_TT})\end{aligned} \tag{34}$$

holds in state $N^{\#} := N^{\dagger}$. To see that it still holds in state $N^{\#} := N$, we argue that it is preserved under changes to the routing table of node nhip between states $N^{\dagger}$ and N. Since the state N is fixed in (34), the value $\mathtt{ltime}^{\mathit{ip}}_N(\mathit{oip})$ does not change, so only changes to $\mathit{oip} \in \mathtt{vD}^{\mathit{nhip}}_{N\#}$, $\mathit{oip} \in \mathtt{iD}^{\mathit{nhip}}_{N\#}$ and $\mathtt{ltime}^{\mathit{nhip}}_{N\#}(\mathit{oip})$ need to be considered. These could be made by the functions `update`, `invalidate`, `addpreRT`, `setTime_rt` or `exp_rt`. An `update` cannot make a valid entry invalid, erase an invalid entry, or shorten the lifetime of an entry. For this reason, (34) is maintained under applications of `update`. The same applies to applications of `setTime_rt`. Applications of `addpreRT` have no impact on (34) either.

If the routing table entry to oip is invalidated by `invalidate`, its expiration time $\mathtt{ltime}^{\mathit{nhip}}_{N\#}(\mathit{oip})$ is set to $\mathit{now}_{N\#} + \mathtt{DELETE\_PERIOD}$. So Equation (29), applied to $\mathit{oip} \in \mathtt{vD}^{\mathit{ip}}_N$, yields $\mathtt{ltime}^{\mathit{nhip}}_{N\#}(\mathit{oip}) = \mathit{now}_{N\#} + \mathtt{DELETE\_PERIOD} \geq \mathit{now}_{N^{\dagger}} + \mathtt{DELETE\_PERIOD} > \mathit{now}_N + \mathtt{DELETE\_PERIOD} - \mathtt{NET\_TT} > \mathtt{ltime}^{\mathit{ip}}_N(\mathit{dip}) + \mathtt{NODE\_TT}$. Hence invariant (34) is maintained.

Using the antecedent of (33), $\mathtt{ltime}^{\mathit{ip}}_N(\mathit{oip}) \geq \mathit{now}_N \geq \mathit{now}_{N\#}$, so (34) implies that the routing table entry to oip cannot be deleted by `exp_rt`. If the routing table entry to oip is invalidated by `exp_rt`, its expiration time $\mathtt{ltime}^{\mathit{nhip}}_{N\#}(\mathit{oip})$ is extended by `DELETE_PERIOD`. This preserves (34).

Pro 5, Line 2: The argument is exactly as in the previous case, but using RREP instead of RREQ and dip instead of oip. Moreover, we call Proposition B.11 instead of Assumption 3.

Pro. 1, Line 32; Pro. 3, Line 9: When this instruction is executed, a data packet is underway to $\mathit{nhip} := \mathtt{nhop}^{\mathit{ip}}_N(\mathit{dip})$ (Pro. 1, Line 29, or Pro. 3, Line 7, resp.), so the antecedent of the invariant is not satisfied.

Pro. 1, Line 46; Pro. 3, Line 26: As this affects an invalid route, the invariant is preserved.

Pro. 4, Line 7: Let $\mathit{nhip} := \mathtt{nhop}^{ip}_{N}(\mathit{oip})$ be the next hop to oip (before and after the the call of $\mathtt{setTime\_rt}$), and let $\mathit{osn} := \mathtt{sqn}^{ip}_{N}(\mathit{oip})$ be the destination sequence number of this route. Then $\mathit{osn} \geq \xi^{ip}_{N}(\mathtt{osn})$, where $\xi^{ip}_{N}(\mathtt{osn})$ is the sequence number for oip carried in the route request. Let $N^{\ddagger}$ be the state in which node oip initiated the route request, and thus incremented its own sequence number to $\xi^{ip}_{N}(\mathtt{osn})$. By Postulate 2, $\mathit{now}_N \leq \mathit{now}_{N^{\ddagger}} + \mathtt{NET\_TT}$. Assume $\mathit{oip} \in \mathtt{vD}^{ip}_{N} \wedge \mathtt{ltime}^{ip}_{N}(\mathit{oip}) > \mathit{now}_N \wedge \mathit{nhip} \neq \mathit{oip} \wedge \neg\mathtt{pkt}^{\mathit{nhip}}_{N}(\mathit{oip})$ as otherwise the invariant is maintained. Then, by induction, we have $\mathit{oip} \in \mathtt{kD}^{\mathit{nhip}}_{N}$ right before the update, so also right afterwards.

Suppose $\mathit{oip} \in \mathtt{vD}^{\mathit{nhip}}_{N}$. Then, by Proposition B.19 (A3) and the above calculation, $\mathtt{ltime}^{\mathit{nhip}}_{N}(\mathit{oip}) \geq \mathit{now}_{N^{\ddagger}} \geq \mathit{now}_N - \mathtt{NET\_TT}$. Using that $\mathit{oip} \in \mathtt{vD}^{ip}_{N}$ in combination with (29) we have $\mathit{now}_N - \mathtt{NET\_TT} > \mathtt{ltime}^{ip}_{N}(\mathit{dip}) + \mathtt{NODE\_TT} - \mathtt{DEL\_PRD}$; hence the invariant is maintained.

Suppose $\mathit{oip} \in \mathtt{iD}^{\mathit{nhip}}_{N}$. Then $\mathtt{ltime}^{\mathit{nhip}}_{N}(\mathit{oip}) \geq \mathit{now}_{N^{\ddagger}} + \mathtt{DELETE\_PRD}$ by Proposition B.19 (A3), so $\mathtt{ltime}^{\mathit{nhip}}_{N}(\mathit{oip}) \geq \mathit{now}_N + \mathtt{DELETE\_PERIOD} - \mathtt{NET\_TT} > \mathtt{ltime}^{ip}_{N}(\mathit{oip}) + \mathtt{NODE\_TT}$, using (29) and that $\mathit{oip} \in \mathtt{vD}^{ip}_{N}$.

Pro. 1, Line 33; Pro. 3, Lines 10, 11, 12; Pro. 5, Line 13: WRONG. □

We have shown that, under Assumption 3, only 5 lines of the AODV specification invalidate Invariant (33) of the Intended Theorem B.3, namely Line 33 of Pro. 1, Lines 10, 11, 12 of Pro. 3 and Line 13 of Pro. 5. If these lines were to be changed in a way that preserves this invariant, then Assumption 1 holds as well. Assumption 2 holds as a consequence of Assumption 3 (Theorem B.2). Hence, if A3 can be ensured and B.3 can be repaired then AODV becomes loop free (Theorem B.1). In the next section we will show how this can be achieved.

B.4 Six Routing Loops and their Repair

The loop freedom proof above broke down in six places: the unwarranted Assumption 3 we needed to make, and the five lines that do not preserve our main invariant. Below we sketch scenarios showing that each of these flaws actually leads to a case of premature route expiration, and consequently a routing loop. We also describe how the protocol could be fixed to avoid these loops.

Assumption 3. Assume a 5-node linear topology $C{-}B{-}A{-}E{-}D$. It can happen that B has an invalid routing table entry to D with a sequence number that is substantially larger than D's own sequence number.[28] By waiting sufficiently

[28] This can happen when B maintains a valid route to D and the link $D{-}B$ breaks down and reappears multiple times: each time a link break occurs, B invalidates the route to D, thereby incrementing its destination sequence number; and each time the link reappears, D forwards a message to B, causing B to validate its route to D (Pro. 1, Lines 16, 20 or 24) without changing its destination sequence number.

long, it can moreover happen that this routing table entry has almost reached the end of its lifespan. Assume that in such a state D initiates two route requests, say RREQ_{DA} with destination A and RREQ_{DE} for E. Right after RREQ_{DA} is sent on its way via E to A, the link $D{-}C$ emerges, so that RREQ_{DE} travels via C and B towards A. When RREQ_{DE} is forwarded by node B, B will not update its route to D, because it already has an (invalid) route to D with a higher sequence number than the one carried by the route request. Yet, it extends the expiration time of its (invalid) route to D to the value

$$\mathtt{now} + 2 \cdot \mathtt{NET\_TT} - 2 \cdot (2+1) \cdot \mathtt{NODE\_TT}$$

following Pro. 4, Line 7. When the message reaches A, A creates a routing table entry for D, with next hop B and the sequence number carried by RREQ_{DE}.

Although RREQ_{DA} has to travel only two hops before reaching A, it is possible that it arrives there after RREQ_{DE}. When this happens, Line 6 of Pro. 4 does not give rise to an update of the routing table entry at A for D, since that entry has already a higher destination sequence number than the one carried by RREQ_{DA}. Yet, by Pro. 4, Line 7 the entry to D (with next hop B) has its expiration time extended to at least

$$\mathtt{now} + 2 \cdot \mathtt{NET\_TT} - 2 \cdot (2+1) \cdot \mathtt{NODE\_TT}\ ,$$

which by now is strictly past the expiration time of the route to D maintained by B. A case of premature route expiration, and a possible routing loop, results.

The repair of this loop is already sketched in Footnote 14: simply make the broadcast forwarding the route request (Pro. 4, Line 43) conditional on the existence of a valid route to $\mathtt{oip}$. This assures that Assumption 3 is met. An even better solution is to make execution of all of Pro. 4, Lines 9–46 conditional on $\mathtt{oip} \in \mathtt{vD}(\mathtt{rt})$. Besides preventing the routing loop indicated above, this is a strict improvement of AODV on all counts, as none of the actions taken in Pro. 4, Lines 9–46 makes any sense if $\mathtt{oip} \notin \mathtt{vD}(\mathtt{rt})$.

Pro. 1, Line 33. We now consider the topology $A\overset{\frown}{{-}B{-}}C{-}D$. It can happen that a node A has a valid routing table entry to a destination D with next hop C, but the routing table entry for C at node A has a hop count strictly larger than 1, and next hop $B \neq C$. One of the ways this can happen is when the link $A{-}C$ breaks down (after the route $A{-}C{-}D$ has been established) and C initiates a route request that reaches A via B; right afterwards, the link $A{-}C$ is restored, before the link break could impede any unicast.[29] In such a situation, whenever A sends a data packet to D, via C, Line 33 extends the lifetime of A's routing table entry to C. Yet the route from B to C is never used and will eventually expire and be deleted. This gives rise a case of premature route expiration.

The resulting routing loop can be avoided by changing the AODV specification in such a way that Line 33 is executed only when the hop count of the

[29] The described situation can also arise without any link breaks; we leave the "how" as a puzzle for the reader.

route to $\mathtt{nhop}(\mathtt{rt},\mathtt{dip})$ is 1. This is the only situation where there is a rationale for this instruction in the first place. The invariant of the Intended Theorem B.3 is then preserved by Line 33 because afterwards the antecedent $\mathit{nhip} \neq \mathit{dip}$ is not met. An alternative is to skip this line altogether; of course this could yield shorter lifetimes of certain routing table entries.

Pro. 3, Line 10. The routing loop arises just as in the previous case, except that the data packets from A to D are now forwarded by A rather than originating from A. The repair is the same as well.

Pro. 3, Line 11. Let us now have a look at the following topology: $A{<}\genfrac{}{}{0pt}{}{B_1}{B_2}{>}C{-}D$. It can happen that first a route $A{-}B_1{-}C{-}D$ is established, and later C finds a fresher route to A via node B_2. A stream of data packets from A to D, via B_1 and C, will cause node C, at Pro. 3, Line 11 to extend the lifetime of the route to A (via B_2). But the routing table entry for A at B_2 will eventually expire and disappear, giving rise to a case of premature route expiration.

A possible repair is to simply skip Line 11. For routes need not be bidirectional: if the route to `oip` is not used, a stream of data packets *from* `oip` is no reason to keep the route in the direction `oip` alive.

Pro. 3, Line 12. This routing loop arises by combining the scenarios of Lines 10 and 11. The repair is to simply delete this line.

Pro. 5, Line 13. Assume a topology $B_2\overparen{{-}B_1\cdots A{-}B{-}}C{-}D$ in which the link $B_1{-}A$ recently broke, and suppose a route request from A looking for node D travels via B and C. Afterwards the link $B{-}C$ breaks and during that time C searches for a new route to A. Node B_2 answers this route request, and a route $C{-}B_2{-}B_1{-}A$ is established. This can happen even if the routing table entry for A at B_1 is invalid, namely when B_2 missed all RERR messages announcing this. Only afterwards comes the route reply from D, passing through C and B_2 on its way to A. At B_2 Pro. 5, Line 13 causes the lifetime of the route to A to be extended. However, it could be arbitrary long ago that the route from B_1 to A was ever used, and this invalid route may be about to expire. By possibly repeating this scenario various times (since A never finds a route to D), one obtains a valid routing table entry from B_2 to A via B_1, while the routing table entry for A at B_1 is deleted.

This case of premature route expiration can be prevented by simply omitting Line 13. The argument for doing this is again that routes need not be bidirectional: if the route to `oip` is not used, a route reply that is intended to establish a route *from* `oip` is no reason to keep the route in the direction `oip` alive.