A Process Algebra for Link Layer Protocols

Rob van Glabbeek[1,2(✉)], Peter Höfner[1,2], and Michael Markl[1,3]

[1] Data61, CSIRO, Sydney, Australia
rvg@cs.stanford.edu
[2] Computer Science and Engineering, University of New South Wales, Sydney, Australia
[3] Institut für Informatik, Universität Augsburg, Augsburg, Germany

Abstract. We propose a process algebra for link layer protocols, featuring a unique mechanism for modelling frame collisions. We also formalise suitable liveness properties for link layer protocols specified in this framework. To show applicability we model and analyse two versions of the Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol. Our analysis confirms the hidden station problem for the version without virtual carrier sensing. However, we show that the version with virtual carrier sensing not only overcomes this problem, but also the exposed station problem with probability 1. Yet the protocol cannot guarantee packet delivery, not even with probability 1.

1 Introduction

The (data) link layer is the 2nd layer of the ISO/OSI model of computer networking [18]. Amongst others, it is responsible for the transfer of data between adjacent nodes in Wide Area Networks (WANs) and Local Area Networks (LANs).

Examples of link layer protocols are Ethernet for LANs [16], the Point-to-Point Protocol [24] and the High-Level Data Link Control protocol (e.g. [14]). Part of this layer are also multiple access protocols such as the Carrier-Sense Multiple Access with Collision Detection (CSMA/CD) protocol for re-transmission in Ethernet bus networks and hub networks, or the Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol [17,19] in wireless networks.

One of the unique characteristics of the link layer is that when devices attempt to use a medium simultaneously, *collisions of messages* occur. So, any modelling language and formal analysis of layer-2 protocols has to support such collisions. Moreover, some protocols are of probabilistic nature: CSMA/CA for example chooses time slots probabilistically with discrete uniform distribution.

As we are not aware of any formal framework with primitives for modelling data collisions, this paper introduces a process algebra for modelling and analysing link layer protocols. In Sect. 2 we present an algebra featuring a unique mechanism for modelling collisions, 'hard-wired' in the semantics. It is the non-probabilistic fragment of the Algebra for Link Layer protocols (ALL), which we

© The Author(s) 2019
L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 668–693, 2019.
https://doi.org/10.1007/978-3-030-17184-1_24

introduce in Sect. 3. In Sect. 4 we formulate *packet delivery*, a liveness property that ideally ought to hold for link layer protocols, either outright, or with a high probability. In Sect. 5 we use this framework to formally model and analyse the CSMA/CA protocol.

Our analysis confirms the hidden station problem for the version of CSMA/CA without virtual carrier sensing (Sect. 5.2). However, we also show that the version with virtual carrier sensing overcomes not only this problem, but also the exposed station problem with probability 1. Yet the protocol cannot guarantee packet delivery, not even with probability 1.

2 A Non-probabilistic Subalgebra

In this section we propose a timed process algebra that can model the collision of link layer messages, called *frames*.[1] It can be used for link layer protocols that do not feature probabilistic choice, and is inspired by the (Timed) Algebra for Wireless Networks ((T-)AWN) [2,12,13], a process algebra suitable for modelling and analysing protocols on layers 3 (network) and 4 (transport) of the OSI model.

The process algebra models a (wired or wireless) network as an encapsulated parallel composition of network nodes. Due to the nature of the protocols under consideration, on each node exactly one sequential process is running. The algebra features a discrete model of time, where each sequential process maintains a local variable `now` holding its local clock value—an integer. We employ only one clock for each sequential process. All sequential processes in a network synchronise in taking time steps, and at each time step all local clocks advance by one unit. Since this means that all clocks are in sync and do not run at different speeds it is clear that we do not consider the problem of clock shift. For the rest, the variable `now` behaves like any other variable maintained by a process: its value can be read when evaluating guards, thereby making progress time-dependant, and any value can be assigned to it, thereby resetting the local clock. Network nodes communicate with their direct neighbours—those nodes that are in transmission range. The algebra provides a mobility option that allows nodes to 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 (the higher OSI layers).

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. Predicate logic yields terms (or *data expressions*) and formulas

[1] As it is the nonprobabilistic fragment of a forthcoming algebra we do not name it.

to denote data values and statements about them. Our data structure always contains the types $\mathtt{TIME}$, $\mathtt{DATA}$, $\mathtt{MSG}$, $\mathtt{CHUNK}$, $\mathtt{ID}$ and $\mathscr{P}(\mathtt{ID})$ of discrete *time values*, which we take to be integers, *network layer data*, *messages*, *chunks* of messages that take one time unit to transmit, *node identifiers* and *sets of node identifiers*. We further assume that there are variables $\mathtt{now}$ of type $\mathtt{TIME}$ and $\mathtt{rfr}$ of type $\mathtt{CHUNK}$. 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 $n \in \mathbb{N}$, $\mathtt{var}_i$ 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*.[2] 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:

$$P ::= X(exp_1, \ldots, exp_n) \ \mid\ [\varphi]P \ \mid\ [\![\mathtt{var} := exp]\!]P \ \mid\ \alpha.P \ \mid\ P + P$$
$$\alpha ::= \mathbf{transmit}(ms) \ \mid\ \mathbf{newpkt}(\mathtt{data}, \mathtt{dest}) \ \mid\ \mathbf{deliver}(\mathtt{data})$$

Here X is a process name, exp_i a data expression of the same type as $\mathtt{var}_i$, φ a data formula, $\mathtt{var} := exp$ an assignment of a data expression exp to a variable $\mathtt{var}$ of the same type, ms a data expression of type $\mathtt{MSG}$, and $\mathtt{data}$, $\mathtt{dest}$ data variables of types $\mathtt{DATA}$, $\mathtt{ID}$ respectively.

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 process $[\![\mathtt{var} := exp]\!]P$ acts as P, but under an updated valuation of the data variable $\mathtt{var}$. The 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 process $\alpha.P$ first performs the action α and subsequently acts as P. The above behaviour is identical to AWN, and many other standard process algebras. The action $\mathbf{transmit}(ms)$ transmits (the data value bound to the expression) ms to all other network nodes within transmission range. The action $\mathbf{newpkt}(\mathtt{data}, \mathtt{dest})$ models the injection by the network layer of a data packet $\mathtt{data}$ to be transmitted to a destination $\mathtt{dest}$. Technically, $\mathtt{data}$ and $\mathtt{dest}$ are variables that will be bound to the obtained values upon receipt of a **newpkt**. Data is delivered to the network layer by $\mathbf{deliver}(\mathtt{data})$. In contrast to AWN, we do not have a primitive for

[2] An occurrence of a data variable in P is *bound* if it is one of the variables $\mathtt{var}_i$, one of the two special variables $\mathtt{now}$ or $\mathtt{rfr}$, a variable $\mathtt{var}$ occurring in a subexpression $[\![\mathtt{var} := exp]\!]Q$, an occurrence in a subexpression $[\varphi]Q$ of a variable occurring free in φ, or a variable $\mathtt{data}$ or $\mathtt{dest}$ occurring in a subexpression $\mathbf{newpkt}(\mathtt{data}, \mathtt{dest}).Q$. Here Q is an arbitrary sequential process expression.

receiving messages from neighbouring nodes, because our processes are *always* listening to neighbouring nodes, in parallel with anything else they do.

As in AWN, the internal state of a sequential process described by an expression P is determined by P, together with a *valuation* ξ associating values $\xi(\texttt{var})$ 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. We denote the valuation that assigns the value v to the variable var, and agrees with ξ on all other variables, by $\xi[\texttt{var} := v]$. The valuation $\xi_{|S}$ agrees with ξ on all variables $\texttt{var} \in S$ and is undefined otherwise. Moreover we use $\xi[\texttt{var}\,\texttt{++}]$ as an abbreviation for $\xi[\texttt{var} := \xi(\texttt{var}) + 1]$, for suitable types.

To capture the durational nature of transmitting a message between network nodes, we model a message as a sequence of *chunks*, each of which takes one time unit to transmit. The function $\texttt{dur}: \texttt{MSG} \rightarrow \texttt{TIME}_{>0}$ calculates the amount of time steps needed for a sending a message, i.e. it calculates the number of chunks. We employ the internal data type $\texttt{CHUNK} := \{m{:}c \mid m \in \texttt{MSG}, 1 \leq c \leq \texttt{dur}(m)\} \cup \{\underline{\texttt{conflict}}, \underline{\texttt{idle}}\}$. The chunk $m{:}c$ indicates the cth fragment of a message m. Data conflicts—junk transmitted via the medium—is modelled by the special chunk $\underline{\texttt{conflict}}$, and the absence of an incoming chunk is modelled by $\underline{\texttt{idle}}$.

Our process algebra maintains a variable rfr of type CHUNK, storing the fragment of the current message received so far.
As a value of this variable, $m{:}c$ indicates that the first c chunks of message m have been received in order; $\underline{\texttt{conflict}}$ indicates that the last incoming chunk was not the expected (next) part of a message in progress, and $\underline{\texttt{idle}}$ indicates that the channel was idle during the last time step. The table on the right, with $*$ a wild card, shows how the value of rfr evolves upon receiving a new chunk *ch*.

rfr	*ch*	rfr $\star$ *ch*
$*$	$\underline{\texttt{conflict}}$	$\underline{\texttt{conflict}}$
$*$	$\underline{\texttt{idle}}$	$\underline{\texttt{idle}}$
$*$	$m{:}1$	$m{:}1$
$m{:}c$	$m{:}c{+}1$	$m{:}c{+}1$
rfr	$m{:}c{+}1$	$\underline{\texttt{conflict}}$ if $\mathit{rfr} \neq m{:}c$

Specifications may refer to the data type CHUNK only through the Boolean functions NEW—having a single argument *msg* of type MSG—and IDLE, defined by $\text{NEW}(\mathit{msg}) := (\texttt{rfr} = (\mathit{msg} : \texttt{dur}(\mathit{msg}))$ and $\text{IDLE} := (\texttt{rfr} = \underline{\texttt{idle}})$. A guard $[\text{NEW}(\mathit{msg})]$ evaluates to true iff a new message *msg* has just been received; [IDLE] evaluates to true iff in the last time slice the medium was idle.

The structural operational semantics of Table 1 describes how one internal state can evolve into another by performing an *action*. The set Act of actions consists of **transmit**$(m{:}c, \mathit{ch})$, **wait**(ch), **newpkt**(d, dest), **deliver**(d), and internal actions τ, for each choice of $m \in \texttt{MSG}$, $c \in \{1, \ldots, \texttt{dur}(m)\}$, $\mathit{ch} \in \texttt{CHUNK}$, $d \in \texttt{DATA}$ and $\mathit{dest} \in \texttt{ID}$, where the first two actions are time consuming. On every time-consuming action, each process receives a chunk *ch* and updates the variable rfr accordingly; moreover, the variable now is incremented on all process expressions in a (complete) network synchronously.

Besides the special variables now and rfr, the formal semantics employs an internal variable $\texttt{cntr} \in \mathbb{N}$ that enumerates the chunks of split messages and is

Table 1. Structural operational semantics for sequential process expressions

(1) $$\xi, \mathbf{transmit}(ms).P \xrightarrow{\mathbf{transmit}(\xi(ms):\texttt{c+},ch)} \xi\begin{bmatrix}\texttt{cntr}\,\texttt{++}\\ \texttt{rfr} := \texttt{rfr} \star ch\\ \texttt{now}\,\texttt{++}\end{bmatrix}, \mathbf{transmit}(\xi(ms)).P \qquad \begin{array}{r}(\text{if } \texttt{c+} < \texttt{dur}(\xi(ms)))\\ (\forall ch \in \texttt{CHUNK})\end{array}$$

(2) $$\xi, \mathbf{transmit}(ms).P \xrightarrow{\mathbf{transmit}(\xi(ms):\texttt{c+},ch)} \xi\begin{bmatrix}\texttt{cntr} := 0\\ \texttt{rfr} := \texttt{rfr} \star ch\\ \texttt{now}\,\texttt{++}\end{bmatrix}, P \qquad \begin{array}{r}(\text{if } \texttt{c+} = \texttt{dur}(\xi(ms)))\\ (\forall ch \in \texttt{CHUNK})\end{array}$$

(3) $$\xi, \mathbf{newpkt}(\texttt{data},\texttt{dest}).P \xrightarrow{\mathbf{newpkt}(d,dest)} \xi\begin{bmatrix}\texttt{data} := d\\ \texttt{dest} := dest\end{bmatrix}, P \qquad (\forall d \in \texttt{DATA},\ dest \in \texttt{ID})$$

(4) $$\xi, \mathbf{newpkt}(\texttt{data},\texttt{dest}).P \xrightarrow{\mathbf{wait}(ch)} \xi\begin{bmatrix}\texttt{rfr} := \texttt{rfr} \star ch\\ \texttt{now}\,\texttt{++}\end{bmatrix}, \mathbf{newpkt}(\texttt{data},\texttt{dest}).P \qquad (\forall ch \in \texttt{CHUNK})$$

(5) $$\xi, \mathbf{deliver}(\texttt{data}).P \xrightarrow{\mathbf{deliver}(\xi(\texttt{data}))} \xi, P$$

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

(7) $$\frac{\xi_{|\texttt{RO}}[\texttt{var}_i := \xi(exp_i)]_{i=1}^{n}, P \xrightarrow{a} \zeta, P'}{\xi, X(exp_1,\ldots,exp_n) \xrightarrow{a} \zeta, P'}\ \left(X(\texttt{var}_1,\ldots,\texttt{var}_n) \stackrel{def}{=} P\right) \qquad (\forall a \in \text{Act} - \{\mathbf{wait}(ch) \mid ch \in \texttt{CHUNK}\})$$

(8) $$\frac{\xi_{|\texttt{RO}}[\texttt{var}_i := \xi(exp_i)]_{i=1}^{n}, P \xrightarrow{\mathbf{wait}(ch)} \zeta, P'}{\xi, X(exp_1,\ldots,exp_n) \xrightarrow{\mathbf{wait}(ch)} \xi\begin{bmatrix}\texttt{rfr} := \texttt{rfr} \star ch\\ \texttt{now}\,\texttt{++}\end{bmatrix}, X(exp_1,\ldots,exp_n)}\ \left(X(\texttt{var}_1,\ldots,\texttt{var}_n) \stackrel{def}{=} P\right) \qquad (\forall ch \in \texttt{CHUNK})$$

(9) $$\xi, P \xrightarrow{\mathbf{wait}(ch)} \xi\begin{bmatrix}\texttt{rfr} := \texttt{rfr} \star ch\\ \texttt{now}\,\texttt{++}\end{bmatrix}, P \qquad \begin{array}{r}(\text{if } \xi(P)\uparrow)\\ (\forall ch \in \texttt{CHUNK})\end{array}$$

(10) $$\frac{\xi, P \xrightarrow{a} \zeta, P'}{\xi, P+Q \xrightarrow{a} \zeta, P'} \qquad \frac{\xi, Q \xrightarrow{a} \zeta, Q'}{\xi, P+Q \xrightarrow{a} \zeta, Q'} \qquad (\forall a \in \text{Act} - \{\mathbf{wait}(ch) \mid ch \in \texttt{CHUNK}\})$$

(11) $$\frac{\xi, P \xrightarrow{\mathbf{wait}(ch)} \zeta, P' \quad \xi, Q \xrightarrow{\mathbf{wait}(ch)} \zeta', Q'}{\xi, P+Q \xrightarrow{\mathbf{wait}(ch)} \zeta, P'+Q'} \qquad (\forall ch \in \texttt{CHUNK})$$

(12) $$\frac{\xi \stackrel{\varphi}{\to} \zeta}{\xi, [\varphi]P \xrightarrow{\tau} \zeta, P} \qquad \frac{\xi \stackrel{\varphi}{\not\to}}{\xi, [\varphi]P \xrightarrow{\mathbf{wait}(ch)} \xi\begin{bmatrix}\texttt{rfr} := \texttt{rfr} \star ch\\ \texttt{now}\,\texttt{++}\end{bmatrix}, [\varphi]P} \qquad (\forall ch \in \texttt{CHUNK})$$

used to identify which chunk needs to be sent next. The variables now, rfr and cntr are not meant to be changed by ALL specifications, e.g. by using assignments. We call them read-only and collect them in the set $\texttt{RO} = \{\texttt{now}, \texttt{rfr}, \texttt{cntr}\}$.

Let us have a closer look at the rules of Table 1.

The first two rules describe the sending of a message *ms*. Remember that $\texttt{dur}(ms)$ calculates the time needed to send *ms*. The counter cntr keeps track of the time passed already. The action **transmit**$(m{:}c, ch)$ occurs when the node transmits the fragment $m{:}c$; simultaneously, it receives the fragment *ch*.[3] The counter cntr is 0 before a message is sent, and is incremented before the transmission of each chunk. So, each chunk sent has the form $\xi(ms){:}\xi(\texttt{cntr}){+}1$. To ease readability we abbreviate $\xi(\texttt{cntr}){+}1$ by c+. In case the (already incremented) counter c+ is strictly smaller than the number of chunks needed to send $\xi(ms)$, another **transmit**-action is needed (Rule 1); if the last fragment has been sent ($\texttt{c+} = \texttt{dur}(\xi(ms))$) the process can continue to act as P (Rule 2).

The actions **newpkt**$(d, dest)$ and **deliver**(d) are instantaneous and model the submission of data d from the network layer, destined for *dest*, and the delivery of data d to the network layer, respectively. The process **newpkt**$(d, dest).P$ has also the possibility to wait, namely if no network layer instruction arrives.

Rule 6 defines a rule for assignment in a straightforward fashion; only the valuation of the variable var is updated.

In Rules 7 and 8, which define recursion, $\xi_{|\texttt{RO}}[\texttt{var}_i := \xi(exp_i)]_{i=1}^{n}$ is the valuation that *only* assigns the values $\xi(exp_i)$ to the variables $\texttt{var}_i$, for $i = 1, \ldots, n$, and maintains the values of the variables now, rfr and cntr. These rules state that a defined process X has the same transitions as the body p of its defining equation. In case of a **wait**-transition, the sequential process does not progress, and accordingly the recursion is not yet unfolded.

Most transition rules so far feature statements of the form $\xi(exp)$ where *exp* is a data expression. The application of the rule depends on $\xi(exp)$ being defined. Rule 9 covers all cases where the above rules cannot be applied since at least one data expression in an action α is not defined. A state ξ, P is *unvalued*, denoted by $\xi(p)\!\uparrow$, if P has the form **transmit**$(ms).P$, **deliver**$(\texttt{data}).P$, $[\![\texttt{var} := exp]\!]P$ or $X(exp_1, \ldots, exp_n)$ with either $\xi(ms)$ or $\xi(\texttt{data})$ or $\xi(exp)$ or some $\xi(exp_i)$ undefined. From such a state the process can merely wait.

A process $P + Q$ can wait *only* if both P and Q can do the same; if either P or Q can achieve 'proper' progress, the choice process $P + Q$ always chooses progress over waiting. A simple induction shows that if $\xi, P \xrightarrow{\textbf{wait}(ch)} \zeta, P'$ and $\xi, Q \xrightarrow{\textbf{wait}(ch)} \zeta', Q'$ then $P = P'$, $Q = Q'$ and $\zeta = \zeta'$.

The first rule of (12), describing the semantics of guards $[\varphi]$, is taken from AWN. Here $\xi \xrightarrow{\varphi} \zeta$ 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. Its negation $\xi \stackrel{\varphi}{\not\rightarrow}$ says

[3] Normally, a node is in its own transmission range. In that case the received chunk *ch* will be either the chunk $m{:}c$ it is transmitting itself, or $\underline{\texttt{conflict}}$ in case some other node within transmission range is transmitting as well.

that no such extension exists, and thus, that φ is false in the current state, no matter how we interpret the variables whose values are still undefined. If that is the case, the process $[\varphi]p$ will idle by performing the action $\mathbf{wait}(ch)$.

2.2 A Language for Node Expressions

We model network nodes in the context of a (wireless) network by *node expressions* of the form

$$id{:}(\xi, P){:}R\,.$$

Here $id \in \mathtt{ID}$ is the *address* of the node, P is a sequential process expression with a valuation ξ, and $R \in \mathscr{P}(\mathtt{ID})$ is the *range* of the node, defined as the set of nodes within transmission range of id. Unlike AWN, the process algebra does not offer a parallel operator for combining sequential processes; such an operator is not needed due to the nature of link layer protocols.

In the semantics of this layer it is crucial to handle frame collisions. The idea is that all chunks sent are recorded, together with the respective recipient. In case a node receives more than one chunk at a time, a conflict is raised, as it is impossible to send two or more messages via the same medium at the same time.

The formal semantics for node expressions, presented in Table 2, uses transition labels $\mathbf{traffic}(\mathcal{T},\mathcal{R})$, $id{:}\mathbf{deliver}(d)$, $id{:}\mathbf{newpkt}(d, id')$, $\mathbf{connect}(id, id')$, $\mathbf{disconnect}(id, id')$ and τ, with partial functions $\mathcal{T},\mathcal{R} : \mathtt{ID} \rightharpoonup \mathtt{CHUNK}$, $id, id' \in \mathtt{ID}$, and $d \in \mathtt{DATA}$.

Table 2. Structural operational semantics for node expressions

$$\frac{P \xrightarrow{\mathbf{wait}(\underline{\mathtt{idle}})} P'}{id{:}P{:}R \xrightarrow{\mathbf{traffic}(\emptyset,\emptyset)} id{:}P'{:}R} \qquad \frac{P \xrightarrow{\mathbf{transmit}(m:c,\underline{\mathtt{idle}})} P'}{id{:}P{:}R \xrightarrow{\mathbf{traffic}(\{(r,\,m:c)\,|\,r\in R\},\emptyset)} id{:}P'{:}R}$$

$$\frac{P \xrightarrow{\mathbf{wait}(ch)} \quad (ch \neq \underline{\mathtt{idle}})}{id{:}P{:}R \xrightarrow{\mathbf{traffic}(\emptyset,\{(id,\,ch)\})} id{:}P'{:}R} \qquad \frac{P \xrightarrow{\mathbf{transmit}(m:c,ch)} P' \quad (ch \neq \underline{\mathtt{idle}})}{id{:}P{:}R \xrightarrow{\mathbf{traffic}(\{(r,\,m:c)\,|\,r\in R\},\{(id,\,ch)\})} id{:}P'{:}R}$$

$$\frac{P \xrightarrow{\mathbf{deliver}(d)} P'}{id{:}P{:}R \xrightarrow{id\,:\,\mathbf{deliver}(d)} id{:}P'{:}R} \qquad \frac{P \xrightarrow{\mathbf{newpkt}(d,dest)} P'}{id{:}P{:}R \xrightarrow{id\,:\,\mathbf{newpkt}(d,dest)} id{:}P'{:}R} \qquad \frac{P \xrightarrow{\tau} P'}{id{:}P{:}R \xrightarrow{\tau} id{:}P'{:}R}$$

$$id{:}P{:}R \xrightarrow{\mathbf{connect}(id,id')} id{:}P{:}R \cup \{id'\} \qquad id{:}P{:}R \xrightarrow{\mathbf{disconnect}(id,id')} id{:}P{:}R - \{id'\}$$

$$id{:}P{:}R \xrightarrow{\mathbf{connect}(id',id)} id{:}P{:}R \cup \{id'\} \qquad id{:}P{:}R \xrightarrow{\mathbf{disconnect}(id',id)} id{:}P{:}R - \{id'\}$$

$$\frac{id \notin \{id', id''\}}{id{:}P{:}R \xrightarrow{\mathbf{connect}(id',id'')} id{:}P{:}R} \qquad \frac{id \notin \{id', id''\}}{id{:}P{:}R \xrightarrow{\mathbf{disconnect}(id',id'')} id{:}P{:}R}$$

All time-consuming actions on process level ($\mathbf{transmit}(m{:}c, ch)$ and $\mathbf{wait}(ch)$) are transformed into an action $\mathbf{traffic}(\mathcal{T},\mathcal{R})$ on node level: the first argument

Table 3. Structural operational semantics for network expressions

$$\frac{M \xrightarrow{a} M'}{M \| N \xrightarrow{a} M' \| N} \qquad \frac{N \xrightarrow{a} N'}{M \| N \xrightarrow{a} M \| N'} \qquad \frac{M \xrightarrow{a} M'}{[M] \xrightarrow{a} [M']} \qquad \left(\forall a \in \begin{Bmatrix} \tau,\, id\!:\!\textbf{deliver}(d), \\ id\!:\!\textbf{newpkt}(d, id), \end{Bmatrix}\right)$$

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

$$\frac{M \xrightarrow{\textbf{traffic}(\mathcal{T}_1,\mathcal{R}_1)} M' \quad N \xrightarrow{\textbf{traffic}(\mathcal{T}_2,\mathcal{R}_2)} N'}{M \| N \xrightarrow{\textbf{traffic}(\mathcal{T}_1 \uplus \mathcal{T}_2,\mathcal{R}_1 \uplus \mathcal{R}_2)} M' \| N'} \qquad \frac{M \xrightarrow{\textbf{traffic}(\mathcal{R},\mathcal{R})} M'}{[M] \xrightarrow{\textbf{tick}} [M']}$$

$\mathcal{T}$ maps *dest* to $m{:}c$ if and only if the chunk $m{:}c$ is transmitted to *dest*. The second argument $\mathcal{R}$ maps *id* to $m{:}c$ if and only if the chunk $m{:}c$ is received on process level at node *id*. For the sos-rules of Table 2 we use the set-theoretic presentation of partial functions. The two rules for **wait** set $\mathcal{T} := \emptyset$, as no chunks are transmitted; the rules for **transmit** allow a transmitted chunk $m{:}c$ to travel to all nodes within transmission range: $\mathcal{T} := \{(r, m{:}c) \mid r \in R\}$. In case that during the transmission or waiting no chunk is received ($ch = \underline{\texttt{idle}}$) we set $\mathcal{R} = \emptyset$; otherwise $\mathcal{R} = \{(id, ch)\}$, indicating that chunk *ch* is received by node *id*.

The actions $id\!:\!\textbf{newpkt}(d, dest)$ and $id\!:\!\textbf{deliver}(d)$ as well as the internal actions τ are simply inherited by node expressions from the processes that run on these nodes.

The remaining rules of Table 2 model the mobility aspect of wireless networks; the rules are taken straight from AWN [12,13]. We allow actions $\textbf{connect}(id, id')$ and $\textbf{disconnect}(id, id')$ for $id, id' \in \texttt{ID}$ 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 id' is in the range of node *id*, meaning that id' can receive messages sent by *id*, if and only if *id* is in the range of id'. To break this symmetry, one just skips the last four rules of Table 2 and replaces the synchronisation rules for **connect** and **disconnect** in Table 3 by interleaving rules (like the ones for **deliver**, **newpkt** and τ) [12]. For some applications a wired or non-mobile network need to be considered. In such cases the last six rules of Table 2 are dropped.

Whether a node $id\!:\!P\!:\!R$ receives its own transmissions depends on whether $id \in R$. Only if $id \in R$ our process algebra will disallow the transmission from and to a single node *id* at the same time, yielding a $\underline{\texttt{conflict}}$.

2.3 A Language for Networks

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

The syntax of networks is described by the following grammar:

$$N ::= [M^T_T] \qquad M^T_{S_1 \dot\cup S_2} ::= M^T_{S_1} \| M^T_{S_2} \qquad M^T_{\{id\}} ::= id\!:\!(\xi, P)\!:\!R\ ,$$

with $\{id\} \cup R \subseteq T \subseteq \mathtt{ID}$. Here M^T_S models a partial network describing the behaviour of all nodes $id \in S$. The set T contains the identifiers of all nodes that are part of the complete network. This grammar guarantees that node identifiers of node expressions—the first component of $id\!:\!P\!:\!R$—are unique.

The operational semantics of network expressions is given in Table 3. Internal actions τ as well as the actions $id\!:\!\mathbf{deliver}(d)$ and $id\!:\!\mathbf{newpkt}(d,id)$ are interleaved in the parallel composition of nodes that makes up a network, and then lifted to encapsulated networks (Line 1 of Table 3).

Actions **traffic** and (**dis**)**connect** are synchronised. The rule for synchronising the action **traffic** (Line 3), the only action that consumes time on the network layer, uses the union $\uplus$ of partial functions. It is formally defined as

$$(\mathcal{R}_1 \uplus \mathcal{R}_2)(id) := \begin{cases} \underline{\mathtt{conflict}} & \text{if } id \in \mathtt{dom}(\mathcal{R}_1) \cap \mathtt{dom}(\mathcal{R}_2) \\ \mathcal{R}_1(id) & \text{if } id \in \mathtt{dom}(\mathcal{R}_1) - \mathtt{dom}(\mathcal{R}_2) \\ \mathcal{R}_2(id) & \text{if } id \in \mathtt{dom}(\mathcal{R}_2) - \mathtt{dom}(\mathcal{R}_1)\ . \end{cases}$$

The synchronisation of the sets $\mathcal{R}_i$ and $\mathcal{T}_i$ has the following intuition: if a node identifier $id \in \mathtt{ID}$ is in both $\mathtt{dom}(\mathcal{T}_1)$ and $\mathtt{dom}(\mathcal{T}_2)$ then there exist two nodes that transmit to node id at the same time, and therefore a frame collision occurs. In our algebra this is modelled by the special chunk $\underline{\mathtt{conflict}}$. The sos rules of Tables 2 and 3 guarantee that there cannot be collisions within the set of received chunks $\mathcal{R}$. The reason is that each node merely contributes to $\mathcal{R}$ a chunk for itself; it can be the chunk $\underline{\mathtt{conflict}}$ though. Therefore we could have written $\mathcal{R}_1 \cup \mathcal{R}_2$ instead of $\mathcal{R}_1 \uplus \mathcal{R}_2$ in the sixth rule of Table 3.

The last rule propagates a $\mathbf{traffic}(\mathcal{T}, \mathcal{R})$-action of a partial network M to a complete network $[M]$. By then $\mathcal{T}$ consists of all chunks (after collision detection) that are being transmitted by any member in the network, and $\mathcal{R}$ consists of all chunks that are received. The condition $\mathcal{R} = \mathcal{T}$ determines the content of the messages in $\mathcal{R}$. The $\mathbf{traffic}(\mathcal{T}, \mathcal{R})$-actions become internal at this level, as they cannot be steered by the outside world; all that is left is a time-step **tick**.

2.4 Results on the Process Algebra

As for the process algebra T-AWN [2], but with a slightly simplified proof, one can show that our processes have no *time deadlocks*:

Theorem 2.1. *A complete network N in our process algebra always admits a transition, independently of the outside environment, i.e. $\forall N, \exists a$ such that $N \xrightarrow{a}$ and $a \notin \{\boldsymbol{connect}(id, id'), \boldsymbol{disconnect}(id, id'), id\!:\!\mathtt{newpkt}(d, dest)\}$.*
More precisely, either $N \xrightarrow{\mathbf{tick}}$, or $N \xrightarrow{id:\,\boldsymbol{deliver}(d)}$ or $N \xrightarrow{\tau}$.

The following results (statements and proofs) are very similar to the results about the process algebra AWN, as presented in [13]. A rich body of foundational

meta theory of process algebra allows the transfer of the results to our setting, without too much overhead work.

Identical to AWN and its timed version T-AWN, our process algebra admits a translation into one without data structures (although we cannot describe the target algebra without using data structures). The idea is to replace any variable by all possible values it can take. The target algebra differs from the original only on the level of sequential processes; the subsequent layers are unchanged. The construction closely follows the one given in the appendix of [2]. The inductive definition contains the rules

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

$\mathscr{T}_\xi(\llbracket\texttt{var} := \mathit{exp}\rrbracket P) = \tau.\mathscr{T}_{\xi[\texttt{var} := \xi(\mathit{exp})]}(P)$.

Most other rules require extra operators that keep track of the passage of time and the evolution of other internal variables. The resulting process algebra has a structural operational semantics in the (infinitary) *de Simone* format, generating the same transition system—up to strong bisimilarity, $\leftrightarroweq$ —as the original. It follows that $\leftrightarroweq$, and many other semantic equivalences, are congruences on our language [23].

Theorem 2.2. *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. [25]). 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 [23]. □

Theorem 2.3. *The operator $\|$ is associative and commutative, up to $\leftrightarroweq$.*

Proof. The operational rules for this operator fits a format presented in [6], guaranteeing associativity up to $\leftrightarroweq$. The *A*SSOC-de Simone format of [6] 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 four rules for $\|$ of Table 3 are of types 1, 2 and 7, respectively. To be precise, it has rules 1_a and 2_a for $a \in \{\tau, \mathit{id}\colon\mathbf{deliver}(d), \mathit{id}\colon\mathbf{newpkt}(d, \mathit{dest})\}$, rules $7_{(a,b)}$ for

$$(a,b) \in \{(\mathbf{traffic}(\mathcal{T}_1,\mathcal{R}_1),\mathbf{traffic}(\mathcal{T}_2,\mathcal{R}_2)) \mid \mathcal{R}_1,\mathcal{R}_2,\mathcal{T}_1,\mathcal{T}_2 \in \texttt{ID} \rightharpoonup \texttt{CHUNK}\}$$

and rules $7_{(c,c)}$ for $c \in \{\mathbf{connect}(\mathit{id},\mathit{id}'),\mathbf{disconnect}(\mathit{id},\mathit{id}') \mid \mathit{id},\mathit{id}' \in \texttt{ID}\}$. Moreover, the partial *communication function* $\gamma : \mathrm{Act} \times \mathrm{Act} \rightharpoonup \mathrm{Act}$ is given by $\gamma(\mathbf{traffic}(\mathcal{T}_1,\mathcal{R}_1),\mathbf{traffic}(\mathcal{T}_2,\mathcal{R}_2)) = \mathbf{traffic}(\mathcal{T}_1 \uplus \mathcal{T}_2, \mathcal{R}_1 \uplus \mathcal{R}_2)$ and $\gamma(c,c) = c$. The main result of [6] 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 trivially met for $\|$ as there neither exists a rule of the form $1_{\textbf{traffic}(\mathcal{T},\mathcal{R})}$ nor of the form $2_{\textbf{traffic}(\mathcal{T},\mathcal{R})}$, or 1_c, 2_c with c as above. As on **traffic** actions γ is basically the union of partial functions ($\uplus$), where a collision in domains is indicated by an error `conflict`, it is straightforward to prove associativity of γ.

Commutativity of $\|$ follows by symmetry of the sos rules. □

3 An Algebra for Link Layer Protocols

We now introduce ALL, the *Algebra for Link Layer protocols*. It is obtained from the process algebra presented in the previous section by the addition of a probabilistic choice operator $\bigoplus_0^n$. As a consequence, the semantics of the algebra is no longer a labelled transition system, but a *probabilistic labelled transition system* (pLTS) [8]. This is a triple $(S, \mathrm{Act}, \rightarrow)$, where

(i) S is a set of states
(ii) Act is a set of actions
(iii) $\rightarrow \subseteq S \times \mathrm{Act} \times \mathcal{D}(S)$, where $\mathcal{D}(S)$ is the set of all (discrete) probability distributions over S: functions $\Delta : S \rightarrow [0,1]$ with $\sum_{s\in S} \Delta(s) = 1$.

As with LTSs, we usually write $s \xrightarrow{\alpha} \Delta$ instead of $(s, \alpha, \Delta) \in \rightarrow$. The *point distribution* δ_s, for $s \in S$, is the distribution with $\delta_s(s) = 1$. We simply write $s \xrightarrow{\alpha} t$ for $s \xrightarrow{\alpha} \delta_t$. An LTS may be viewed as a degenerate pLTS, in which only point distributions occur. For a uniform distribution over $s_0, \ldots, s_n \in S$ we write $\mathcal{U}_{i=0}^n s_i$. The pLTS associated to ALL takes S to be the disjoint union of the pairs ξ, P, with P a sequential process expression, and the network expressions. Act is the collection of transition labels, and $\rightarrow$ consists of the transitions derivable from the structural operational semantics of the language.

Rules (1)–(6), (9), (11) and (12) of Table 1 are adopted to ALL unchanged, whereas in Rules (7), (8) and (10) the state ζ, P' (or ζ, Q') is replaced by an arbitrary distribution Δ. Add to those the following rule for the probabilistic choice operator:

$$\xi, \bigoplus_{\mathtt{i}=0}^{n} P \xrightarrow{\tau} \mathcal{U}_{i=0}^{\xi(n)}\ \xi[\mathtt{i} := i]\ , P$$

Here the data variable i may occur in P. The rules of Tables 2 and 3 are adapted to ALL unchanged, except that P', M' and N' are now replaced by arbitrary distributions over sequential processes and network expressions, respectively. Here we adapt the convention that a unary or binary operation on states lifts to distributions in the standard manner. For example, if Δ is a distribution over sequential processes, $id \in \mathtt{ID}$ and $R \subseteq \mathtt{ID}$, then $id{:}\Delta{:}R$ describes the distribution over node expressions that only has probability mass on nodes with address id and range R, and for which the probability of $id{:}P{:}R$ is $\Delta(P)$. Likewise, if Δ and Θ are distributions over network expressions, then $\Delta\|\Theta$ is the distribution over network expressions of the form $M\|N$, where $(\Delta\|\Theta)(M\|N) = \Delta(M) \cdot \Theta(N)$.

4 Formalising Liveness Properties of Link Layer Protocols

Link layer protocols communicate with the network layer through the actions $id\colon\mathbf{newpkt}(d, dest)$ and $id\colon\mathbf{deliver}(d)$. The typical liveness property expected of a link layer protocol is that if the network layer at node id injects a data packet d for delivery at destination $dest$ then this packet is delivered eventually. In terms of our process algebra, this says that every execution of the action $id\colon\mathbf{newpkt}(d, dest)$ ought to be followed by the action $dest\colon\mathbf{deliver}(d)$. This property can be formalised in Linear-time Temporal Logic [22] as

$$\mathbf{G}\big(id\colon\mathbf{newpkt}(d, dest) \Rightarrow \mathbf{F}(dest\colon\mathbf{deliver}(d))\big) \tag{1}$$

for any $id, dest \in \mathtt{ID}$ and $d \in \mathtt{DATA}$. This formula has the shape $\mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}\phi^{post}\big)$, and is called an *eventuality property* in [22]. It says that whenever we reach a state in which the precondition ϕ^{pre} is satisfied, this state will surely be followed by a state were the postcondition ϕ^{post} holds. In [7,13] it is explained how action occurrences can be seen or encoded as state-based conditions. Here we will not define how to interpret general LTL-formula in pLTSs, but below we do this for eventuality properties with specific choices of ϕ^{pre} and ϕ^{post}.

Formula (1) is too strong and does not hold in general: in case the nodes id and $dest$ are not within transmission range of each other, the delivery of messages from id to $dest$ is doomed to fail. We need to postulate two side conditions to make this liveness property plausible. Firstly, when the request to deliver the message comes in, id needs to be connected to $dest$. We introduce the predicate $\mathbf{cntd}(id, dest)$ to express this, and hence take ϕ^{pre} to be $\mathbf{cntd}(id, dest) \wedge id\colon\mathbf{newpkt}(d, dest)$. Secondly, we assume that the link between id and $dest$ does not break until the message is delivered. As remarked in [13], such a side condition can be formalised by taking ϕ^{post} to be $dest\colon\mathbf{deliver}(d) \vee \mathbf{disconnect}(id, dest)$. Thus the liveness property we are after is

$$\begin{gathered}\mathbf{G}\big(\mathbf{cntd}(id, dest) \wedge id\colon\mathbf{newpkt}(d, dest) \Rightarrow \\ \mathbf{F}(dest\colon\mathbf{deliver}(d) \vee \mathbf{disconnect}(id, dest) \vee \mathbf{disconnect}(dest, id))\big)\end{gathered} \tag{2}$$

We now define the validity of eventuality properties $\mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}\phi^{post}\big)$. Here ϕ^{pre} and ϕ^{post} denote sets of transitions and actions, respectively, and hold if one of the transitions or actions in the set occurs. In (2), ϕ^{pre} denotes the transitions with label $id\colon\mathbf{newpkt}(d, dest)$ that occur when the side condition $\mathbf{cntd}(id, dest)$ is met, whereas $\phi^{post} = \{dest\colon\mathbf{deliver}(d), \mathbf{disconnect}(id, dest), \mathbf{disconnect}(dest, id)\}$ is a set of actions.

A *path* in a pLTS $(S, \mathrm{Act}, \rightarrow)$ is an alternating sequence $s_0, \alpha_1, s_1, \alpha_2, \ldots$ of states and actions, starting with a state and either being infinite or ending with a state, such that there is a transition $s_i \xrightarrow{\alpha_{i+1}} \Delta_{i+1}$ with $\Delta_{i+1}(s_{i+1}) > 0$ for each i. The path is *rooted* if it starts with a state marked as 'initial', and *complete* if either it is infinite, or there is no transition starting from its last state. A state or transition is *reachable* if it occurs in a rooted path.

In a pLTS with an initial state, an eventually formula $\mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}\phi^{post}\big)$, with ϕ^{pre} and ϕ^{post} denoting sets of transitions and actions, *holds outright* if all complete paths starting with a reachable transition from ϕ^{pre} contain a transition with a label from ϕ^{post}.

Definitions 3 and 5 in [9] define the set of probabilities that a pLTS with an initial state will ever execute the action ω. One obtains a set of probabilities rather than a single probability due to the possibility of nondeterministic choice. This definition generalises to *sets* of actions ϕ^{post} (seen as disjunctions) by first renaming all actions in such a set into ω. It also generalises trivially to pLTSs with an *initial transition*. For t a transition in a pLTS, let $Prob(t, \phi^{post})$ be the infimum of the set of probabilities that the pLTS in which t is taken to be the initial transition will ever execute ϕ^{post}. Now in a pLTS with an initial state, an eventually formula $\mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}\phi^{post}\big)$ *holds with probability at least* p if for all reachable transitions t in ϕ^{pre} we have $Prob(t, \phi^{post}) \geq p$.

Possible correctness criteria for link layer protocols are that the liveness property (2) either holds outright, holds with probability 1, or at least holds with probability p for a sufficiently high value of p.

Sometimes we are content to establish that (2) holds under the additional assumptions that the network is stable until our packet is delivered, meaning that no links between any nodes are broken or established, and/or that the network layer refrains from injecting more packets. This is modelled by taking

$$\phi^{post} = \{dest\!:\mathbf{deliver}(d), \mathbf{disconnect}(*,*), \mathbf{connect}(*,*), \mathbf{newpkt}(*,*)\}. \tag{3}$$

We will refer to this version of (2) as the *weak packet delivery* property. *Packet delivery* is the strengthening without $\mathbf{newpkt}(*,*)$ in (3), i.e. not assuming that the network layer refrains from injecting more packets.

5 Modelling and Analysing the CSMA/CA Protocol

In this section we model two versions of the CSMA/CA protocol, using the process algebra ALL. Moreover, we briefly discuss some results we obtained while analysing these protocols.

The *Carrier-Sense Multiple Access* (CSMA) protocol is a media access control (MAC) protocol in which a node verifies the absence of other traffic before transmitting on a shared transmission medium. If a carrier is sensed, the node waits for the transmission in progress to end before initiating its own transmission. Using CSMA, multiple nodes may, in turn, send and receive on the same medium. Transmissions by one node are generally received by all other nodes connected to the medium.

The CSMA protocol with Collision Avoidance (CSMA/CA) [17,19][4] improves the performance of CSMA. If the transmission medium is sensed busy

[4] The primary medium access control (MAC) technique of IEEE 802.11 [19] is called *distributed coordination function* (DCF), which is a CSMA/CA protocol.

before transmission then the transmission is deferred for a *random* time interval. This interval reduces the likelihood that two or more nodes waiting to transmit will simultaneously begin transmission upon termination of the detected transmission. CSMA/CA is used, for example, in Wi-Fi.

It is well known that CSMA/CA suffers from the *hidden station problem* (see Sect. 5.2). To overcome this problem, CSMA/CA is often supplemented by the request-to-send/clear-to-send (RTS/CTS) handshaking [19]. This mechanism is known as the IEEE 802.11 RTS/CTS exchange, or *virtual carrier sensing*. While this extension reduces the amount of collisions, wireless 802.11 implementations do not typically implement RTS/CTS for all transmissions because the transmission overhead is too great for small data transfers.

We use the process algebra ALL to model both the CSMA/CA without and with virtual carrier sensing.

5.1 A Formal Model for CSMA/CA

Our formal specification of CSMA/CA consists of four short processes written in ALL. It is precise and free of ambiguities—one of the many advantages formal methods provide, in contrast to specifications written in English prose.

The syntax of ALL is intended to look like pseudo code, and it is our belief that the specification can easily be read and understood by software engineers, who may or may not have experience with process algebra.

As the underlying data structure of our model is straightforward, we do not present it explicitly, but introduce it while describing the different processes.

The basic process `CSMA`, depicted in Process 1, is the protocol's entry point.

Process 1. The Basic Routine

$\texttt{CSMA}(\texttt{id}) \stackrel{def}{=}$

1. $\textbf{newpkt}(\texttt{data},\texttt{dest}).\ \texttt{INIT}(\texttt{id},0,\texttt{dataframe}(\texttt{data},\texttt{id},\texttt{dest}))$
2. $+\ [\textsc{new}(\texttt{dataframe}(\texttt{data},\texttt{src},\texttt{id}))]\ \textbf{deliver}(\texttt{data})\ .$
3. $($
4. $\quad [\![\texttt{timeout} := \texttt{now} + \underline{\texttt{sifs}}]\!]\ [\texttt{now} \geq \texttt{timeout}]$
5. $\quad \textbf{transmit}(\texttt{ackframe}(\texttt{src}))\ .\ \texttt{CSMA}(\texttt{id})$
6. $)$

This process maintains a single data variable `id` in which it stores its own identity. It waits until either it receives a request from the network layer to transmit a packet `data` to destination `dest`, or it receives from another node in the network a CSMA message (data frame) destined for itself.

In case of a newly injected data packet (Line 1), the process `INIT` is called; this process (described below) initiates the sending of the message via the medium. When passing the message on to `INIT` we use a function $\texttt{dataframe} : \texttt{DATA} \times \texttt{ID} \times \texttt{ID} \rightarrow \texttt{MSG}$ that generates a message in a format used by the protocol: next to the header fields (from which we abstract) it contains the injected `data` as well as the designated receiver `dest` and the sender `id`—the current node.

In case of an incoming dataframe destined for this node (the third argument carrying the destination is id) (Line 2)—any other incoming message is ignored by this process—the data is handed over to the network layer (**deliver**(data)) followed by the transmission of an acknowledgement back to the sender of the message (src). CSMA/CA requires a short period of idling medium before sending the acknowledgement: in [19] this interval is called *short interframe space* (sifs). The process waits until the time of the interframe spacing has passed, and then transmits the acknowledgement. The acknowledgement sent is not always received by src, e.g. due to data collision; therefore src could send the same message again (see Process 4) and id could deliver the same data to the network layer again.

Process 2. Protocol Initialisation

```
INIT(id,tries,dframe) def=
 1. [tries ≤ max_retransmit]
 2.    [[cw := cwmin × 2^tries]]
 3.    ⊕_{b=0}^{cw−1} CCA(id,b,tries,dframe)      /* choose a backoff from {0,...,cw−1} */
 4. + [tries > max_retransmit]
 5.    deliver(channel_access_failure) . CSMA(id)
```

The process INIT (Process 2) initiates the sending of a message via the medium. Next to the variable id, which is maintained by all processes, it maintains the variable tries and dframe: tries stores the number of attempts already made to send message dframe. When the process is called the first time for a message dframe (Line 1 of Process 1) the value of tries is 0.

The constant max_retransmit specifies the maximum number of attempts the protocol is allowed to retransmit the same message. If the limit is not yet reached (Line 1) the message dframe is sent. As mentioned above, CSMA/CA defers messages for a *random* time interval to avoid collision. The node must start transmission within the contention window cw, a.k.a. backoff time. cw is calculated in Line 2; it increases exponentially.[5] After cw is determined, the process CCA is called, which performs the actual **transmit**-action. In case the maximum number of retransmits is reached (Line 4), the process notifies the network layer and restarts the protocol, awaiting new instructions from the application layer, or a new incoming message.

Process 3 takes care of the actual transmission of dframe. However, the protocol has a complicated procedure when to send this message.

First, the process senses the medium and awaits the point in time when it is idle (Line 6). In case, before this happens, it receives from another node in the network a CSMA message destined for itself (Line 1), this message is handled just as in Process 1, except that after acknowledging this message the protocol returns to Process 3.

[5] A typical value for cwmin is 16; it must satisfy cwmin > 0.

Process 3. Clear Channel Assessment With Physical Carrier Sense

```
CCA(id,b,tries,dframe) def=
 1. [NEW(dataframe(data,src,id))] deliver(data) .
 2.    (
 3.       [[timeout := now + sifs]] [now ≥ timeout]
 4.       transmit(ackframe(src)) . CCA(id,b,tries,dframe)
 5.    )
 6. + [IDLE]
 7.    [[timeout:=now+difs]]       /* start wait for duration difs */
 8.    (
 9.       [¬IDLE] CCA(id,b,tries,dframe)
10.       + [IDLE ∧ now ≥ timeout]
11.          [[timeout := now + b]]
12.          (
13.             [¬IDLE]       /* busy during backoff time */
14.                [[b := timeout − now]] CCA(id,b,tries,dframe)
15.             + [IDLE ∧ now ≥ timeout]       /* idle for backoff time */
16.                transmit(dframe) .
17.                ACKRECV(id,tries,now+max_ack_wait,dframe)
18.          )
19.    )
```

To guarantee a gap between messages sent via the medium, CSMA/CA (as well as other protocols) specifies the *distributed (coordination function) interframe space* (difs ∈ TIME), which is usually small,[6] but larger than sifs, so that acknowledgements get priority over new data frames. When the medium becomes busy during the interframe space, another node started transmitting and the process goes back to listening to the medium (Line 9). In case nothing happens on the medium and the end of the interframe space is reached (Line 10), the process determines the actual time to start transmitting the message, taking the backoff time b into account (Line 11). If the medium is idle for the entire backoff period (Line 15), the message is transmitted (Line 16), and the process calls the process ACKRECV that will await an acknowledgement from the recipient of dframe (Line 17); the third argument specifies the maximum time the process should wait for such an acknowledgement. (As mentioned before an acknowledgement may never arrive.) If another node transmits on the medium during the backoff period, the protocol restarts the routine (Lines 13 and 14), with an adjusted backoff value b—the process already started waiting and should not be punished when the waiting is restarted; this update guarantees fairness of the protocol.

The process awaiting an acknowledgement (Process 4) is straightforward. It waits until either it receives a CSMA message destined for itself (Line 1), or it receives an acknowledgement (Line 6), or it has waited for this acknowledgement as long as it is going to (Line 8).

[6] Recommended values for the constant difs are given in [19].

In the first case, the message is handled just as in Process 1, except that after acknowledging this message the protocol returns to Process 4. In the second case the network layer is informed that the sending of `dframe` was successful and the process loops back to Process 1 (Line 7). Line 8 describes the situation where no acknowledgement message arrives and the process times out. Here CSMA/CA retries to send the message; the counter `tries` is incremented.

Process 4. Receiving an ACK

```
ACKRECV(id,tries,acktimeout,dframe) ≝
 1. [NEW(dataframe(data,src,id))] deliver(data) .
 2.    (
 3.       [[timeout := now + sifs]] [now ≥ timeout]
 4.       transmit(ackframe(src)) . ACKRECV(id,tries,acktimeout,dframe)
 5.    )
 6. + [NEW(ackframe(id))]      /* acknowledgement received */
 7.    deliver(success) . CSMA(id)
 8. + [now ≥ acktimeout] INIT(id,tries+1,dframe)
```

5.2 The Hidden Station Problem

As mentioned in the introduction to this section, CSMA/CA suffers from the hidden station problem. This refers to the situation where two nodes A and C are not within transmission range of each other, while a node B is in range of both. In this situation C may be transmitting to B, but A is not able to sense this, and thus may start a transmission to B at roughly the same time, leading to data collisions at B.

While CSMA/CA is not able to avoid such collisions as a whole—it is always possible that two (or more) nodes hidden from each other happen to (randomly) choose the same backoff time to send messages—it is the exponential growth of the backoff slots that makes the problem less pressing in the long run, as the following theorem shows.

Theorem 5.1. If `max_retransmit`$=\infty$ then weak packet delivery holds with probability 1.

Proof sketch. Since the number of messages that nodes transmit is bounded, and all nodes select random times to start transmitting out of an increasing longer time span, with probability 1 each message will eventually go through. □

In practice, `max_retransmit` is set to a value that is not high enough to approximate the idea behind the above proof. In fact, the transmission time of a single message may be larger than the maximal backoff period allowed. For this reason the hidden station problem does occur when running the CSMA/CA protocol, as studies have shown [5]. Nevertheless, the above analysis still shows that link layer protocols can be formally analysed by process algebra in general, and ALL in particular.

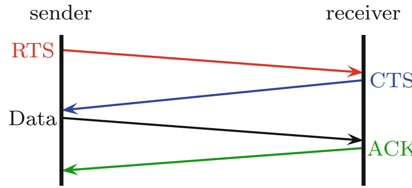


Fig. 1. RTS/CTS exchange

5.3 A Formal Model for CSMA/CA with Virtual Carrier Sensing

To overcome the hidden station problem the usage of a request-to-send/clear-to-send (RTS/CTS) handshaking [19] mechanism is available. This mechanism is also known as *virtual carrier sensing*. The exchange of RTS/CTS messages happens just before the actual data is sent, see Fig. 1. The mechanism serves two purposes: (a) As the RTS and CTS messages are very short—they only contain two node identifiers as well as a natural number indicating the time it will take to send the actual `data` (plus overhead)—the likelihood of a collision is reduced. (b) While the handshaking does not help with solving the hidden station problem for the RTS message itself, it avoids the problem for the sending of `data`. The reason is that a hidden node, which could interfere with the sending of `data` will receive the CTS message from the designated recipient of `data`, and the hidden node will remain silent until the `data` has been sent.

As for the CSMA/CA protocol we have modelled this extension in ALL, based on the model of CSMA/CA we presented earlier.

Our extended model uses two functions to generate `rts` and `cts` messages, respectively. The signature of both is $\texttt{ID} \times \texttt{ID} \times \texttt{TIME} \to \texttt{MSG}$. The first argument carries the sender (source) of the message, the second the indented destination, and the third argument a duration (time period) of silence that is requested/granted. For example, before the message `rts(src,dest,d)` is transmitted, the time period `d` is calculated by
The calculation is straightforward as it follows the protocol logic and determines the amount of time needed until the acknowledgement would be received (see Fig. 2). After the `rts` message has been received the medium should be idle for the interframe space $\underline{\texttt{sifs}}$; then a `cts` message is sent back, which takes time $\underline{\texttt{dur_cts}}$; then another interframe space is needed, followed by the actual transmission of the message—the sending will take `dur(dataframe(data,id,dest))` time units; after the message is received (hopefully) another interframe space is required before the acknowledgement is sent back.

$$[\![\texttt{d} := \underline{\texttt{sifs}} + \underline{\texttt{dur_cts}} + \underline{\texttt{sifs}} + \texttt{dur(dataframe(data,id,dest))} + \underline{\texttt{sifs}} + \underline{\texttt{dur_ack}}]\!] \,.$$

Process 2 remains essentially unchanged; it is merely equipped with the destination `dest` of the message that needs to be transmitted, and an additional timed variable $\texttt{nav} \in \texttt{TIME}$. These variables are not used in this process, but required later on. Variable `nav` holds the point in time until the process should

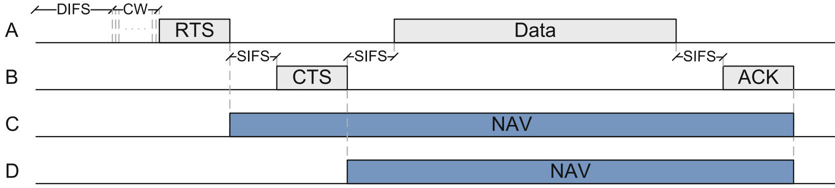


Fig. 2. The use of virtual channel sensing using CSMA/CA [3]

not transmit any `rts` or `cts` message. This period of silence is necessary as the node figures out that until time `nav` another node will transmit message(s).[7]

Process 5 is the modified version of Process 1. Identical to Process 1 it awaits an instruction from the network layer, or an incoming CSMA message destined for itself. Lines 1–3 are identical to Process 1. Lines 4–11 handle the two new message types. In case an `rts` message `rts`(`src`,`dest`,`d`) is received that is intended for another recipient (`dest` $\neq$ `id`) the node concludes that another node wants to use the medium for the amount of `d` time units; the process updates the variable `nav` if needed, indicating the period the node should remain silent, by taking the maximum of the current value of `nav`, and `now`+`d`, the point in time until the sender `src` of the `rts` message requires the medium. The same behaviour occurs if a `cts` message is received that is not intended for the node itself (Line 4). If the incoming message is an `rts` message intended for the node itself (Line 6) by default the node answers with a clear-to-send message back to the sender (Line 9). However, when the receiver of the `rts` has knowledge about other nodes requiring the medium (`now` $\leq$ `nav`), a clear-to-send cannot be granted, and the request is dropped (Line 6). Similar to the sending of an acknowledgement (Line 2), the process waits for the short interframe space (`sifs`) before sending the CTS (Line 6). Line 8 handles the case where the medium becomes busy ($\neg$IDLE) during this period; also here a clear-to-send cannot be granted, and the request is dropped.[8] Only when the medium stays idle during the entire interframe space the node `id` can inform the source of the `rts` message that the medium is clear to send; the `cts` is transmitted in Line 9. The time a receiver of this message has to be silent is adjusted by deducting the time elapsed before this happens. In Line 10 the process resets `nav` to remind itself not to issue any `rts` message until the present exchange has been completed.[9]

[7] After a successful RTS/CTS exchange, communicating nodes proceed with transmitting the data and an acknowledgement regardless of the value of `nav`.

[8] The condition `now` > `timeout`−`sifs` prevents the process from dropping the request in the very first time slice that `CSMA` is running. Here the medium counts as busy, but only because we have just received an `rts` message.

[9] A case NEW(`cts`(`src`,`dest`,`d`)) $\wedge$ `dest` = `id` is not required as a `cts` message is only expected in case an `rts` was sent, and hence handled in process `RTSREACT`.

Process 5. The Basic Routine (RTS/CTS)

```
CSMA(id,nav) ≝
 1. newpkt(data,dest). INIT(id,dest,0,dataframe(data,id,dest),nav)
 2. + [NEW(dataframe(data,src,id))] deliver(data) . ⟦timeout := now + sifs⟧
 3.    [now ≥ timeout] transmit(ackframe(src)) . CSMA(id,nav)
 4. + [(NEW(rts(src,dest,d)) ∨ NEW(cts(src,dest,d))) ∧ dest ≠ id ∧ nav < now+d]
 5.    ⟦nav := now+d⟧ CSMA(id, nav)
 6. + [NEW(rts(src,id,d)) ∧ now > nav] ⟦timeout := now + sifs⟧
 7.    (
 8.       [¬IDLE ∧ now > timeout−sifs] CSMA(id, nav)
 9.       + [IDLE ∧ now ≥ timeout] transmit(cts(id,src,d−dur_cts−sifs)) .
10.          ⟦nav := now+d−dur_cts−sifs⟧ CSMA(id, nav)
11.    )
```

Process 6. Clear Channel Assessment With Virtual Carrier Sense

```
CCA(id,dest,b,tries,dframe,nav) ≝
 1. [NEW(dataframe(data,src,id))] deliver(data) . ⟦timeout := now + sifs⟧
 2.    [now ≥ timeout] transmit(ackframe(src)) . CCA(id,dest,b,tries,dframe,nav)
 3. + [(NEW(rts(src,dest,d)) ∨ NEW(cts(src,dest,d))) ∧ dest ≠ id ∧ nav < now+d]
 4.    ⟦nav := now+d⟧ CCA(id,dest,b,tries,dframe,nav)
 5. + [NEW(rts(src,id,d)) ∧ now > nav] ⟦timeout := now + sifs⟧
 6.    (
 7.       [¬IDLE ∧ now > timeout−sifs] CCA(id,dest,b,tries,dframe,nav)
 8.       + [IDLE ∧ now ≥ timeout] transmit(cts(id,src,d−dur_cts−sifs)) .
 9.          ⟦nav := now+d−dur_cts−sifs⟧ CCA(id,dest,b,tries,dframe,nav)
10.    )
11. + [IDLE ∧ now > nav]
12.    ⟦timeout:=now+difs⟧
13.    (
14.       [¬IDLE] CCA(id,dest,b,tries,dframe,nav)
15.       + [IDLE ∧ now ≥ timeout]
16.          ⟦timeout := now + b⟧
17.          (
18.             [¬IDLE]       /* busy during backoff time */
19.                ⟦b := timeout − now⟧ CCA(id,dest,b,tries,dframe,nav)
20.             + [IDLE ∧ now ≥ timeout]       /* idle for backoff time */
21.                ⟦d := sifs + dur_cts + sifs + dur(dframe) + sifs + dur_ack⟧
22.                transmit(rts(id,dest,d)) .
23.                CTSRECV(id,dest,tries,now + max_cts_wait,dframe,nav)
24.          )
25.    )
```

Process 6 is the modified version of Process 3. The goal of this process is to send an `rts` message (Line 22). Before it can start its work, it waits until the medium is idle, and any time it is required to be silent has elapsed (Line 11).

Until this happens incoming data frames, `rts` or `cts` messages are treated just as in Process 5: Lines 1–10 copy Lines 2–11 of Process 5, except that afterwards the process returns to itself. Then Lines 12–20 are copied from Lines 7–15 from Process 3. Line 21 calculates the time other nodes ought to keep silent when receiving the `rts` message, and Line 23 passes control to the process `CTSRECV`, which awaits a `cts` response to the `rts` message transmitted in Line 22. The fourth argument of `CTSRECV` specifies the maximum time that process should wait for such a response; a good value for max_cts_wait is sifs + dur_cts.

Process `CTSRECV` listens for this time to a `cts` message with source `dest` and destination `id`. In case the expected `cts` message arrives in time (Line 1), the node waits for a time sifs (Line 2) and then transmits the data frame and proceeds to await an acknowledgement (Line 3). The fourth argument of `ACKRECV` specifies the maximum time the process should wait for such an acknowledgement; a good value for max_ack_wait is sifs + dur_ack. If the `cts` message does not arrive in time (Line 6), the process returns to `INIT` to send another `rts` message, while incrementing the counter `tries` (Line 7). While waiting for the `cts` message, any incoming `rts` or `cts` message destined for another node is treated exactly as in Process 5 (Lines 4–5). Incoming data frames cannot arrive when this process is running, and incoming `rts` messages to `id` are ignored.

Process 7. Receiving a CTS

CTSRECV(id,dest,tries,ctstimeout,dframe,nav) $\stackrel{def}{=}$
1. [NEW(cts(dest,id,d))]
2. ⟦timeout := now + sifs⟧ [now $\geq$ timeout]
3. **transmit**(dframe) . ACKRECV(id,dest,tries,now + max_ack_wait,dframe,nav)
4. \+ [(NEW(rts(src,dest,d)) $\vee$ NEW(cts(src,dest,d))) $\wedge$ dest $\neq$ id $\wedge$ nav $<$ now+d]
5. ⟦nav := now+d⟧ CTSRECV(id,dest,tries,ctstimeout,dframe,nav)
6. \+ [now $\geq$ ctstimeout]
7. INIT(id,dest,tries+1,dframe,nav)

Process 8. Receiving an ACK

ACKRECV(id,dest,tries,acktimeout,dframe,nav) $\stackrel{def}{=}$
1. [NEW(ackframe(id))]
2. **deliver**(success) . CSMA(id,nav)
3. \+ [(NEW(rts(src,dest,d)) $\vee$ NEW(cts(src,dest,d))) $\wedge$ dest $\neq$ id $\wedge$ nav $<$ now + d]
4. ⟦nav := now+d⟧ ACKRECV(id,dest,tries,acktimeout,dframe,nav)
5. \+ [now $\geq$ timeout] /* nothing received */
6. INIT(id,dest,tries+1,dframe,nav)

Process 8 handles the receipt of an acknowledgement in response to a successful data transmission. If an acknowledgement arrives, it must be from the node to which `id` has transmitted a data frame. In that case (Line 1), the network layer is informed that the sending of `dframe` was successful and the process loops back to Process 5 (Line 2). Line 5 describes the situation where no acknowledgement message arrives and the process times out. Also here CSMA/CA retries

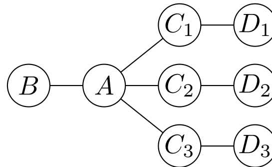
to send the message; the counter `tries` is incremented. Lines 3–4 describe the usual handling of incoming `rts` or `cts` messages destined for another node.

5.4 The Exposed Station Problem

Another source of collisions in CSMA/CA is the well-known *exposed station problem*. This refers to a linear topology $A - B - C - D$, where an unending stream of messages between C and D interferes with attempts by A to get a message across to B. In the default CSMA/CA protocol as formalised in Sect. 5.1, transmissions from A to B may perpetually collide at B with transmissions from C destined for D. CSMA/CA with virtual carrier sensing mitigates this problem, for a `cts` sent by B in response to an `rts` sent by A will tell C to keep silent for the required duration. In fact, we can show that in the above topology, if `max_retransmit`$=\infty$ then packet delivery holds with probability 1. A non-probabilistic guarantee cannot be given since nodes A and C could behave in the same way, meaning if one node is sending out a message the other does the same at the very same moment, and if one is silent the other remains silent as well. In this scenario all messages to be sent are doomed.

Based on our formalisation, we can prove that once the RTS/CTS handshake has been successfully concluded, meaning that all nodes within range of the intended recipient have received the `cts`, then packet delivery holds outright. So the only problem left is to achieve a successful RTS/CTS handshake. Since `rts` and `cts` messages are rather short, even by modest values of `max_retransmit` it becomes likely that such messages do not collide.

In spite of this, CSMA/CA with (or without) virtual channel sensing cannot achieve packet delivery with probability 1 for general topologies. Assume the following network topology



Here it may happen that one of the C_is is always busy transmitting a large message to D_i; any given C_i is occasionally silent (not sending any message), but then one of the others is transmitting. As C_i is disconnected from C_j, for $j \neq i$, coordination between the nodes is impossible. As a consequence, the medium at A will always be busy, so that A cannot send an `rts` message from B.

6 Related Work

The CSMA protocol in its different variants has been analysed with different formalisms in the past.

Multiple analyses were performed for the CSMA/CD protocol (CSMA with collision detection), a predecessor of CSMA/CA that has a constant backoff, i.e.

the backoff time is not increased exponentially, see [10,11,20,21,26]. In all these approaches frame collisions have to be modelled explicitly, as part of the protocol description. In contrast, our approach handles collisions in the semantics; thereby achieving a clear separation between protocol specifications and link layer behaviour.

Duflot et al. [10,11] use probabilistic timed automata (PTAs) to model the protocol, and use probabilistic model checking (PRISM) and approximate model checking (APMC) for their analysis. The model explained in [26] is based on PTAs as well, but uses the model checker Uppaal as verification tool. These approaches, although formal, have very little in common with our approach. On the one hand it is not easy to change the model from CSMA/CD to CSMA/CA, as the latter requires unbounded data structures (or alike) to model the exponential backoff. On the other hand, as usual, model checking suffers from state space explosion and only small networks (usually fewer than ten nodes) can be analysed. This is sufficient and convenient when it comes to finding counter examples, but these approaches cannot provide guarantees for arbitrary network topologies, as ours does.

Jensen et al. [20] use models of CSMA/CD to compare the tools SPIN and Uppaal. Their models are much more abstract than ours. It is proven that no collisions will ever occur, without stating the exact conditions under which this statement holds.

To the best of our knowledge, Parrow [21] is the only one who used process algebra (CCS) to model and analyse CSMA. His untimed model of CSMA/CD is extremely abstract and the analysis performed is limited to two nodes only, avoiding scenarios such as the hidden station problem.

There are far fewer formal analyses techniques available when it comes to CSMA/CA (with and without virtual medium sensing). Traditional approaches to the analysis of network protocols are simulation and test-bed experiments. This is also the case for CSMA/CA (e.g. [4]). While these are important and valid methods for protocol evaluation, in particular for quantitative performance evaluation, they have limitations in regards to the evaluation of basic protocol correctness properties.

Following the spirit of the above-mentioned research of model checking CSMA, Fruth [15] analyses CSMA/CA using PTAs and PRISM. He considers properties such as the minimum probability of two nodes successfully completing their transmissions, and maximum expected number of collisions until two nodes have successfully completed their transmissions. As before, this analysis technique does not scale; in [15] the experiments are limited to two contending nodes only.

Beyond model checking, simulation and test-bed experiments, we are only aware of two other formal approaches. In [1] Markov chains are used to derive an accurate, analytical model to compute the throughput of CSMA/CA. Calculating throughput is an orthogonal task to our vision of proving (functional) correctness.

An approach aiming at proving the correctness of CSMA/CA with virtual carrier sensing (RTS/CTS), and hence related to ours, is presented in [3]. Based

on stochastic bigraphs with sharing it uses rewrite rules to analyse quantitative properties. Although it is an approach that is capable to analyse arbitrary topologies, to apply the rewrite rules a particular topology needs to be modelled by a directed acyclic graph structure, which is part of the bigraph.

7 Conclusion

In this paper we have proposed a novel process algebra, called ALL, that can be used to model, verify and analyse link layer protocols. Since we aimed at a process algebra featuring aspects of the link layer such as frame collisions, as well as arbitrary data structures (to model a rich class of protocols), we could not use any of the existing algebras. The design of ALL is layered. The first layer allows modelling protocols in some sort of pseudo code, which hopefully makes our approach accessible for network and software researchers/engineers. The other layers are mainly for giving a formal semantics to the language. The layer of partial network expressions, the third layer, provides a unique and sophisticated mechanism for modelling the collision of frames. As it is hard-wired in the semantics there is no need to model collisions manually when modelling a protocol, as it was done before [21]. Next to primitives needed for modelling link layer protocols (e.g. **transmit**) and standard operators of process algebra (e.g. nondeterministic choice), ALL provides an operator for probabilistic choice.

This operator is needed to model aspects of link layer protocols such as the exponential backoff for the Carrier-Sense Multiple Access with Collision Avoidance protocol, the case study we have chosen to demonstrate the applicability of ALL. We have modelled and analysed two versions of CSMA/CA, without and with virtual carrier sensing. Our analysis has confirmed the hidden station problem for the version without virtual carrier sensing. However, we have also shown that the version with virtual carrier sensing overcomes not only this problem, but also the exposed station problem with probability 1. Yet the protocol cannot guarantee packet delivery, not even with probability 1.

To perform this analysis we had to formalise suitable liveness properties for link layer protocols specified in our framework.

Acknowledgement. We thank Tran Ngoc Ma for her involvement in this project in a very early phase. We also like to thank the German Academic Exchange Service (DAAD) that funded an internship of the third author at Data61, CSIRO.

References

1. Bianchi, G.: Performance analysis of the IEEE 802.11 distributed coordination function. IEEE J. Sel. Areas Commun. **18**(3), 535–547 (2000). https://doi.org/10.1109/49.840210
2. Bres, E., van Glabbeek, R.J., Höfner, P.: A timed process algebra for wireless networks with an application in routing. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 95–122. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_5

3. Calder, M., Sevegnani, M.: Modelling IEEE 802.11 CSMA/CA RTS/CTS with stochastic bigraphs with sharing. Formal Aspects Comput. **26**(3), 537–561 (2014). https://doi.org/10.1007/s00165-012-0270-3
4. Chhaya, H.S., Gupta, S.: Performance modeling of asynchronous data transfer methods of IEEE 802.11 MAC Protocol. Wirel. Netw. **3**, 217–234 (1997). https://doi.org/10.1023/A:1019109301754
5. Comer, D.: Computer Networks and Internets. Pearson Education Inc., UpperSaddle River (2009)
6. 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). https://doi.org/10.1007/978-3-540-85361-9_35
7. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM **42**(2), 458–487 (1995). https://doi.org/10.1145/201019.201032
8. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.C., Zhang, C.: Remarks on testing probabilistic processes. In: Cardelli, L., Fiore, M., Winskel, G. (eds.) Computation, Meaning, and Logic: Articles Dedicated to Gordon Plotkin, Electronic Notes in Theoretical Computer Science, vol. 172, pp. 359–397. Elsevier (2007). https://doi.org/10.1016/j.entcs.2007.02.013
9. Deng, Y., van Glabbeek, R.J., Morgan, C.C., Zhang, C.: Scalar outcomes suffice for finitary probabilistic testing. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 363–378. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_25
10. Duflot, M., et al.: Probabilistic model checking of the CSMA/CD, protocol using PRISM and APMC. In: Automated Verification of Critical Systems (AVoCS 2004). Electronic Notes in Theoretical Computer Science Series, vol. 128, pp. 195–214 (2004). https://doi.org/10.1016/j.entcs.2005.04.012
11. Duflot, M., et al.: Practical applications of probabilistic model checking to communication protocols. In: Gnesi, S., Margaria, T. (eds.) Formal Methods for Industrial Critical Systems: A Survey of Applications, pp. 133–150. IEEE (2013). https://doi.org/10.1002/9781118459898.ch7
12. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 295–315. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_15
13. 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). http://arxiv.org/abs/1312.7645
14. Friend, G.E., Fike, J.L., Baker, H.C., Bellamy, J.C.: Understanding Data Communications, 2nd edn. Howard W. Sams & Company, Indianapolis (1988)
15. Fruth, M.: Probabilistic model checking of contention resolution in the IEEE 802.15.4 low-rate wireless personal area network protocol. In: Leveraging Applications of Formal Methods, Second International Symposium (ISoLA 2006), pp. 290–297. IEEE Computer Society (2006). https://doi.org/10.1109/ISoLA.2006.34
16. IEEE: IEEE standard for ethernet (2016). https://doi.org/10.1109/IEEESTD.2016.7428776
17. IEEE: IEEE standard for low-rate wireless networks (2016). https://doi.org/10.1109/IEEESTD.2016.7460875
18. ISO/IEC 7498–1: Information technology—open systems interconnection—basic reference model: The basic model (1994). https://www.iso.org/standard/20269.html

19. ISO/IEC/IEEE 8802–11: 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 (2018). https://www.iso.org/standard/73367.html
20. Jensen, H.E., Larsen, K.G., Skou, A.: Modelling and analysis of a collision avoidance protocol using Spin and Uppaal. In: The Spin Verification System. Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 33–50. DIMACS/AMS (1996). https://doi.org/10.7146/brics.v3i24.20005
21. Parrow, J.: Verifying a CSMA/CD-protocol with CCS. In: Aggarwal, S. (eds.) IFIP Symposium on Protocol Specification, Testing and Verification (PSTV 1988), North-Holland, pp. 373–384 (1988)
22. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science (FOCS 1977), pp. 46–57. IEEE (1977). https://doi.org/10.1109/SFCS.1977.32
23. de Simone, R.: Higher-level synchronising devices in Meije-SCCS. TCS **37**, 245–267 (1985). https://doi.org/10.1016/0304-3975(85)90093-3
24. Simpson, W.: The point-to-point protocol (PPP). RFC 1661 Internet Standard (1994). http://www.ietf.org/rfc/rfc1661.txt
25. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. **75**, 440–469 (2010). https://doi.org/10.1016/j.scico.2009.07.008
26. Zhao, J., Li, X., Zheng, T., Zheng, G.: Removing irrelevant atomic formulas for checking timed automata efficiently. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 34–45. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-40903-8_4

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.