A Process Algebra for Wireless Mesh Networks

Ansgar Fehnker[1,4], Rob van Glabbeek[1,4], Peter Höfner[1,4], Annabelle McIver[2,1], Marius Portmann[1,3], and Wee Lum Tan[1,3]

[1] NICTA
[2] Department of Computing, Macquarie University
[3] School of ITEE, The University of Queensland
[4] Computer Science and Engineering, University of New South Wales

Abstract. We propose a process algebra for wireless mesh networks that combines novel treatments of local broadcast, conditional unicast and data structures. In this framework, we model the Ad-hoc On-Demand Distance Vector (AODV) routing protocol and (dis)prove crucial properties such as loop freedom and packet delivery.

1 Introduction

Wireless Mesh Networks (WMNs) have recently gained considerable popularity and are increasingly deployed in a wide range of application scenarios, including emergency response communication, intelligent transportation systems, mining, video surveillance, etc. WMNs are essentially self-organising wireless ad-hoc networks that can provide broadband communication without relying on a wired backhaul infrastructure. This has the benefit of rapid and low-cost network deployment. WMNs can be considered a superset of Mobile Ad-hoc Networks (MANETs), where a network consists exclusively of mobile end user devices such as laptops or smartphones. In contrast to MANETs, WMNs typically also contain stationary infrastructure devices called mesh routers. However, this distinction is not relevant for the purpose of this paper; what matters is that both MANETs and WMNs share the characteristic of highly dynamic network topologies, due to node mobility and the variable nature of wireless links.

In WMNs, a routing protocol is used to establish and maintain network connectivity through paths between source and destination node pairs. As a consequence, the routing protocol is one of the key factors determining the performance and reliability of WMNs. Traditionally, the main tools for evaluating and validating network protocols are simulation and test-bed experiments. The key limitations of these approaches are that they are very expensive, time consuming and non-exhaustive, i.e., they only cover a very limited set of network scenarios. As a result, protocol errors and limitations are still found many years after the definition and standardisation; for example, see [14].

Formal methods have a great potential in helping to address this problem, and can provide valuable tools for design, evaluation and verification of WMN routing protocols. 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.

H. Seidl (Ed.): ESOP 2012, LNCS 7211, pp. 295–315, 2012.

In this paper, we propose a process algebra that provides a step towards this goal. It combines novel treatments of data structures, conditional unicast and local broadcast, and allows formalisation of all important aspects of a routing protocol. All these features are necessary to model "real life" WMNs. Data structures are used to store and maintain information, e.g. routing tables. The conditional unicast construct allows us to model that a node in a network sends a message to a particular neighbour, and if this fails, for example because the receiver has moved out of transmission range, error handling is initiated. Finally, the local broadcast primitive, which allows a node to send messages to all its immediate neighbours, models the wireless broadcast mechanism implemented by the physical and data link layer of wireless standards relevant for WMNs. Our formalisation assumes that any broadcast message *is* received by all nodes within transmission range.[1] This abstraction enables us to interpret a failure of guaranteed message delivery as an imperfection in the protocol, rather than as a result of a chosen formalism not allowing guaranteed delivery.

To demonstrate the use of our algebra, in [6] we use it to formally model and reason about the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [16]—we outline this work here. AODV is one of the most relevant and widely used routing protocols in WMNs. Our model covers the complete core functionality of AODV and abstracts from timing and optional features only. The process algebra proposed in this paper allows us to prove critical protocol properties of AODV, such as loop freedom. We also use our model to show limitations of AODV, e.g. that AODV does not guarantee that messages are always delivered to their destinations, even if a stable route exists (cf. Section 3.4).

2 A Process Algebra for Wireless Routing Protocols

In this section we propose AWN, a process algebra for the specification of WMN routing protocols such as AODV. It models a WMN as an encapsulated parallel composition of network nodes. On each node several sequential processes may be running in parallel. Network nodes communicate with their direct neighbours—those nodes that are in transmission range—using either broadcast or unicast. Due to mobility of nodes and variability of wireless links, nodes can move in or out of transmission range. The encapsulation of the entire network inhibits communications between network nodes and the outside world, with the exception of the receipt and delivery of data packets from or to clients [2] of the modelled protocol that may be hooked up to various nodes.

[1] In reality, communication is only half-duplex: a network node cannot receive messages while sending and hence messages can be lost. However, the CSMA protocol used at the link layer—not modelled here—keeps the probability of packet loss due to two nodes (within range) sending at the same time rather low. Since we are examining imperfect protocols, we first of all want to establish how they behave under optimal conditions. For this reason we abstract from probabilistic reasoning by assuming no message loss at all, rather than working with a lossy broadcast formalism that offers no guarantees that any message will ever arrive.

[2] The application layer that initiates packet sending and awaits receipt of a packet.

2.1 A Language for Sequential Processes

The internal state of a process is determined, in part, by the values of certain data variables that are maintained by that process. To this end, we assume a data structure with several types, variables ranging over these types, operators and predicates. First order predicate logic yields terms (or *data expressions*) and formulas to denote data values and statements about them. Our data structure always contains the types $\mathtt{DATA}$, $\mathtt{MSG}$, $\mathtt{IP}$ and $\mathcal{P}(\mathtt{IP})$ of *application layer data*, *messages*, *IP addresses*—or any other node identifiers—and *sets of IP addresses.*

In addition, we assume a set of *process names.* Each process name X comes with a *defining equation*

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

in which $n \in \mathbb{N}$, the $\mathtt{var}_i$ are variables and p is a *sequential process expression* defined by the grammar below. It may contain the variables $\mathtt{var}_i$ as well as X. However, all occurrences of data variables in p have to be *bound.*[3] The choice of the underlying data structure and the process names with their defining equations can be tailored to any particular application of our language.

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

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

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

Given a valuation of the data variables by concrete data values, the sequential process $[\varphi]p$ acts as p if φ evaluates to $\mathtt{true}$, and deadlocks if φ evaluates to $\mathtt{false}$. In case φ contains free variables that are not yet interpreted as data values, values are assigned to these variables in any way that satisfies φ, if possible. The sequential process $[\![\mathtt{var} := \mathit{exp}]\!]p$ acts as p, but under an updated valuation of the data variable $\mathtt{var}$. The sequential process $p+q$ may act either as p or as q, depending on which of the two processes is able to act at all. In a context where both are able to act, it is not specified how the choice is made. The sequential process $\alpha.p$ first performs the action α and subsequently acts as p. The action $\mathbf{broadcast}(\mathit{ms})$ broadcasts (the data value bound to the expression) ms to the other network nodes within transmission range, whereas $\mathbf{unicast}(\mathit{dest}, \mathit{ms}).p \blacktriangleright q$ is a process that tries to unicast the message ms to

[3] An occurrence of a data variable in p is *bound* if it is one of the variables $\mathtt{var}_i$, a variable $\mathtt{msg}$ occurring in a subexpression $\mathbf{receive}(\mathtt{msg}).q$, a variable $\mathtt{var}$ occurring in a subexpression $[\![\mathtt{var} := \mathit{exp}]\!]q$, or an occurrence in a subexpression $[\varphi]q$ of a variable occurring free in φ. Here q is an arbitrary sequential process expression.

Table 1. Structural operational semantics for sequential process expressions

$$\xi, \textbf{broadcast}(\mathit{ms}).p \xrightarrow{\textbf{broadcast}(\xi(\mathit{ms}))} \xi, p$$

$$\xi, \textbf{groupcast}(\mathit{dests}, \mathit{ms}).p \xrightarrow{\textbf{groupcast}(\xi(\mathit{dests}),\xi(\mathit{ms}))} \xi, p$$

$$\xi, \textbf{unicast}(\mathit{dest}, \mathit{ms}).p \blacktriangleright q \xrightarrow{\textbf{unicast}(\xi(\mathit{dest}),\xi(\mathit{ms}))} \xi, p$$

$$\xi, \textbf{unicast}(\mathit{dest}, \mathit{ms}).p \blacktriangleright q \xrightarrow{\neg\textbf{unicast}(\xi(\mathit{dest}))} \xi, q$$

$$\xi, \textbf{send}(\mathit{ms}).p \xrightarrow{\textbf{send}(\xi(\mathit{ms}))} \xi, p$$

$$\xi, \textbf{deliver}(\mathit{data}).p \xrightarrow{\textbf{deliver}(\xi(\mathit{data}))} \xi, p$$

$$\xi, \textbf{receive}(\texttt{msg}).p \xrightarrow{\textbf{receive}(m)} \xi[\texttt{msg} := m], p \qquad (\forall m \in \texttt{MSG})$$

$$\xi, [\![\texttt{var} := \mathit{exp}]\!]p \xrightarrow{\tau} \xi[\texttt{var} := \xi(\mathit{exp})], p$$

$$\frac{\emptyset[\texttt{var}_i := \xi(\mathit{exp}_i)]_{i=1}^{n}, p \xrightarrow{a} \zeta, p'}{\xi, X(\mathit{exp}_1, \ldots, \mathit{exp}_n) \xrightarrow{a} \zeta, p'} \ (X(\texttt{var}_1, \ldots, \texttt{var}_n) \stackrel{def}{=} p) \qquad (\forall a \in \text{Act})$$

$$\frac{\xi, p \xrightarrow{a} \zeta, p'}{\xi, p + q \xrightarrow{a} \zeta, p'} \quad \frac{\xi, q \xrightarrow{a} \zeta, q'}{\xi, p + q \xrightarrow{a} \zeta, q'} \quad \frac{\xi \xrightarrow{\varphi} \zeta}{\xi, [\varphi]p \xrightarrow{\tau} \zeta, p} \qquad (\forall a \in \text{Act})$$

the destination *dest*; if successful it continues to act as p and otherwise as q. In other words, **unicast**(*dest*, *ms*).p is prioritised over q; only if the action **unicast**(*dest*, *ms*) is not possible, the alternative q will happen. It models an abstraction of an acknowledgment-of-receipt mechanism that is typical for unicast communication but absent in broadcast communication, as implemented by the link layer of relevant wireless standards such as IEEE 802.11. The process **groupcast**(*dests*, *ms*).p tries to transmit *ms* to all destinations *dests*, and proceeds as p regardless of whether any of the transmissions is successful. Unlike **unicast** and **broadcast**, the expression **groupcast** does not have a unique counterpart in networking. Depending on the protocol and the implementation it can be an iterative unicast, a broadcast, or a multicast; thus **groupcast** abstracts from implementation details. The action **send**(*ms*) synchronously transmits a message to another process running on the same node; this action can occur only when this other sequential process is able to receive the message. The sequential process **receive**(`msg`).p receives any message m (a data value of type `MSG`) either from another node, from another sequential process running on the same node or from the client hooked up to the local node. It then proceeds as p, but with the data variable `msg` bound to the value m. The submission of data from a client is modelled by the receipt of a message `newpkt`(d, *dip*), where the function `newpkt` generates a message containing the data d and the intended destination *dip*. Data is delivered to the client by **deliver**(*data*).

The internal state of a sequential process described by an expression p is determined by p, together with a *valuation* ξ associating values ξ(`var`) to variables `var` maintained by this process. Valuations naturally extend to *ξ-closed* expressions—those in which all variables are either bound or in the domain of ξ. The structural operational semantics of Table 1 is in the style of Plotkin [17] and describes how one internal state can evolve into another by performing

Table 2. Structural operational semantics for parallel process expressions

$$\frac{P \xrightarrow{a} P'}{P \langle\!\langle Q \xrightarrow{a} P' \langle\!\langle Q} \ (\forall a \neq \mathbf{receive}(m)) \qquad \frac{Q \xrightarrow{a} Q'}{P \langle\!\langle Q \xrightarrow{a} P \langle\!\langle Q'} \ (\forall a \neq \mathbf{send}(m))$$

$$\frac{P \xrightarrow{\mathbf{receive}(m)} P' \qquad Q \xrightarrow{\mathbf{send}(m)} Q'}{P \langle\!\langle Q \xrightarrow{\tau} P' \langle\!\langle Q'} \ (\forall m \in \mathtt{MSG})$$

an *action.* The set Act of actions consists of $\mathbf{broadcast}(m)$, $\mathbf{groupcast}(D,m)$, $\mathbf{unicast}(\mathit{dip},m)$, $\neg\mathbf{unicast}(\mathit{dip})$, $\mathbf{send}(m)$, $\mathbf{deliver}(d)$, $\mathbf{receive}(m)$ and internal actions τ, for each choice of $m \in \mathtt{MSG}$, $\mathit{dip} \in \mathtt{IP}$, $D \in \mathcal{P}(\mathtt{IP})$ and $d \in \mathtt{DATA}$. Here, $\neg\mathbf{unicast}(\mathit{dip})$ denotes a failed unicast. Moreover $\xi[\mathtt{var} := v]$ denotes the valuation that assigns the value v to the variable $\mathtt{var}$, and agrees with ξ on all other variables. The empty valuation $\emptyset$ assigns values to no variables. Hence $\emptyset[\mathtt{var}_i := v_i]_{i=1}^{n}$ is the valuation that *only* assigns the values v_i to the variables $\mathtt{var}_i$ for $i = 1, \ldots, n$. The rule for process names in Table 1 (Line 9) says that a process, named X, has the same transitions as the body p of its defining equation. (See [6] for details.) Finally, $\xi \xrightarrow{\varphi} \zeta$ says that ζ is an extension of ξ, i.e., a valuation that agrees with ξ on all variables on which ξ is defined, and evaluates other variables occurring free in φ, such that the formula φ holds under ζ. All variables not free in φ and not evaluated by ξ are also not evaluated by ζ.

2.2 A Language for Parallel Processes

Parallel process expressions are given by the grammar

$$\mathit{PP} \ ::= \ \xi, \mathit{SP} \ \mid \ \mathit{PP} \langle\!\langle \mathit{PP} \ ,$$

where SP is a sequential process expression and ξ a valuation. An expression ξ, p denotes a sequential process expression equipped with a valuation of the variables it maintains. The process $P \langle\!\langle Q$ is a parallel composition of P and Q, running on the same network node. As formalised in Table 2, an action $\mathbf{receive}(m)$ of P synchronises with an action $\mathbf{send}(m)$ of Q into an internal action τ. These receive actions of P and send actions of Q cannot happen separately. All other actions of P and Q occur interleaved in $P \langle\!\langle Q$. The variables of sequential processes running on the same node are maintained separately, and thus cannot be shared.

Though $\langle\!\langle$ is a restricted version of synchronisation, which only allows information flow "in one direction", it reflects reality of WMNs. Usually two sequential processes run on the same node: $P \langle\!\langle Q$. The main process P deals with all protocol details of the node, e.g., message handling and maintaining the data such as routing tables. The process Q manages the queueing of messages as they arrive; it is always able to receive a message even if P is busy. The use of message queueing in combination with $\langle\!\langle$ is crucial, since otherwise incoming messages would be lost when the process is busy dealing with other messages[4], which would not be an accurate model of what happens in real implementations.

[4] Assuming that one employs the optional augmentation of Section 2.5.

Table 3. Structural operational semantics for node expressions

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

$$\frac{P \xrightarrow{\textbf{unicast}(dip,m)} P' \qquad dip \in R}{ip\!:\!P\!:\!R \xrightarrow{\{dip\}\,:\,\textbf{*cast}(m)} ip\!:\!P'\!:\!R} \qquad \frac{P \xrightarrow{\neg\textbf{unicast}(dip)} P' \qquad dip \notin R}{ip\!:\!P\!:\!R \xrightarrow{\tau} ip\!:\!P'\!:\!R}$$

$$\frac{P \xrightarrow{\textbf{deliver}(d)} P'}{ip\!:\!P\!:\!R \xrightarrow{ip\,:\,\textbf{deliver}(d)} ip\!:\!P'\!:\!R} \qquad \frac{P \xrightarrow{\textbf{receive}(m)} P'}{ip\!:\!P\!:\!R \xrightarrow{\{ip\}\neg\emptyset\,:\,\textbf{arrive}(m)} ip\!:\!P'\!:\!R}$$

$$\frac{P \xrightarrow{\tau} P'}{ip\!:\!P\!:\!R \xrightarrow{\tau} ip\!:\!P'\!:\!R} \qquad ip\!:\!P\!:\!R \xrightarrow{\emptyset\neg\{ip\}\,:\,\textbf{arrive}(m)} ip\!:\!P\!:\!R$$

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

2.3 A Language for Networks

We model network nodes in the context of a WMN by *node expressions* of the form $ip\!:\!PP\!:\!R$. Here $ip \in \texttt{IP}$ is the *address* of the node, PP is a parallel process expression, and $R \in \mathcal{P}(\texttt{IP})$ is the *range* of the node—the set of nodes that are currently within transmission range of ip.

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

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

The operational semantics of node and network expressions of Tables 3 and 4 uses transition labels $R\!:\!\textbf{*cast}(m)$, $H\neg K\!:\!\textbf{arrive}(m)$, $\textbf{connect}(ip,ip')$, $\textbf{disconnect}(ip,ip')$, $ip\!:\!\textbf{newpkt}(d,dip)$, $ip\!:\!\textbf{deliver}(d)$ and τ. Again, $m \in \texttt{MSG}$, $d \in \texttt{DATA}$, $R \in \mathcal{P}(\texttt{IP})$, and $ip, ip' \in \texttt{IP}$. Moreover, $H, K \in \mathcal{P}(\texttt{IP})$ are sets of IP addresses. The action $R\!:\!\textbf{*cast}(m)$ casts a message m that can be received by the set R of network nodes. We do not distinguish whether this message has been broadcast, groupcast or unicast—the differences show up merely in the value of R. Recall that $D \in \mathcal{P}(\texttt{IP})$ denotes a set of intended destinations, and $dip \in \texttt{IP}$ a single destination. A failed unicast attempt on the part of its process is modelled as an internal action τ on the part of a node expression. The action $\textbf{send}(m)$ of a process does not give rise to any action of the corresponding node—this action of a sequential process cannot occur without communicating with a receive action of another sequential process running on the same node.

The action $H\neg K\!:\!\textbf{arrive}(m)$ states that the message m simultaneously arrives at all addresses $ip{\in}H$, and fails to arrive at all addresses $ip{\in}K$. The rules of

Table 4. Structural operational semantics for network expressions

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

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

$$\frac{M \xrightarrow{R\,:\,\textbf{*cast}(m)} M'}{[M] \xrightarrow{\tau} [M']} \qquad \frac{M \xrightarrow{\{ip\}\neg K\,:\,\textbf{arrive}(\texttt{newpkt}(d,dip))} M'}{[M] \xrightarrow{ip\,:\,\textbf{newpkt}(d,dip)} [M']}$$

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

Table 4 let an $R\!:\textbf{*cast}(m)$-action of one node synchronise with an $\textbf{arrive}(m)$ of all other nodes, where this $\textbf{arrive}(m)$ amalgamates the arrival of message m at the nodes in the transmission range R, and the non-arrival at the other nodes. The rules for $\textbf{arrive}(m)$ in Table 3 state that arrival of a message at a node happens if and only if the node receives it, whereas non-arrival can happen at any time. This embodies our assumption that, at any time, any message that is transmitted to a node within range of the sender is actually received by that node. (The eighth rule in Table 3, having no premises, may appear to say that any node ip has the option to disregard any message at any time. However, the encapsulation operator (below) prunes away all such disregard-transitions that do not synchronise with a cast action for which ip is out of range.)

Internal actions τ and the action $ip\!:\textbf{deliver}(d)$ are simply inherited by node expressions from the processes that run on these nodes, and are interleaved in the parallel composition of nodes that makes up a network. Finally, we allow actions $\textbf{connect}(ip, ip')$ and $\textbf{disconnect}(ip, ip')$ for $ip, ip' \in \texttt{IP}$ modelling a change in network topology. These actions can be thought of as occurring nondeterministically, or as actions instigated by the environment of the modelled network protocol. In this formalisation node ip' may be in the range of node ip, meaning that ip can send to ip', even when the reverse does not hold. For some applications, in particular the one to AODV in [6], it is useful to assume that ip' is in the range of ip if and only if ip is in the range of ip'. This symmetry can be enforced by adding the following rules to Table 3

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

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

and replacing the last three rules for (dis)connect actions by

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

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

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

2.4 Results on the Process Algebra

Our process algebra admits translation into one without data structures (although we cannot describe the target algebra without using data structures): the idea is to replace processes ξ,p by $\mathcal{T}_\xi(p)$, where $\mathcal{T}_\xi$ is defined inductively by

$\mathcal{T}_\xi(\mathbf{broadcast}(\mathit{ms}).p) = \mathbf{broadcast}(\xi(\mathit{ms})).\mathcal{T}_\xi(p)$,
$\mathcal{T}_\xi(\mathbf{receive}(\mathtt{msg}).p) = \sum_{m\in\mathtt{MSG}}\mathbf{receive}(m).\mathcal{T}_{\xi[\mathtt{msg}:=m]}(p)$,
$\mathcal{T}_\xi(X(\mathit{exp}_1,\ldots,\mathit{exp}_n)) = X_{\xi(\mathit{exp}_1),\ldots,\xi(\mathit{exp}_n)}$, etc.

This requires the introduction of a process name $X_{\vec{v}}$ for every substitution instance $\vec{v}$ of the arguments of X. The resulting process algebra has a structural operational semantics in the *de Simone* format, generating the same transition system—up to strong bisimilarity, $\underline{\leftrightarrow}$ —as the original. It follows that $\underline{\leftrightarrow}$, and many other semantic equivalences, are congruences on our language [19].

Theorem 2.1. *Strong bisimilarity is a congruence for all operators of our language.*

This is a deep result that usually takes many pages to establish (e.g., [20]). Here we get it directly from the existing theory on structural operational semantics, as a result of carefully designing our language within the disciplined framework described by de Simone [19]. □

Theorem 2.2. *$\langle\!\langle$ is associative, and $\|$ is associative and commutative, up to $\,\underline{\leftrightarrow}$.*

Proof. The operational rules for these operators fit a format presented in [5], guaranteeing associativity up to $\underline{\leftrightarrow}$. The ASSOC-de Simone format of [5] applies to all transition system specifications (TSSs) in de Simone format, and allows 7 different types of rules (named 1–7) for the operators in question. Our TSS is in De Simone format; the three rules for $\langle\!\langle$ of Table 2 are of types 1, 2 and 7, respectively. To be precise, it has rules 1_a for $a \in \mathrm{Act} \setminus \{\mathbf{receive}(m) \mid m \in \mathtt{MSG}\}$, rules 2_a for $a \in \mathrm{Act} \setminus \{\mathbf{send}(m) \mid m \in \mathtt{MSG}\}$, and rules $7_{(a,b)}$ for $(a,b) \in \{(\mathbf{receive}(m), \mathbf{send}(m)) \mid m \in \mathtt{MSG}\}$. Moreover, the partial *communication function* $\gamma : \mathrm{Act} \times \mathrm{Act} \rightharpoonup \mathrm{Act}$ is given by $\gamma(\mathbf{receive}(m), \mathbf{send}(m)) = \tau$. The main result of [5] is that an operator is guaranteed to be associative, provided that γ is associative and six conditions are fulfilled. In the absence of rules of types 3, 4, 5 and 6, five of these conditions are trivially fulfilled, and the remaining one reduces to

$$7_{(a,b)} \;\Rightarrow\; (1_a \Leftrightarrow 2_b) \;\wedge\; (2_a \Leftrightarrow 2_{\gamma(a,b)}) \;\wedge\; (1_b \Leftrightarrow 1_{\gamma(a,b)}) \;.$$

Here 1_a says that rule 1_a is present, etc. This condition is met for $\langle\!\langle$ because the antecedent holds only when taking $(a,b) = (\mathbf{receive}(m), \mathbf{send}(m))$ for some $m \in \mathtt{MSG}$. In that case 1_a is false, 2_b is false, and 2_a, 2_τ, 1_b and 1_τ are true. Moreover, $\gamma(\gamma(a,b),c)$ and $\gamma(a,\gamma(b,c))$ are never defined, thus making γ trivially associative. The argument for $\|$ being associative proceeds likewise. Here the only nontrivial condition is the associativity of γ, given by

$$\begin{aligned}\gamma(R\!:\!\mathbf{*cast}(m), H\neg K\!:\!\mathbf{arrive}(m)) &= \gamma(H\neg K\!:\!\mathbf{arrive}(m), R\!:\!\mathbf{*cast}(m))\\ &= R\!:\!\mathbf{*cast}(m)\;,\end{aligned}$$

provided $H \subseteq R$ and $K \cap R = \emptyset$, and

$$\gamma(H\neg K\!:\!\mathbf{arrive}(m), H'\neg K'\!:\!\mathbf{arrive}(m)) = (H \cup H')\neg(K \cup K')\!:\!\mathbf{arrive}(m)\;.$$

Commutativity of $\|$ follows by symmetry. □

2.5 Optional Augmentation to Ensure Non-blocking Broadcast

Our process algebra, as presented above, is intended for networks in which each node is *input enabled* [11], meaning that it is always ready to receive any message, i.e., able to engage in the transition $\mathbf{receive}(m)$ for any $m \in \mathtt{MSG}$. In our model of AODV [6] we ensure this by equipping each node with a message queue that is always able to accept messages for later handling—even when the main sequential process is currently busy. This makes our model *non-blocking*, meaning that no sender can be delayed in transmitting a message simply because one of the potential recipients is not ready to receive it.

However, the operational semantics does allow blocking if one would (mis)use the process algebra to model nodes that are not input enabled. This is a logical consequence of insisting that any broadcast message *is* received by all nodes within transmission range.

Since the possibility of blocking can be regarded as a bad property of broadcast formalisms, one may wish to take away the expressiveness of the language that allows modelling a blocking broadcast. This is the purpose of the following optional augmentations of our operational semantics.

The first possibility is the addition of the rule $\dfrac{P \overset{\textbf{receive}(m)}{\not\longrightarrow}}{ip{:}P{:}R \xrightarrow{\{ip\}\neg\emptyset\,:\,\textbf{arrive}(m)} ip{:}P{:}R}$. It states that a message may arrive at a node *ip* regardless whether the node is ready to receive it; if it is not ready, the message is simply ignored, and the process running remains in the same state.

A variation on the same idea, elaborated in [6, Sect. 4.5], stems from the *Calculus of Broadcasting Systems* [18]. It consists in eliminating the negative premise in the above rule in favour of *discard* actions, thereby remaining within the de Simone format of structural operational semantics. Either of these two optional augmentations of our semantics gives rise to the same transition system. Moreover, when modelling networks in which all nodes are input enabled—as we do in [6]—the added rule for node expressions will never be used, and the resulting transition system is the same whether we use augmentation or not.

2.6 Illustrative Example

To illustrate the use of our process algebra AWN, we consider a network of two nodes a and b ($a, b \in \mathtt{IP}$) on which the same process is running, although starting in different states. The process describes a simple (toy-)protocol: whenever a new data packet for destination *dip* "appears",[5] the data is broadcast through the network until it finally reaches *dip*. A node alternates between broadcasting, and receiving and handling a message. The *data* stemming from a message received by node *ip* will be delivered to the application layer if the message is destined for *ip* itself. Otherwise the node forwards the message. Every message travelling through the network and handled by the protocol has the form $\mathtt{mg}(\mathit{data}, \mathit{dip})$, where $\mathit{data} \in \mathtt{DATA}$ is the data to be sent and $\mathit{dip} \in \mathtt{IP}$ is its destination. The behaviour of each node can be modelled by:

$$\begin{aligned}
&\mathtt{X}(\mathtt{ip};\mathtt{data},\mathtt{dip}) \overset{\mathit{def}}{=} \textbf{broadcast}(\mathtt{mg}(\mathtt{data},\mathtt{dip})).\mathtt{Y}(\mathtt{ip})\\
&\mathtt{Y}(\mathtt{ip}) \overset{\mathit{def}}{=} \textbf{receive}(\mathtt{m}).([\mathtt{m}{=}\mathtt{mg}(\mathtt{data},\mathtt{dip}) \wedge \mathtt{dip}{=}\mathtt{ip}]\ \textbf{deliver}(\mathtt{data}).\mathtt{Y}(\mathtt{ip})\\
&\qquad\qquad\qquad\qquad + [\mathtt{m}{=}\mathtt{mg}(\mathtt{data},\mathtt{dip}) \wedge \mathtt{dip}{\neq}\mathtt{ip}]\ \mathtt{X}(\mathtt{ip};\mathtt{data},\mathtt{dip}))\ .
\end{aligned}$$

If a node is in a state $\mathtt{X}(\mathit{ip}; \mathit{data}, \mathit{dip})$, where $\mathit{ip} \in \mathtt{IP}$ is the node's stored value of its own IP address, it will broadcast $\mathtt{mg}(\mathit{data}, \mathit{dip})$ and continue in state $\mathtt{Y}(\mathit{ip})$, meaning that all information about the message is dropped. If a node in state $\mathtt{Y}(\mathit{ip})$ receives a message m—a value that will be assigned to the variable $\mathtt{m}$—it has two ways to continue: process $[\mathtt{m}{=}\mathtt{mg}(\mathtt{data},\mathtt{dip}) \wedge \mathtt{dip}{=}\mathtt{ip}]\ \textbf{deliver}(\mathtt{data}).\mathtt{Y}(\mathtt{ip})$ is enabled if the incoming message has the form $\mathtt{mg}(\mathit{data}, \mathit{dip})$ and the node itself is the destination of the message ($\mathtt{dip}{=}\mathtt{ip}$). In that case the data distilled from m will be delivered to the application layer, and the process returns to $\mathtt{Y}(\mathit{ip})$. Alternatively, if $[\mathtt{m}{=}\mathtt{mg}(\mathtt{data},\mathtt{dip}) \wedge \mathtt{dip}{\neq}\mathtt{ip}]$, the process continues as

[5] In this small example, we assume that new data packets just appear "magically"; of course one could use the message newpkt(data,dip) instead.

$\mathtt{X}(ip; data, dip)$, which will then broadcast another message with contents *data* and *dip*. Note that calls to processes use expressions as parameters.

Let us have a look at three scenarios. First, assume that the nodes a and b are within transmission range of each other; node a in state $\mathtt{X}(a; d, b)$, and node b in $\mathtt{Y}(b)$. This is formally expressed as $[a : \mathtt{X}(a; d, b) : \{b\} \parallel b : \mathtt{Y}(b) : \{a\}]$, although for compactness of presentation, below we just write $[\mathtt{X}(a; d, b) \parallel \mathtt{Y}(b)]$. In this case, node a broadcasts the message $\mathtt{mg}(d, b)$ and continues as $\mathtt{Y}(a)$. Node b receives the message, **deliver**s d (after evaluation of the message) and continues as $\mathtt{Y}(a)$. Formally, we get transitions from one state to the other:

$$[\mathtt{X}(a; d, b) \parallel \mathtt{Y}(b)] \xrightarrow{a:\textbf{*cast}(\mathtt{mg}(d,b))} \xrightarrow{\tau} \xrightarrow{b:\textbf{deliver}(d)} [\mathtt{Y}(a) \parallel \mathtt{Y}(b)].$$

Here, the τ-transition is the action of evaluating the first of the two guards of a process Y, and we left out the two intermediate expressions.

Second, assume that the nodes are not within transmission range, with the initial process of a and b the same as above; formally $[a : \mathtt{X}(a; d, b) : \emptyset \parallel b : \mathtt{Y}(b) : \emptyset]$. As before, node a broadcasts $\mathtt{mg}(d, b)$ and continues in $\mathtt{Y}(a)$; but this time the message is not received by any node; hence no message is forwarded or delivered and both nodes end up running process Y.

For the last scenario, we assume that a and b are within transmission range and that they have the initial states $\mathtt{X}(a; d, b)$ and $\mathtt{X}(b; e, a)$. Without the augmentation of Section 2.5, the network expression $[\mathtt{X}(a; d, b) \parallel \mathtt{X}(b; e, a)]$ admits no transitions at all; neither node can broadcast its message, because the other node is not listening. With the optional augmentation, assuming that node a sends first:

$$\begin{aligned}[\mathtt{X}(a; d, b) \parallel \mathtt{X}(b; e, a)] &\xrightarrow{a:\textbf{*cast}((\mathtt{mg}(d,b))} [\mathtt{Y}(a) \parallel \mathtt{X}(b; e, a)] \\ &\xrightarrow{b:\textbf{*cast}(\mathtt{mg}(e,a))} \xrightarrow{\tau} \xrightarrow{a:\textbf{deliver}(e)} [\mathtt{Y}(a) \parallel \mathtt{Y}(b)].\end{aligned}$$

Unfortunately, node b is initially 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 3.2). Using a message queue, the optional augmentation is not needed, since any node is always in a state where it can receive a message.

3 Routing Protocols

The features of our process algebra were largely determined by what we needed to enable a complete and accurate formalisation of wireless network protocols and their properties.

We use the proposed algebra to formally model and reason about the Ad hoc On-demand Distance Vector (AODV) routing protocol [16]. Due to lack of space, we can only briefly report on our formalisation and the properties proved. All details can be found in [6].

Since routing protocols for WMNs are based on common concepts in wireless networks in general, such as local broadcast, we do expect that our process algebra can easily be used to model other wireless network protocols.

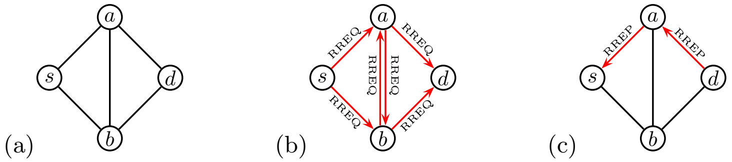

Fig. 1. Example network topology

3.1 Ad-Hoc On-Demand Distance Vector Routing Protocol

AODV [16] is a widely-used routing protocol designed for MANETs, and is one of the four protocols currently standardised by the IETF MANET working group[6]. It also forms the basis of new WMN routing protocols, including the upcoming IEEE 802.11s wireless mesh network standard [10].

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, …, 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 Fig. 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.[7] Therefore, they simply re-broadcast the message, as shown in Fig. 1(b). Each RREQ message has a unique identifier which allows nodes to ignore duplicate RREQ messages that they have handled before.

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 sending a route reply (RREP) message back to node s, as shown in Fig. 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 are another important aspect

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

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

Table 5. Data structure

Type	Variables	Description
IP	ip, dip, oip, rip, sip, nhip	node identifiers
SQN	sn, dsn, rsn	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		pending-request flag
STORE	store	store of queued data packets
MSG	msg	messages
[MSG]	msgs	message queues
$\mathcal{P}$(IP)	pre	sets of identifiers (precursors, destinations, ...)
IP $\rightharpoonup$ SQN	dests	sets of destinations with sequence numbers
$\mathcal{P}$(IP $\times$ RREQID)	rreqs	sets of request identifiers with originator IP

Constant/Predicate	Description
0 : SQN, 1 : SQN	unknown, smallest sequence number
$<$ $\subseteq$ SQN $\times$ 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
pen, non-pen : P	constants to distinguish (non-)pending RREQs
[ ] : [MSG]	empty queue

Operator	Description
setP : STORE $\times$ IP $\times$ P $\rightarrow$ STORE	set the pending-request flag
(_, _, _, _, _, _, _) : IP$\times$SQN$\times$K$\times$F$\times$ $\mathbb{N}$ $\times$IP$\times\mathcal{P}$(IP) $\rightarrow$ R	generates a routing table entry
inc : SQN $\rightarrow$ SQN	increments the sequence number
sqn : RT $\times$ IP $\rightarrow$ SQN	returns the sequence number of a particular route
flag : RT $\times$ IP $\rightharpoonup$ F	returns the validity of a particular route
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 $\rightharpoonup$ $\mathcal{P}$(IP)	returns the set of precursors of a particular route
vD, kD : RT $\rightarrow$ $\mathcal{P}$(IP)	returns the set of valid, known destinations
addpreRT : RT $\times$ IP $\times$ $\mathcal{P}$(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) $\rightarrow$ RT	invalidates a set of routes within a routing table
rrep : $\mathbb{N}$ $\times$IP $\times$ SQN $\times$ IP $\times$ IP $\rightarrow$ MSG	generates a route reply
rerr : (IP $\rightharpoonup$ SQN) $\times$ IP $\rightarrow$ MSG	generates a route error message

of AODV, and are used to indicate the freshness of routing table entries for the purpose of preventing routing loops.

3.2 A Formal Model of AODV

Our formalisation of AODV is a faithful rendering of the IETF's specification [16] with the exception of time and any optional features. Additionally, we model the submission, forwarding and delivery of data packets—this is not part of the AODV standard, but crucial to trigger the route discovery process of AODV.

In this section we give an overview of the formal model, setting out the details only for the RREP message handling. Full details are available in [6, Sect. 6].

Table 5 lists the types and operators needed for the formalisation presented in this section. For example, RT is the type of routing tables—modelled as set of entries (*dip*, *dsn*, *dsk*, *flag*, *hops*, *nhip*, *pre*), each providing information on a route of length *hops* with ultimate destination *dip*. The next hop address on that route is *nhip*. The value *dsn* is a *sequence number*, intended to describe the "freshness" of this entry, with *dsk* a Boolean indicating whether or not that number is known to be an up-to-date indicator of the freshness of the entry. The values *flag* and *pre*, respectively, describe the validity of the entry, and its *precursors*—a set of nodes that "rely" on it to ensure the validity of their own entries. In a routing table *rt* there is at most one entry for each destination *dip*; $\mathtt{sqn}(rt, dip)$ denotes the sequence number of that entry and likewise for the operators `flag` and `dhops`. Another example is $\mathtt{update}(rt, r)$, which updates a routing table *rt* with an entry *r*. This is one of the major activities of AODV. It adds $r := (dip, dsn, dsk, flag, hops, nhip, pre)$ to the routing table *rt* if no entry for *dip* is present. The existing entry is overwritten by *r* if the latter's sequence number is larger ($dsn > \mathtt{sqn}(rt, dip)$) or, in case of equal sequence numbers, the existing entry is invalid, or the new hop count smaller ($dsn = \mathtt{sqn}(rt, dip) \wedge (\mathtt{flag}(rt, dip) = \mathtt{inv} \vee hops < \mathtt{dhops}(rt, dip))$).

A network is modelled as a parallel composition of its constituent nodes.[8] For all nodes of a network—characterised by a set $\mathbf{IP} \subseteq \mathtt{IP}$ of unique identifiers $ip \in \mathbf{IP}$—the node expression $ip : P : R$ is initialised with the parallel process

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

The sequential process $\mathtt{AODV}(\mathtt{ip}, \mathtt{rt}, \mathtt{sn}, \mathtt{rreqs}, \mathtt{store})$ deals with the detailed message handling of the node, manages its routing table `rt`, stores its own sequence number in `sn`, records all route requests seen so far in `rreqs` and maintains in `store` packets to be sent. The process `QMSG(msgs)` manages the queueing of messages as they arrive; it is always able to receive a message even if `AODV` is busy updating `rt`, forwarding requests etc. Whenever a message arrives `QMSG(msgs)` appends it to the queue `msgs`, passing it on to `AODV` whenever it can. The composition $\langle\!\langle$ is crucial here to express this "buffering mechanism" occurring in actual implementations of AODV.

Any node is initialised with its own identifier stored in the variable `ip`, an empty routing table, the sequence number 1, and empty sets of seen route requests and stored data packets. Also the queue of received messages is empty.

The process `AODV` receives messages from `QMSG` and then, depending on their types, delegates the response to the appropriate process : `PKT` (for data), `RREQ` (for requests), `RREP` (for replies) and `RERR` (for errors). In this paper we give only the specification of `RREP` (cf. Process 1); the specifications of the other processes can be found in [6, Sect. 6].

Usually, `RREP` updates the routing table with information from the route reply message `rrep(hops, dip, dsn, oip, sip)`, meaning that it is a reply to a former request initiated by `oip` for destination `dip`, that it was sent by (1-hop neighbour) `sip`, and that it takes `hops` hops from `sip` to `dip`. The sequence number `dsn` measures the "freshness" of this information. In case the current node is `oip`, receipt of this message establishes a route from `oip` to `dip`. Only when the new information leads to an actual update of the routing table (Line 2), and the

[8] Here, associativity and commutativity of $\|$ (Theorem 2.2) is essential.

Process 1. RREP handling

```
RREP(hops , dip , dsn , oip , sip ; ip , rt , sn , rreqs , store) $\overset{def}{=}$
 1. /*routing that describes the handling of a received route reply*/
 2. [ $\mathtt{rt} \neq \mathtt{update}(\mathtt{rt}, (\mathtt{dip}, \mathtt{dsn}, \mathtt{kno}, \mathtt{val}, \mathtt{hops}+1, \mathtt{sip}, \emptyset))$ ]   /*the routing table has to be updated*/
 3.   (
 4.     $[\![\mathtt{rt} := \mathtt{update}(\mathtt{rt}, (\mathtt{dip}, \mathtt{dsn}, \mathtt{kno}, \mathtt{val}, \mathtt{hops} + 1, \mathtt{sip}, \emptyset))]\!]$
 5.     [ $\mathtt{oip} = \mathtt{ip}$ ]   /*this node is the originator of the corresponding RREQ*/
 6.       $[\![\mathtt{store} := \mathtt{setP}(\mathtt{store}, \mathtt{dip}, \mathtt{non\text{-}pen})]\!]$   /*set queue-flag to non-pending*/
 7.       /*a packet may now be sent; this is done in the process AODV*/
 8.       AODV(ip,sn,rt,rreqs,store)
 9.     + [ $\mathtt{oip} \neq \mathtt{ip}$ ]   /*this node is not the originator; forward RREP*/
10.       (
11.         [ $\mathtt{oip} \in \mathtt{vD}(\mathtt{rt})$ ]   /*valid route to oip*/
12.           /*add next hop towards oip as precursor and forward the route reply*/
13.           $[\![\mathtt{rt} := \mathtt{addpreRT}(\mathtt{rt}, \mathtt{dip}, \{\mathtt{nhop}(\mathtt{rt}, \mathtt{oip})\})]\!]$
14.           $[\![\mathtt{rt} := \mathtt{addpreRT}(\mathtt{rt}, \mathtt{nhop}(\mathtt{rt}, \mathtt{dip}), \{\mathtt{nhop}(\mathtt{rt}, \mathtt{oip})\})]\!]$
15.           unicast(nhop(rt,oip),rrep(hops + 1,dip,dsn,oip,ip)) .
16.             AODV(ip,sn,rt,rreqs,store)
17.           ▶ /*If the packet transmission is unsuccessful, a RERR message is generated*/
18.             $[\![\mathtt{dests} := \{(\mathtt{rip}, \mathtt{inc}(\mathtt{sqn}(\mathtt{rt}, \mathtt{rip}))) \mid \mathtt{rip} \in \mathtt{vD}(\mathtt{rt}) \wedge \mathtt{nhop}(\mathtt{rt}, \mathtt{rip}) = \mathtt{nhop}(\mathtt{rt}, \mathtt{oip})\}]\!]$
19.             $[\![\mathtt{rt} := \mathtt{invalidate}(\mathtt{rt}, \mathtt{dests})]\!]$
20.             $[\![\mathtt{pre} := \bigcup\{\mathtt{precs}(\mathtt{rt}, \mathtt{rip}) \mid (\mathtt{rip}, *) \in \mathtt{dests}\}]\!]$
21.             $[\![\mathtt{dests} := \{(\mathtt{rip}, \mathtt{rsn}) \mid (\mathtt{rip}, \mathtt{rsn}) \in \mathtt{dests} \ \wedge \ \mathtt{precs}(\mathtt{rt}, \mathtt{rip}) \neq \emptyset]\!]$
22.             groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
23.         + [ $\mathtt{oip} \notin \mathtt{vD}(\mathtt{rt})$ ]   /*no valid route to oip*/
24.           AODV(ip,sn,rt,rreqs,store)
25.       )
26.   )
27. + [ $\mathtt{rt} = \mathtt{update}(\mathtt{rt}, (\mathtt{dip}, \mathtt{dsn}, \mathtt{kno}, \mathtt{val}, \mathtt{hops}+1, \mathtt{sip}, \emptyset))$ ]   /*the routing table is not updated*/
28.   (
29.     [ $\mathtt{oip} = \mathtt{ip}$ ]   /*this node is the originator of the corresponding RREQ*/
30.       $[\![\mathtt{store} := \mathtt{setP}(\mathtt{store}, \mathtt{dip}, \mathtt{non\text{-}pen})]\!]$   /*set queue-flag to non-pending*/
31.       AODV(ip,sn,rt,rreqs,store)
32.     + [ $\mathtt{oip} \neq \mathtt{ip}$ ]   /*this node is not the originator; drop RREP*/
33.       AODV(ip,sn,rt,rreqs,store)
34.   )
```

current node is not the final destination `oip` of the route reply (Line 9), the RREP message will be forwarded (Line 15). In case the unicast is unsuccessful (Line 17), the link connecting the current node to $\mathtt{nhop}(\mathtt{rt}, \mathtt{oip})$ must be broken and the process initiates the procedure for error reporting (Lines 18–22). This involves determining which other nodes are "interested" in that link, because it contributes to their routes. Those interested nodes are stored in the precursor lists inside `rt` and an error message is sent to the nodes it finds there via the action **groupcast**. Before that, the node marks as invalid all routes in its routing table which use the failed link, and increments their sequence numbers (Lines 18–19).

3.3 Invariants

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

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

where P is a state either in the process AODV, PKT, RREQ, RREP or RERR. 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 $\mathbf{IP} \subseteq \mathtt{IP}$ of nodes, a reachable network expression N is an encapsulated parallel composition of node expressions—one for each $ip \in \mathbf{IP}$. To distil current information about a node from N, we define the following projections for valuation ξ and range R:

$$\begin{array}{lll} R_N^{ip} := R, & \text{where } ip : (*, * \langle\!\langle *, *) : R & \text{is a node expression of } N \ , \\ \xi_N^{ip} := \xi, & \text{where } ip : (\xi, * \langle\!\langle *, *) : * & \text{is a node expression of } N \ . \end{array}$$

For example, $\xi_N^{ip}(\mathtt{rt})$ evaluates the current routing table maintained by node ip in the network expression N.

Proposition 3.1. 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, i.e., if

$$N \xrightarrow{R:\mathbf{*cast}(\mathtt{rrep}(hops_c, dip_c, dsn_c, *, ip_c))} N'$$

then $dip_c \in \mathtt{kD}(\xi_N^{ip_c}(\mathtt{rt}))$, $\mathtt{sqn}(\xi_N^{ip_c}(\mathtt{rt}), dip_c) = dsn_c$, $\mathtt{dhops}(\xi_N^{ip_c}(\mathtt{rt}), dip_c) = hops_c$, and $\mathtt{flag}(\xi_N^{ip_c}(\mathtt{rt}), ip_c) = \mathtt{val}$.

Proof. We have to check all cases where a route reply is sent. Here we restrict ourselves to RREP; the entire proof can be found in [6, Prop. 7.10(b)]. A route reply occurs only in Line 15, where a message $\xi(\mathtt{rrep}(\mathtt{hops}+1, \mathtt{dip}, \mathtt{dsn}, \mathtt{oip}, \mathtt{ip}))$ is unicast. Here ξ is the current valuation ξ_N^{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 4. With exception of its precursors, which are irrelevant here, the routing table does not change between Lines 4 and 15; nor do the values of the variables hops, dip and dsn. Line 2 guarantees that during the update in Line 4, the new entry is inserted into the routing table, so

$$\begin{array}{lllll} \mathtt{sqn}(\xi(\mathtt{rt}), \xi(\mathtt{dip})) & = & \xi(\mathtt{dsn}) & = & dsn_c \\ \mathtt{dhops}(\xi(\mathtt{rt}), \xi(\mathtt{dip})) & = & \xi(\mathtt{hops}) + 1 & = & hops_c \\ \mathtt{flag}(\xi(\mathtt{rt}), \xi(\mathtt{dip})) & = & \xi(\mathtt{val}) & = & \mathtt{val}\ . \end{array}$$

□

The classical 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 "bad". This can happen when the destination is highly mobile and the packet "follows" the destination and keeps travelling through the network. Therefore, 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.

To this end we define the *routing graph* of 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 $(dip,*,*,\texttt{val},*,ip',*) \in \xi_N^{ip}(\texttt{rt})$.

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.

To prove loop freedom of AODV, we first establish a useful invariant.

Theorem 3.2. 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 without a valid routing table entry to dip, either the sequence number strictly increases, or this number stays the same and the hop count strictly decreases.

$$\begin{aligned} & dip \in \texttt{vD}(\xi_N^{ip}(\texttt{rt})) \cap \texttt{vD}(\xi_N^{nhip}(\texttt{rt})) \ \wedge \ nhip \neq dip \\ \Rightarrow\ & \texttt{sqn}(\xi_N^{ip}(\texttt{rt}),dip) < \texttt{sqn}(\xi_N^{nhip}(\texttt{rt}),dip) \ \vee \ \big(\texttt{sqn}(\xi_N^{ip}(\texttt{rt}),dip) = \texttt{sqn}(\xi_N^{nhip}(\texttt{rt}),dip) \\ & \wedge\ \texttt{dhops}(\xi_N^{ip}(\texttt{rt}),dip) > \texttt{dhops}(\xi_N^{nhip}(\texttt{rt}),dip)\big)\ , \end{aligned}$$

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

The proof uses Proposition 3.1; it can be found in [6].

From this, we immediately conclude that AODV is loop free.

More precisely, *our* AWN-specification of AODV is loop free. It is our belief that, up to the abstraction of time and any optional features presented in [16], it reflects precisely the intention and the meaning of the RFC. However, when formalising AODV, we came across ambiguities, which yield different possible interpretations. Such interpretations can be seen as variants of AODV and, as we discovered, only a few of them are loop free. Since loop freedom is a sine qua non for routing protocols like AODV, we endeavour to resolve the ambiguities as much as possible by discarding the interpretations that lead to loops.

We briefly explain one of the problems found. A crucial requirement in the proof of Theorem 3.2 is that sequence numbers in routing table entries are never decreased, and increased upon invalidating the entry. Following the RFC literally, a “node initiates processing for a RERR message”[9], “if it receives a RERR from a neighbor”[9]. For every destination to be invalidated the “destination sequence number”[9] is “copied from the incoming RERR”[9]. We have shown that this copying in combination with *self-entries* (entries for *ip* in *ip*’s own routing table)[10] violate the above requirement and yield loops; a detailed example is given in [6]. In our specification this behaviour does not occur since we slightly modified the invalidation procedure [6, Sect. 5] to ensure an increase of sequence number for an invalidated entry, in the spirit of Section 6.2 of the RFC.

[9] Section 6.11 of the RFC [16].

[10] In our model we allow self-entries, since they are not explicitly forbidden; they also occur in real implementations, e.g., Kernel AODV [1]; they are forbidden by others such as AODV-UU [2].

3.4 Formalising Temporal Properties

Our formalism enables verification of correctness properties. While some properties, such as loop freedom, are invariants on the routing tables, others require reasoning about the temporal order of transitions. We use Computation Tree Logic (CTL) to specify and discuss one such property, namely *packet delivery.*

CTL uses the path quantifiers $\mathbf{A}$ and $\mathbf{E}$, and the temporal operators $\mathbf{G}, \mathbf{F}, \mathbf{X}$, and $\mathbf{U}$. The (state) formula $\mathbf{A}\phi$ is satisfied in a state if all paths starting in that state satisfy ϕ, while $\mathbf{E}\phi$ is satisfied if some path satisfies ϕ. The (path) formulas $\mathbf{G}\phi, \mathbf{F}\phi$ and $\mathbf{X}\phi$ mean that ϕ holds globally in all states, in some state, and in the next state of a path, respectively. The *until* $\phi\mathbf{U}\psi$ means that, until a state occurs along the path that satisfies ψ, property ϕ has to hold. In CTL a temporal operator is always immediately preceded by a path quantifier. Here CTL is interpreted on the unfolding into a tree of the transition system generated by our operational semantics.

The property of *packet delivery* says that if a client submits a packet, it will eventually be delivered to the destination. However, in a WMN it is not guaranteed that this property holds, since nodes can get disconnected, e.g. due to node mobility. A useful formulation has to be weaker. AODV should guarantee *packet delivery* only if an end-to-end route exists long enough. More precisely, AODV should guarantee delivery of a packet submitted by a client at node *oip* with destination *dip*, when *oip* is connected to *dip* and afterwards no link in the network gets disconnected. This means that for any pair *oip* and *dip*, and any data *d*, the following should hold:

$$\begin{aligned}&\mathbf{AG}(\mathit{oip} : \mathbf{newpkt}(d, \mathit{dip}) \wedge \mathbf{connected}^*(\mathit{oip}, \mathit{dip}))\\ &\quad \Rightarrow \mathbf{AF}(\mathbf{disconnect}(*, *) \vee (\mathit{dip} : \mathbf{deliver}(d))) \ .\end{aligned}$$

$\mathit{oip}\!:\!\mathbf{newpkt}(d, \mathit{dip})$ models submission of a new packet at *oip*, $\mathit{dip}\!:\!\mathbf{deliver}(d)$ that the destination receives it, and $\mathbf{disconnect}(*, *)$ the action of disconnecting. We treat these transitions as predicates, with the understanding that along a path the state immediately succeeding such a transition satisfies it. The predicate $\mathbf{connected}^*(\mathit{oip}, \mathit{dip})$ is true if there are exist nodes $\mathit{ip}_0, \ldots, \mathit{ip}_n$ such that $\mathit{ip}_0 = \mathit{oip}$, $\mathit{ip}_n = \mathit{dip}$ and $\mathit{ip}_i \in R_N^{\mathit{ip}_{i-1}}$. for $i = 1, \ldots, n$.

Surprisingly, AODV does not satisfy this property. One cause is that AODV nodes do not forward route replies from which they do not learn anything new. However, the information may be vital for the potential recipients of the forwarding. See [6, Sect. 8] for further discussion of a counterexample.

4 Related Work

Several process algebras for MANETs have been proposed: CBS# [15], CWS [13], CMAN [8], CMN [12], the ω-calculus [20] and RBPT [7]. All these languages, as well as ours, feature a form of local broadcast, in which a single message, sent by one node, can be received by other nodes within transmission range, given an arbitrary topology. In CWS the topology is fixed, whereas the other formalisms deal with arbitrary changes in topology.

The latter four formalisms model a *lossy* broadcast, in which a potential receiver may lose a message; in CBS# and CWS, any node within range must receive a message m sent to it, provided the node is *ready* to receive it, i.e., in a state that admits a transition **receive**(m). This proviso makes all these calculi *non-blocking*, meaning that no sender can be delayed in sending a message simply because one of the potential recipients is not ready to receive it.

The syntax of CBS# and CWS does not permit the construction of meaningful nodes that are always ready to receive a message. Hence our model is the first that assumes that any message is received by a potential recipient within range. It is this feature that allows us to evaluate whether a protocol satisfies the *packet delivery* property. *Any* routing protocol formalised in any of the other formalisms would automatically fail to satisfy such a property.

Besides this ensured broadcast, the novel *conditional unicast* operator chooses a continuation process dependent on whether the message can be delivered. This operator is essential for the correct formalisation of AODV. In practice such an operator may be implemented by means of an acknowledgement mechanism; however, this is done at the link layer, from which the AODV specification [16], and hence our formalism, abstracts. One could formalise a conditional unicast as a standard unicast in the scope of a priority operator [4]; however, our operator prioritises, while allowing an operational semantics within the de Simone format.

Although our treatment of data structures follows the classical approach of universal algebra, and is in the spirit of formalisms like μCRL [9], we have not seen a process algebra that freely mixes in imperative programming constructs like variable assignment. Yet this helps to properly capture AODV and other routing protocols.

Our formalisation of AODV [6], which is partly shown here, has grown from elaborating a partial formalisation of AODV in [20]. The features of our process algebra were largely determined by what we needed to enable a complete and accurate formalisation of this protocol. We conjecture that the same formalism is also applicable to a wide range of other wireless protocols.

Loop freedom is a crucial property of network protocols, commonly claimed to hold for AODV [16]. In [6] we show that several *interpretations* of AODV—consistent ways to revolve the ambiguities in the RFC—fail to be loop free, while proving loop freedom of others. A preliminary draft of AODV has been shown to be not loop free (for other reasons) in [3]. In [21] a proof sketch of loop freedom for a restricted version of AODV is given, using an interactive theorem prover.

5 Conclusion and Outlook

We have proposed a novel algebra covering major aspects of WMN routing protocols. We have accurately modelled the core of AODV, a widely used protocol of practical relevance. In contrast to other works, our model covers the crucial aspect of data handling, such as maintaining routing table information. We have formalised and proven some of AODV's general properties. Our model provides, in combination with abstraction from lower network layers, a practical and powerful tool for WMN protocol specification, evaluation and verification.

Our analysis of AODV uncovered several ambiguities in the RFC [16]. Finding ambiguities and unexpected behaviour is not uncommon for RFCs in general. This shows that the specification of a reasonably rich protocol such as AODV cannot be described precisely and unambiguously by simple (English) text only; formal methods are indispensable for this purpose.

More detailed analysis requires the addition of 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.

References

1. Kernel AODV (v. 2.2.2), NIST, `http://www.antd.nist.gov/wctg/aodv_kernel/` (accessed January 6, 2012)
2. AODV-UU: An implementation of the AODV routing protocol (IETF RFC 3561), `http://sourceforge.net/projects/aodvuu/` (accessed January 6, 2012)
3. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM 49(4), 538–576 (2002), `http://dx.doi.org/10.1145/581771.581775`
4. Cleaveland, R., Lüttgen, G., Natarajan, V.: Priority in process alegbra. In: Handbook of Process Algebra, ch.12, pp. 711–765. Elsevier (2001)
5. Cranen, S., Mousavi, M.R., Reniers, M.A.: A Rule Format for Associativity. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 447–461. Springer, Heidelberg (2008)
6. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Tech. Rep. 5513, NICTA (2012), `http://www.nicta.com.au/pub?id=5513`
7. Ghassemi, F., Fokkink, W.J., Movaghar, A.: Restricted broadcast process theory. In: Proc. IEEE SEFM 2008 (2008)
8. Godskesen, J.C.: A Calculus for Mobile Ad Hoc Networks. In: Murphy, A.L., Ryan, M. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 132–150. Springer, Heidelberg (2007)
9. Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. In: Algebra of Communicating Processes 1994. Workshops in Computing, pp. 26–62. Springer, Heidelberg (1995)
10. Hiertz, G.R., Denteneer, D., Max, S., Taori, R., Cardona, J., Berlemann, L., Walke, B.: IEEE 802.11s: the WLAN mesh standard. IEEE Wireless Communications 17(1), 104–111 (2010), `http://dx.doi.org/10.1109/MWC.2010.5416357`
11. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI-Quarterly 2(3), 219–246 (1989); centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands
12. Merro, M.: An observational theory for mobile ad hoc networks. Inf. Comput. 207(2), 194–208 (2009)
13. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electr. Notes Theor. Comput. Sci. 158, 331–353 (2006)
14. Miskovic, S., Knightly, E.W.: Routing primitives for wireless mesh networks: Design, analysis and experiments. In: IEEE INFOCOM, pp. 2793–2801 (2010), `http://dx.doi.org/10.1109/INFCOM.2010.5462111`

15. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theor. Comput. Sci. 367, 203–227 (2006)
16. Perkins, C., Belding-Royer, E., Das, S.: Ad hoc on-demand distance vector (AODV) routing. RFC 3561 (2003), `http://www.ietf.org/rfc/rfc3561.txt`
17. Plotkin, G.: A structural approach to operational semantics. The Journal of Logic and Algebraic Programming 60- 61, 17–139 (2004) (originally appeared in 1981)
18. Prasad, K.V.S.: A calculus of broadcasting systems. Sci. Comput. Program. 25(2-3), 285–327 (1995)
19. de Simone, R.: Higher-level synchronising devices in Meije-SCCS. Theor. Comput. Sci. 37, 245–267 (1985)
20. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Science of Computer Programming 75, 440–469 (2010)
21. Zhou, M., Yang, H., Zhang, X., Wang, J.: The proof of AODV loop freedom. In: Proc. IEEE WCSP (2009)