To appear in **Distributed Computing**
Accepted December 1, 2015

Modelling and Verifying the AODV Routing Protocol

Rob van Glabbeek · Peter Höfner · Marius Portmann · Wee Lum Tan

Abstract This paper presents a formal specification of the Ad hoc On-Demand Distance Vector (AODV) routing protocol using AWN (Algebra for Wireless Networks), a recent process algebra which has been tailored for the modelling of Mobile Ad Hoc Networks and Wireless Mesh Network protocols. Our formalisation models the exact details of the core functionality of AODV, such as route discovery, route maintenance and error handling. We demonstrate how AWN can be used to reason about critical protocol properties by providing detailed proofs of loop freedom and route correctness.

Keywords Wireless mesh networks; mobile ad-hoc networks; routing protocols; AODV; process algebra; AWN; loop freedom.

1 Introduction

Routing protocols are crucial to the dissemination of data packets between nodes in Wireless Mesh Networks (WMNs) and Mobile Ad Hoc Networks (MANETs). One of the most popular protocols that is widely used in WMNs is the Ad hoc On-Demand Distance Vector (AODV) routing protocol [39]. It is one of the four protocols standardised by the IETF MANET working group, and it also forms the basis of new WMN routing protocols, including the Hybrid Wireless Mesh Protocol (HWMP) in the IEEE 802.11s wireless mesh network standard [27]. The details of the AODV protocol are standardised in IETF RFC 3561 [39]. However, due to the use of English prose, this specification contains ambiguities and contradictions. This can lead to significantly different implementations of the AODV routing protocol, depending on the developer's understanding and reading of the AODV RFC. In the worst case scenario, an AODV implementation may contain serious flaws, such as routing loops [20].

Traditional approaches to the analysis of AODV and many other AODV-based protocols [41, 27, 46, 50, 43] are simulation and test-bed experiments. While such methods are important and valid for protocol evaluation, in particular for quantitative performance evaluation, they have limitations in regards to the evaluation of basic protocol correctness properties. Experimental evaluation is resource intensive and time consuming, and, even after a very long time of evaluation, only a finite set of network scenarios can be considered—no general guarantee can be given about correct protocol behaviour for a wide range of unpredictable deployment scenarios [2]. This problem is illustrated by recent discoveries of limitations in AODV-like protocols that have been under intense scrutiny over many years [35].

We believe that formal methods can help in this regard; they complement simulation and test-bed experiments as methods for protocol evaluation and verification, and provide stronger and more general assurances about protocol properties and behaviour. The overall goal is to reduce the "time-to-market" for better (new or modified) WMN protocols, and to increase the reliability and performance of the corresponding networks.

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

R. J. van Glabbeek
NICTA and UNSW. E-mail: rvg@cs.stanford.edu

P. Höfner
NICTA and UNSW. E-mail: Peter.Hoefner@nicta.com.au

M. Portmann
The University of Queensland. E-mail: marius@itee.uq.edu.au

W. L. Tan
Griffith University. E-mail: w.tan@griffith.edu.au

In this paper we provide a complete and accurate formal specification of the core functionality of AODV using the specification language AWN (Algebra of Wireless Networks) [15]. AWN provides the right level of abstraction to model key features such as unicast and broadcast, while abstracting from implementation-related details. As its semantics is completely unambiguous, specifying a protocol in such a framework enforces total precision and the removal of any ambiguities. A key contribution is to demonstrate how AWN can be used to support reasoning about protocol behaviour and to provide rigorous proofs of key protocol properties, using the examples of loop freedom and route correctness. In contrast to what can be achieved by model checking and test-bed experiments, our proofs apply to all conceivable dynamic network topologies.

Route correctness is a minimal sanity requirement for a routing protocol; it is the property that the routing table entries stored at a node are entirely based on information on routes to other nodes that either is currently valid or was valid at some point in the past. Loop freedom is a critical property for any routing protocol, but it is particularly relevant and challenging for WMNs. Descriptions as in [17] capture the common understanding of loop freedom: "*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.*" Packets caught in a routing loop, until they are discarded by the IP Time-To-Live (TTL) mechanism, can quickly saturate the links and have a detrimental impact on network performance. It is therefore critical to ensure that protocols prevent routing loops. We show that loop freedom can be guaranteed only if sequence numbers are used in a careful way, considering further rules and assumptions on the behaviour of the protocol. The problem is, as shown in the case of AODV, that these additional rules and assumptions are not explicitly stated in the RFC, and that the RFC has significant ambiguities in regards to this. To the best of our knowledge we are the first to give a complete and detailed proof of loop freedom.[1] [2]

The rigorous protocol analysis discussed in this paper has the potential to save a significant amount of time in the development and evaluation of new network protocols, can provide increased levels of assurance of protocol correctness, and complements simulation and other experimental protocol evaluation approaches.

The remainder of this paper is organised as follows. Section 2 gives an informal introduction to AODV. We briefly recapitulate AWN in Section 3. Section 5 provides a detailed formal specification of AODV in AWN.[3] To achieve this, we present the basic data structure needed in Section 4. In Section 6 we formally prove some properties of AODV that can be expressed as invariants, in particular loop freedom and route correctness.[4] Section 7 describes related work, and in Section 8 we summarise our findings and point at work that is yet to be done.

2 The AODV Routing Protocol

The Ad hoc On-Demand Distance Vector (AODV) routing protocol [39] is a widely-used routing protocol designed for MANETs, and is one of the four protocols currently standardised by the IETF MANET working group[5]. It also forms the basis of new WMN routing protocols, including the Hybrid Wireless Mesh Protocol (HWMP) in the IEEE 802.11s wireless mesh network standard [27].

AODV is a reactive protocol: routes are established only on demand. A route from a source node s to a destination node d is a sequence of nodes $[s, n_1, \ldots, n_k, d]$, where $n_1, \ldots, n_k$ are intermediate nodes located on the path from s to d. Its basic operation can best be explained using a simple example topology shown in Figure 1(a), where edges connect nodes within transmission range. We assume node s wants to send a data packet to node d, but s does not have a valid routing table entry for d. Node s initiates a route discovery mechanism by broadcasting a route request (RREQ) message, which is received by s's immediate neighbours a and b. We assume that neither a nor b knows a route to the destination node d.[6] Therefore, they simply rebroadcast the message, as shown in Figure 1(b). Each RREQ message has a unique identifier which allows nodes to ignore duplicate RREQ messages that they have handled before.

[1] Loop freedom of AODV has been "proven" at least thrice [42,3,55], but the proofs in [42] and [3] are not correct, and the one in [55] is based on a simple subset of AODV only, not including the "intermediate route reply" feature—a most likely source of loops. We elaborate on this in Section 7.

[2] In this paper, we abstract from timing issues by postulating that routing table entries never expire. Consequently, we can make no claim on routing loops resulting from premature expiration of routing tables entries. This will be the subject of a forthcoming paper [8].

[3] Parts of the specification have been published before in "A Process Algebra for Wireless Mesh Networks" [15], in "Automated Analysis of AODV using UPPAAL" [14] and in "A Rigorous Analysis of AODV and its Variants" [26].

[4] A sketch of the loop freedom proof is given in [15] and in [26].

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

[6] In case an intermediate node knows a route to d, it directly sends a route reply back.

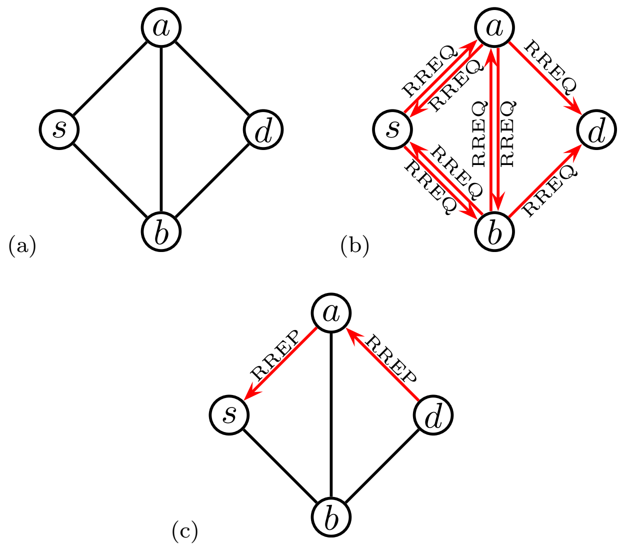

Fig. 1 Example network topology

When forwarding the RREQ message, each intermediate node updates its routing table and adds a "reverse route" entry to s, indicating via which next hop the node s can be reached, and the distance in number of hops. Once the first RREQ message is received by the destination node d (we assume via a), d also adds a reverse route entry in its routing table, saying that node s can be reached via node a, at a distance of 2 hops.

Node d then responds by generating a route reply (RREP) message and sending it back to node s, as shown in Figure 1(c). In contrast to the RREQ message, the RREP is unicast, i.e., it is sent to an individual next hop node only. The RREP is sent from d to a, and then to s, using the reverse routing table entries created during the forwarding of the RREQ message. When processing the RREP message, a node creates a "forward route" entry into its routing table. For example, upon receiving the RREP via a, node s creates an entry saying that d can be reached via a, at a distance of 2 hops. At the completion of the route discovery process, a route has been established from s to d, and data packets can start to flow.

In the event of link and route breaks, AODV uses route error (RERR) messages to inform affected nodes.

Sequence numbers, another important aspect of AODV, indicate the freshness of routing information. AODV "*uses destination sequence numbers to ensure loop freedom at all times (even in the face of anomalous delivery of routing control messages), ...*" [39]. A proof of loop freedom of AODV has been sketched in [42]. Nodes maintain their own sequence number as well as a destination sequence number for each route discovered. This use of sequence numbers can be an efficient approach to address the problem of routing loops, but has to be taken with caution, since loop freedom cannot be guaranteed a priori [20].

3 The Specification Language AWN

Ideally, any specification is free of ambiguities and contradictions. Using English prose only, this is nearly impossible to achieve. Hence every specification should be equipped with a formal specification. The choice of an appropriate specification language is often secondary, although it has high impact on the analysis. The use of *any* formal specification language helps to avoid ambiguities and to precisely describe the intended behaviour. Examples of modelling languages are (i) the Alloy language, used by Zave to model Chord [54]; (ii) timed automata, which are the input language for the UPPAAL model checker and used by Chiyangwa, Kwiatkowska [9] and others [14] to reason about AODV; (iii) routing algebra as introduced by Griffin and Sobrinho [23], or (iv) AWN, a process algebra particularly tailored for (wireless mesh) routing protocols [15,26].

For this paper we choose the modelling language AWN: on the one hand it is tailored for wireless protocols and therefore offers primitives such as **broadcast**; on the other hand, it defines the protocol in a pseudocode that is easily readable. (The language itself is implementation independent). AWN is a variant of standard process algebras [34,25,1,4], extended with a local broadcast mechanism and a novel *conditional unicast* operator—allowing error handling in response to failed communications while abstracting from link layer implementations of the communication handling—and incorporating data structures with assignments. It also describes the interaction between nodes in a network with a dynamic network topology. Process algebras such as AWN are equipped with an operational semantics [15,16]: once a model has been described, its behaviour is governed by the transitions allowed by the algebra's semantics. This can significantly reduce the burden of proofs. In this paper we abstain from a formal definition of the operational semantics.[7] Instead, we employ a correspondence between the transitions of AWN processes and the execution of *actions*—subexpressions as occur in Entries 3–10 of Table 1—identified by line numbers in protocol specifications in AWN.

[7] Thereby we also abstain from explaining the modelling of the dynamic network topology in the semantics, i.e., the mechanism by which links between nodes break. This matter is explained in [15,16], and completely orthogonal to the formal specification of the AODV protocol and the correctness properties that are the focus of this paper. In particular, the correctness properties hold independently of the number of link breaks or link occurrences.

Table 1 process expressions

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

We use an underlying data structure (described in detail in Section 4) with several types, variables ranging over these types, operators and predicates. First order predicate logic yields terms (or *data expressions*) and formulas to denote data values and statements about them. The AWN data structure always contains the types DATA, MSG, IP and $\mathscr{P}$(IP) of *application layer data*, *messages*, *IP addresses*—or any other node identifiers—and *sets of IP addresses*. The messages comprise *data packets*, containing application layer data, and *control messages*. The rest of the data structure is customisable for any application of AWN.

In AWN, a WMN is modelled as an encapsulated parallel composition of network nodes. On each node several processes may be running in parallel. Network nodes communicate with their direct neighbours—those nodes that are currently in transmission range—using either broadcast, unicast, or an iterative unicast/multicast (here called *groupcast*). The *process expressions* are given in Table 1. A process name X comes with a *defining equation*

$$X(\mathtt{var}_1,\dots,\mathtt{var}_n) \stackrel{def}{=} p\,,$$

where p is a process expression, and the $\mathtt{var}_i$ are data variables maintained by process X. A named process is like a *procedure*; when it is called, data expressions exp_i of the appropriate type are filled in for the variables $\mathtt{var}_i$. Furthermore, φ is a condition, $\mathtt{var}:=exp$ an assignment of a data expression exp to a variable var of the same type, $dest$, $dests$, $data$ and ms data expressions of types IP, $\mathscr{P}$(IP), DATA and MSG, respectively, and msg a data variable of type MSG.

Given a valuation of the data variables by concrete data values, the process $[\varphi]p$ acts as p if φ evaluates to true, and deadlocks if φ evaluates to false.[8] In case φ contains free variables that are not yet interpreted as data values, values are assigned to these variables in any way that satisfies φ, if possible. The process $[\![\mathtt{var}:=exp]\!]p$ acts as p, but under an updated valuation of the data variables. The process $p+q$ may act either as p or as q, depending on which of the two 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 process **broadcast**$(ms).p$ broadcasts (the data value bound to the expression) ms to the other network nodes within range, and subsequently acts as p, whereas the process **unicast**$(dest, ms).p \blacktriangleright q$ tries to unicast the message ms to the destination $dest$; if successful it continues to act as p and otherwise as q.[9] The latter models an abstraction of an acknowledgment-of-receipt mechanism that is typical for unicast communication but absent in broadcast communication, as implemented by the link layer of wireless standards such as IEEE 802.11. The process **groupcast**$(dests, ms).p$ tries to transmit ms to all destinations $dests$, and proceeds as p regardless of whether any of the transmissions is successful. The process **send**$(ms).p$ synchronously transmits a message to another process running on the same network node; this action can occur only when the other process is able to receive the message. The process **receive**$(\mathtt{msg}).p$ receives any message m (a data value of type MSG) either from another node, from another process running on the same node or from the application layer process on the local node. It then proceeds as p, but with the data variable msg bound to the value m. In particular, **receive**$(\mathtt{newpkt}(data, dip))$ models the injection of a data from the application layer, where the function newpkt generates a message containing the application layer $data$ and the intended destination address dip. Data is delivered to the application layer by **deliver**$(data)$.

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

In the full process algebra [15], *node expressions* $a:P:R$ are given by process expressions P, annotated with an *address* a and a set of nodes R that are within

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

[9] The unicast is unsuccessful if the destination $dest$ is out of transmission range of the node ip performing the unicast, i.e., if in the dynamic network topology there is currently no link between ip and $dest$.

transmission range of a. A partial network is then modelled as a parallel composition of node expressions, using the operator $\|$, and a complete network is obtained by placing this composition in the scope of an encapsulation operator $[\_]$. The main purpose of the encapsulation operator is to prevent the receipt of messages that have never been sent by other nodes in the network—with the exception of messages $\texttt{newpkt}(\mathit{data},\mathit{dip})$ stemming from the application layer of a node. More details on the language AWN can be found in [16].

To illustrate the use of AWN we consider a network of two nodes on which the same process is running. One node broadcasts an integer value. A received broadcast message will be delivered to the application layer if its value is 1. Otherwise the node decrements its value and broadcasts the new value. The behaviour of each node can be modelled by:

$$\begin{aligned}\texttt{X}(\texttt{n}) &\stackrel{def}{=} \textbf{broadcast}(\texttt{n}).\texttt{Y}()\\ \texttt{Y}() &\stackrel{def}{=} \textbf{receive}(\texttt{m}).([\texttt{m}{=}1]\ \textbf{deliver}(\texttt{m}).\texttt{Y}()\ .\\ &\qquad + [\texttt{m}{\neq}1]\ \texttt{X}(\texttt{m}-1))\end{aligned}$$

If a node is in a state $\texttt{X}(\texttt{n})$ it will broadcast $\texttt{n}$ and continue in state $\texttt{Y}()$. If a node is in state $\texttt{Y}()$, and it receives $\texttt{m}$, it has two ways to continue. Process $[\texttt{m}{=}1]$ $\textbf{deliver}(\texttt{m}).\texttt{Y}()$ is enabled if $\texttt{m}{=}1$. In that case $\texttt{m}$ will be delivered to the application layer, and the process returns to $\texttt{Y}()$. Alternatively, if $\texttt{m}\neq 1$, the process continues as $\texttt{X}(\texttt{m}-1)$. Note that calls to processes use expressions as parameters, in this case $\texttt{m}-1$.

Let us have a look at two network topologies. First, assume that the nodes a and b are within transmission range of each other; node a in state $\texttt{X}(2)$, and node b in $\texttt{Y}()$. In AWN this is formally expressed as $[a{:}\texttt{X}(2){:}\{b\} \,\|\, b{:}\texttt{Y}(){:}\{a\}]$, although below we simply write $\texttt{X}(2) \,\|\, \texttt{Y}()$. Then, node a broadcasts 2 and continues as $\texttt{Y}()$. Node b receives 2, and continues as $\texttt{X}(1)$. Next b broadcasts 1, and continues as $\texttt{Y}()$, while node a receives 1, and, since the condition $\texttt{m}{=}1$ is satisfied, **deliver**s 1 and continues as $\texttt{Y}()$. This gives rise to transitions from one state to the other:

$$\begin{aligned}\texttt{X}(2) \,\|\, \texttt{Y}() &\xrightarrow{a:\textbf{broadcast}(2)} \texttt{Y}() \,\|\, \texttt{X}(1) \xrightarrow{b:\textbf{broadcast}(1)}\\ &\xrightarrow{a:\textbf{deliver}(1)} \texttt{Y}() \,\|\, \texttt{Y}()\ .\end{aligned}$$

In state $\texttt{Y}() \,\|\, \texttt{Y}()$ no further activity is possible; the network has reached a *deadlock*.

Second, assume that the nodes are not within transmission range; formally $[a{:}\texttt{X}(2):\emptyset \,\|\, b{:}\texttt{Y}():\emptyset]$. Again a is in state $\texttt{X}(2)$, and b in $\texttt{Y}()$. As before, node a broadcasts 2 and continues as $\texttt{Y}()$; but this time the message is not received by any node; hence no message is forwarded or delivered and both nodes end up in state $\texttt{Y}()$.

For the last scenario, we assume that a and b are within transmission range and that both nodes have the same initial state $\texttt{X}(1)$. Assuming that no packet collisions occur, and node a sends first:

$$\begin{aligned}\texttt{X}(1) \,\|\, \texttt{X}(1) &\xrightarrow{a:\textbf{broadcast}(1)} \texttt{Y}() \,\|\, \texttt{X}(1) \xrightarrow{b:\textbf{broadcast}(1)}\\ &\xrightarrow{a:\textbf{deliver}(1)} \texttt{Y}() \,\|\, \texttt{Y}()\ .\end{aligned}$$

Unfortunately, node b is in a state where it cannot receive a message, so a's message "remains unheard" and b will never deliver that message. To avoid this behaviour, and ensure that both messages get delivered, as happens in real WMNs, a message queue can be introduced (see Section 5.6).

4 Data Structure for AODV

In this section we present the data structure needed for the detailed formal specification of AODV. As well as describing *types* for the information handled at the nodes during the execution of the protocol we also define functions which will be used to describe the precise intention—and overall effect—of the various update mechanisms in an AODV implementation. The definitions are grouped roughly according to the various "aspects" of AODV and the host network.

Many of the presented type and function definitions are straightforward; so in principle this section can be skipped or be used as reference material.

4.1 Mandatory Types

As stated in the previous section, the data structure always consists of application layer data, messages, IP addresses and sets of IP addresses.

(a) The ultimate purpose of AODV is to deliver *application layer data*. The type $\texttt{DATA}$ describes a set of application layer data items. An item of data is thus a particular element of that set, denoted by the variable $\texttt{data} \in \texttt{DATA}$.

(b) *Messages* are used to send information via the network. In our specification we use the variable $\texttt{msg}$ of the type $\texttt{MSG}$. We distinguish AODV control messages (route request, route reply, and route error) as well as *data packets*: messages for sending application layer data (see Section 4.8).

(c) The type $\texttt{IP}$ describes a set of IP addresses or, more generally, a *set of node identifiers*. In the RFC 3561 [39], $\texttt{IP}$ is defined as the set of all IP addresses. We assume that each node has a unique identifier $\mathit{ip} \in \texttt{IP}$. Moreover, in our model, each node ip maintains a variable $\texttt{ip}$ which always has the value ip. In any AODV control message, the variable $\texttt{sip}$ holds the IP address of the sender, and if the message is part of the *route discovery process*—a route request or route reply message—we use `oip` and `dip` for the origin and destination of the route sought. Furthermore, `rip` denotes an unreachable destination (a destination to which a route was established earlier, but this route is now broken) and `nhip` the next hop on some route.

4.2 Sequence Numbers

As explained in Section 2, any node maintains its own *sequence number*—the value of the variable `sn`—and a routing table whose entries describe routes to other nodes. The value of `sn` increases over time. In AODV each routing table entry is equipped with a sequence number to constitute a measure approximating the relative freshness of the information held—a smaller number denotes older information. All sequence numbers of routes to $dip \in \mathtt{IP}$ stored in routing tables are ultimately derived from *dip*'s own sequence number at the time such a route was discovered.

We denote the set of sequence numbers by `SQN` and assume it to be totally ordered. By default we take `SQN` to be $\mathbb{N}$, and use standard functions such as max. The initial sequence number of any node is 1. We reserve a special element $0 \in \mathtt{SQN}$ to be used for the sequence number of a route, whose semantics is that no sequence number for that route is known. Sequence numbers are incremented by the function

$$\mathtt{inc} : \mathtt{SQN} \rightarrow \mathtt{SQN}$$
$$\mathtt{inc}(sn) \;=\; \begin{cases} sn+1 & \text{if } sn \neq 0 \\ sn & \text{otherwise}\,. \end{cases}$$

The variables `osn`, `dsn` and `rsn` of type `SQN` are used to denote the sequence numbers of routes leading to the nodes `oip`, `dip` and `rip`.

AODV tags sequence numbers of routes as "known" or "unknown". This indicates whether the value of the sequence number can be trusted. The sequence-number-status flag is set to unknown (`unk`) when a routing table entry is updated with information that is not equipped with a sequence number itself. In such a case the old sequence number of the entry is maintained; hence the value `unk` does not indicate that no sequence number for the entry is known. Here we use the set $\mathtt{K} = \{\mathtt{kno}, \mathtt{unk}\}$ for the possible values of the sequence-number-status flag; we use the variable `dsk` to range over type `K`.

4.3 Modelling Routes

In a network, pairs $(ip_0, ip_k) \in \mathtt{IP} \times \mathtt{IP}$ of nodes are considered to be "connected" if ip_0 can send to ip_k directly, i.e., ip_0 is in transmission range of ip_k and vice versa. We say that such nodes are connected by a single *hop*. When ip_0 is not connected to ip_k then messages from ip_0 directed to ip_k need to be "routed" through intermediate nodes. We say that a *route* (from ip_0 to ip_k) is made up of a sequence $[ip_0, ip_1, ip_2, \ldots, ip_{k-1}, ip_k]$, where (ip_i, ip_{i+1}), $i = 0, \ldots, k{-}1$, are connected pairs; the *length* or *hop count* of the route is the number of single hops, and any node ip_i needs only to know the "next hop" address ip_{i+1} in order to be able to route messages intended for the final destination ip_k.

In operation, information about routes to certain destinations is stored in *routing tables* maintained at each node. This information sometimes needs to be re-evaluated in regard to its validity. Routes may become *invalid* if one of the pairs (ip_i, ip_{i+1}) in the hop-to-hop sequence gets disconnected. Then AODV may be reinvoked, as the need arises, to discover alternative routes.

In addition to the next hop and hop count, AODV also "tags" a route with its validity, sequence number and sequence-number status. Information about invalid routes is preserved until fresh information is received that establishes a valid replacement route. The purpose of this is to compare the sequence number and hop count of the replacement route with that of the invalid one, to check that the information is indeed fresher (or equally fresh while the replacement route is shorter). For every route, a node moreover stores a list of *precursors*, modelled as a set of `IP` addresses. This set collects all nodes which are currently potential users of the route, and are located one hop further "upstream". When the interest of other nodes emerges, these nodes are added to the precursor list;[10] the main purpose of recording this information is to inform those nodes when the route becomes invalid.

In summary, following the RFC, a routing table entry (or entry for short) is given by 7 components:

(a) The destination IP address—an element of `IP`;
(b) The destination sequence number—an element of `SQN`;
(c) The sequence-number-status flag—an element of the set $\mathtt{K} = \{\mathtt{kno}, \mathtt{unk}\}$;
(d) A flag tagging the route as being valid or invalid—an element of the set $\mathtt{F} = \{\mathtt{val}, \mathtt{inv}\}$. We use the variable `flag` to range over type `F`;
(e) The hop count, which is an element of $\mathbb{N}$. The variable `hops` ranges over the type $\mathbb{N}$ and we make use of the standard function $+1$;
(f) The next hop, which is again an element of `IP`; and

[10] The RFC does not mention a situation where nodes are dropped from the list, which seems curious.

(g) A precursor list, which is modelled as an element of $\mathscr{P}(\mathtt{IP})$.[11] The variable $\mathtt{pre}$ ranges over $\mathscr{P}(\mathtt{IP})$.

We denote the type of routing table entries by $\mathtt{R}$, and use the variable $\mathtt{r}$. A tuple

$$(\mathit{dip}, \mathit{dsn}, \mathit{dsk}, \mathit{flag}, \mathit{hops}, \mathit{nhip}, \mathit{pre})$$

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* denotes whether the sequence number is "outdated" or can be used to reason about freshness of the route. Finally, *pre* is a set of all neighbours who are "interested" in the route to *dip*. A node being "interested" in the route is somewhat sketchily defined as one which has previously used the current node to route messages to *dip*. Interested nodes are recorded in case the route to *dip* should ever become invalid, so that they may subsequently be informed. We use projections $\pi_1, \ldots \pi_7$ to select the corresponding component from the 7-tuple: For example, $\pi_6 : \mathtt{R} \to \mathtt{IP}$ determines the next hop.

4.4 Routing Tables

Nodes store all their information about routes in their *routing tables*; a node *ip*'s routing table consists of a set of routing table entries, exactly one for each known destination. Thus, a routing table is defined as a set of entries, with the restriction that each has a different destination *dip*, i.e., the first component of each entry in a routing table is unique.[12] Formally, we define the type $\mathtt{RT}$ of routing tables by

$$\mathtt{RT} := \{rt \in \mathscr{P}(\mathtt{R}) \mid \forall r, s \in rt : r \neq s \Rightarrow \pi_1(r) \neq \pi_1(s)\}\,.$$

AODV chooses between alternative routes if necessary to ensure that only one route per destination ends up in a given node's routing table. In our model, each node *ip* maintains a variable $\mathtt{rt}$, whose value is the current routing table of the node.

In the formal model (and indeed in any AODV implementation) we need to extract the components of the entry for any given destination from a routing table. To this end, we define the following partial functions—they are partial because the routing table need not have an entry for the given destination. We begin by selecting the entry in a routing table corresponding to a given destination *dip*:

$$\sigma_{route} : \mathtt{RT} \times \mathtt{IP} \rightharpoonup \mathtt{R}$$
$$\sigma_{route}(rt, dip) := \begin{cases} r & \text{if } r \in rt \wedge \pi_1(r) = dip \\ \text{undefined} & \text{otherwise}\,. \end{cases}$$

Through the projections $\pi_1, \ldots, \pi_7$, defined above, we can now select the components of a selected entry:

(a) The *destination sequence number* relative to *dip*:

$$\mathtt{sqn} : \mathtt{RT} \times \mathtt{IP} \to \mathtt{SQN}$$
$$\mathtt{sqn}(rt, dip) := \begin{cases} \pi_2(\sigma_{route}(rt, dip)) \\ \quad \text{if } \sigma_{route}(rt, dip) \text{ is defined} \\ 0 \quad \text{otherwise}\,. \end{cases}$$

(b) The *"known" status* of a route's sequence number:

$$\mathtt{sqnf} : \mathtt{RT} \times \mathtt{IP} \to \mathtt{K}$$
$$\mathtt{sqnf}(rt, dip) := \begin{cases} \pi_3(\sigma_{route}(rt, dip)) \\ \quad \text{if } \sigma_{route}(rt, dip) \text{ is defined} \\ \mathtt{unk} \quad \text{otherwise}\,. \end{cases}$$

(c) The *validity status* of a recorded route:

$$\mathtt{flag} : \mathtt{RT} \times \mathtt{IP} \rightharpoonup \mathtt{F}$$
$$\mathtt{flag}(rt, dip) := \pi_4(\sigma_{route}(rt, dip))\,.$$

(d) The *hop count* of the route from the current node (hosting *rt*) to *dip*:

$$\mathtt{dhops} : \mathtt{RT} \times \mathtt{IP} \rightharpoonup \mathbb{N}$$
$$\mathtt{dhops}(rt, dip) := \pi_5(\sigma_{route}(rt, dip))\,.$$

(e) The *identity of the next node on the route to dip* (if such a route is known):

$$\mathtt{nhop} : \mathtt{RT} \times \mathtt{IP} \rightharpoonup \mathtt{IP}$$
$$\mathtt{nhop}(rt, dip) := \pi_6(\sigma_{route}(rt, dip))\,.$$

(f) The set of *precursors* or neighbours interested in using the route from *ip* to *dip*:

$$\mathtt{precs} : \mathtt{RT} \times \mathtt{IP} \rightharpoonup \mathscr{P}(\mathtt{IP})$$
$$\mathtt{precs}(rt, dip) := \pi_7(\sigma_{route}(rt, dip))\,.$$

The domain of these partial functions changes during the operation of AODV as more routes are discovered and recorded in the routing table *rt*. The first two functions are extended to be total functions: whenever there is no route to *dip* inside the routing table under consideration, the sequence number is set to "unknown" (0) and the sequence-number-status flag is set to "unknown" ($\mathtt{unk}$), respectively. In the same style each partial function could be turned into a total one. However, in the specification we use these functions only when they are defined.

We are not only interested in information about a single route, but also in information on a routing table:

[11] The word "precursor list" is used in the RFC, but no properties of lists are used.

[12] As an alternative to restricting the set, we could have defined routing tables as partial functions from $\mathtt{IP}$ to $\mathtt{R}$, in which case it makes more sense to define an entry as a 6-tuple, not including the the destination IP as the first component.

(a) The set of destination IP addresses for *valid* routes in rt is given by

$$\mathtt{vD} : \mathtt{RT} \to \mathscr{P}(\mathtt{IP})$$
$$\mathtt{vD}(rt) := \{dip \mid (dip, *, *, \mathtt{val}, *, *, *) \in rt\}\,.$$ [13]

(b) The set of destination IP addresses for *invalid* routes in rt is

$$\mathtt{iD} : \mathtt{RT} \to \mathscr{P}(\mathtt{IP})$$
$$\mathtt{iD}(rt) := \{dip \mid (dip, *, *, \mathtt{inv}, *, *, *) \in rt\}\,.$$

(c) Last, we define the set of destination IP addresses for *known* routes by

$$\mathtt{kD} : \mathtt{RT} \to \mathscr{P}(\mathtt{IP})$$
$$\mathtt{kD}(rt) := \mathtt{vD}(rt) \cup \mathtt{iD}(rt) = \{dip \mid (dip, *, *, *, *, *, *) \in rt\}\,.$$

The partial functions σ_{route}, $\mathtt{flag}$, $\mathtt{dhops}$, $\mathtt{nhop}$ and $\mathtt{precs}$ are defined for rt and dip iff $dip \in \mathtt{kD}(rt)$.

4.5 Updating Routing Tables

Routing tables can be updated for three principal reasons. The first is when a node needs to adjust its list of precursors relative to a given destination; the second is when a received request or response carries information about network connectivity; and the last when information is received to the effect that a previously valid route should now be considered invalid. We define an update function for each case.

4.5.1 Updating Precursor Lists

Recall that the precursors of a given node ip relative to a particular destination dip are the nodes that are "interested" in a route to dip via ip. The function $\mathtt{addpre}$ takes a routing table entry and a set of IP addresses $npre$ and updates the entry by adding $npre$ to the list of precursors already present:

$$\mathtt{addpre} : \mathtt{R} \times \mathscr{P}(\mathtt{IP}) \to \mathtt{R}$$
$$\mathtt{addpre}((dip, dsn, dsk, flag, hops, nhip, pre), npre) := (dip, dsn, dsk, flag, hops, nhip, pre \cup npre)\,.$$

Often it is necessary to add precursors to an entry of a given routing table. For that, we define the function $\mathtt{addpreRT}$, which takes a routing table rt, a destination dip and a set of IP addresses $npre$ and updates the entry with destination dip by adding $npre$ to the list of precursors already present. It is only defined if an entry for destination dip exists.

$$\mathtt{addpreRT} : \mathtt{RT} \times \mathtt{IP} \times \mathscr{P}(\mathtt{IP}) \rightharpoonup \mathtt{RT}$$
$$\mathtt{addpreRT}(rt, dip, npre) := (rt - \{\sigma_{route}(rt, dip)\}) \cup \{\mathtt{addpre}(\sigma_{route}(rt, dip), npre)\}$$

Formally, we remove the entry with destination dip from the routing table and insert a new entry for that destination. This new entry is the same as before—only the precursors have been added.

4.5.2 Inserting New Information in Routing Tables

If a node gathers new information about a route to a destination dip, then it updates its routing table depending on its existing information on a route to dip. If no route to dip was known at all, it inserts a new entry in its routing table recording the information received. If it already has some (partial) information then it may update this information, depending on whether the new route is fresher or shorter than the one it has already. We define an update function $\mathtt{update}(rt, r)$ 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 $\pi_3(r) = \mathtt{unk} \Rightarrow \pi_5(r) = 1$. After we have introduced our formal specification for AODV in Section 5, we will show that we only use the function $\mathtt{update}$ if this condition is satisfied (Proposition 15); hence this definition is sufficient.

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

where $s := \sigma_{route}(rt, \pi_1(r))$ is the current entry in the routing table for the destination of r (if it exists), and $nrt := rt - \{s\}$ is the routing table without that entry. The entry $nr := \mathtt{addpre}(r, \pi_7(s))$ is identical to r except that the precursors from s are added and $ns := \mathtt{addpre}(s, \pi_7(r))$ is generated from s by adding the precursors from r. Lastly, nr' is identical to nr except that the sequence number is replaced by the one

[13] We use "$*$" as a wildcard.

from the route s. More precisely, $nr' := (dip_{nr}, \pi_2(s), dsk_{nr}, flag_{nr}, hops_{nr}, nhip_{nr}, pre_{nr})$ if $nr = (dip_{nr}, *, dsk_{nr}, flag_{nr}, hops_{nr}, nhip_{nr}, pre_{nr})$. In the situation where $\mathtt{sqn}(rt, \pi_1(r)) = \pi_2(r)$ both routes nr and nr' are equal. Therefore, though the cases of the above definition are not mutually exclusive, the function is well defined.

The first case describes the situation where the routing table does not contain any information on a route to dip. The second case models the situation where the new route has a greater sequence number. As a consequence all the information from the incoming information is copied into the routing table. In the third and fourth case the sequence numbers are the same and cannot be used to identify better information. Hence other measures are used. The route inside the routing table is only replaced if either the new hop count is strictly smaller—a shorter route has been found—or if the route inside the routing table is marked as invalid. The fifth case deals with the situation where a new route to a known destination has been found without any information on its sequence number ($\pi_2(r) = 0 \wedge \pi_3(r) = \mathtt{unk}$). In that case the routing table entry to that destination is always updated, but the existing sequence number is maintained, and marked as "unknown".

Note that we do not update if we receive a new entry where the sequence number and the hop count are identical to the current entry in the routing table. Following the RFC, the time period (till the valid route becomes invalid) should be reset; however at the moment we do not model timing aspects.

4.5.3 Invalidating Routes

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 or the hop count, remains accessible. The process of invalidating a routing table entry follows four rules: (a) any sequence number is incremented by 1, except (b) the truly unknown sequence number ($sqn = 0$, which will only occur if $dsk = \mathtt{unk}$) is not incremented, (c) the validity flag of the entry is set to $\mathtt{inv}$, and (d) an invalid entry cannot be invalidated again. However, in exception to (a) and (b), when the invalidation is in response to an error message, this message also contains a new (and already incremented) sequence number for each destination to be invalidated.

The function for invalidating routing table entries takes as arguments a routing table and a set of destinations $dests \in \mathscr{P}(\mathtt{IP} \times \mathtt{SQN})$. Elements of this set are (rip, rsn)-pairs that not only identify an unreachable destination rip, but also a sequence number that describes the freshness of the faulty route. As for routing tables, we restrict ourselves to sets that have at most one entry for each destination; this time we formally define $dests$ as a *partial function* from IP to SQN, i.e. a subset of $\mathtt{IP} \times \mathtt{SQN}$ satisfying

$$(rip, rsn), (rip, rsn') \in dests \;\Rightarrow\; rsn = rsn' .$$

We use the variable `dests` to range over such sets. When invoking `invalidate` we either distil $dests$ from an error message, or determine $dests$ as a set of pairs $(rip, \mathtt{inc}(\mathtt{sqn}(rt, rip)))$, where the operator `inc` (from Section 4.2) takes care of (a) and (b). Moreover, we will distil or construct $dests$ in such a way that it only lists destinations for which there is a valid entry in the routing table—this takes care of (d).

$$\begin{aligned}
&\mathtt{invalidate} : \mathtt{RT} \times (\mathtt{IP} \rightharpoonup \mathtt{SQN}) \to \mathtt{RT}\\
&\mathtt{invalidate}(rt, dests) := \{r \in rt \mid (\pi_1(r), *) \notin dests\}\\
&\quad \cup \{(\pi_1(r), rsn, \pi_3(r), \mathtt{inv}, \pi_5(r), \pi_6(r), \pi_7(r)) \mid\\
&\qquad r \in rt \wedge (\pi_1(r), rsn) \in dests\}
\end{aligned}$$

All entries in the routing table for a destination rip in $dests$ are modified. The modification replaces the value `val` by `inv` and the sequence number in the entry by the corresponding sequence number from $dests$.

Copying the sequence number from $dests$ leaves the possibility that the destination sequence number of an entry is decreased, which would violate one of the fundamental assumption of AODV and may yield unexpected behaviour. However, we will show that a decrease of a destination sequence number does not occur in our model of AODV.

4.6 Route Requests

A route request—RREQ—for a destination dip is initiated by a node (with routing table rt) if this node wants to transmit a data packet to dip but there is no valid entry for dip in the routing table, i.e. $dip \notin \mathtt{vD}(rt)$. When a new route request is sent out it contains the identity of the originating node oip, and a *route request identifier* (RREQ ID); the type of all such identifiers is denoted by `RREQID`, and the variable `rreqid` ranges over this type. This information does not change, even when the request is re-broadcast by any receiving node that does not already know a route to the requested destination. In this way any request still circulating through the network can be uniquely identified by the pair $(oip, rreqid) \in \mathtt{IP} \times \mathtt{RREQID}$. For our specification we set $\mathtt{RREQID} = \mathbb{N}$. In our model, each node maintains a variable `rreqs` of type $\mathscr{P}(\mathtt{IP} \times \mathtt{RREQID})$ of sets of such pairs to store the sets of route requests seen by the node so far. Within this set, the node records the requests it has previously initiated itself. To ensure a fresh *rreqid* for each new RREQ it generates, the node *ip* applies the following function:

$$\texttt{nrreqid} : \mathscr{P}(\texttt{IP} \times \texttt{RREQID}) \times \texttt{IP} \rightarrow \texttt{RREQID}$$
$$\texttt{nrreqid}(\mathit{rreqs}, \mathit{ip}) := \max\{n \mid (\mathit{ip}, n) \in \mathit{rreqs}\} + 1\,,$$

where we take the maximum of the empty set to be 0.

4.7 Queued Packets

Strictly speaking the task of sending data packets is not regarded as part of the AODV protocol—however, failure to send a packet because either a route to the destination is unknown, or a previously known route has become invalid, prompts AODV to be activated. In our modelling we describe this interaction between packet sending and AODV, providing the minimal infrastructure for our specification.

If a new packet is submitted by a client of AODV to a node, it may need to be stored until a route to the packet's destination has been found and the node is not busy carrying out other AODV tasks. We use a queue-style data structure for modelling the store of packets at a node, noting that at each node there may be many data queues, one for each destination. In general, we denote queues of type `TYPE` by [`TYPE`], denote the empty queue by [ ], and make use of the standard (partial) functions $\texttt{head} : [\texttt{TYPE}] \rightharpoonup \texttt{TYPE}$, $\texttt{tail} : [\texttt{TYPE}] \rightharpoonup [\texttt{TYPE}]$ and $\texttt{append} : \texttt{TYPE} \times [\texttt{TYPE}] \rightarrow [\texttt{TYPE}]$ that return the "oldest" element in the queue, remove the "oldest" element, and add a packet to the queue, respectively.

The data type

$$\texttt{STORE} := \big\{\mathit{store} \in \mathscr{P}(\texttt{IP} \times \texttt{P} \times [\texttt{DATA}]) \mid \big((\mathit{dip}, p, q), (\mathit{dip}, p', q') \in \mathit{store} \Rightarrow p = p' \wedge q = q'\big)\big\}$$

describes stores of enqueued data packets for various destinations, where $\texttt{P} := \{\texttt{no-req}, \texttt{req}\}$. An element $(\mathit{dip}, p, q) \in \texttt{IP} \times \texttt{P} \times [\texttt{DATA}]$ denotes the queue q of packets destined for *dip*; the request-required flag p is `req` if a new route discovery process for *dip* still needs to be initiated, i.e., a route request message needs to be sent. The value `no-req` indicates that such a RREQ message has been sent already, and either the reply is still pending or a route to *dip* has been established. The flag is set to `req` when a routing table entry is invalidated.

As for routing tables, we require that there is at most one entry for every IP address. In our model, each node maintains a variable `store` of type `STORE` to record its current store of data packets.

We define some functions for inspecting a store:

(a) Similar to σ_{route}, we need a function that is able to extract the queue for a given destination:

$$\sigma_{queue} : \texttt{STORE} \times \texttt{IP} \rightarrow [\texttt{DATA}]$$
$$\sigma_{queue}(\mathit{store}, \mathit{dip}) := \begin{cases} q & \text{if } (\mathit{dip}, *, q) \in \mathit{store} \\ [\,] & \text{otherwise}\,. \end{cases}$$

(b) We define a function `qD` to extract the destinations for which there are unsent packets:

$$\texttt{qD} : \texttt{STORE} \rightarrow \mathscr{P}(\texttt{IP})$$
$$\texttt{qD}(\mathit{store}) := \{\mathit{dip} \mid (\mathit{dip}, *, *) \in \mathit{store}\}\,.$$

Next, we define operations for adding and removing data packets from a store.

(c) Adding a data packet for a particular destination to a store is defined by:

$$\texttt{add} : \texttt{DATA} \times \texttt{IP} \times \texttt{STORE} \rightarrow \texttt{STORE}$$
$$\texttt{add}(d, \mathit{dip}, \mathit{store}) := \begin{cases} \mathit{store} \cup \{(\mathit{dip}, \texttt{req}, \texttt{append}(d, [\,]))\} \\ \quad \text{if } (\mathit{dip}, *, *) \notin \mathit{store} \\ \mathit{store} - \{(\mathit{dip}, p, q)\} \cup \{(\mathit{dip}, p, \texttt{append}(d, q))\} \\ \quad \text{if } (\mathit{dip}, p, q) \in \mathit{store}\,. \end{cases}$$

Informally, the process selects the entry $(\mathit{dip}, p, q) \in \mathit{store} \in \texttt{STORE}$, where *dip* is the destination of the application layer data d, and appends d to queue q of *dip* in that triple; the request-required flag p remains unchanged. In case there is no entry for *dip* in *store*, the process creates a new queue $[d]$ of stored packets that only contains the data packet under consideration and inserts it—together with *dip*—into the store; the request-required flag is set to `req`, since a route request needs to be sent.

(d) To delete the oldest packet for a particular destination from a store, we define:

$$\texttt{drop} : \texttt{IP} \times \texttt{STORE} \rightharpoonup \texttt{STORE}$$
$$\texttt{drop}(\mathit{dip}, \mathit{store}) := \begin{cases} \mathit{store} - \{(\mathit{dip}, *, q)\} \\ \quad \text{if } \texttt{tail}(q) = [\,] \\ \mathit{store} - \{(\mathit{dip}, p, q)\} \cup \{(\mathit{dip}, p, \texttt{tail}(q))\} \\ \quad \text{otherwise}\,, \end{cases}$$

where $q = \sigma_{queue}(\mathit{store}, \mathit{dip})$ is the selected queue for destination *dip*. If $\mathit{dip} \notin \texttt{qD}(\mathit{store})$ then $q = [\,]$. Therefore $\texttt{tail}(q)$ and hence also $\texttt{drop}(\mathit{dip}, \mathit{store})$ is undefined. Note that if d is the last queued packet for a specific destination, the whole entry for the destination is removed from *store*.

In our model of AODV we use only `add` and `drop` to update a store. This ensures that the store will never contain a triple $(\mathit{dip}, *, [\,])$ with an empty data queue, that is

$$\mathit{dip} \in \texttt{qD}(\mathit{store}) \Rightarrow \sigma_{queue}(\mathit{store}, \mathit{dip}) \neq [\,]\,. \tag{1}$$

Finally, we define operations for reading and manipulating the request-required flag of a queue.

(e) We define a partial function $\sigma_{p\text{-}flag}$ to extract the flag for a destination for which there are unsent packets:

$$\sigma_{p\text{-}flag} : \texttt{STORE} \times \texttt{IP} \rightharpoonup \texttt{P}$$
$$\sigma_{p\text{-}flag}(store, dip) := \begin{cases} p & \text{if } (dip, p, *) \in store \\ \text{undefined otherwise} . \end{cases}$$

(f) To change the status of the request-required flag, we define functions `setRRF` and `unsetRRF`. After a route request for destination *dip* has been initiated, the request-required flag for *dip* has to be set to `no-req`.

$$\texttt{unsetRRF} : \texttt{STORE} \times \texttt{IP} \to \texttt{STORE}$$
$$\texttt{unsetRRF}(store, dip) := \begin{cases} store - \{(dip, *, q)\} \cup \{(dip, \texttt{no-req}, q)\} \\ \quad \text{if } \{(dip, *, q)\} \in store \\ store \ \ \text{otherwise} . \end{cases}$$

In case that there is no queued data for destination *dip*, the *store* remains unchanged.
Whenever a route is invalidated the corresponding request-required flag has to be set to `req`; this indicates that the protocol might need to initiate a new route discovery process. Since the function `invalidate` invalidates sets of routing table entries, we define a function with a set of destinations $dests \in \mathscr{P}(\texttt{IP} \times \texttt{SQN})$ as one of its arguments (annotated with sequence numbers, which are not used here).

$$\texttt{setRRF} : \texttt{STORE} \times (\texttt{IP} \rightharpoonup \texttt{SQN}) \to \texttt{STORE}$$
$$\texttt{setRRF}(store, dests) :=$$
$$\{(dip, p, q) \in store \mid (dip, *) \notin dests\}$$
$$\cup \{(dip, \texttt{req}, q) \mid (dip, p, q) \in store \wedge (dip, *) \in dests\} .$$

4.8 Messages and Message Queues

Messages are the main ingredient of any routing protocol. The message types used in the AODV protocol are route request, route reply, and route error. To generate theses messages, we use functions

$$\texttt{rreq} : \mathbb{N} \times \texttt{RREQID} \times \texttt{IP} \times \texttt{SQN} \times \texttt{K} \times \texttt{IP} \times \texttt{SQN} \times \texttt{IP} \to \texttt{MSG}$$
$$\texttt{rrep} : \mathbb{N} \times \texttt{IP} \times \texttt{SQN} \times \texttt{IP} \times \texttt{IP} \to \texttt{MSG}$$
$$\texttt{rerr} : (\texttt{IP} \rightharpoonup \texttt{SQN}) \times \texttt{IP} \to \texttt{MSG} .$$ [14]

The function $\texttt{rreq}(hops, rreqid, dip, dsn, dsk, oip, osn, sip)$ generates a route request. Here, *hops* indicates the hop count from the originator *oip*—that, at the time of sending, had the sequence number *osn*—to the sender of the message *sip*; *rreqid* uniquely identifies the route request; *dsn* is the least level of freshness of a route to dip that is acceptable to *oip*—it has been obtained by incrementing the latest sequence number received in the past by *oip* for a route towards *dip*; and *dsk* indicates whether we can trust that number. In case no sequence number is known, *dsn* is set to 0 and *dsk* to `unk`. By $\texttt{rrep}(hops, dip, dsn, oip, sip)$ a route reply message is obtained. Originally, it was generated by *dip*—where *dsn* denotes the sequence number of *dip* at the time of sending—and is destined for *oip*; the last sender of the message was the node with IP address *sip* and the distance between *dip* and *sip* is given by *hops*. The error message is generated by $\texttt{rerr}(dests, sip)$, where $dests : \texttt{IP} \rightharpoonup \texttt{SQN}$ is the list of unreachable destinations and *sip* denotes the sender. Every unreachable destination *rip* comes together with the incremented last-known sequence number *rsn*.

Next to these AODV control messages, we use for our specification also data packets: messages that carry application layer data.

$$\texttt{newpkt} : \texttt{DATA} \times \texttt{IP} \to \texttt{MSG}$$
$$\texttt{pkt} : \texttt{DATA} \times \texttt{IP} \times \texttt{IP} \to \texttt{MSG}$$

Although these messages are not part of the protocol itself, they are necessary to initiate error messages, and to trigger the route discovery process. $\texttt{newpkt}(d, dip)$ generates a message containing new application layer data *d* destined for a particular destination *dip*. Such a message is submitted to a node by a client of the AODV protocol hooked up to that node. The function $\texttt{pkt}(d, dip, sip)$ generates a message containing application layer data *d*, that is sent by the sender *sip* to the next hop on the route towards *dip*.

All messages received by a particular node are first stored in a queue (see Section 5.6 for a detailed description). To model this behaviour we use a message queue, denoted by the variable `msgs` of type [`MSG`]. As for every other queue, we will freely use the functions `head`, `tail` and `append`.

Table 2 provides a summary of the entire data structure we use.

5 Modelling AODV

Our formalisation of AODV tries to accurately model the protocol as defined in the IETF RFC 3561 specification [39]. The model focusses on layer 3 of the protocol stack, i.e., the routing and forwarding of messages and

[14] The ordering of the arguments follows the RFC.

Table 2 Data structure

Basic Type	Variables	Description
IP	ip, dip, oip, rip, sip, nhip	node identifiers
SQN	dsn, osn, rsn, sn	sequence numbers
K	dsk	sequence-number-status flag
F	flag	route validity
$\mathbb{N}$	hops	hop counts
R	r	routing table entries
RT	rt	routing tables
RREQID	rreqid	request identifiers
P		request-required flag
DATA	data	application layer data
STORE	store	store of queued data packets
MSG	msg	messages

Complex Type	Variables	Description
[TYPE]		queues with elements of type TYPE
[MSG]	msgs	message queues
$\mathscr{P}$(TYPE)		sets consisting of elements of type TYPE
$\mathscr{P}$(IP)	pre	sets of identifiers (precursors, destinations, ...)
$\mathscr{P}$(IP $\times$ RREQID)	rreqs	sets of request identifiers with originator IP
$\mathtt{TYPE}_1 \rightharpoonup \mathtt{TYPE}_2$		partial functions from $\mathtt{TYPE}_1$ to $\mathtt{TYPE}_2$
IP $\rightharpoonup$ SQN	dests	sets of destinations with sequence numbers

Constant/Predicate	Description
$0:\mathtt{SQN},\ 1:\mathtt{SQN}$	unknown, smallest sequence number
$< \subseteq \mathtt{SQN}\times\mathtt{SQN}$	strict order on sequence numbers
kno, unk : K	constants to distinguish known and unknown sqns
val, inv : F	constants to distinguish valid and invalid routes
no-req, req : P	constants indicating whether a RREQ is required
$0:\mathbb{N},\ 1:\mathbb{N},\ < \subseteq \mathbb{N}\times\mathbb{N}$	standard constants/predicates of natural numbers
$[\,]:[\mathtt{TYPE}],\ \emptyset:\mathscr{P}(\mathtt{TYPE})$	empty queue, empty set
$\in \subseteq \mathtt{TYPE}\times\mathscr{P}(\mathtt{TYPE})$	membership, standard set theory

Function	Description
head : [TYPE] $\rightharpoonup$ TYPE	returns the "oldest" element in the queue
tail : [TYPE] $\rightharpoonup$ [TYPE]	removes the "oldest" element in the queue
append : TYPE $\times$ [TYPE] $\to$ [TYPE]	inserts a new element into the queue
drop : IP $\times$ STORE $\rightharpoonup$ STORE	deletes a packet from the queued data packets
add : DATA $\times$ IP $\times$ STORE $\to$ STORE	adds a packet to the queued data packets
unsetRRF : STORE $\times$ IP $\to$ STORE	set the request-required flag to no-req
setRRF : STORE $\times$ (IP $\rightharpoonup$ SQN) $\to$ STORE	set the request-required flag to req
σ_{queue} : STORE $\times$ IP $\to$ [DATA]	selects the data queue for a particular destination
$\sigma_{p\text{-}flag}$: STORE $\times$ IP $\rightharpoonup$ P	selects the flag for a destination from the store
σ_{route} : RT $\times$ IP $\rightharpoonup$ R	selects the route for a particular destination
$(\_,\_,\_,\_,\_,\_,\_)$: IP$\times$SQN$\times$K$\times$F$\times\mathbb{N}\times$IP$\times\mathscr{P}$(IP) $\to$ R	generates a routing table entry
inc : SQN $\to$ SQN	increments the sequence number
max : SQN $\times$ SQN $\to$ SQN	returns the larger sequence number
sqn : RT $\times$ IP $\to$ SQN	returns the sequence number of a particular route
sqnf : RT $\times$ IP $\to$ K	determines whether the sequence number is known
flag : RT $\times$ IP $\rightharpoonup$ F	returns the validity of a particular route
$+1:\mathbb{N}\to\mathbb{N}$	increments the hop count
dhops : RT $\times$ IP $\rightharpoonup \mathbb{N}$	returns the hop count of a particular route
nhop : RT $\times$ IP $\rightharpoonup$ IP	returns the next hop of a particular route
precs : RT $\times$ IP $\rightharpoonup \mathscr{P}$(IP)	returns the set of precursors of a particular route
vD, iD, kD : RT $\to \mathscr{P}$(IP)	returns the set of valid, invalid, known destinations
qD : STORE $\to \mathscr{P}$(IP)	returns the set of destinations with unsent packets
$\cap,\ \cup,\ \bigcup\{\ldots\},\ \ldots$	standard set-theoretic functions
addpre : R $\times \mathscr{P}$(IP) $\to$ R	adds a set of precursors to a routing table entry
addpreRT : RT $\times$ IP $\times \mathscr{P}$(IP) $\rightharpoonup$ RT	adds a set of precursors to an entry inside a table
update : RT $\times$ R $\rightharpoonup$ RT	updates a routing table with a route (if fresh enough)
invalidate : RT $\times$ (IP $\rightharpoonup$ SQN) $\to$ RT	invalidates a set of routes within a routing table
nrreqid : $\mathscr{P}$(IP $\times$ RREQID) $\times$ IP $\to$ RREQID	generates a new route request identifier
newpkt : DATA $\times$ IP $\to$ MSG	generates a message with new application layer data
pkt : DATA $\times$ IP $\times$ IP $\to$ MSG	generates a message containing application layer data
rreq : $\mathbb{N}\times$RREQID$\times$IP$\times$SQN$\times$K$\times$IP$\times$SQN$\times$IP $\to$ MSG	generates a route request
rrep : $\mathbb{N}\times$IP $\times$ SQN $\times$ IP $\times$ IP $\to$ MSG	generates a route reply
rerr : (IP $\rightharpoonup$ SQN) $\times$ IP $\to$ MSG	generates a route error message

packets, and abstracts from lower layer network protocols and mechanisms such as the Carrier Sense Multiple Access (CSMA) protocol.

The presented formalisation includes all core components of the protocol, but, at the moment, abstracts from timing issues and optional protocol features. This keeps our specification manageable. A consequence of not modelling timing issues is that statements such as "*Can a route expire before a data packet is transmitted?*" [9] cannot be analysed, for in our model routes do not expire at all. Our plan is to extend our model step by step. The model allows us to reason about protocol behaviour and to prove critical protocol characteristics. A detailed list of abstractions made can be found in [16, Section 3].

In this section, we present a specification of the AODV protocol using process algebra. The model includes a mechanism to describe the delivery of data packets; though this is not part of the protocol itself it is necessary to trigger any AODV activity. Our model consists of 7 processes, named `AODV`, `NEWPKT`, `PKT`, `RREQ`, `RREP`, `RERR` and `QMSG`:

– The basic process `AODV` reads a message from the message queue and, depending on the type of the message, calls other processes. When there is no message handling going on, the process initiates the transmission of queued data packets or generates a new route request (if packets are stored for a destination, no route to this destination is known and no route request for this destination is pending).
– The processes `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. This includes accepting the packet (if the node is the destination), forwarding the packet (if the node is not the destination) and sending an error message (if forwarding fails).
– The process `RREQ` models all events that might occur after a route request has been received. This includes updating the node's routing table, forwarding the route request as well as the initiation of a route reply if a route to the destination is known.
– Similarly, the `RREP` process describes the reaction of the protocol to an incoming route reply.
– The process `RERR` models the part of AODV which handles error messages. In particular, it describes the modification and forwarding of the AODV error message.
– The last process `QMSG` concerns message handling. Whenever a message is received, it is first stored in a message queue. If the corresponding node is able to handle a message it pops the oldest message from the queue and handles it. An example where a node is not ready to process an incoming message immediately is when it is already handling a message.

In the remainder of the section, we provide a formal specification for each of these processes and explain them step by step. Our specification can be split into three parts: the brown lines describe updates to be performed on the node's data, e.g., its routing table; the black lines are other process algebra constructs (cf. Section 3); and the blue lines are ordinary comments.

5.1 The Basic Routine

The basic process `AODV` either reads 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, and its current store of queued data packets that await transmission (cf. Section 4).

The message handling is described in Lines 1–20. First, the message has to be read from the queue of stored messages (**receive**(`msg`)). After that, the process `AODV` checks the type of the message and calls a process that can handle the message: in case of a newly injected data packet, the process `NEWPKT` is called; in case of an incoming data packet, the process `PKT` is called; in case that the incoming message is an AODV control message (route request, route reply or route error), the node updates its routing table. More precisely, if there is no entry to the message's sender `sip`, the receiver-node creates an entry with the unknown sequence number 0 and hop count 1; in case there is already a routing table entry $(\texttt{sip}, \texttt{dsn}, *, *, *, *, \texttt{pre})$, then this entry is updated to $(\texttt{sip}, \texttt{dsn}, \texttt{unk}, \texttt{val}, 1, \texttt{sip}, \texttt{pre})$ (cf. Lines 10, 14 and 18). Afterwards, the processes `RREQ`, `RREP` and `RERR` are called, respectively.

The second part of `AODV` (Lines 21–32) initiates the sending of a data packet. For that, it has to be checked if there is a queued data packet for a destination that has a known and valid route in the routing table ($\texttt{qD}(\texttt{store}) \cap \texttt{vD}(\texttt{rt}) \neq \emptyset$). In case that there is more than one destination with stored data and a known route, an arbitrary destination is chosen and denoted by `dip` (Line 21).[15] Moreover `data` is set to the first queued data packet from the application layer that should be sent (`data` :=

[15] Although the word "let" is not part of the syntax, we add it to stress the nondeterminism happening here.

Process 1 The basic routine

```
AODV(ip,sn,rt,rreqs,store) ≝
1.  receive(msg) .
2.  /* depending on the message, the node calls different processes */
3.  (
4.    [ msg = newpkt(data,dip) ]   /* new DATA packet */
5.      NEWPKT(data,dip , ip,sn,rt,rreqs,store)
6.    + [ msg = pkt(data,dip,oip) ]   /* incoming DATA packet */
7.      PKT(data,dip,oip , ip,sn,rt,rreqs,store)
8.    + [ msg = rreq(hops,rreqid,dip,dsn,dsk,oip,osn,sip) ]   /* RREQ */
9.      /* update the route to sip in rt */
10.     ⟦rt := update(rt,(sip,0,unk,val,1,sip,∅))⟧   /* 0 is used since no sequence number is known */
11.     RREQ(hops,rreqid,dip,dsn,dsk,oip,osn,sip , ip,sn,rt,rreqs,store)
12.   + [ msg = rrep(hops,dip,dsn,oip,sip) ]   /* RREP */
13.     /* update the route to sip in rt */
14.     ⟦rt := update(rt,(sip,0,unk,val,1,sip,∅))⟧
15.     RREP(hops,dip,dsn,oip,sip , ip,sn,rt,rreqs,store)
16.   + [ msg = rerr(dests,sip) ]   /* RERR */
17.     /* update the route to sip in rt */
18.     ⟦rt := update(rt,(sip,0,unk,val,1,sip,∅))⟧
19.     RERR(dests,sip , ip,sn,rt,rreqs,store)
20. )
21. + [ Let dip ∈ qD(store) ∩ vD(rt) ]   /* send a queued data packet if a valid route is known */
22.   ⟦data := head(σ_queue(store,dip))⟧
23.   unicast(nhop(rt,dip),pkt(data,dip,ip)) .
24.     ⟦store := drop(dip,store)⟧   /* drop data from the store for dip if the transmission was successful */
25.     AODV(ip,sn,rt,rreqs,store)
26.   ▶ /* an error is produced and the routing table is updated */
27.     ⟦dests := {(rip,inc(sqn(rt,rip))) | rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,dip)}⟧
28.     ⟦rt := invalidate(rt,dests)⟧
29.     ⟦store := setRRF(store,dests)⟧
30.     ⟦pre := ⋃{precs(rt,rip) | (rip,∗) ∈ dests}⟧
31.     ⟦dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}⟧
32.     groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
33. + [ Let dip ∈ qD(store) − vD(rt) ∧ σ_p-flag(store,dip) = req ]   /* a route discovery process is initiated */
34.   ⟦store := unsetRRF(store,dip)⟧   /* set request-required flag to no-req */
35.   ⟦sn := inc(sn)⟧   /* increment own sequence number */
36.   /* update rreqs by adding (ip,nrreqid(rreqs,ip)) */
37.   ⟦rreqid := nrreqid(rreqs,ip)⟧
38.   ⟦rreqs := rreqs ∪ {(ip,rreqid)}⟧
39.   broadcast(rreq(0,rreqid,dip,sqn(rt,dip),sqnf(rt,dip),ip,sn,ip)) . AODV(ip,sn,rt,rreqs,store)
```

head(σ_{queue}(store,dip))).[16] This data packet is unicast to the next hop on the route to dip. If the unicast is successful, the data packet data is removed from store (Line 24). Finally, the process calls itself—stating that the node is ready for handling a new message, initiating the sending of another packet towards a destination, etc. In case the unicast is not successful, the data packet has not been transmitted. Therefore data is not removed from store. Moreover, the node knows that the link to the next hop on the route to dip is faulty and, most probably, broken. An error message is initiated. Generally, route error and link breakage processing requires the following steps: (a) invalidating existing routing table entries, (b) listing affected destinations, (c) determining which neighbours may be affected (if any), and (d) delivering an appropriate AODV error message to such neighbours [39]. Therefore, the process determines all valid destinations dests that have this unreachable node as next hop (Line 27) and marks the routing table entries for these destinations as invalid (Line 28), while incrementing their sequence numbers (Line 27). In Line 29, we set, for all invalidated routing table entries, the request-required flag to req, thereby indicating that a new route discovery process may need to be initiated. In Line 30 the recipients of the error message are determined. These are the precursors of the invalidated destinations, i.e., the neighbouring nodes listed as having a route to one of the affected destinations passing through the broken link. Finally, an error message is sent to them (Line 32), listing only those invalidated destinations with a non-empty set of precursors (Line 31).

[16] Following the RFC, data packets waiting for a route should be buffered "first-in, first-out" (FIFO).

Process 2 Routine for handling a newly injected data packet

```
NEWPKT(data,dip , ip,sn,rt,rreqs,store) ≝
1. [ dip = ip ]      /* the DATA packet is intended for this node */
2.    deliver(data) . AODV(ip,sn,rt,rreqs,store)
3. + [ dip ≠ ip ]      /* the DATA packet is not intended for this node */
4.    ⟦store := add(data,dip,store)⟧ . AODV(ip,sn,rt,rreqs,store)
```

The third and final part of AODV (Lines 33–39) initiates a route discovery process. This is done when there is at least one queued data packet for a destination without a valid routing table entry, that is not waiting for a reply in response to a route request process initiated before. Following the RFC, the process generates a new route request. This is achieved in four steps: First, the request-required flag is set to no-req (Line 34), meaning that no further route discovery processes for this destination need to be initiated.[17] Second, the node's own sequence number is increased by 1 (Line 35). Third, by determining nrreqid(rreqs , ip), a new route request identifier is created and stored—together with the node's ip—in the set rreqs of route requests already seen (Line 38). Fourth, the message itself is sent (Line 39) using broadcast. In contrast to **unicast**, transmissions via **broadcast** are not checked on success. The information inside the message follows strictly the RFC. In particular, the hop count is set to 0, the route request identifier previously created is used, etc. This ends the initiation of the route discovery process.

5.2 Data Packet Handling

The processes NEWPKT and PKT describe all actions performed by a node when a data packet is injected by a client hooked up to the local node or received via the protocol, respectively. For the process PKT, this includes the acceptance (if the node is the destination), the forwarding (if the node is not the destination), as well as the sending of an error message in case something went wrong. The process NEWPKT does not include the initiation of a new route request; this is part of the process AODV. Although packet handling itself is not part of AODV, it is necessary to include it in our formalisation, since a failure to transmit a data packet triggers AODV activity.

The process NEWPKT first checks whether the node is the intended addressee of the data packet. If this is the case, it delivers the data and returns to the basic routine AODV. If the node is not the intended destination ($\texttt{dip} \neq \texttt{ip}$, Line 3), the data is added to the data queue for dip (Line 4),[18] which finishes the handling of a newly injected data packet. The further handling of queued data (forwarding it to the next hop on the way to the destination in case a valid route to the destination is known, and otherwise initiating a new route request if still required) is the responsibility of the main process AODV.

Similar to NEWPKT, the process PKT first checks if it is the intended addressee of the data packet. If this is the case, it delivers the data and returns to the basic routine AODV. If the node is not the intended destination ($\texttt{dip} \neq \texttt{ip}$, Line 3) more activity is needed.

In case that the node has a valid route to the data's destination dip ($\texttt{dip} \in \texttt{vD}(\texttt{rt})$), it forwards the packet using a unicast to the next hop nhop(rt , dip) on the way to dip. Similar to the unicast of the process AODV, it has to be checked whether the transmission is successful: no further action is necessary if the transmission succeeds, and the node returns to the basic routine AODV. If the transmission fails, the link to the next hop nhop(rt , dip) is assumed to be broken. As before, all destinations dests that are reached via that broken link are determined (Line 9) and all precursors interested in at least one of these destinations are informed via an error message (Line 14). Moreover, all the routing table entries using the broken link have to be invalidated in the node's routing table rt (Line 10), and all corresponding request-required flags are set to req (Line 11).

In case that the node has no valid route to the destination dip ($\texttt{dip} \notin \texttt{vD}(\texttt{rt})$), the data packet is lost and possibly an error message is sent. If there is an (invalid) route to the data's destination dip in the routing table (Line 18), the possibly affected neighbours can be determined and the error message is sent to these precursors (Line 20). If there is no information about a route towards dip nothing happens (and the basic process AODV is called again).

[17] The RFC does not describe packet handling in detail; hence the request-required flag is not part of the RFC's RREQ generation process.

[18] If no data for destination dip was already queued, the function add creates a fresh queue for dip, and set the request-required flag to req; otherwise, the request-required flag keeps the value it had already.

Process 3 Routine for handling a received data packet

```
PKT(data,dip,oip , ip,sn,rt,rreqs,store) ≝
[ dip = ip ]      /* the DATA packet is intended for this node */
  deliver(data) . AODV(ip,sn,rt,rreqs,store)
+ [ dip ≠ ip ]      /* the DATA packet is not intended for this node */
  (
    [ dip ∈ vD(rt) ]      /* valid route to dip */
      /* forward packet */
      unicast(nhop(rt,dip),pkt(data,dip,oip)) . AODV(ip,sn,rt,rreqs,store)
      ▶ /* If the packet transmission is unsuccessful, a RERR message is generated */
        [[dests := {(rip, inc(sqn(rt,rip))) | rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,dip)}]]
        [[rt := invalidate(rt,dests)]]
        [[store := setRRF(store,dests)]]
        [[pre := ⋃{precs(rt,rip) | (rip, ∗) ∈ dests}]]
        [[dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
        groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
    + [ dip ∉ vD(rt) ]      /* no valid route to dip */
      /* no local repair occurs; data is lost */
      (
        [ dip ∈ iD(rt) ]      /* invalid route to dip */
          /* if the route is invalid, a RERR is sent to the precursors */
          groupcast(precs(rt,dip),rerr({(dip,sqn(rt,dip))},ip)) . AODV(ip,sn,rt,rreqs,store)
        + [ dip ∉ iD(rt) ]      /* route not in rt */
          AODV(ip,sn,rt,rreqs,store)
      )
  )
```

5.3 Receiving Route Requests

The process RREQ models all events that may occur after a route request has been received.

RREQ first reads the unique identifier (oip, rreqid) of the route request received. If this pair is already stored in the node's data rreqs, the route request has been handled before and the message can silently be ignored (Lines 1–2).

If the received message is new to this node, i.e.,

$$(\texttt{oip},\texttt{rreqid}) \notin \texttt{rreqs} \quad \text{(Line 3)}\,,$$

the node establishes a route of length hops+1 back to the originator oip of the message. If this route is "better" than the route to oip in the current routing table, the routing table is updated by this route (Line 4). Moreover the unique identifier has to be added to the set rreqs of already seen (and handled) route requests (Line 5).

After these updates the process checks if the node is the intended destination (dip = ip, Line 7). In that case, a route reply must be initiated: first, the node's sequence number is—according to the RFC—set to the maximum of the current sequence number and the destination sequence number stemming from the RREQ message (Line 8).Then the reply is unicast to the next hop on the route back to the originator oip of the route request. The content of the new route reply is as follows: the hop count is set to 0, the destination and originator are copied from the route request received and the destination's sequence number is the node's own sequence number sn; of course the sender's IP of this message has to be set to the node's ip. As before (cf. Sections 5.1 and 5.2), the process invalidates the corresponding routing table entries, sets request-required flags and sends an error message to all relevant precursors if the unicast transmission fails (Lines 12–17).

If the node is not the destination dip of the message but an intermediate hop along the path from the originator to the destination, it is allowed to generate a route reply only if the information in its own routing table is fresh enough. This means that (a) the node has a valid route to the destination, (b) the destination sequence number in the node's existing routing table entry for the destination (sqn(rt,dip)) is greater than or equal to the requested destination sequence number dsn of the message and (c) the sequence number sqn(rt,dip) is known, i.e., sqnf(rt,dip) = kno. If these three conditions are satisfied—the check is done in Line 20—the node generates a new route reply and sends it to the next hop on the way back to the originator oip of the received route request.[19]. To this end, it 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 25). The

[19] This next hop will often, but not always, be sip; see [16].

Process 4 RREQ handling

```
RREQ(hops,rreqid,dip,dsn,dsk,oip,osn,sip , ip,sn,rt,rreqs,store) ≝
1.  [ (oip , rreqid) ∈ rreqs ]      /* the RREQ has been received previously */
2.    AODV(ip,sn,rt,rreqs,store)      /* silently ignore RREQ, i.e. do nothing */
3.  + [ (oip , rreqid) ∉ rreqs ]      /* the RREQ is new to this node */
4.    [[rt := update(rt, (oip, osn, kno, val, hops + 1, sip, ∅))]]      /* update the route to oip in rt */
5.    [[rreqs := rreqs ∪ {(oip, rreqid)}]]      /* update rreqs by adding (oip , rreqid) */
6.    (
7.      [ dip = ip ]      /* this node is the destination node */
8.        [[sn := max(sn, dsn)]]      /* update the sqn of ip */
9.        /* unicast a RREP towards oip of the RREQ */
10.       unicast(nhop(rt,oip),rrep(0,dip,sn,oip,ip)) . AODV(ip,sn,rt,rreqs,store)
11.       ▶ /* If the transmission is unsuccessful, a RERR message is generated */
12.         [[dests := {(rip, inc(sqn(rt,rip))) | rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}]]
13.         [[rt := invalidate(rt,dests)]]
14.         [[store := setRRF(store,dests)]]
15.         [[pre := ⋃{precs(rt,rip) | (rip, ∗) ∈ dests}]]
16.         [[dests := {(rip, rsn) | (rip, rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
17.         groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
18.     + [ dip ≠ ip ]      /* this node is not the destination node */
19.       (
20.         [dip ∈ vD(rt) ∧ dsn ≤ sqn(rt,dip) ∧ sqnf(rt,dip) = kno ]      /* valid route to dip that is fresh enough */
21.           /* update rt by adding precursors */
22.           [[rt := addpreRT(rt,dip,{sip})]]
23.           [[rt := addpreRT(rt,oip,{nhop(rt,dip)})]]
24.           /* unicast a RREP towards the oip of the RREQ */
25.           unicast(nhop(rt,oip),rrep(dhops(rt,dip),dip,sqn(rt,dip),oip,ip)) .
26.             AODV(ip,sn,rt,rreqs,store)
27.           ▶ /* If the transmission is unsuccessful, a RERR message is generated */
28.             [[dests := {(rip, inc(sqn(rt,rip))) | rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}]]
29.             [[rt := invalidate(rt,dests)]]
30.             [[store := setRRF(store,dests)]]
31.             [[pre := ⋃{precs(rt,rip) | (rip, ∗) ∈ dests}]]
32.             [[dests := {(rip, rsn) | (rip, rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
33.             groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
34.       + [ dip ∉ vD(rt) ∨ sqn(rt,dip) < dsn ∨ sqnf(rt,dip) = unk ]      /* no valid route that is fresh enough */
35.           /* no further update of rt */
36.           broadcast(rreq(hops+1,rreqid,dip,max(sqn(rt,dip),dsn),dsk,oip,osn,ip)) .
37.           AODV(ip,sn,rt,rreqs,store)
38.       )
39.   )
```

unicast might fail, which causes the usual error handling (Lines 28–33). Just before transmitting the unicast, the intermediate node updates the forward route entry to `dip` by placing the last hop node (`sip`)[20] into the precursor list for the forward route entry (Line 22). Likewise, it updates the reverse route entry to `oip` by placing the first hop `nhop(rt,dip)` towards `dip` in the precursor list for that entry (Line 23).[21]

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 36. 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.

[20] This is a mistake in the RFC; it should be `nhop(rt,oip)`.

[21] Unless the *gratuitous RREP flag* is set, which we do not model in this paper, this update is rather useless, as the precursor `nhop(rt,dip)` in general is not aware that it has a route to `oip`.

5.4 Receiving Route Replies

The process `RREP` describes the reaction of the protocol to an incoming route reply. Our model first checks if a forward routing table entry is going to be created or updated (Line 1). This is the case if (a) the node has no known route to the destination, or (b) the destination sequence number in the node's existing routing table entry for the destination (`sqn(rt,dip)`) is smaller

Process 5 RREP handling

```
RREP(hops,dip,dsn,oip,sip , ip,sn,rt,rreqs,store) ≝
1.  [ rt ≠ update(rt,(dip,dsn,kno,val,hops + 1,sip,∅)) ]      /* the routing table has to be updated */
2.    [[rt := update(rt,(dip,dsn,kno,val,hops + 1,sip,∅))]]
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 ≠ ip ]      /* this node is not the originator; forward RREP */
8.        (
9.          [ oip ∈ 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.           unicast(nhop(rt,oip),rrep(hops+1,dip,dsn,oip,ip)) .
14.             AODV(ip,sn,rt,rreqs,store)
15.           ▶ /* If the transmission is unsuccessful, a RERR message is generated */
16.             [[dests := {(rip, inc(sqn(rt,rip))) | rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}]]
17.             [[rt := invalidate(rt,dests)]]
18.             [[store := setRRF(store,dests)]]
19.             [[pre := ⋃{precs(rt,rip) | (rip,∗) ∈ dests}]]
20.             [[dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
21.             groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
22.       + [ oip ∉ vD(rt) ]      /* no valid route to oip */
23.           AODV(ip,sn,rt,rreqs,store)
24.       )
25.   )
26. + [ rt = update(rt,(dip,dsn,kno,val,hops + 1,sip,∅)) ]      /* the routing table is not updated */
27.   AODV(ip,sn,rt,rreqs,store)
```

than the destination sequence number dsn in the RREP message, or (c) the two destination sequence numbers are equal and, in addition, either the incremented hop count of the RREP received is strictly smaller than the one in the routing table, or the entry for dip in the routing table is invalid. Hence Line 1 could be replaced by

$$[\mathtt{dip} \notin \mathtt{kD}(\mathtt{rt}) \lor \mathtt{sqn}(\mathtt{rt},\mathtt{dip}) < \mathtt{dsn} \lor (\mathtt{sqn}(\mathtt{rt},\mathtt{dip}) = \mathtt{dsn} \land (\mathtt{dhops}(\mathtt{rt},\mathtt{dip}) > \mathtt{hops}{+}1 \lor \mathtt{flag}(\mathtt{rt},\mathtt{dip}) = \mathtt{inv}))]\,.$$ [22]

In case that one of these conditions is true, the routing table is updated in Line 2. If the node is the intended addressee of the route reply ($\mathtt{oip} = \mathtt{ip}$) the protocol returns to its basic process AODV. Otherwise ($\mathtt{oip} \neq \mathtt{ip}$) the message should be forwarded. Following the RFC [39], “If the current node is not the node indicated by the Originator IP Address in the RREP message AND a forward route has been created or updated [. . .], the node consults its route table entry for the originating node to determine the next hop for the RREP packet, and then forwards the RREP towards the originator using the information in that route table entry.” This action needs a valid route to the originator oip of the route request to which the current message is a reply ($\mathtt{oip} \in \mathtt{vD}(\mathtt{rt})$, Line 9). The content of the RREP message to be sent is mostly copied from the RREP received; only the sender has to be changed (it is now the node's ip) and the hop count is incremented. Prior to the unicast, the node **nhop**(rt , oip), to which the message is sent, is added to the list of precursors for the routes to dip (Line 11) and to the next hop on the route to dip (Line 12). Although not specified in the RFC, it would make sense to also add a precursor to the reverse route by $[\![\mathtt{rt} := \mathtt{addpreRT}(\mathtt{rt}, \mathtt{oip}, \{\mathtt{nhop}(\mathtt{rt}, \mathtt{dip})\})]\!]$. As usual, if the unicast fails, the affected routing table entries are invalidated and the precursors of all routes using the broken link are determined and an error message is sent (Lines 16–21). In the unlikely situation that a reply should be forwarded but no valid route is known by the node, nothing happens. Following the RFC, no precursor has to be notified and no error message has to be sent—even if there is an invalid route.

If a forward routing table entry is not created nor updated, the reply is silently ignored and the basic process is called (Lines 26–27).

[22] In case $\mathtt{dip} \notin \mathtt{kD}(\mathtt{rt})$, the terms dhops(rt , dip) and flag(rt , dip) are not defined. In such a case, according to the convention of Footnote 8 in Section 3, the atomic formulas $\mathtt{dhops}(\mathtt{rt},\mathtt{dip}) > \mathtt{hops}{+}1$ and $\mathtt{flag}(\mathtt{rt},\mathtt{dip}) = \mathtt{inv}$ evaluate to false. However, in case one would use lazy evaluation of the outermost disjunction, the evaluation of the expression would be independent of the choice of a convention for interpreting undefined terms appearing in formulas.

Process 6 RERR handling

```
RERR(dests,sip , ip,sn,rt,rreqs,store) ≝
1. /* invalidate broken routes */
2. [[dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧ rip ∈ vD(rt) ∧ nhop(rt,rip) = sip ∧ sqn(rt,rip) < rsn}]]
3. [[rt := invalidate(rt,dests)]]
4. [[store := setRRF(store,dests)]]
5. /* forward the RERR to all precursors for rt entries for broken connections */
6. [[pre := ⋃{precs(rt,rip) | (rip,∗) ∈ dests}]]
7. [[dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
8. groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
```

5.5 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.

If a node receives an error message from a neighbour for one or more valid routes, it has—under some conditions—to invalidate the entries for those routes in its own routing table and forward the error message. The node compares the set dests of unavailable destinations from the incoming error message with its own entries in the routing table. If the routing table lists a valid route with a (rip, rsn)-combination from dests and if the next hop on this route is the sender sip of the error message, this entry may be affected by the error message. In our formalisation, we have added the requirement $\texttt{sqn}(\texttt{rt},\texttt{rip}) < \texttt{rsn}$, saying that the entry is affected by the error message only if the "incoming" sequence number is larger than the one stored in the routing table, meaning that it is based on fresher information.[23] In this case, the entry has to be invalidated and all precursors of this particular route have to be informed. This has to be done for all affected routes.

In fact, the process first determines all (rip, rsn)-pairs that have effects on its own routing table and that may have to be forwarded as content of a new error message (Line 2). After that, all entries to unavailable destinations are invalidated (Line 3), and as usual when routing table entries are invalidated, the request-required flags are set to req (Line 4). In Line 6 the set of all precursors (affected neighbours) of the unavailable destinations are summarised in the set pre. Then, the set dests is "thinned out" to only those destinations that have at least one precursor— only these destinations are transmitted in the forwarded error message (Line 7). Finally, the message is sent (Line 8).

[23] This additional requirement is in the spirit of Section 6.2 of the RFC [39] on updating routing table entries, but in contradiction with Section 6.11 of the RFC on handling RERR messages. In [20] we show that the reading of Section 6.11 of the RFC gives rise to routing loops.

5.6 The Message Queue and Synchronisation

We assume that any message sent by a node *sip* to a node *ip* that happens to be within transmission range of *sip* is actually received by *ip*. For this reason, *ip* should always be able to perform a receive action, regardless of which state it is in. However, the main process AODV that runs on the node *ip* can reach a state, such as PKT, RREQ, RREP or RERR, in which it is not ready to perform a receive action. 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. Every incoming message is first stored in this queue, and piped from there to the process AODV, whenever AODV is ready to handle a new message. The process QMSG is always ready to receive a new message, even when AODV is not. The whole parallel process running on a node is then given by an expression of the form

$$(\xi, \texttt{AODV}(\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store})) \;\langle\!\langle\; (\zeta, \texttt{QMSG}(\texttt{msgs}))\,.$$

5.7 Initial State

To finish our specification, we have to define an initial state. The initial network expression is an encapsulated parallel composition of node expressions $ip : P : R$, where the (finite) number of nodes and the range R of each node expression is left unspecified (can be anything). However, each node in the parallel composition is required to have a unique IP address *ip*. The initial process P of *ip* is given by the expression

$$(\xi, \texttt{AODV}(\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store})) \;\langle\!\langle\; (\zeta, \texttt{QMSG}(\texttt{msgs}))\,,$$

with

$$\begin{aligned}\xi(\texttt{ip}) = ip \wedge \xi(\texttt{sn}) = 1 \wedge \xi(\texttt{rt}) = \emptyset \wedge \xi(\texttt{rreqs}) = \emptyset \\ \wedge\, \xi(\texttt{store}) = \emptyset \wedge \zeta(\texttt{msgs}) = [\,]\,.\end{aligned} \tag{2}$$

This says that initially each node is correctly informed about its own identity; its own sequence number is initialised with 1 and its routing table, the list of RREQs seen, the store of queued data packets as well as the message queue are empty.

Process 7 Message queue

```
QMSG(msgs) ≝
    /* store incoming message at the end of msgs */
    receive(msg) . QMSG(append(msg,msgs))
  + [ msgs ≠ [] ]     /* the queue is not empty */
    (
      /* pop top message and send it to another sequential process */
      send(head(msgs)) . QMSG(tail(msgs))
      /* or receive and store an incoming message */
      + receive(msg) . QMSG(append(msg,msgs))
    )
```

6 Invariants

Using AWN and the proposed model of AODV we can now formalise and prove crucial properties of AODV. In this section we verify properties that can be expressed as invariants, i.e., statements that hold all the time when the protocol is executed.

The most important invariant we establish is *loop freedom*; most prior results can be regarded as stepping stones towards this goal. Next to that we also formalise and discuss *route correctness*.

6.1 State and Transition Invariants

A *(state) invariant* is a statement that holds for all reachable states of our model. Here states are network expressions, as formally defined in [16] and described in Section 3. An invariant is usually verified by showing that it holds for all possible initial states, and that, for any transition $N \xrightarrow{\ell} N'$ between network expressions derived by our operational semantics, if it holds for state N then it also holds for state N'. In this paper we abstain from a formal definition of the operational semantics, and hence do not define the labelled transition relation $\longrightarrow$ between network states. Instead we verify invariants by checking that they are preserved under any execution of any line in one of the Processes 1–7. In [16] we formally document that such a check yields the required result.

Besides (state) invariants, we also establish statements we call *transition invariants*. A transition invariant is a statement that holds for each reachable transition $N \xrightarrow{\ell} N'$ between network expressions derived by the operational semantics. Again these transitions correspond with lines in one of the Processes 1–7; they either describe a relation between the states N and N' before and after executing the instruction—e.g. that the value of a specific variable maintained by our processes will never decrease—or they describe a relation between the instruction being executed (such as a broadcast of a message involving a certain value) and the state right beforehand (such as a comparable value maintained by the broadcasting node). Transition invariants are simply checked by going though all appropriate lines in Processes 1–7. In a few cases we use *induction on reachability*; this amounts to assuming that the same relation holds for instructions executed earlier. Again we refer to [16] for the soundness of this approach.

In our formalisation of transition invariants, we write $N \xrightarrow{R:\textbf{*cast}(m)}_{ip} N'$ to indicate that our network moves from state N to state N' by means of a **broadcast**, **unicast** or **groupcast** of the message m, executed by node ip, while the current transmission range of this node is R.

The following observations are crucial in establishing many of our invariants.

Proposition 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 1 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 $\textbf{*cast}(m)$ of a node ip' of the network.

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

Proof The only way Line 1 of Pro. 1 can be executed, is through a synchronisation of the main process AODV with the message queue QMSG (Pro. 6) running on the same node. This involves the action **send**(m) of QMSG. Here m is popped from the message queue msgs, which started out empty. So at some point QMSG must have performed the action **receive**(m). However, this action is blocked by the encapsulation operator $[\_]$, except when m has the form $\texttt{newpkt}(d,dip)$ or when it synchronises with an action ***cast**(m) of another node. $\square$

At first glance Part(b) does not seem to reflect reality. Of course, an application running on a local node has to be able to send data packets to another application running on the same node. However, in any practical implementation, when a node sends a message to itself, the message will be delivered to the corresponding application on the local node without ever being "seen" by AODV or any other routing protocol. Therefore, from AODV's perspective, no node can receive a message (directly) from itself.

6.2 Notions and Notations

Before formalising and proving invariants, we introduce some useful notions and notations.

All processes except QMSG maintain the five data variables ip, sn, rt, rreqs and store. Next to that QMSG maintains the variable msgs. Hence, these 6 variables can be evaluated at any time. Moreover, every node expression in the transition system looks like

$$ip : (\xi, P \;\langle\!\langle\; \zeta, \texttt{QMSG(msgs)}) : R\,,$$

where P is a state in one of the following sequential processes:

```
AODV(ip,sn,rt,rreqs,store),
NEWPKT(data,dip , ip,sn,rt,rreqs,store),
PKT(data,dip,oip , ip,sn,rt,rreqs,store),
RREQ(hops,rreqid,dip,dsn,dsk,oip,osn,sip ,
     ip,sn,rt,rreqs,store)
RREP(hops,dip,dsn,oip,sip ,
     ip,sn,rt,rreqs,store)
RERR(dests,sip , ip,sn,rt,rreqs,store).
```

Hence the state of the transition system for a node ip is determined by the process P, the range R, and the two valuations ξ and ζ. If a network consists of a (finite) set $\mathbf{IP} \subseteq \texttt{IP}$ of nodes, a reachable network expression N is an encapsulated parallel composition of node expressions—one for each $ip \in \mathbf{IP}$. In this section, we assume N and N' to be reachable network expressions in our model of AODV. To distill information about a node from N, we define the following projections:

$$\begin{aligned}
P_N^{ip} &:= P, \text{ where } ip:(*, P \,\langle\!\langle\, *, *): * \text{ is a node expr. of } N,\\
R_N^{ip} &:= R, \text{ where } ip:(*, * \,\langle\!\langle\, *, *):R \text{ is a node expr. of } N,\\
\xi_N^{ip} &:= \xi, \text{ where } ip:(\xi, * \,\langle\!\langle\, *, *): * \text{ is a node expr. of } N,\\
\zeta_N^{ip} &:= \zeta, \text{ where } ip:(*, * \,\langle\!\langle\, \zeta, *): * \text{ is a node expr. of } N.
\end{aligned}$$

For example, P_N^{ip} determines the sequential process the node is currently working in, R_N^{ip} denotes the set of all nodes currently within transmission range of ip, and $\xi_N^{ip}(\texttt{rt})$ evaluates the current routing table maintained by node ip in the network expression N. In the forthcoming proofs, when discussing the effects of an action, identified by a line number in one of the processes of our model, ξ denotes the current valuation ξ_N^{ip}, where ip is the address of the local node, executing the action under consideration, and N is the network expression obtained right before this action occurs, corresponding with the line number under consideration. When considering the effects of several actions, corresponding to several line numbers, ξ is always interpreted most locally. For instance, in the proof of Proposition 12(a), case **Pro. 4, Line 36**, we write

> Hence ... $ip_c := \xi(\texttt{ip}) = ip$ and $\xi_N^{ip_c} = \xi$ (by (3)). At Line 4 we update the routing table using $r := \xi(\texttt{oip},\texttt{osn},\texttt{kno},\texttt{val},\texttt{hops}{+}1,\texttt{sip},\emptyset)$ as new entry. The routing table does not change between Lines 4 and 36; nor do the values of hops, oip and osn.

Writing N_k for a network expression in which the local node ip is about to execute Line k, this passage can be reworded as

> Hence ... $ip_c := \xi_{N_{36}}^{ip}(\texttt{ip}) = ip$ and $\xi_{N_{36}}^{ip_c} = \xi_{N_{36}}^{ip}$ (by (3)). $\xi_{N_5}^{ip}(\texttt{rt}) :=$
> $\xi_{N_4}^{ip}(\texttt{update}(\texttt{rt},(\texttt{oip},\texttt{osn},\texttt{kno},\texttt{val},\texttt{hops}{+}1,\texttt{sip},\emptyset)))$
> $:= \texttt{update}(\xi_{N_4}^{ip}(\texttt{rt}),(\xi_{N_4}^{ip}(\texttt{oip}),\xi_{N_4}^{ip}(\texttt{osn}),\ldots))$.
> $\xi_{N_5}^{ip}(\texttt{rt}) = \xi_{N_{36}}^{ip}(\texttt{rt}) \;\wedge \xi_{N_4}^{ip}(\texttt{hops}) = \xi_{N_{36}}^{ip}(\texttt{hops}) \wedge$
> $\xi_{N_4}^{ip}(\texttt{oip}) = \xi_{N_{36}}^{ip}(\texttt{oip}) \wedge\; \xi_{N_4}^{ip}(\texttt{osn}) = \xi_{N_{36}}^{ip}(\texttt{osn})$.

In all of case **Pro. 4, Line 36**, through the statement of the proposition, N is bound to N_{36}, so that $\xi_N^{ip} = \xi_{N_{36}}^{ip}$.

In Section 4.4 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 $\texttt{sqn}_N^{ip}(dip)$, $\texttt{dhops}_N^{ip}(dip)$, $\texttt{flag}_N^{ip}(dip)$, $\sigma_{route}{}_N^{ip}(dip)$, $\texttt{kD}_N^{ip}$, $\texttt{vD}_N^{ip}$ and $\texttt{iD}_N^{ip}$ for $\texttt{sqn}(\xi_N^{ip}(\texttt{rt}),dip)$, $\texttt{dhops}(\xi_N^{ip}(\texttt{rt}),dip)$, $\texttt{flag}(\xi_N^{ip}(\texttt{rt}),dip)$, $\sigma_{route}(\xi_N^{ip}(\texttt{rt}),ip)$, $\texttt{kD}(\xi_N^{ip}(\texttt{rt}))$, $\texttt{vD}(\xi_N^{ip}(\texttt{rt}))$ and $\texttt{iD}(\xi_N^{ip}(\texttt{rt}))$, respectively.

6.3 Basic Properties

In this section we show some of the most fundamental invariants for AODV. The first one is already stated in the RFC [39, Sect. 3].

Proposition 2 Any sequence number of a 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 In all initial states the invariant is satisfied, as all sequence numbers of all nodes are set to 1 (see (2) in Section 5.7). The Processes 1–7 of Section 5 change a node's sequence number only through the functions inc and max. This occurs at two places only:

Pro. 1, Line 35: Here $\xi_N^{ip}(\mathtt{sn}) \leq \mathtt{inc}(\xi_N^{ip}(\mathtt{sn})) = \xi_{N'}^{ip}(\mathtt{sn})$.
Pro. 4, Line 8: Here $\xi_N^{ip}(\mathtt{sn}) \leq \max(\xi_N^{ip}(\mathtt{sn}),*) = \xi_{N'}^{ip}(\mathtt{sn})$.

From this and the fact that all sequence numbers are initialised with 1 we get $1 \leq \xi_N^{ip}(\mathtt{sn})$. □

The proof strategy used above can be generalised.

Remark 3 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` or `addpreRT`. 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.

To ease readability we defer most of the proofs to the appendix; and show only the most important ones in the main text.

Proposition 4 The set of known destinations of any node increases monotonically. That is, for $ip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $\mathtt{kD}_N^{ip} \subseteq \mathtt{kD}_{N'}^{ip}$.

Proposition 5 In each node's routing table, the sequence number for a given destination increases monotonically, i.e., never decreases. That is, for $ip, dip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $\mathtt{sqn}_N^{ip}(dip) \leq \mathtt{sqn}_{N'}^{ip}(dip)$.

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

Proposition 6 For each $ip \in \mathbf{IP}$ and each reachable state N we have $\xi_N^{ip}(\mathtt{ip}) = ip$.

This proposition will be used implicitly in many of the proofs to follow. In particular, for all $ip', ip'' \in \mathbf{IP}$

$$\xi_N^{ip'}(\mathtt{ip}) = ip'' \Rightarrow ip' = ip'' \wedge \xi_N^{ip'} = \xi_N^{ip''} . \tag{3}$$

Next, we show that every AODV control message contains the IP address of the sender.

Proposition 7 If an AODV control message is sent by node $ip \in \mathbf{IP}$, the node sending this message identifies itself correctly: $N \xrightarrow{R:\mathbf{*cast}(m)}_{ip} N' \Rightarrow ip = ip_c$, where the message m is either $\mathtt{rreq}(*,*,*,*,*,*,*,ip_c)$, $\mathtt{rrep}(*,*,*,*,ip_c)$, or $\mathtt{rerr}(*,ip_c)$.

The proof is straightforward: whenever such a message is sent in one of the processes of Section 5, $\xi(\mathtt{ip})$ is set as the last argument. □

Proposition 8 All routing table entries have a hop count greater than or equal to 1.

$$(*,*,*,*,hops,*,*) \in \xi_N^{ip}(\mathtt{rt}) \Rightarrow hops \geq 1 \tag{4}$$

Proposition 9

(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.

$$N \xrightarrow{R:\mathbf{*cast}(\mathtt{rreq}(0,*,*,*,*,oip_c,*,ip_c))}_{ip} N' \Rightarrow oip_c = ip_c \tag{5}$$

(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.

$$N \xrightarrow{R:\mathbf{*cast}(\mathtt{rrep}(0,dip_c,*,*,ip_c))}_{ip} N' \Rightarrow dip_c = ip_c \tag{6}$$

Proposition 10

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

$$(dip,0,f,*,*,*,*) \in \xi_N^{ip}(\mathtt{rt}) \Rightarrow f = \mathtt{unk} \tag{7}$$

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

$$(*,*,\mathtt{unk},*,hops,*,*) \in \xi_N^{ip}(\mathtt{rt}) \Rightarrow hops = 1 \tag{8}$$

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

$$(dip,*,*,*,1,nhip,*) \in \xi_N^{ip}(\mathtt{rt}) \Rightarrow dip = nhip \tag{9}$$

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

$$\begin{aligned} &(dip,0,f,*,hops,nhip,*) \in \xi_N^{ip}(\mathtt{rt}) \\ &\Rightarrow f = \mathtt{unk} \wedge hops = 1 \wedge dip = nhip \end{aligned} \tag{10}$$

Proposition 11

(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.

$$N \xrightarrow{R:\mathbf{*cast}(\mathtt{rreq}(*,*,*,*,*,*,osn_c,*))}_{ip} N' \Rightarrow osn_c \geq 1 \tag{11}$$

(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.

$$N \xrightarrow{R:\mathbf{*cast}(\mathtt{rrep}(*,*,dsn_c,*,*))}_{ip} N' \Rightarrow dsn_c \geq 1 \tag{12}$$

Proposition 12

(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.

$$\begin{aligned} &N \xrightarrow{R:\mathbf{*cast}(\mathtt{rreq}(hops_c,*,*,*,*,oip_c,osn_c,ip_c))}_{ip} N' \\ &\wedge\ ip_c \neq oip_c \\ \Rightarrow\ & oip_c \in \mathtt{kD}_N^{ip_c} \wedge \big(\mathtt{sqn}_N^{ip_c}(oip_c) > osn_c \\ &\vee (\mathtt{sqn}_N^{ip_c}(oip_c) = osn_c \wedge \mathtt{dhops}_N^{ip_c}(oip_c) \leq hops_c \\ &\quad \wedge \mathtt{flag}_N^{ip_c}(oip_c) = \mathtt{val})\big) \end{aligned} \tag{13}$$

(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.

$$\begin{aligned} &N \xrightarrow{R:\mathbf{*cast}(\mathtt{rrep}(hops_c,dip_c,dsn_c,*,ip_c))}_{ip} N' \\ &\wedge\ ip_c \neq dip_c \\ \Rightarrow\ & dip_c \in \mathtt{kD}_N^{ip_c} \wedge \mathtt{sqn}_N^{ip_c}(dip_c) = dsn_c \\ &\wedge \mathtt{dhops}_N^{ip_c}(dip_c) = hops_c \wedge \mathtt{flag}_N^{ip_c}(dip_c) = \mathtt{val} \end{aligned} \tag{14}$$

Proposition 13 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.

$$\begin{aligned} &N \xrightarrow{R:\mathbf{*cast}(\mathtt{rerr}(dests_c,ip_c))}_{ip} N' \\ &\wedge (rip_c, rsn_c) \in dests_c \\ \Rightarrow\ & rip_c \in \mathtt{iD}_N^{ip} \wedge rsn_c = \mathtt{sqn}_N^{ip}(rip_c) \end{aligned} \tag{15}$$

6.4 Well-Definedness

We have to ensure that our specification of AODV is actually well defined. Since many functions introduced in Section 4 are only partial, it has to be checked that these functions are either defined when they are used, or are subterms of atomic formulas. In the latter case, those formula would evaluate to `false` (cf. Footnote 8).

The first proposition shows that the functions defined in Section 4 respect the data structure. In fact, these properties are required (or implied) by our data structure.

Proposition 14

(a) In each routing table there is at most one entry for each destination.
(b) In each store of queued data packets there is at most one data queue for each destination.
(c) Whenever a set of pairs (rip, rsn) is assigned to the variable `dests` of type $\mathtt{IP} \rightharpoonup \mathtt{SQN}$, or to the first argument of the function `rerr`, this set is a partial function, i.e., there is at most one entry (rip, rsn) for each destination rip.

Property (a) is stated in the RFC [39].

Next, we show that a function is used in the specification of AODV only when it is defined, with `nhop` and $\sigma_{p\text{-}flag}$ as possible exceptions. In this paper, we only give the proof for `update`; for the remaining functions σ_{route}, `flag`, `dhops`, `precs`, `addpreRT`, `head`, `tail` and `drop` the proofs are straightforward, inspecting the locations of function calls; detailed proofs can be found in [16, Section 7.4].

Proposition 15 In our specification of AODV, the function `update` is used only when it is defined.

The functions `nhop` and $\sigma_{p\text{-}flag}$ need a closer inspection.

Proposition 16 In our specification of AODV, the function `nhop` is either used within formulas or if it is defined; hence it is only used in a meaningful way.

If one chooses to use lazy evaluation for conjunction, then `nhop` is only used where it is defined. Lastly, the function $\sigma_{p\text{-}flag}$ is called only in Pro. 1 in Line 33, within a formula. Again, if one uses lazy evaluation for conjunction, then $\sigma_{p\text{-}flag}$ is used only where it is defined.

6.5 The Quality of Routing Table Entries

In this section we define a total preorder $\sqsubseteq_{dip}$ on routing table entries for a given destination dip. Entries are ordered by the *quality* of the information they provide. This order will be defined in such a way that (a) the quality of a node's routing table entry for dip will only increase over time, and (b) the quality of valid routing table entries along a route to dip strictly increases every hop (at least prior to reaching dip). This order allows us to prove *loop freedom* of AODV in the next section.

A main ingredient in the definition of the quality preorder is the sequence number of a routing table entry. A higher sequence number denotes fresher information. However, it generally is not the case that along a route to dip found by AODV the sequence numbers are only increasing. This is since AODV increases the sequence number of an entry at an intermediate node when invalidating it. To "compensate" for that we introduce the concept of a *net sequence number*. It is defined by a function $\mathtt{nsqn} : \mathtt{R} \to \mathtt{SQN}$

$$\mathtt{nsqn}(r) := \begin{cases} \pi_2(r) & \text{if } \pi_4(r) = \mathtt{val} \vee \pi_2(r) = 0 \\ \pi_2(r) - 1 & \text{otherwise}. \end{cases}$$

For $n \in \mathbb{N}$ define $n \dot{-} 1 := \max(n-1, 0)$; hence $\texttt{inc}(n) \dot{-} 1 = n$. Then $\texttt{nsqn}(r) = \pi_2(r) \dot{-} 1$ if $\pi_4(r) = \texttt{inv}$.

To model increase in quality, we define $\sqsubseteq_{dip}$ by first comparing the net sequence numbers of two entries—a larger net sequence number denotes fresher and higher quality information. In case the net sequence numbers are equal, we decide on their hop counts—the entry with the least hop count is the best. This yields the following lexicographical order:

Assume two routing table entries $r, r' \in \texttt{R}$ with $\pi_1(r) = \pi_1(r') = dip$. Then

$$r \sqsubseteq_{dip} r' :\Leftrightarrow \texttt{nsqn}(r) < \texttt{nsqn}(r') \\ \vee \left(\texttt{nsqn}(r) = \texttt{nsqn}(r') \wedge \pi_5(r) \geq \pi_5(r')\right).$$

To reason about AODV, net sequence numbers and the quality preorder are lifted to routing tables. As for sqn we define a total function to determine net sequence numbers.

$$\texttt{nsqn} : \texttt{RT} \times \texttt{IP} \to \texttt{SQN}$$

$$\texttt{nsqn}(rt, dip) := \begin{cases} \texttt{nsqn}(\sigma_{route}(rt, dip)) \\ \quad \text{if } \sigma_{route}(rt, dip) \text{ is defined} \\ 0 \ \text{ otherwise}. \end{cases} \\ = \begin{cases} \texttt{sqn}(rt, dip) \\ \quad \text{if } \texttt{flag}(rt, dip) = \texttt{val} \\ \texttt{sqn}(rt, dip) \dot{-} 1 \ \text{ otherwise}. \end{cases}$$

If two routing tables rt and rt' have a routing table entry to dip, i.e., $dip \in \texttt{kD}(rt) \cap \texttt{kD}(rt')$, the preorder can be lifted as well.

$$rt \sqsubseteq_{dip} rt' :\Leftrightarrow \sigma_{route}(rt, dip) \sqsubseteq_{dip} \sigma_{route}(rt', dip) \\ \Leftrightarrow \texttt{nsqn}(rt, dip) < \texttt{nsqn}(rt', dip) \vee \\ \left(\texttt{nsqn}(rt, dip) = \texttt{nsqn}(rt', dip) \\ \wedge \texttt{dhops}(rt, dip) \geq \texttt{dhops}(rt', dip)\right)$$

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

As with sqn, we abbreviate nsqn: $\texttt{nsqn}^{ip}_N(dip) := \texttt{nsqn}(\xi^{ip}_N(\texttt{rt}), dip)$. Note that

$$\texttt{sqn}^{ip}_N(dip) \dot{-} 1 \leq \texttt{nsqn}^{ip}_N(dip) \leq \texttt{sqn}^{ip}_N(dip). \tag{16}$$

After setting up this notion of quality, we now show that routing tables, when modified by AODV, never decrease their quality.

Proposition 17 Assume a routing table $rt \in \texttt{RT}$ with $dip \in \texttt{kD}(rt)$.

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

$$rt \sqsubseteq_{dip} \texttt{update}(rt, r). \tag{17}$$

(b) An invalidate on rt does not change the quality of the routing table if, for each $(rip, rsn) \in dests$, 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 $\texttt{IP} \times \texttt{SQN}$)

$$\big((rip, rsn) \in dests \Rightarrow rip \in \texttt{vD}(rt) \\ \wedge rsn = \texttt{inc}(\texttt{sqn}(rt, rip))\big) \\ \Rightarrow rt \approx_{dip} \texttt{invalidate}(rt, dests). \tag{18}$$

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

$$rt \approx_{dip} \texttt{addpreRT}(rt, dip, npre). \tag{19}$$

We can apply this result to obtain the following theorem.

Theorem 18 In AODV, the quality of routing tables can only be increased, never decreased. Assume $N \xrightarrow{\ell} N'$ and $ip, dip \in \mathbf{IP}$. If $dip \in \texttt{kD}^{ip}_N$, then $dip \in \texttt{kD}^{ip}_{N'}$ and $\xi^{ip}_N(\texttt{rt}) \sqsubseteq_{dip} \xi^{ip}_{N'}(\texttt{rt})$.

Proof If $dip \in \texttt{kD}^{ip}_N$, then $dip \in \texttt{kD}^{ip}_{N'}$ follows by Proposition 4. To show $\xi^{ip}_N(\texttt{rt}) \sqsubseteq_{dip} \xi^{ip}_{N'}(\texttt{rt})$, by Remark 3 and Proposition 17(a) and (c) it suffices to check all calls of invalidate.

Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17: By construction of dests (immediately before the invalidation call)

$$(rip, rsn) \in \xi^{ip}_N(\texttt{dests}) \\ \Rightarrow rip \in \texttt{vD}(\xi^{ip}_N(\texttt{rt})) \wedge rsn = \texttt{inc}(\texttt{sqn}(\xi^{ip}_N(\texttt{rt}), rip))$$

and hence, by Proposition 17(b), $\xi^{ip}_N(\texttt{rt}) \approx_{dip} \texttt{invalidate}(\xi^{ip}_N(\texttt{rt}), \xi^{ip}_N(\texttt{dests})) = \xi^{ip}_{N'}(\texttt{rt})$.

Pro. 6, Line 3: Assume that invalidate modifies an entry having the form $(rip, dsn, *, flag, *, *, *)$. Let $(rip, rsn) \in dests$; then the update results in the entry $(rip, rsn, *, \texttt{inv}, *, *, *)$. Moreover, by Line 2 of Pro. 6, $flag$=val. By definition of net sequence numbers,

$$\texttt{nsqn}(\xi^{ip}_N(\texttt{rt}), rip) = \texttt{sqn}(\xi^{ip}_N(\texttt{rt}), rip) \\ \leq rsn \dot{-} 1 = \texttt{nsqn}(\xi^{ip}_{N'}(\texttt{rt}), rip).$$

The second step holds, since $\texttt{sqn}(\xi^{ip}_{N_2}(\texttt{rt}), rip) < rsn$, using Line 2. Since the hop count is not changed by invalidate, we also have $\texttt{dhops}(\xi^{ip}_N(\texttt{rt}), rip) = \texttt{dhops}(\xi^{ip}_{N'}(\texttt{rt}), rip)$, and hence $\xi^{ip}_N(\texttt{rt}) \sqsubseteq_{dip} \xi^{ip}_{N'}(\texttt{rt})$. ☐

Theorem 18 states in particular that if $N \xrightarrow{\ell} N'$ then $\mathtt{nsqn}^{ip}_{N}(dip) \leq \mathtt{nsqn}^{ip}_{N'}(dip)$.

Proposition 19 If, in a reachable network expression N, a node $ip \in \mathbf{IP}$ has a 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.

$$\begin{aligned} & dip \in \mathtt{kD}^{ip}_{N} \wedge nhip \neq dip \\ \Rightarrow\ & dip \in \mathtt{kD}^{nhip}_{N} \wedge \mathtt{nsqn}^{ip}_{N}(dip) \leq \mathtt{nsqn}^{nhip}_{N}(dip)\,, \end{aligned} \tag{20}$$

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

To prove loop freedom we will show that on any route established by AODV the quality of routing tables 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 20 *The relation $\sqsubset_{dip}$ is irreflexive and transitive.*

Theorem 21 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*.

$$\begin{aligned} & dip \in \mathtt{vD}^{ip}_{N} \cap \mathtt{vD}^{nhip}_{N} \wedge nhip \neq dip \\ \Rightarrow\ & \xi^{ip}_{N}(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N}(\mathtt{rt})\,, \end{aligned} \tag{21}$$

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

Proof As before, we first check the initial states of our transition system and then check all locations in Processes 1–7 where a routing table might be changed. For an initial network expression, the invariant holds since all routing tables are empty. Adding precursors to $\xi^{ip}_{N}(\mathtt{rt})$ or $\xi^{nhip}_{N}(\mathtt{rt})$ does not affect the invariant, since the invariant does not depend on precursors, so it suffices to examine all modifications to $\xi^{ip}_{N}(\mathtt{rt})$ or $\xi^{nhip}_{N}(\mathtt{rt})$ using `update` or `invalidate`. Moreover, without loss of generality we restrict attention to those applications of `update` or `invalidate` that actually modify the entry for *dip*, beyond its precursors; if `update` only adds some precursors in the routing table, the invariant—which is assumed to hold before—is maintained.

Applications of `invalidate` to $\xi^{ip}_{N}(\mathtt{rt})$ or $\xi^{nhip}_{N}(\mathtt{rt})$ lead to a network state in which the antecedent of (21) is not satisfied. Now consider an application of `update` to $\xi^{nhip}_{N}(\mathtt{rt})$. We restrict attention to the case that the antecedent of (21) is satisfied right after the update, so that right before the update we have $dip \in \mathtt{vD}^{ip}_{N} \wedge nhip \neq dip$. In the special case that $\mathtt{sqn}^{nhip}_{N}(dip) = 0$ right before the update, we have $\mathtt{nsqn}^{nhip}_{N}(dip) = 0$ and thus $\mathtt{nsqn}^{ip}_{N}(dip) = 0$ by Invariant (20). Considering that $\mathtt{flag}^{ip}_{N}(dip) = \mathtt{val}$, this implies $\mathtt{sqn}^{ip}_{N}(dip) = 0$. By Proposition 10(d) we have $nhip = dip$, contradicting our assumptions. It follows that right before the update $\mathtt{sqn}^{nhip}_{N}(dip) > 0$; so in particular $dip \in \mathtt{kD}^{nhip}_{N}$.

An application of `update` to $\xi^{nhip}_{N}(\mathtt{rt})$ that changes $\mathtt{flag}^{nhip}_{N}(dip)$ from `inv` to `val` cannot decrease the sequence number of the entry to *dip* and hence strictly increases its net sequence number. Before the `update` we had $\mathtt{nsqn}^{ip}_{N}(dip) \leq \mathtt{nsqn}^{nhip}_{N}(dip)$ by Invariant (20), so afterwards we must have $\mathtt{nsqn}^{ip}_{N}(dip) < \mathtt{nsqn}^{nhip}_{N}(dip)$, and therefore $\xi^{ip}_{N}(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N}(\mathtt{rt})$. An `update` to $\xi^{nhip}_{N}(\mathtt{rt})$ that maintains $\mathtt{flag}^{nhip}_{N}(dip) = \mathtt{val}$ can only increase the quality of the entry to *dip* (cf. Theorem 18), and hence maintains Invariant (21).

It remains to examine the `updates` to $\xi^{ip}_{N}(\mathtt{rt})$.

Pro. 1, Lines 10, 14, 18: The entry $\xi(\mathtt{sip}, 0, \mathtt{unk}, \mathtt{val}, 1, \mathtt{sip}, \emptyset)$ is used for the update; its destination is $dip := \xi(\mathtt{sip})$. Since $dip = \mathtt{nhop}^{ip}_{N}(dip) = nhip$, the antecedent of the invariant to be proven is not satisfied.

Pro. 4, Line 4: We assume that the entry $\xi(\mathtt{oip}, \mathtt{osn}, \mathtt{kno}, \mathtt{val}, \mathtt{hops}+1, \mathtt{sip}, *)$ is inserted into $\xi(\mathtt{rt})$. So $dip := \xi(\mathtt{oip})$, $nhip := \xi(\mathtt{sip})$, $\mathtt{nsqn}^{ip}_{N}(dip) := \xi(\mathtt{osn})$ and $\mathtt{dhops}^{ip}_{N}(dip) := \xi(\mathtt{hops}) + 1$. This information is distilled from a received route request message (cf. Lines 1 and 8 of Pro. 1). By Proposition 1 this message was sent before, say in state $N^{\dagger}$; by Proposition 7 the sender of this message is $\xi(\mathtt{sip})$.
By Invariant (13), with $ip_c := \xi(\mathtt{sip}) = nhip$, $oip_c := \xi(\mathtt{oip}) = dip$, $osn_c := \xi(\mathtt{osn})$ and $hops_c := \xi(\mathtt{hops})$, and using that $ip_c = nhip \neq dip = oip_c$, we get that

$$\begin{aligned} & \mathtt{sqn}^{nhip}_{N^{\dagger}}(dip) = \mathtt{sqn}^{ip_c}_{N^{\dagger}}(oip_c) > osn_c \;=\; \xi(\mathtt{osn})\,, \text{ or} \\ & \mathtt{sqn}^{nhip}_{N^{\dagger}}(dip) = \xi(\mathtt{osn}) \wedge \mathtt{dhops}^{nhip}_{N^{\dagger}}(dip) \leq \xi(\mathtt{hops}) \\ & \qquad\qquad \wedge\, \mathtt{flag}^{nhip}_{N^{\dagger}}(dip) \;=\; \mathtt{val}\,. \end{aligned}$$

We first assume that the first line holds. Then, by the assumption $dip \in \mathtt{vD}(\xi^{nhip}_{N}(\mathtt{rt}))$, the definition of net sequence numbers, and Proposition 5,

$$\begin{aligned} \mathtt{nsqn}^{nhip}_{N}(dip) &= \mathtt{sqn}^{nhip}_{N}(dip) \geq \mathtt{sqn}^{nhip}_{N^{\dagger}}(dip) \\ &> \xi(\mathtt{osn}) = \mathtt{nsqn}^{ip}_{N}(dip)\,. \end{aligned}$$

and hence $\xi^{ip}_{N}(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N}(\mathtt{rt})$.
We now assume the second line to be valid. From this we conclude

$$\begin{aligned} \mathtt{nsqn}^{nhip}_{N^{\dagger}}(dip) &= \mathtt{sqn}^{nhip}_{N^{\dagger}}(dip) = \xi(\mathtt{osn}) \\ &= \mathtt{nsqn}^{ip}_{N}(dip)\,. \end{aligned}$$

Moreover, $\mathtt{dhops}^{nhip}_{N^\dagger}(dip) \leq \xi(\mathtt{hops}) < \xi(\mathtt{hops}) + 1 = \mathtt{dhops}^{ip}_{N}(dip)$. Hence $\xi^{ip}_N(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N^\dagger}(\mathtt{rt})$. Together with Theorem 18 and the transitivity of $\sqsubseteq_{dip}$ this yields $\xi^{ip}_N(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_N(\mathtt{rt})$.

Pro. 5, Line 2: This update is similar to the one of Pro. 4, Line 4. The only difference is that the information stems from an incoming RREP message and that a routing table entry to $\xi(\mathtt{dip})$ (instead of $\xi(\mathtt{oip})$) is established. Therefore, the proof is similar to the one of Pro. 4, Line 4; instead of Invariant (13) we use Invariant (14). □

6.6 Loop Freedom

The "naïve" notion of loop freedom is a term that informally means that "a packet never goes round in cycles without (at some point) being delivered". This dynamic definition is not only hard to formalise, it is also too restrictive a requirement for AODV. There are situations where packets are sent in cycles, but which are not considered harmful. This can happen when the topology keeps changing. We refer to [16, Sect. 7.6] for an example.

Due to this dynamic behaviour, the sense of loop freedom is much better captured by a static invariant, saying that at any given time the collective routing tables of the nodes do not admit a loop. Such a requirement does not rule out the dynamic loop alluded to above. However, in situations where the topology remains stable sufficiently long it does guarantee that packets will not keep going around in cycles.

To this end we define the *routing graph* of a network expression N with respect to destination dip by $\mathcal{R}_N(dip) := (\mathbf{IP}, E)$, where all nodes of the network form the set of vertices and there is an arc $(ip, ip') \in E$ iff $ip \neq dip$ and $(dip, *, *, \mathtt{val}, *, ip', *) \in \xi^{ip}_N(\mathtt{rt})$.

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. 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.

Using this definition of a routing graph, Theorem 21 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 22 The specification of AODV given in Section 5 is loop free.

Proof If there were a loop in a routing graph $\mathcal{R}_N(dip)$, then for any edge $(ip, nhip)$ on that loop one has, by Theorem 21, $\xi^{ip}_N(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_N(\mathtt{rt})$. Thus, by transitivity of $\sqsubset_{dip}$, one has $\xi^{ip}_N(\mathtt{rt}) \sqsubset_{dip} \xi^{ip}_N(\mathtt{rt})$, which contradicts the irreflexivity of $\sqsubset_{dip}$ (cf. Corollary 20). □

According to Theorem 22 any route to a destination dip established by AODV—i.e. a path in $\mathcal{R}_N(dip)$—ends after finitely many hops. There are three possible ways in which it could end:

(1) by reaching the destination,
(2) by reaching a node with an invalid entry to dip, or
(3) by reaching a node without any entry to dip.

(1) is what AODV attempts to accomplish, whereas (2) is an unavoidable due to link breaks in a dynamic topology. It follows directly from Proposition 19 that (3) can never occur.

6.7 Route Correctness

The creation of a routing table entry at node ip for destination dip is no guarantee that a route from ip to dip actually exists. The entry is created based on information gathered from messages received in the past, and at any time link breaks may occur. The best one could require of a protocol like AODV is that routing table entries are based on information that was valid at some point in the past. This is the essence of what we call *route correctness*.

We define a *history* of an AODV-like protocol as a sequence $H = N_0 N_1 \dots N_k$ of network expressions, where N_0 is an initial state of the protocol, and for $1 \leq i \leq k$ there is a transition $N_{i-1} \xrightarrow{\ell} N_i$; we call H a history *of* the state N_k. The *connectivity graph* of a history H is $\mathcal{C}_H := (\mathbf{IP}, E)$, where the nodes of the network form the set of vertices and there is an arc $(ip, ip') \in E$ iff $ip' \in R^{ip}_{N_i}$ for some $0 \leq i \leq k$, i.e. if at some point during that history node ip' was in transmission range of ip. A protocol satisfies the property *route correctness* if for every history H of a reachable state N and for every routing table entry $(dip,*,*,*,hops,nhip,*) \in \xi^{ip}_N(\mathtt{rt})$ there is a path $ip \to nhip \to \cdots \to dip$ in $\mathcal{C}_H$ from ip to dip with $hops$ hops and (if $hops > 0$) next hop $nhip$.[24]

Theorem 23 Let H be a history of a network state N.

(a) For each entry $(dip, *, *, *, hops, nhip, *) \in \xi^{ip}_N(\mathtt{rt})$ there is a path $ip \to nhip \to \cdots \to dip$ in $\mathcal{C}_H$ from ip to dip with $hops$ hops and (if $hops > 0$) next hop $nhip$.

[24] A path with 0 hops consists of a single node only.

(b) For each route request sent in state N there is a corresponding path in the connectivity graph of H.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rreq}(hops_c,*,*,*,*,oip_c,*,ip_c))}_{ip} N' \Rightarrow \text{there is a path } ip_c \to \cdots \to oip_c \text{ in } \mathcal{C}_H \text{ from } ip_c \text{ to } oip_c \text{ with } hops_c \text{ hops} \quad (22)$$

(c) For each route reply sent in state N there is a corresponding path in the connectivity graph of H.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rrep}(hops_c,dip_c,*,*,ip_c))}_{ip} N' \Rightarrow \text{there is a path } ip_c \to \cdots \to dip_c \text{ in } \mathcal{C}_H \text{ from } ip_c \text{ to } dip_c \text{ with } hops_c \text{ hops} \quad (23)$$

Theorem 23(a) says that the AODV protocol is route correct. For the proof it is essential that we use the version of AWN where a node ip' is in the range of node ip, meaning that ip' can receive messages sent by ip, if and only if ip is in the range of ip'. If AWN is modified so as to allow asymmetric connectivity graphs, as contemplated in [15,16], it is trivial to construct a 2-node counterexample to route correctness.

A stronger concept of route correctness could require that for every history H of a state N and for each $(dip,*,*,*,hops,nhip,*) \in \xi_N^{ip}(\texttt{rt})$

- either $hops = 0$ and $dip = ip$,
- or $hops = 1$ and $dip = nhip$ and there is a $N^\dagger$ in H such that $nhip \in R_{N^\dagger}^{ip}$,
- or $hops > 1$ and there is a $N^\dagger$ in H with $nhip \in R_{N^\dagger}^{ip}$ and $(dip,*,*,\texttt{val},hops{-}1,*,*) \in \xi_{N^\dagger}^{nhip}(\texttt{rt})$.

It turns out that this stronger form of route correctness does not hold for AODV. It can be violated when a node forwards a route request without updating its own (fresher) routing table entry for the originator of the route request.

7 Related Work

Several process algebras modelling broadcast communication have been proposed before: the Calculus of Broadcasting Systems (CBS) [45], the $b\pi$-calculus [13], CBS$^\#$[36], the Calculus of Wireless Systems (CWS) [33], the Calculus of Mobile Ad Hoc Networks (CMAN) [21], the Calculus for Mobile Ad Hoc Networks (CMN) [32], the ω-calculus [49], restricted branching process theory (RBPT) [19], $bA\pi$ [22] and the broadcast psi-calculi [5]. The latter eight of these were specifically designed to model MANETs. However, none of these process calculi provides all features needed to fully model routing protocols such as AODV, namely data handling, (conditional) unicast and (local) broadcast. For example, all above-mentioned process algebras lack the feature of guaranteed receipt of messages by destinations within transmission range. Due to this, it is not possible to analyse properties such as route discovery and packet delivery [16]. A more detailed discussion of these process algebras can be found in [16].

Our complete formalisation of AODV has grown from elaborating a partial and simplified formalisation of AODV in [49, Fig. 8]. The features of our process algebra were largely determined by what we needed to enable a complete and accurate formalisation of this protocol. The same formalism has been used to model the Dynamic MANET On-demand (DYMO) Routing Protocol (also known as AODVv2) [12]. We conjecture that AWN is also applicable to a wide range of other wireless protocols, such as the Dynamic Source Routing (DSR) protocol [29], the Lightweight Underlay Network Ad-hoc Routing (LUNAR) protocol [51,52], the Optimized Link State Routing (OSLR) protocol [10] or the Better Approach To Mobile Adhoc Networking (B.A.T.M.A.N.) [37]. The specification and the correctness of the latter three, however, rely heavily on timing aspects; hence an AWN-extension with time [8] appears necessary (see also Section 8).

While process algebras such as AWN can be used to formally model and verify the correctness of network routing protocols, test-bed experiments and simulations are complementary tools that can be used to quantitatively evaluate the performance of the protocols. While test-bed experiments are able to capture the full complex characteristics of the wireless medium and its effect on the network routing protocols [30,44], network simulators [38,48] offer the ease and flexibility of evaluating and comparing the performance of different routing protocols in a large-scale network of hundreds of nodes, coupled with the added advantage of being able to repeat and reproduce the experiments [11,40,28].

Loop freedom is a crucial property of network protocols, commonly claimed to hold for AODV [39]. Merlin and Segall [31] were amongst the first to use sequence numbers to guarantee loop freedom of a routing protocol. In a companion paper [20] we have shown that several *interpretations* of AODV—consistent ways to revolve the ambiguities in the RFC—fail to be loop free, while in [16] we establish loop freedom of others by adaptation of the proof presented here.

A preliminary draft of AODV has been shown to be not loop free by Bhargavan et al. in [3]. Their counterexamples to loop freedom have to do with timing issues: the premature deletion of invalid routes, and a too quick restart of a node after a reboot. Since then, AODV has changed to such a degree that these examples do not apply to the current version [39]. However, similar examples, claimed to apply to the current version, are reported in [18,47]; we discuss them in [8].

All these papers propose repairs that avoid these loops through better timing policies. In contrast, the routing loops documented in [20] are time-independent.

Previous attempts to prove loop freedom of AODV have been reported in [42,3,55], but none of these proofs are complete and valid for the current version of AODV [39]:

- The proof sketch given in [42] uses the fact that when a loop in a route to a destination Z is created, all nodes X_i on that loop must have route entries for destination Z with the same destination sequence number. "*Furthermore, because the destination sequence numbers are all the same, the next hop information must have been derived at every node X_i from the same RREP transmitted by the destination Z*" [42, Page 11]. The latter is not true at all: some of the information could have been derived from RREQ messages, or from a RREP message transmitted by an intermediate node that has a route to Z. More importantly, the nodes on the loop may have acquired their information on a route to Z from different RREP or RREQ messages, that all carried the same sequence number. This is illustrated by the routing loop created in [20, Figure 1].
- Based on an analysis of an early draft of AODV[25] [3] suggests three improvements. The modified version is then proved to be loop free, using the following invariant (written in our notation):

  $$\begin{array}{l}\text{if } \mathit{nhip} = \mathtt{nhop}_N^{ip}(\mathit{dip}), \text{ then}\\ (1)\ \mathtt{sqn}_N^{ip}(\mathit{dip}) \leq \mathtt{sqn}_N^{\mathit{nhip}}(\mathit{dip}), \text{ and}\\ (2)\ \mathtt{sqn}_N^{ip}(\mathit{dip}) = \mathtt{sqn}_N^{\mathit{nhip}}(\mathit{dip})\\ \quad \Rightarrow \mathtt{dhops}_N^{ip}(\mathit{dip}) < \mathtt{dhops}_N^{\mathit{nhip}}(\mathit{dip})\,.\end{array}$$

  This invariant does not hold for this modified version of AODV, nor for the current version, documented in the RFC. It can happen that in a state N where $\mathtt{sqn}_N^{ip}(\mathit{dip}) = \mathtt{sqn}_N^{\mathit{nhip}}(\mathit{dip})$, node *ip* notices that the link to *nhip* is broken. Consequently, *ip* invalidates its route to *dip*, which has *nhip* as its next hop. According to recommendation (**A1**) of [3, Page 561]), node *ip* increments its sequence number for the (invalid) route to *dip*, resulting in a state N' for which $\mathtt{sqn}_{N'}^{ip}(\mathit{dip}) > \mathtt{sqn}_{N'}^{\mathit{nhip}}(\mathit{dip})$, thereby violating the invariant.

  Note that the invariant of [3] does not restrict itself to the case that the routing table entry for *dip* maintained by *ip* is *valid*. Adapting the invariant with such a requirement would give rise to a valid invariant, but one whose verification poses problems, at least for the current version of AODV. These problems led us, in this paper, to use *net sequence numbers* instead (cf. Section 6.5).

  Recommendation (**A1**) is assumed to be in effect for the (improved) version of AODV analysed in [3], although it was not in effect for the draft of AODV existing at the time. Since then, recommendation (**A1**) has been incorporated in the RFC. Looking at the proofs in [3], it turns out that Lemma 20(1) of [3] is invalid. This failure is surprising, given that according to [3] Lemma 20 is automatically verified by SPIN. A possible explanation might be that this lemma *is* obviously valid for the version of AODV prior to the recommendations of [3].
- Zhou, Yang, Zhang, and Wang [55] establish loop freedom of AODV using an adaptation of the invariant from [3] with a validity requirement. However, they do not model route replies by intermediate nodes. This is a core feature of AODV, and a potential source of routing loops [20].

[25] Draft version 2 is analysed, dated November 1998; the RFC can be seen as version 14, dated July 2001.

8 Conclusion and Future Work

In this paper, we have presented a complete and accurate model of the core functionality of the Ad hoc On-Demand Distance Vector routing protocol, a widely used protocol of practical relevance, using the process algebra AWN. We currently do not model optional features such as local route repair, expanding ring search, gratuitous route reply and multicast. We also abstract from all timing issues. In addition to modelling the complete set of core functionalities of the AODV protocol, our model also covers the interface to higher protocol layers via the injection and delivery of application layer data, as well as the forwarding of data packets at intermediate nodes. Although this is not part of the AODV protocol specification, it is necessary for a practical model of any reactive routing protocol, where protocol activity is triggered via the sending and forwarding of data packets. The completeness of our model is in contrast to some prior related work, which either modelled only very simple protocols, or modelled only a subset of the functionality of relevant WMN or MANET routing protocols.

The used modelling language AWN is tailored for WMNs and MANETs and hence covers major aspects of WMN routing protocols, for example the crucial aspect of data handling, such as maintaining routing table information. AWN allows not only the creation of accurate and concise models of relatively complex and practically relevant protocols, but also supports readability.

The currently predominant practice of informally specifying WMN and MANET protocols via English prose has a potential for ambiguity and inconsistent interpretation. The ability to provide a formal and unambiguous specification of such protocols via AWN is a significant benefit in its own right. Through a careful analysis of AODV, in particular with respect to the loop-freedom property, we have demonstrated how AWN can be used as a basis for reasoning about critical protocol correctness properties. By establishing invariants that remain valid in a network running AODV, we have shown that our model is loop free. In contrast to protocol evaluation using simulation, test-bed experiments or model checking, where only a finite number of specific network scenarios can be considered, our reasoning with AWN is generic and the proofs hold for any possible network scenario in terms of topology and traffic pattern. None of the experimental protocol evaluation approaches can deliver this high degree of assurance about protocol behaviour. As a "side product" we have also shown that, in contrast to common belief, sequence numbers do not guarantee loop freedom, even if they are increased monotonically over time and incremented whenever a new route request is generated; this result is presented elsewhere [20].

During creation of our model of AODV we uncovered several ambiguities in the AODV RFC [39]. In [16] we have analysed *all* interpretations of the RFC that stem from the ambiguities revealed. It turned out that several interpretations can yield unwanted behaviour such as routing loops. We also found that implementations of AODV behave differently in crucial aspects of protocol behaviour, although they all follow the lines of the RFC. Of course a specification "*needs to be reasonably implementation independent*"[26] and can leave some decisions to the software engineer; however it is our belief that any specification should be clear and unambiguous enough to guarantee the same behaviour when given to different developers. As demonstrated, this is not the case for AODV, and likely not for many other RFCs provided by the IETF.

To increase the level of confidence of our analysis even further, we mechanised AWN as well as the presented pen-and-paper proof of loop freedom of AODV in the interactive theorem prover Isabelle/HOL. [7,6] When verifying our (pen-and-paper) proof we did not find any major errors: (1) type checking found a minor typo in the model, (2) one proof invoked an incorrect invariant requiring the addition and proof of a new invariant based on an existing one, (3) a minor flaw in another proof required the addition of a new invariant. All these "flaws" have been repaired in the present proof.

There are two directions of future work with regards to AODV: (a) A further analysis of AODV will require an extension of AWN with time and probability: the former to cover aspects such as AODV's handling (deletion) of stale routing table entries and the latter to model the probability associated with lossy links. The loop freedom result presented here is based on a model in which routing table entries never expire. Hence it does not rule out AODV routing loops due to premature deletion of routing table entries. We expect that the resulting algebra will be also applicable to a wide range of other wireless protocols. (b) Since AODV was designed without security features in mind, it is vulnerable to malicious attacks such as routing attacks and forwarding attacks [53]. It may be worthwhile to formally prove that extensions of AODV such as SAODV [24] actually protect the route discovery mechanism by providing security features like integrity and authentication.

Next to this on-going work, we also aim to complement AWN by model checking.[27] Having the ability of automatically deriving a model for model checkers such as Uppaal from an AWN specification allows the confirmation and detailed diagnostics of suspected errors in an early phase of protocol development. Model checking is limited to networks of small size—due to state space explosion— whereas our analysis covers all (static and dynamic) topologies. However, finding shortcomings in some topologies is useful to identify problematic behaviour. These shortcomings can be eliminated, even before a more thorough and general analysis using AWN.

Acknowledgements We thank Annabelle McIver and Ansgar Fehnker for fruitful discussions. Further we thank the anonymous referees for the careful evaluation of this paper and their feedback.

References

1. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theoretical Computer Science **37**(1), 77–121 (1985)
2. Bhargavan, K., Gunter, C.A., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., Viswanathan, M.: Verisim: Formal analysis of network simulations. IEEE Transactions on Software Engineering **28**(2), 129–145 (2002). doi:10.1109/32.988495
3. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. Journal of the ACM **49**(4), 538–576 (2002). doi:10.1145/581771.581775

[26] `http://www.ietf.org/iesg/statement/pseudocode-guidelines.html`

[27] See [14] for the first work in that direction.

4. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks **14**, 25–59 (1987). doi:10.1016/0169-7552(87)90085-7
5. Borgström, J., Huang, S., Johansson, M., Raabjerg, P., Victor, B., Pohjola, J.Å., Parrow, J.: Broadcast psi-calculi with an application to wireless protocols. In: G. Barthe, A. Pardo, G. Schneider (eds.) Software Engineering and Formal Methods (SEFM'11), *Lecture Notes in Computer Science*, vol. 7041, pp. 74–89. Springer (2011). doi:10.1007/978-3-642-24690-6_7
6. Bourke, T., van Glabbeek, R.J., Höfner, P.: A mechanized proof of loop freedom of the (untimed) AODV routing protocol. In: F. Cassez, J.F. Raskin (eds.) Automated Technology for Verification and Analysis (ATVA'14), *Lecture Notes in Computer Science*, vol. 8837, pp. 47–63. Springer (2014). doi:10.1007/978-3-319-11936-6_5
7. Bourke, T., van Glabbeek, R.J., Höfner, P.: Showing invariance compositionally for a process algebra for network protocols. In: G. Klein, R. Gamboa (eds.) Interactive Theorem Proving (ITP'14), *Lecture Notes in Computer Science*, vol. 8558, pp. 144–159. Springer (2014). doi:10.1007/978-3-319-08970-6_10
8. Bres, E., van Glabbeek, R.J., Höfner, P.: T-AWN: A timed process alebra for wireless networks. To appear (2016)
9. Chiyangwa, S., Kwiatkowska, M.: A timing analysis of AODV. In: Formal Methods for Open Object-based Distributed Systems (FMOODS'05), *Lecture Notes in Computer Science*, vol. 3535, pp. 306–322. Springer (2005). doi:10.1007/11494881_20
10. Clausen, T., Jacquet, P.: Optimized link state routing protocol (OLSR). RFC 3626 (Experimental), Network Working Group (2003). URL `http://www.ietf.org/rfc/rfc3626.txt`
11. Das, S.R., Castañeda, R., Yan, J.: Simulation-based performance evaluation of routing protocols for mobile ad hoc networks. Mobile Networks and Applications **5**(3), 179–189 (2000). doi:10.1023/A:1019108612308
12. Edenhofer, S., Höfner, P.: Towards a rigorous analysis of AODVv2 (DYMO). In: Rigorous Protocol Engineering (WRiPE '12). IEEE (2012). doi:10.1109/ICNP.2012.6459942
13. Ene, C., Muntean, T.: A broadcast-based calculus for communicating systems. In: Parallel & Distributed Processing Symposium (IPDPS '01), pp. 1516–1525. IEEE Computer Society (2001). doi:10.1109/IPDPS.2001.925136
14. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: Automated analysis of AODV using UPPAAL. In: C. Flanagan, B. König (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS '12), *Lecture Notes in Computer Science*, vol. 7214, pp. 173–187. Springer (2012). doi:10.1007/978-3-642-28756-5_13
15. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: H. Seidl (ed.) European Symposium on Programming (ESOP '12), *Lecture Notes in Computer Science*, vol. 7211, pp. 295–315. Springer (2012). doi:10.1007/978-3-642-28869-2_15
16. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Technical Report 5513, NICTA (2013). URL `http://arxiv.org/abs/1312.7645`
17. Garcia-Luna-Aceves, J.J.: A unified approach to loop-free routing using distance vectors or link states. In: Symposium Proceedings on Communications, Architectures & Protocols (SIGCOMM '89), *ACM SIGCOMM Computer Communication Review*, vol. 19(4), pp. 212–223. ACM Press (1989). doi:10.1145/75246.75268
18. Garcia-Luna-Aceves, J.J., Rangarajan, H.: A new framework for loop-free on-demand routing using destination sequence numbers. In: Mobile Ad-hoc and Sensor Systems (MASS' 04), pp. 426–435. IEEE (2004). doi:10.1109/MAHSS.2004.1392182
19. Ghassemi, F., Fokkink, W., Movaghar, A.: Restricted broadcast process theory. In: A. Cerone, S. Gruner (eds.) Software Engineering and Formal Methods (SEFM '08), pp. 345–354. IEEE Computer Society (2008). doi:10.1109/SEFM.2008.25
20. van Glabbeek, R.J., Höfner, P., Tan, W.L., Portmann, M.: Sequence numbers do not guarantee loop freedom —AODV can yield routing loops—. In: Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '13), pp. 91–100. ACM Press (2013). doi:10.1145/2507924.2507943
21. Godskesen, J.C.: A calculus for mobile ad hoc networks. In: A.L. Murphy, J. Vitek (eds.) Coordination Models and Languages (COORDINATION '07), *Lecture Notes in Computer Science*, vol. 4467, pp. 132–150. Springer (2007). doi:10.1007/978-3-540-72794-1_8
22. Godskesen, J.C.: Observables for mobile and wireless broadcasting systems. In: D. Clarke, G.A. Agha (eds.) Coordination Models and Languages (COORDINATION '10), *Lecture Notes in Computer Science*, vol. 6116, pp. 1–15. Springer (2010). doi:10.1007/978-3-642-13414-2_1
23. Griffin, T.G., Sobrinho, J.: Metarouting. SIGCOMM Computer Communication Review **35**(4), 1–12 (2005). doi:10.1145/1090191.1080094
24. Guerrero-Zapata, M., Asokan, N.: Securing Ad Hoc Routing Protocols. In: Proceedings of the 2002 ACM Workshop on Wireless Security (WiSe 2002), pp. 1–10. ACM Press (2002). doi:10.1145/570681.570682
25. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
26. Höfner, P., van Glabbeek, R.J., Tan, W.L., Portmann, M., McIver, A.K., Fehnker, A.: A rigorous analysis of AODV and its variants. In: Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '12), pp. 203–212. ACM Press (2012). doi:10.1145/2387238.2387274
27. IEEE: 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 (2011). URL `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6018236`
28. Jacquet, P., Laouiti, A., Minet, P., Viennot, L.: Performance of multipoint relaying in ad hoc mobile routing protocols. In: E. Gregori, M. Conti, A.T. Campbell, G. Omidyar, M. Zukerman (eds.) Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications (NETWORKING '02), Lecture Notes in Computer Science, pp. 387–398. Springer (2002). doi:10.1007/3-540-47906-6_31
29. Johnson, D., Hu, Y., Maltz, D.: The dynamic source routing protocol (DSR) for mobile ad hoc networks for IPv4. RFC 4728 (Experimental), Network Working Group (Errata Exist) (2007). URL `http://www.ietf.org/rfc/rfc4728.txt`

30. Maltz, D., Broch, J., Johnson, D.B.: Lessons from a full-scale multihop wireless ad hoc network testbed. IEEE Personal Communications **8**(1), 8–15 (2001). doi:10.1109/98.904894
31. Merlin, P.M., Segall, A.: A failsafe distributed routing protocol. IEEE Transactions on Communications **27**(9), 1280–1287 (1979). doi:10.1109/TCOM.1979.1094552
32. Merro, M.: An observational theory for mobile ad hoc networks (full version). Information and Computation **207**(2), 194–208 (2009). doi:10.1016/j.ic.2007.11.010
33. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electronic Notes in Theoretical Computer Science **158**, 331–353 (2006). doi:10.1016/j.entcs.2006.04.017
34. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
35. Miskovic, S., Knightly, E.W.: Routing primitives for wireless mesh networks: Design, analysis and experiments. In: Conference on Information Communications (INFOCOM '10), pp. 2793–2801. IEEE (2010). doi:10.1109/INFCOM.2010.5462111
36. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theoretical Computer Science **367**, 203–227 (2006). doi:10.1016/j.tcs.2006.08.036
37. Neumann, A., Aichele C. Lindner, M., Wunderlich, S.: Better approach to mobile ad-hoc networking (B.A.T.M.A.N.). Internet-Draft (Experimental), Network Working Group (2008). URL `http://tools.ietf.org/html/draft-openmesh-b-a-t-m-a-n-00`
38. The network simulator ns-2. URL `http://nsnam.isi.edu/nsnam/index.php/Main_Page`
39. Perkins, C.E., Belding-Royer, E.M., Das, S.: Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), Network Working Group (2003). URL `http://www.ietf.org/rfc/rfc3561.txt`
40. Perkins, C.E., Belding-Royer, E.M., Das, S.R., Marina, M.K.: Performance comparison of two on-demand routing protocols for ad hoc networks. IEEE Personal Communications **8**(1), 16–28 (2001). doi:10.1109/98.904895
41. Perkins, C.E., Ratliff, S., Dowdell, J.: Dynamic MANET on-demand (AODVv2) routing. Internet Draft (Standards Track), Mobile Ad hoc Networks Working Group (2013). URL `http://tools.ietf.org/html/draft-ietf-manet-aodvv2-02`
42. Perkins, C.E., Royer, E.M.: Ad-hoc On-Demand Distance Vector Routing. In: Mobile Computing Systems and Applications (WMCSA '99), pp. 90–100. IEEE (1999). doi:10.1109/MCSA.1999.749281
43. Pirzada, A.A., Portmann, M., Indulska, J.: Performance analysis of multi-radio AODV in hybrid wireless mesh networks. Computer Communications **31**(5), 885–895 (2008). doi:10.1016/j.comcom.2007.12.012
44. Pirzada, A.A., Portmann, M., Wishart, R., Indulska, J.: SafeMesh: a wireless mesh network routing protocol for incident area communications. Pervasive and Mobile Computing **5**(2), 201–221 (2009). doi:10.1016/j.pmcj.2008.11.005
45. Prasad, K.V.S.: A calculus of broadcasting systems. Science of Computer Programming **25**(2-3), 285–327 (1995). doi:10.1016/0167-6423(95)00017-8
46. Ramachandran, K., Buddhikot, M.M., Chandranmenon, G., Miller, S., Belding-Royer, E.M., Almeroth, K.: On the design and implementation of infrastructure mesh networks. In: IEEE Workshop on Wireless Mesh Networks (WiMesh'05). IEEE (2005)
47. Rangarajan, H., Garcia-Luna-Aceves, J.J.: Making on-demand routing protocols based on destination sequence numbers robust. In: Communications (ICC '05), vol. 5, pp. 3068–3072 (2005). doi:10.1109/ICC.2005.1494958
48. SCALABLE Network Technologies: QualNet communications simulation platform. URL `http://web.scalable-networks.com/content/qualnet`
49. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Science of Computer Programming **75**, 440–469 (2010). doi:10.1016/j.scico.2009.07.008
50. Subramanian, A.P., Buddhikot, M.M., Miller, S.: Interference aware routing in multi-radio wireless mesh networks. In: IEEE Workshop on Wireless Mesh Networks (WiMesh '06). IEEE (2006)
51. Tschudin, C.F.: Lightweight underlay network ad hoc routing (LUNAR) protocol. Internet Draft (Expired), Mobile Ad Hoc Networking Working Group (2004). URL `http://user.it.uu.se/~rmg/pub/draft-tschudin-manet-lunar-00.txt`
52. Tschudin, C.F., Gold, R., Rensfelt, O., Wibling, O.: LUNAR: a lightweight underlay network ad-hoc routing protocol and implementation. In: Y. Koucheryavy, J. Harju, A. Koucheryavy (eds.) Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN '04) (2004)
53. Yang, H., Luo, H., Ye, F., Lu, S., Zhang, L.: Security in Mobile Ad Hoc Networks: Challenges and Solutions. IEEE Wireless Communications, **11**(1), 38–47 (2004). doi:10.1109/MWC.2004.1269716
54. Zave, P.: Using lightweight modeling to understand Chord. SIGCOMM Comput. Commun. Rev. **42**(2), 49–57 (2012). doi:10.1145/2185376.2185383
55. Zhou, M., Yang, H., Zhang, X., Wang, J.: The proof of AODV loop freedom. In: Wireless Communications & Signal Processing (WCSP '09). IEEE (2009). doi:10.1109/WCSP.2009.5371479

A Omitted Proofs

Proof of Proposition 4 None of the functions used to change routing tables removes an entry altogether. ☐

Proof of Proposition 5 The only function that can decrease a destination sequence number is `invalidate`. When invalidating routing table entries using the function `invalidate(rt, dests)`, sequence numbers are copied from `dests` to the corresponding entry in `rt`. It is sufficient to show that for all $(rip, rsn) \in \xi_N^{ip}(\texttt{dests})$ we have $\texttt{sqn}_N^{ip}(rip) \leq rsn$, as all other sequence numbers in routing table entries remain unchanged.

Pro. 1, Line 28; Pro. 3, Line 10;Pro. 4, Lines 13, 29; Pro. 5, Line 17: The set `dests` is constructed immediately before the invalidation procedure. For $(rip, rsn) \in \xi_N^{ip}(\texttt{dests})$, we have $\texttt{sqn}_N^{ip}(rip) \leq \texttt{inc}(\texttt{sqn}_N^{ip}(rip)) = rsn$.

Pro. 6, Line 3: When constructing `dests` in Line 2, the side condition $\xi_{N_2}^{ip}(\texttt{sqn(rt,rip)}) < \xi_{N_2}^{ip}(\texttt{rsn})$ is taken into account, which immediately yields the claim for $(rip, rsn) \in \xi_N^{ip}(\texttt{dests})$. ☐

Proof of Proposition 6 According to Section 5.7 the claim holds for each initial state, and none of our processes has an assignment changing the value of the variable `ip`. ☐

Proof of Proposition 8 All initial states trivially satisfy the invariant since all routing tables are empty. The functions `invalidate` and `addpreRT` do not affect the invariant, since they do not change the hop count of a routing table entry. Therefore, we only have to look at the application calls of `update`. In each case, if the update does not change the routing table entry beyond its precursors (the last clause of `update`), the invariant is trivially preserved; hence we examine the cases that an update actually occurs.

Pro. 1, Lines 10, 14, 18: All these updates have a hop count equal to 1; hence the invariant is preserved.

Pro. 4, Line 4; Pro. 5, Line 2: Here, $\xi(\texttt{hops})+1$ is used for the update. Since $\xi(\texttt{hops}) \in \mathbb{N}$, the invariant is maintained. ◻

Proof of Proposition 9

(a) We have to check that the consequent holds whenever a route request is sent. In all the processes there are only two locations where this happens.

Pro. 1, Line 39: A request with content $\xi(0,*,*,*,*,\texttt{ip},*,\texttt{ip})$ is sent. Since the sixth and the eighth component are the same ($\xi(\texttt{ip})$), the claim holds.

Pro. 4, Line 36: The message has the form $\texttt{rreq}(\xi(\texttt{hops})+1,*,*,*,*,*,*,*)$ Since $\xi(\texttt{hops}) \in \mathbb{N}$, $\xi(\texttt{hops})+1 \neq 0$ and hence the antecedent does not hold.

(b) We have to check that the consequent holds whenever a route reply is sent. In all the processes there are only three locations where this happens.

Pro. 4, Line 10: A reply with content $\xi(0,\texttt{dip},*,*,\texttt{ip})$ is sent. By Line 7 we have $\xi(\texttt{dip}) = \xi(\texttt{ip})$, so the claim holds.

Pro. 4, Line 25: $\texttt{rrep}(\texttt{dhops}(\texttt{rt},\texttt{dip}),*,*,*,*)$ is the form of the message. By Proposition 8, $\texttt{dhops}(\texttt{rt},\texttt{dip}) > 0$, so the antecedent does not hold.

Pro. 5, Line 13: $\texttt{rrep}(\xi(\texttt{hops})+1,*,*,*,*)$ is the form of the message. Since $\xi(\texttt{hops}) \in \mathbb{N}$, $\xi(\texttt{hops})+1 \neq 0$ and hence the antecedent does not hold. ◻

Proof of Proposition 10 At the initial states all routing tables are empty. Since `invalidate` and `addpreRT` change neither the sequence-number-status flag, nor the next hop or the hop count of a routing table entry, and—by Proposition 5—cannot decrease the sequence number of a destination, we only have to look at the application calls of `update`. As before, we only examine the cases that an update actually occurs.

(a) Function calls of the form $\texttt{update}(rt,r)$ always preserve the invariant: in case `update` is given an argument for which it is not defined,[28] the process algebra blocks and no change of the routing table is performed [16, Section 4]; in case one of the first four clauses in the definition of `update` is used, this follows because $\texttt{update}(rt,r)$ is defined only when $\pi_2(r) = 0 \Leftrightarrow \pi_3(r) = \texttt{unk}$; in case the fifth clause is used it follows because $\pi_3(r) = \texttt{unk}$; and in case the last clause is used, it follows by induction, since the invariant was already valid before the update.

(b) *Pro. 1, Lines 10, 14, 18:* All these updates have an unknown sequence number and hop count equal to 1. By Clause 5 of `update`, these sequence-number-status flag and hop count are transferred literally into the routing table; hence the invariant is preserved.

Pro. 4, Line 4 and Pro. 5, Line 2: In these updates the sequence-number-status flag is set to `kno`. By the definition of `update`, this value ends up in the routing table. Hence the assumption of the invariant to be proven is not satisfied.

(c) *Pro. 1, Lines 10, 14, 18:* The new entries, which have the form $\xi(\texttt{sip},0,\texttt{unk},\texttt{val},1,\texttt{sip},\emptyset)$, satisfy the invariant; even if the routing table is actually updated with one of the new routes, the invariant holds afterwards.

Pro. 4, Line 4; Pro. 5, Line 2: The route that might be inserted into the routing table has hop count $\texttt{hops}+1$, $\texttt{hops} \in \mathbb{N}$. It can only be equal to 1 if the received message had hop count $\texttt{hops} = 0$. In that case Invariant (5), resp. (6), guarantees that the invariant remains unchanged.

(d) Immediate from Parts (a) to (c). ◻

Proof of Proposition 11

(a) We have to check that the consequent holds whenever a route request is sent.

Pro. 1, Line 39: A route request is initiated. The originator sequence number is a copy of the node's own sequence number, i.e., $osn_c = \xi(\texttt{sn})$. By Proposition 2, we get $osn_c \geq 1$.

Pro. 4, Line 36: $osn_c := \xi(\texttt{osn})$ is not changed within Pro. 4; it stems, through Line 8 of Pro. 1, from an incoming RREQ message (Pro. 1, Line 1). For this incoming RREQ message, using Proposition 1(a) and induction on reachability, the invariant holds; hence the claim follows immediately.

(b) We have to check that the consequent holds whenever a route reply is sent.

Pro. 4, Line 10: The destination initiates a route reply. The sequence number is a copy of the node's own sequence number, i.e., $dsn_c = \xi(\texttt{sn})$. By Proposition 2, we get $dsn_c \geq 1$.

Pro. 4, Line 25: The sequence number used for the message is copied from the routing table; its value is $dsn_c := \texttt{sqn}(\xi(\texttt{rt}),\xi(\texttt{dip}))$. By Line 20, we know that $\texttt{flag}(\xi(\texttt{rt}),\xi(\texttt{dip})) = \texttt{kno}$ and hence, by Invariant (7), $dsn_c \geq 1$. Thus the invariant is maintained.

Pro. 5, Line 13: $dsn_c := \xi(\texttt{dsn})$ is not changed within Pro. 5; it stems, through Line 12 of Pro. 1, from an incoming RREP message (Pro. 1, Line 1). For this incoming RREP message the invariant holds and hence the claim follows immediately. ◻

Proof of Proposition 12

(a) We have to check all cases where a route request is sent:

Pro. 1, Line 39: A new route request is initiated with $ip_c = oip_c := \xi(\texttt{ip}) = ip$. Here the antecedent of (13) is not satisfied.

Pro. 4, Line 36: The broadcast message has the form $\xi(\texttt{rreq}(\texttt{hops}+1,\texttt{rreqid},\texttt{dip},\max(\texttt{sqn}(\texttt{rt},\texttt{dip}),\texttt{dsn}),\texttt{dsk},\texttt{oip},\texttt{osn},\texttt{ip}))$. Hence $hops_c := \xi(\texttt{hops})+1$, $oip_c := \xi(\texttt{oip})$, $osn_c := \xi(\texttt{osn})$, $ip_c := \xi(\texttt{ip}) = ip$ and $\xi_N^{ip_c} = \xi$ (by (3)). At Line 4 we update the routing table using $r := \xi(\texttt{oip},\texttt{osn},\texttt{kno},\texttt{val},\texttt{hops}+1,\texttt{sip},\emptyset)$ as new entry. The routing table does not change between Lines 4 and 36; nor do the values of the variables `hops`, `oip` and `osn`. If the new (valid) entry is inserted into the routing table, then one of the first four cases in the definition of `update` must have applied—the fifth case cannot apply, since $\pi_3(r) = \texttt{kno}$. Thus, using that $oip_c \neq ip_c$,

$$\begin{aligned}
\texttt{sqn}_N^{ip_c}(oip_c) &= \texttt{sqn}(\xi(\texttt{rt}),\xi(\texttt{oip})) = \xi(\texttt{osn}) = osn_c \\
\texttt{dhops}_N^{ip_c}(oip_c) &= \texttt{dhops}(\xi(\texttt{rt}),\xi(\texttt{oip})) = \xi(\texttt{hops})+1 \\
&= hops_c \\
\texttt{flag}_N^{ip_c}(oip_c) &= \texttt{flag}(\xi(\texttt{rt}),\xi(\texttt{oip})) = \xi(\texttt{val}) = \texttt{val}\,.
\end{aligned}$$

[28] In Section 6.4 we will show that this cannot occur.

In case the new entry is not inserted into the routing table (the sixth case of update), we have $\mathtt{sqn}_N^{ip_c}(oip_c) = \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{oip})) \geq \xi(\mathtt{osn}) = osn_c$, and in case that $\mathtt{sqn}_N^{ip_c}(oip_c) = osn_c$ we see that $\mathtt{dhops}_N^{ip_c}(oip_c) = \mathtt{dhops}(\xi(\mathtt{rt}),\xi(\mathtt{oip})) \leq \xi(\mathtt{hops})+1 = hops_c$ and moreover $\mathtt{flag}_N^{ip_c}(oip_c) = \mathtt{val}$. Hence the invariant holds.

(b) We have to check all cases where a route reply is sent.

Pro. 4, Line 10: A new route reply with $ip_c := \xi(\mathtt{ip}) = ip$ is initiated. Moreover, by Line 7, $dip_c := \xi(\mathtt{dip}) = \xi(\mathtt{ip}) = ip$ and thus $ip_c = dip_c$. Hence, the antecedent of (14) is not satisfied.

Pro. 4, Line 25: We have $ip_c := \xi(\mathtt{ip}) = ip$, so $\xi_N^{ip_c} = \xi$. This time, by Line 18, $dip_c := \xi(\mathtt{dip}) \neq \xi(\mathtt{ip}) = ip_c$. By Line 20 there is a valid routing table entry for $dip_c := \xi(\mathtt{dip})$.

$$\begin{aligned} dsn_c &:= \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) = \mathtt{sqn}_N^{ip_c}(dip_c)\,, \\ hops_c &:= \mathtt{dhops}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) = \mathtt{dhops}_N^{ip_c}(dip_c)\,. \end{aligned}$$

Pro. 5, Line 13: The RREP message has the form

$$\xi(\mathtt{rrep}(\mathtt{hops}+1,\mathtt{dip},\mathtt{dsn},\mathtt{oip},\mathtt{ip}))\,.$$

Hence $hops_c := \xi(\mathtt{hops})+1$, $dip_c := \xi(\mathtt{dip})$, $dsn_c := \xi(\mathtt{dsn})$, $ip_c := \xi(\mathtt{ip}) = ip$ and $\xi_N^{ip_c} = \xi$. Using $(\xi(\mathtt{dip}), \xi(\mathtt{dsn}), \mathtt{kno}, \mathtt{val}, \xi(\mathtt{hops})+1, \xi(\mathtt{sip}), \emptyset)$ as new entry, the routing table is updated at Line 2. With exception of its precursors, which are irrelevant here, the routing table does not change between Lines 2 and 13; nor do the values of the variables hops, dip and dsn. Line 1 guarantees that during the update in Line 2, the new entry is inserted into the routing table, so

$$\begin{aligned} \mathtt{sqn}_N^{ip_c}(dip_c) &= \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) = \xi(\mathtt{dsn}) = dsn_c \\ \mathtt{dhops}_N^{ip_c}(dip_c) &= \mathtt{dhops}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) = \xi(\mathtt{hops})+1 \\ &= hops_c \\ \mathtt{flag}_N^{ip_c}(dip_c) &= \mathtt{flag}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) = \xi(\mathtt{val}) \\ &= \mathtt{val}\,. \end{aligned}$$ □

Proof of Proposition 13 We have to check that the consequent holds whenever a route error message is sent. In all the processes there are only seven locations where this happens.

Pro. 1, Line 32: The set $dests_c$ is constructed in Line 31 as a subset of $\xi_{N_{31}}^{ip}(\mathtt{dests}) = \xi_{N_{28}}^{ip}(\mathtt{dests})$. For each pair $(rip_c, rsn_c) \in \xi_{N_{28}}^{ip}(\mathtt{dests})$ one has $rip_c = \xi_{N_{27}}^{ip}(\mathtt{rip}) \in \mathtt{vD}_{N_{27}}^{ip}$. Then in Line 28, using the function invalidate, $\mathtt{flag}(\xi(\mathtt{rt}),rip_c)$ is set to inv and $\mathtt{sqn}(\xi(\mathtt{rt}),rip_c)$ to rsn_c. Thus we obtain $rip_c \in \mathtt{iD}_N^{ip}$ and $\mathtt{sqn}_N^{ip}(rip_c) = rsn_c$.

Pro. 3, Line 14; Pro. 4, Lines 17, 33; Pro. 5, Line 21; Pro. 6, Line 8: Exactly as above.

Pro. 3, Line 20: The set $dests_c$ contains only one single element. Hence $rip_c := \xi_N^{ip}(\mathtt{dip})$ and $rsn_c := \xi_N^{ip}(\mathtt{sqn}(\mathtt{rt},\mathtt{dip}))$. By Line 18, we have $rip_c = \xi_N^{ip}(\mathtt{dip}) \in \mathtt{iD}_N^{ip}$. The remaining claim follows by $rsn_c = \xi_N^{ip}(\mathtt{sqn}(\mathtt{rt}\,,\mathtt{dip})) = \mathtt{sqn}(\xi_N^{ip}(\mathtt{rt}),\xi_N^{ip}(\mathtt{dip})) = \mathtt{sqn}_N^{ip}(rip_c)$. □

Proof of Proposition 14

(a) In all initial states the invariant is satisfied, as a routing table starts out empty (see (2) in Section 5.7). None of the Processes 1–7 of Section 5 changes a routing table directly; the only way a routing table can be changed is through the functions update, invalidate and addpreRT. The latter two only change the sequence number, the validity status and the precursors of an existing route. This kind of update has no effect on the invariant. The first function inserts a new entry into a routing table only if the destination is unknown, that is, if no entry for this destination already exists in the routing table; otherwise the existing entry is replaced. Therefore the invariant is maintained.

(b) In any initial state the invariant is satisfied, as each store of queued data packets starts out empty. In Processes 1–7 of Section 5 a store is updated only through the functions add and drop. These functions respect the invariant.

(c) This is checked by inspecting all assignments to dests in Processes 1–7.

Pro. 1, Line 16: The message $\xi(\mathtt{msg})$ is received in Line 1, and hence, by Proposition 1(a), sent by some node before. The content of the message does not change during transmission, and we assume there is only one way to read a message $\xi(\mathtt{msg})$ as $\mathtt{rerr}(\xi(\mathtt{dests}),\xi(\mathtt{sip}))$. By induction, we may assume that when the other node composed the message, a partial function was assigned to the first argument $\xi(\mathtt{dests})$ of rerr.

Pro. 1, Line 27; Pro. 3, Line 9; Pro. 4, Lines 12, 28; Pro. 5, Line 16: The assigned sets have the form $\{(\xi(\mathtt{rip}), \mathtt{inc}(\mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{rip})))) \mid \dots\})$. Since inc and sqn are functions, for each $\xi(\mathtt{rip})$ there is only one pair $(\xi(\mathtt{rip}), \mathtt{inc}(\mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{rip}))))$.

Pro. 1, Line 31; Pro. 3, Line 13; Pro. 4, Lines 16, 32; Pro. 5, Line 20; Pro. 6, Line 7: In each of these cases a set $\xi(\mathtt{dests})$ constructed four lines before is used to construct a new set. By the invariant to be proven, these sets are already partial functions. From these sets some values are removed. Since subsets of partial functions are again partial functions, the claim follows immediately.

Pro. 6, Line 2: Similar to the previous case except that the set $\xi(\mathtt{dests})$ to be thinned out is not constructed before but stems from an incoming RERR message.

Pro. 3, Lines 20: The set is explicitly given and consists of only one element; thus the claim is trivial. □

Proof of Proposition 15 $\mathtt{update}(rt, r)$ is defined only under the assumptions $\pi_4(r) = \mathtt{val}$, $\pi_2(r) = 0 \Leftrightarrow \pi_3(r) = \mathtt{unk}$ and $\pi_3(r) = \mathtt{unk} \Rightarrow \pi_5(r) = 1$. In Pro. 1, Lines 10, 14 and 18, the entry $\xi(\mathtt{sip}, 0, \mathtt{unk}, \mathtt{val}, 1, \mathtt{sip}, \emptyset)$ is used as second argument, which obviously satisfies the assumptions. The function is used at four other locations:

Pro. 4, Line 4: Here, the entry $\xi(\mathtt{oip}, \mathtt{osn}, \mathtt{kno}, \mathtt{val}, \mathtt{hops}+1, \mathtt{sip}, \emptyset)$ is used as r to update the routing table. This entry fulfils $\pi_4(r) = \mathtt{val}$. Since $\pi_3(r) = \mathtt{kno}$, it remains to show that $\pi_2(r) = \xi(\mathtt{osn}) \geq 1$. The sequence number $\xi(\mathtt{osn})$ stems, through Line 8 of Pro. 1, from an incoming RREQ message and is not changed within Pro. 4. Hence, by Invariant (11), $\xi(\mathtt{osn}) \geq 1$.

Pro. 5, Lines 1, 2, 26: The update is similar to the one of Pro. 4, Line 4. The only difference is that the information stems from an incoming RREP message and that a routing table entry to $\xi(\mathtt{dip})$ (instead of $\xi(\mathtt{oip})$) is established. Therefore, the proof is similar to the one of Pro. 4, Line 4; instead of Invariant (11) we use Invariant (12). □

Proof of Proposition 16 The function $\mathtt{nhop}(rt, dip)$ is defined iff $dip \in \mathtt{kD}(rt)$.

Pro. 1, Line 27; Pro. 3, Line 9; Pro. 4, Lines 12, 28; Pro. 5, Line 16; Pro. 6, Line 2: The function is used within a formula.

Pro. 1, Line 23: Line 21 states $\xi(\mathtt{dip}) \in \mathtt{vD}(\xi(\mathtt{rt}))$; hence $\mathtt{nhop}(\xi(\mathtt{rt}),\xi(\mathtt{dip}))$ is defined.

Pro. 3, Line 7: By Line 5, $\xi(\mathtt{dip}) \in \mathtt{vD}(\xi(\mathtt{rt}))$.

Pro. 4, Lines 10, 25: In Line 4 the entry for destination $\xi(\texttt{oip})$ is updated; by this $\xi(\texttt{oip}) \in \texttt{kD}(\xi(\texttt{rt}))$.
Pro. 4, Line 23: By Line 20 $\xi(\texttt{dip}) \in \texttt{vD}(\xi(\texttt{rt}))$.
Pro. 5, Lines 11, 13: By Line 9 $\xi(\texttt{oip}) \in \texttt{vD}(\xi(\texttt{rt}))$.
Pro. 5, Line 12: In Line 2 the entry for destination $\xi(\texttt{dip})$ is updated; by this $\xi(\texttt{dip}) \in \texttt{kD}(\xi(\texttt{rt}))$. By Line 9 $\xi(\texttt{oip}) \in \texttt{vD}(\xi(\texttt{rt}))$.

If nhop is used within a formula, then $\texttt{nhop}(rt,rip)$ may not be defined, namely if $rip \notin \texttt{kD}(rt)$. In such a case, according to the convention of Footnote 8 in Section 3, the atomic formula in which this term occurs evaluates to false, and thereby is defined properly. □

Proof of Proposition 17 For the proof we denote the routing table after the update by rt'.

(a) By assumption, there is an entry $(dip, dsn_{rt}, *, f_{rt}, hops_{rt}, *, *)$ for dip in rt. In case $\pi_1(r) \neq dip$ the quality of the routing table w.r.t. dip stays the same, since the entry for dip is not changed.
We first assume that $r := (dip, 0, \texttt{unk}, \texttt{val}, 1, *, *)$. This means that the Clause 5 in the definition of update is used. The updated routing table entry to dip has the form $(dip, dsn_{rt}, \texttt{unk}, \texttt{val}, 1, *, *)$. So

$$\texttt{nsqn}(rt, dip) \leq \texttt{sqn}(rt, dip) = dsn_{rt} = \texttt{nsqn}(rt', dip)\,,$$
$$\text{and}\quad \texttt{dhops}(rt, dip) = hops_{rt} \geq 1 = \texttt{dhops}(rt', dip)\,.$$

The first inequality holds by (16); the penultimate step by Invariant (4).
Next, we assume that the sequence number is known and therefore the route used for the update has the form $r = (dip, dsn, \texttt{kno}, \texttt{val}, hops, *, *)$ with $dsn \geq 1$. After the performed update the routing entry for dip either has the form $(dip, dsn_{rt}, *, f_{rt}, hops_{rt}, *, *)$ or $(dip, dsn, \texttt{kno}, \texttt{val}, hops, *, *)$. In the former case the invariant is trivially preserved; in the latter, we know, by definition of update, that either (i) $dsn_{rt} < dsn$, (ii) $dsn_{rt} = dsn \wedge hops_{rt} > hops$, or (iii) $dsn_{rt} = dsn \wedge f_{rt} = \texttt{inv}$ holds. We complete the proof of the invariant by a case distinction.
(i) holds: First, $\texttt{nsqn}(rt, dip) \leq dsn_{rt} < dsn = \texttt{sqn}(rt', dip) = \texttt{nsqn}(rt', dip)$. Since dsn_{rt} is strictly smaller than $\texttt{nsqn}(rt', dip)$, there is nothing more to prove.
(iii) holds: We have $\texttt{nsqn}(rt, dip) = dsn_{rt} \mathbin{\dot{-}} 1 < dsn = \texttt{sqn}(rt', dip) = \texttt{nsqn}(rt', dip)$. The inequality holds since either $dsn_{rt} \mathbin{\dot{-}} 1 = 0 < 1 \leq dsn$ or $dsn_{rt} \mathbin{\dot{-}} 1 = dsn_{rt} - 1 < dsn_{rt} = dsn$.
(ii) holds but (iii) does not: Then $f_{rt} = \texttt{val}$. In this case the update does not change the net sequence number for dip: $\texttt{nsqn}(rt, dip) = dsn_{rt} = dsn = \texttt{nsqn}(rt', dip)$. By (ii), the hop count decreases:
$$\texttt{dhops}(rt, dip) = hops_{rt} > hops = \texttt{dhops}(rt', dip)\,.$$

(b) Assume that invalidate modifies an entry of the form $(rip, dsn, *, flag, *, *, *)$. Let $(rip, rsn) \in dests$; then $flag = \texttt{val}$ and the update results in the entry $(rip, \texttt{inc}(dsn), *, \texttt{inv}, *, *, *)$. By definition of net sequence numbers,

$$\begin{aligned}\texttt{nsqn}(rt, rip) &= \texttt{sqn}(rt, rip) = dsn = \texttt{inc}(dsn) \mathbin{\dot{-}} 1\\ &= \texttt{nsqn}(rt', rip)\,.\end{aligned}$$

Since the hop count is not changed by invalidate, we also have $\texttt{dhops}(rt,rip) = \texttt{dhops}(rt',rip)$, and hence $rt \approx_{dip} \texttt{invalidate}(rt, dests)$.

(c) The function addpreRT only modifies a set of precursors; it does not change the sequence number, the validity, the flag, nor the hop count of any entry of the routing table rt. □

Proof of Proposition 19 As before, we first check the initial states of our transition system and then check all locations in Processes 1–7 where a routing table might be changed. For an initial network expression, the invariant holds since all routing tables are empty.

A modification of $\xi_N^{nhip}(\texttt{rt})$ is harmless, as it can only increase $\texttt{kD}_N^{nhip}$ (cf. Proposition 4) as well as $\texttt{nsqn}_N^{nhip}(dip)$ (cf. Theorem 18).

Adding precursors to $\xi_N^{ip}(\texttt{rt})$ does not harm since the invariant does not depend on precursors. It remains to examine all calls of update and invalidate to $\xi_N^{ip}(\texttt{rt})$. Without loss of generality we restrict attention to those applications of update or invalidate that actually modify the entry for dip, beyond its precursors; if update only adds some precursors in the routing table, the invariant—which is assumed to hold before—is maintained. If invalidate occurs, the next hop $nhip$ is not changed. Since the invariant has to hold before the execution, it follows that $dip \in \texttt{kD}_N^{nhip}$ also holds after execution.

Pro. 1, Lines 10, 14, 18: The entry $\xi(\texttt{sip},0,\texttt{unk},\texttt{val},1,\texttt{sip},\emptyset)$ is used for the update; its destination is $dip := \xi(\texttt{sip})$. Since $dip = \xi(\texttt{sip}) = \texttt{nhop}_N^{ip}(\xi(\texttt{sip})) = \texttt{nhop}_N^{ip}(dip) = nhip$, the antecedent of the invariant to be proven is not satisfied.
Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17: In each of these cases, the precondition of (18) is satisfied by the executions of the line immediately before the call of invalidate (Pro. 1, Line 27, Pro. 3, Line 9; Pro. 4, Lines 12, 28; Pro. 5, Line 16). Thus, the quality of the routing table w.r.t. dip, and thereby the net sequence number of the routing table entry for dip, remains unchanged. Therefore the invariant is maintained.
Pro. 4, Line 4: Let us assume that the routing table entry $\xi(\texttt{oip}, \texttt{osn}, \texttt{kno}, \texttt{val}, \texttt{hops}+1, \texttt{sip}, *)$ is inserted into $\xi(\texttt{rt})$. So $dip := \xi(\texttt{oip})$, $nhip := \xi(\texttt{sip})$, $\texttt{nsqn}_N^{ip}(dip) := \xi(\texttt{osn})$ and $\texttt{dhops}_N^{ip}(dip) := \xi(\texttt{hops}) + 1$. This information is distilled from a received route request message (cf. Lines 1 and 8 of Pro. 1). By Proposition 1 this message was sent before, say in state $N^\dagger$; by Proposition 7 the sender of this message is $\xi(\texttt{sip})$.
By Invariant (13), with $ip_c := \xi(\texttt{sip}) = nhip$, $oip_c := \xi(\texttt{oip}) = dip$, $osn_c := \xi(\texttt{osn})$ and $hops_c := \xi(\texttt{hops})$, and using that $ip_c = nhip \neq dip = oip_c$, we get that $dip \in \texttt{kD}_{N^\dagger}^{nhip}$ and

$$\texttt{sqn}_{N^\dagger}^{nhip}(dip) = \texttt{sqn}_{N^\dagger}^{ip_c}(oip_c) > osn_c = \xi(\texttt{osn})\,,\ \text{or}$$
$$\texttt{sqn}_{N^\dagger}^{nhip}(dip) = \xi(\texttt{osn}) \wedge \texttt{flag}_{N^\dagger}^{nhip}(dip) = \texttt{val}\,.$$

We first assume that the first line holds. Then, by Theorem 18 and (16),

$$\begin{aligned}\texttt{nsqn}_N^{nhip}(dip) &\geq \texttt{nsqn}_{N^\dagger}^{nhip}(dip) \geq \texttt{sqn}_{N^\dagger}^{nhip}(dip) \mathbin{\dot{-}} 1\\ &\geq \xi(\texttt{osn}) = \texttt{nsqn}_N^{ip}(dip)\,.\end{aligned}$$

We now assume the second line to be valid. From this we conclude

$$\begin{aligned}\texttt{nsqn}_N^{nhip}(dip) &\geq \texttt{nsqn}_{N^\dagger}^{nhip}(dip) = \texttt{sqn}_{N^\dagger}^{nhip}(dip)\\ &= \xi(\texttt{osn}) = \texttt{nsqn}_N^{ip}(dip)\,.\end{aligned}$$

Pro. 5, Line 2: The update is similar to the one of Pro. 4, Line 4. The only difference is that the information stems from an incoming RREP message and that a routing table entry to $\xi(\texttt{dip})$ (instead of $\xi(\texttt{oip})$) is established. Therefore, the proof is similar to the one of Pro. 4, Line 4; instead of Invariant (13) we use Invariant (14).

Pro. 6, Line 3: Let N_3 and N be the network expressions right before and right after executing Pro. 6, Line 3. The entry for destination dip can be affected only if $(dip, dsn) \in \xi_{N_2}^{ip}(\texttt{dests})$ for some $dsn \in \texttt{SQN}$. In that case, by Line 2, $(dip, dsn) \in \xi_{N_2}^{ip}(\texttt{dests})$, $dip \in \texttt{vD}_{N_2}^{ip}$, and $\texttt{nhop}_{N_2}^{ip}(dip) = \xi_{N_2}^{ip}(\texttt{sip})$. By definition of $\texttt{invalidate}$, $\texttt{sqn}_N^{ip}(dip) = dsn$ and $\texttt{flag}_N^{ip}(dip) = \texttt{inv}$, so

$$\texttt{nsqn}_N^{ip}(dip) = \texttt{sqn}_N^{ip}(dip) \dot{-} 1 = dsn \dot{-} 1\,.$$

Hence we need to show that $dsn \dot{-} 1 \leq \texttt{nsqn}_N^{nhip}(dip)$.
The values $\xi_{N_2}^{ip}(\texttt{dests})$ and $\xi_{N_2}^{ip}(\texttt{sip})$ stem from a received route error message (cf. Lines 1 and 16 of Pro. 1). By Proposition 1(a), a transition labelled

R:***cast**$(\texttt{rerr}(dests_c\,, ip_c))$

with $dests_c := \xi_{N_2}^{ip}(\texttt{dests})$ and $ip_c := \xi_{N_2}^{ip}(\texttt{sip})$ must have occurred before, say in state $N^\dagger$. By Proposition 7, the node casting this message is $ip_c = \xi_{N_2}^{ip}(\texttt{sip}) = \texttt{nhop}_{N_2}^{ip}(dip) = \texttt{nhop}_N^{ip}(dip) = nhip$. The penultimate equation holds since the next hop to dip is not changed during the execution of Pro. 6. By Proposition 13 we have $dip \in \texttt{iD}_{N^\dagger}^{nhip}$ and $dsn \leq \texttt{sqn}(\xi_{N^\dagger}^{nhip}(\texttt{rt})\,, dip)$. Hence

$$\begin{aligned}\texttt{nsqn}_N^{nhip}(dip) &\geq \texttt{nsqn}_{N^\dagger}^{nhip}(dip) = \texttt{nsqn}(\xi_{N^\dagger}^{nhip}(\texttt{rt})\,, dip)\\ &= \texttt{sqn}(\xi_{N^\dagger}^{nhip}(\texttt{rt})\,, dip) \dot{-} 1 \geq dsn \dot{-} 1\,,\end{aligned}$$

where the first inequality follows by Theorem 18. □

Proof of Theorem 23 In the course of running the protocol, the set of edges E in the connectivity graph $\mathcal{C}_H$ only increases, so the properties are invariants. We prove them by simultaneous induction.

(a) In an initial state the invariant is satisfied because the routing tables are empty. Since entries can never be removed, and the functions $\texttt{addpreRT}$ and $\texttt{invalidate}$ do not affect $hops$ and $nhip$, it suffices to check all application calls of $\texttt{update}$. In each case, if the update does not change the routing table entry beyond its precursors (the last clause of $\texttt{update}$), the invariant is trivially preserved; hence we examine the cases that an update actually occurs.

Pro. 1, Lines 10, 14, 18: The update changes the entry into $\xi(\texttt{sip}, *, \texttt{unk}, \texttt{val}, 1, \texttt{sip}, *)$; hence $hops = 1$ and $nhip = dip := \xi(\texttt{sip})$. The value $\xi(\texttt{sip})$ stems through Lines 8, 12 or 16 of Pro. 1 from an incoming AODV control message. By Proposition 1 this message was sent before, say in state $N^\dagger$; by Proposition 7 the sender of this message is $\xi(\texttt{sip}) = nhip$. Since in state $N^\dagger$ the message must have reached the queue of incoming messages of node ip, it must be that $ip \in R_{N^\dagger}^{nhip}$. In our formalisation of AWN the connectivity graph is always symmetric [16]: $nhip \in R_{N^\dagger}^{ip}$ iff $ip \in R_{N^\dagger}^{nhip}$. It follows that $(ip, nhip) \in E$, so there is a 1-hop path in $\mathcal{C}_H$ from ip to dip.

Pro. 4, Line 4: Here $dip := \xi(\texttt{oip})$, $hops := \xi(\texttt{hops})+1$ and $nhip := \xi(\texttt{sip})$. These values stem from an incoming RREQ message, which must have been sent beforehand, say in state $N^\dagger$. As in the previous case we obtain $(ip, nhip) \in E$. By Invariant (22), with $oip_c := \xi(\texttt{oip}) = dip$, $hops_c := \xi(\texttt{hops})$ and $ip_c := \xi(\texttt{sip}) = nhip$, there is a path $nhip \to \cdots \to dip$ in $\mathcal{C}_H$ from ip_c to oip_c with $hops_c$ hops. It follows that there is a path $ip \to nhip \to \cdots \to dip$ in $\mathcal{C}_H$ from ip to dip with $hops$ hops and next hop $nhip$.

Pro. 5, Line 2: Here $dip := \xi(\texttt{dip})$, $hops := \xi(\texttt{hops})+1$ and $nhip := \xi(\texttt{sip})$. The reasoning is exactly as in the previous case, except that we deal with an incoming RREP message and use Invariant (23).

(b) We check all occasions where a route request is sent.

Pro. 1, Line 39: A new route request is initiated with $ip_c = oip_c := \xi(\texttt{ip}) = ip$ and $hops_c := 0$. Indeed there is a path in $\mathcal{C}_H$ from ip_c to oip_c with 0 hops.

Pro. 4, Line 36: The broadcast message has the form

$$\begin{aligned}&\xi(\texttt{rreq}(\texttt{hops}{+}1\,, \texttt{rreqid}\,, \texttt{dip}\,, \max(\texttt{sqn}(\texttt{rt}\,, \texttt{dip}), \texttt{dsn}),\\ &\qquad \texttt{dsk}\,, \texttt{oip}\,, \texttt{osn}\,, \texttt{ip}))\,.\end{aligned}$$

So $hops_c{:=}\xi(\texttt{hops}){+}1$, $oip_c{:=}\xi(\texttt{oip})$ and $ip_c{:=}\xi(\texttt{ip}) = ip$. The values $\xi(\texttt{hops})$ and $\xi(\texttt{oip})$ stem through Line 8 of Pro. 1 from an incoming RREQ message of the form

$$\xi(\texttt{rreq}(\texttt{hops}\,, \texttt{rreqid}\,, \texttt{dip}\,, \texttt{dsn}\,, \texttt{dsk}\,, \texttt{oip}\,, \texttt{osn}\,, \texttt{sip}))\,.$$

By Proposition 1 this message was sent before, say in state $N^\dagger$; by Proposition 7 the sender of this message is $sip := \xi(\texttt{sip})$. By induction, using Invariant (22), there is a path $sip \to \cdots \to oip_c$ in $\mathcal{C}_{H^\dagger} \subseteq \mathcal{C}_H$ from sip to oip_c with $\xi(\texttt{hops})$ hops. It remains to show that there is a 1-hop path from ip to sip. In state $N^\dagger$ the message sent by sip must have reached the queue of incoming messages of node ip, and therefore ip was in transmission range of sip, i.e., $ip \in R_{N^\dagger}^{sip}$. Since the connectivity graph of AWN is always symmetric, $ip \in R_{N^\dagger}^{sip}$ holds as well. Hence it follows that $(ip, sip) \in E$.

(c) We check all occasions where a route reply is sent.

Pro. 4, Line 10: A new route reply with $hops_c := 0$ and $ip_c := \xi(\texttt{ip}) = ip$ is initiated. Moreover, by Line 7, $dip_c := \xi(\texttt{dip}) = \xi(\texttt{ip}) = ip$. Thus there is a path in $\mathcal{C}_H$ from ip_c to dip_c with 0 hops.

Pro. 4, Line 25: We have $ip_c{:=}\xi(\texttt{ip}){=}ip$, $dip_c{:=}\xi(\texttt{dip})$ and $hops_c := \texttt{dhops}_N^{ip}(dip_c)$. By Line 20 there is a routing table entry $(dip_c, *, *, *, hops_c, *, *) \in \xi_N^{ip}(\texttt{rt})$. Hence by Invariant (a), which we may assume to hold when using simultaneous induction, there is a path $ip \to \cdots \to dip_c$ in $\mathcal{C}_H$ from $ip = ip_c$ to dip_c with $hops_c$ hops.

Pro. 5, Line 13: The RREP message has the form $\xi(\texttt{rrep}(\texttt{hops}+1\,, \texttt{dip}\,, \texttt{dsn}\,, \texttt{oip}\,, \texttt{ip}))$ and the proof goes exactly as for Pro. 4, Line 36 of Part (b), by using $dip_c := \xi(\texttt{dip})$ instead of $oip_c := \xi(\texttt{oip})$, and an incoming RREP message instead of an incoming RREQ message. □