Mechanizing a Process Algebra for Network Protocols

Timothy Bourke, Robert Van Glabbeek, Peter Höfner

Noname manuscript No.
(will be inserted by the editor)

Mechanizing a Process Algebra for Network Protocols

Timothy Bourke · Robert J. van Glabbeek · Peter Höfner

Received: date / Accepted: date

Abstract This paper presents the mechanization of a process algebra for Mobile Ad hoc Networks and Wireless Mesh Networks, and the development of a compositional framework for proving invariant properties. Mechanizing the core process algebra in Isabelle/HOL is relatively standard, but its layered structure necessitates special treatment. The control states of reactive processes, such as nodes in a network, are modelled by terms of the process algebra. We propose a technique based on these terms to streamline proofs of inductive invariance. This is not sufficient, however, to state and prove invariants that relate states across multiple processes (entire networks). To this end, we propose a novel compositional technique for lifting global invariants stated at the level of individual nodes to networks of nodes.

Keywords Interactive Theorem Proving · Isabelle/HOL · Process Algebra · Compositional Invariant Proofs · Wireless Mesh Networks · Mobile Ad hoc Networks

1 Introduction and related work

The Algebra for Wireless Networks (AWN) is a process algebra developed in particular for modelling and analysing protocols for Mobile Ad hoc Networks (MANETs)

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

T. Bourke
Inria Paris and École normale supérieure, Paris, France
E-mail: Timothy.Bourke@inria.fr

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

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

and Wireless Mesh Networks (WMNs) [10, 11], but that can be used for reasoning about routing and communication protocols in general. This paper reports on both its mechanization in Isabelle/HOL [29] and the development of a compositional framework for showing invariant properties of models.[1] The techniques we describe are a response to problems encountered during the mechanization of a model and proof of a crucial correctness property for the Ad hoc On-demand Distance Vector (AODV) routing protocol, a widely used protocol, standardized by the IETF [31]. The AODV case study is described in detail elsewhere [5] and we only refer to it briefly in this paper. The property we study is *loop freedom*, meaning that no data packet is sent in cycles forever. Such a property can only be expressed by relating states of different (neighbouring) network nodes. Encoding such inter-node properties in an Interactive Theorem Prover (ITP) proved quite challenging, since the proof is performed inductively for an arbitrary number of nodes and the base case is a single node whose neighbours do not yet exist. We develop a novel compositional technique to address this challenge.

Despite extensive research on related problems [34] and several mechanized frameworks for reactive systems [9, 18, 27], we are not aware of other solutions that allow the compositional statement and proof of properties relating the states of different nodes in a message-passing model—at least not within the strictures imposed by an ITP.

Related work. AWN is a process algebra, but for the purposes of proving properties we treat it essentially as a structured programming language and employ a technique originally proposed by Floyd [13] and later developed by Manna and Pnueli [23], whereby a set of semantic rules is defined to link the syntax of a program to an induced transition system. Safety properties are then shown to hold for all reachable states by induction from a set of initial states over the set of transitions. Rather than define the induced transition system in terms of labels and (virtual) program counters [23, Chapter 1], we use term derivatives and Structural Operational Semantics (SOS) rules [32].

This separation between language and model differs from the approach taken in formalisms like UNITY [8] and I/O Automata [22], where initial states and sets of transitions are specified directly, and also from that of TLA$^+$ [21], where the initial states and transition relation are written as a formula of first-order logic. The advantage of the language-plus-semantics approach is that sequencing and branching in models is expressed by syntactic operators with the implied changes in the underlying control state being managed by the semantic rules. Arguably, this permits models that are easier to understand by experts in the system being modelled. The disadvantage is some extra complexity and layers of definitions. We find, however, that these details are well managed by ITPs and—once defined—intrude little on the verification task.

AWN provides a unique mix of communication primitives and a treatment of data structures that are essential for studying MANET and WMN protocols with dynamic topologies and sophisticated routing logic [11, §1]. It supports communication primitives for one-to-one (unicast), one-to-many (groupcast), and one-to-all (broadcast) message passing. AWN comprises distinct layers for expressing the structure of nodes and networks. We exploit this structure critically in our proofs, and

[1] The Isabelle/HOL source files can be found in the Archive of Formal Proofs (AFP) [4].

we expect the techniques proposed in Sections 3 and 4 to also apply to similar layered modelling languages [15, 16, 24, 25, 28, 33].

Besides this, our work differs from other mechanizations for verifying reactive systems, like UNITY [18], TLA$^+$ [9], or I/O Automata [27] (from which we drew the most inspiration), in its explicit treatment of control states, in the form of process algebra terms, as distinct from data states. In this respect, our approach is close to that of Isabelle/Circus [12], but it differs in (1) the treatment of operators for composing nodes, which we model directly as functions on automata, (2) the treatment of recursive invocations, which we do not permit, and (3) our inclusion of a framework for compositional proofs.

Within the process algebraic tradition, other work in ITPs focuses on showing properties of process algebras, such as the treatment of binders [1], that bisimulation equivalence is a congruence [17, 19], or properties of fix-point induction [36], while we focus on what has been termed 'proof methodology' [14], and develop a compositional method for showing correctness properties of protocols specified in a process algebra.

As an alternative to the frameworks cited above, and the work we present, Paulson's inductive approach [30] can be applied to show properties of protocols specified with less generic infrastructure. In fact, it has also been applied to model the AODV protocol [39]; a detailed comparison is given elsewhere [5, §9]. But we think this approach to be better suited to systems specified in a 'declarative' style as opposed to the strongly operational models we consider. The question of style has practical implications. It determines the 'distance' between the original specification and the formal model—perhaps surprisingly protocol descriptions are often quite operational (this is the case for AODV [31]). It also likely influences proofs of refinement between abstract and implementation models.

Structure and contributions. Section 2 describes the mechanization of AWN. The basic definitions are routine but the layered structure of the language and the treatment of operators on networks as functions on automata are relatively novel and essential to understanding later sections. Section 3 describes our mechanization of the theory of inductive invariants, closely following [23]. We exploit the structure of AWN to generate verification conditions corresponding to those of pen-and-paper proofs [11, §7]. Section 4 presents a compositional technique for stating and proving invariants that relate states across multiple nodes. Basically, we substitute 'open' SOS rules over the global state for the standard rules over local states (Section 4.1), show the property over a single sequential process (Section 4.2), 'lift' it successively over layers that model message queueing and network communication (Section 4.3), and, ultimately, 'transfer' it to the original model (Section 4.4).

Note. This paper is an extended version of [6]. It presents all details with regards to the mechanization—many of which were skipped in [6] due to lack of space. We also present more details about the novel compositional technique for lifting global invariants, including motivation and examples. As a case study, the framework we present in this paper was successfully applied in the mechanization of a proof of AODV's loop freedom, the details of which are available in the AFP [7] and presented elsewhere [5].

{l}⟦u⟧ p	'l ⇒ ('k ⇒ 'k) ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
{l}⟨g⟩ p	'l ⇒ ('k ⇒ 'k set) ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
{l}unicast(s_{id}, s_{msg}) . p ▷ q	'l ⇒ ('k ⇒ ip) ⇒ ('k ⇒ msg) ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
{l}broadcast(s_{msg}) . p	'l ⇒ ('k ⇒ msg) ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
{l}groupcast(s_{ids}, s_{msg}) . p	'l ⇒ ('k ⇒ ip set) ⇒ ('k ⇒ msg) ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
{l}send(s_{msg}) . p	'l ⇒ ('k ⇒ msg) ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
{l}receive(u_{msg}) . p	'l ⇒ (msg ⇒ 'k ⇒ 'k) ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
{l}deliver(s_{data}) . p	'l ⇒ ('k ⇒ data) ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
p ⊕ q	('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp ⇒ ('k, 'p, 'l) seqp
call(pn)	'p ⇒ ('k, 'p, 'l) seqp

Fig. 1 Term constructors for sequential processes: ('k, 'p, 'l) seqp.
(Leading λ-abstractions are omitted, for example, λl u p. {l}⟦u⟧ p is written {l}⟦u⟧ p.)

2 The process algebra AWN

The Algebra for Wireless Networks (AWN) comprises five layers [11, §4]: (1) *sequential processes* for encoding the protocol logic as a recursive specification; (2) *parallel composition* of sequential processes for running multiple processes simultaneously on a single node; (3) *node expressions* for encapsulating processes running on a node and tracking a node's address and neighbours (other nodes within transmission range); (4) *partial network expressions* for describing networks as parallel compositions of nodes and (5) *complete network expressions* for closing partial networks to further interactions with an environment. We treat each layer as an automaton with states of a specific form and a given set of transition rules. We describe the layers from the bottom up over the following sections.

2.1 Sequential processes

Sequential processes are used to encode protocol logic. Each is modelled by a *(recursive) specification* Γ of type 'p ⇒ ('k, 'p, 'l) seqp, which maps process names of type 'p to terms of type ('k, 'p, 'l) seqp, also parameterized by 'k, data states, and 'l, labels. States of sequential processes have the form (ξ, p) where ξ is a data state of type 'k and p is a control term of type ('k, 'p, 'l) seqp.[2]

Process terms are built from the constructors that are shown with their types in Figure 1. Here we make use of types data, msg, and ip of *application layer data*, *messages* and *IP addresses* (or any other node identifiers). These are to be defined separately for any application of AWN. Furthermore, for any type 't, the type of sets of objects of type 't is denoted 't set. The inductive set seqp-sos, shown in Figure 2, contains SOS rules for each constructor. It is parameterized by a specification Γ and relates triples of source states, actions, and destination states.

The 'prefix' constructors are each labelled with an {l}. Labels are used to strengthen invariants when a property is only true in or between certain states; they have no influence on control flow (unlike in [23]). The prefix constructors are

[2] In fact, control terms are also parameterized by the type of messages, which are specific to a given protocol, but we prefer to omit this detail from the presentation given here.

$$\frac{\xi' = \mathsf{u}\ \xi}{((\xi,\ \{\mathsf{I}\}[\![\mathsf{u}]\!]\ \mathsf{p}),\ \tau,\ (\xi',\ \mathsf{p})) \in \mathsf{seqp\text{-}sos}\ \Gamma} \qquad \frac{\xi' \in \mathsf{g}\ \xi}{((\xi,\ \{\mathsf{I}\}\langle \mathsf{g}\rangle\ \mathsf{p}),\ \tau,\ (\xi',\ \mathsf{p})) \in \mathsf{seqp\text{-}sos}\ \Gamma}$$

$$((\xi,\ \{\mathsf{I}\}\mathsf{unicast}(\mathsf{s}_{id},\ \mathsf{s}_{msg})\ .\ \mathsf{p} \triangleright \mathsf{q}),\ \mathsf{unicast}\ (\mathsf{s}_{id}\ \xi)\ (\mathsf{s}_{msg}\ \xi),\ (\xi,\ \mathsf{p})) \in \mathsf{seqp\text{-}sos}\ \Gamma$$

$$((\xi,\ \{\mathsf{I}\}\mathsf{unicast}(\mathsf{s}_{id},\ \mathsf{s}_{msg})\ .\ \mathsf{p} \triangleright \mathsf{q}),\ \neg\mathsf{unicast}\ (\mathsf{s}_{id}\ \xi),\ (\xi,\ \mathsf{q})) \in \mathsf{seqp\text{-}sos}\ \Gamma$$

$$((\xi,\ \{\mathsf{I}\}\mathsf{broadcast}(\mathsf{s}_{msg})\ .\ \mathsf{p}),\ \mathsf{broadcast}\ (\mathsf{s}_{msg}\ \xi),\ (\xi,\ \mathsf{p})) \in \mathsf{seqp\text{-}sos}\ \Gamma$$

$$((\xi,\ \{\mathsf{I}\}\mathsf{groupcast}(\mathsf{s}_{ids},\ \mathsf{s}_{msg})\ .\ \mathsf{p}),\ \mathsf{groupcast}\ (\mathsf{s}_{ids}\ \xi)\ (\mathsf{s}_{msg}\ \xi),\ (\xi,\ \mathsf{p})) \in \mathsf{seqp\text{-}sos}\ \Gamma$$

$$((\xi,\ \{\mathsf{I}\}\mathsf{send}(\mathsf{s}_{msg})\ .\ \mathsf{p}),\ \mathsf{send}\ (\mathsf{s}_{msg}\ \xi),\ (\xi,\ \mathsf{p})) \in \mathsf{seqp\text{-}sos}\ \Gamma$$

$$((\xi,\ \{\mathsf{I}\}\mathsf{receive}(\mathsf{u}_{msg})\ .\ \mathsf{p}),\ \mathsf{receive\ msg},\ (\mathsf{u}_{msg}\ \mathsf{msg}\ \xi,\ \mathsf{p})) \in \mathsf{seqp\text{-}sos}\ \Gamma$$

$$((\xi,\ \{\mathsf{I}\}\mathsf{deliver}(\mathsf{s}_{data})\ .\ \mathsf{p}),\ \mathsf{deliver}\ (\mathsf{s}_{data}\ \xi),\ (\xi,\ \mathsf{p})) \in \mathsf{seqp\text{-}sos}\ \Gamma$$

$$\frac{((\xi,\ \mathsf{p}),\ \mathsf{a},\ (\xi',\ \mathsf{p}')) \in \mathsf{seqp\text{-}sos}\ \Gamma}{((\xi,\ \mathsf{p} \oplus \mathsf{q}),\ \mathsf{a},\ (\xi',\ \mathsf{p}')) \in \mathsf{seqp\text{-}sos}\ \Gamma} \qquad \frac{((\xi,\ \mathsf{q}),\ \mathsf{a},\ (\xi',\ \mathsf{q}')) \in \mathsf{seqp\text{-}sos}\ \Gamma}{((\xi,\ \mathsf{p} \oplus \mathsf{q}),\ \mathsf{a},\ (\xi',\ \mathsf{q}')) \in \mathsf{seqp\text{-}sos}\ \Gamma}$$

$$\frac{((\xi,\ \Gamma\ \mathsf{pn}),\ \mathsf{a},\ (\xi',\ \mathsf{p}')) \in \mathsf{seqp\text{-}sos}\ \Gamma}{((\xi,\ \mathsf{call}(\mathsf{pn})),\ \mathsf{a},\ (\xi',\ \mathsf{p}')) \in \mathsf{seqp\text{-}sos}\ \Gamma}$$

Fig. 2 SOS rules for sequential processes: seqp-sos.

assignment, guard/bind, network synchronizations unicast/broadcast/groupcast/receive, and internal communications send/receive/deliver.

The *assignment* $\{\mathsf{I}\}[\![\mathsf{u}]\!]\ \mathsf{p}$ transforms the data state ξ deterministically into the data state ξ', according to the function u, and then acts as p. 'During' the update a τ-action is performed. In the original AWN [10,11], the data state ξ was defined as a partial function from data variables to values of the appropriate type, and the assignment u modified or extended this partial function by (re)mapping a specific variable to a new value, which could depend on the current data state. In our mechanization the type of data states is given as an abstract parameter of the language that is not yet instantiated in any particular way. Consequently, u is taken to be any function of type 'k ⇒ 'k, modifying the data state. In comparison with [10,11], our current treatment is less syntactic and more general.

The *guard/bind* statement $\{\mathsf{I}\}\langle \mathsf{g}\rangle\ \mathsf{p}$ encodes both guards and variable bindings. Here g is of type 'k ⇒ 'k set, a function from data states to sets of data states. Executing a guard amounts to making a nondeterministic choice of one of the data states obtainable from the current state ξ by applying g; in case $\mathsf{g}(\xi)$ is empty no transition is possible. For a valuation function h of type 'k ⇒ bool the guard statement is implemented as $\{\mathsf{I}\}\langle\lambda\xi.\ \mathsf{if\ h}\ \xi\ \mathsf{then}\ \{\xi\}\ \mathsf{else}\ \emptyset\rangle\ \mathsf{p}$, which has no outgoing transition if h evaluates to false. Variable binding like $\langle\lambda\xi.\ \{\xi(\!|\mathsf{no} := \mathsf{n}|\!) \mid \mathsf{n} < 5\}\rangle\ \mathsf{p}$ returns all possible states that satisfy the binding constraint. In the original AWN [10,11], where the data state ξ was a partial function from data variables to values, the execution of a guard/bind construct could only extend the domain of ξ, thereby assigning values to previously unbound variables. In our more abstract approach to data states, we must allow any manipulation of the (as of yet unspecified) data state. As this includes changing values of already bound variables, the guard/bind construct strictly subsumes assignment. Since this 'misuse' of a guard as assign-

ment is not allowed in the original semantics of AWN [10, 11], we prefer to keep both.

The sequential process $\{|\}\mathsf{unicast}(\mathsf{s}_{id}, \mathsf{s}_{msg})\ .\ \mathsf{p} \triangleright \mathsf{q}$ tries to unicast the message s_{msg} to the destination s_{id}; if successful it continues to act as p and otherwise as q. In other words, $\mathsf{unicast}(\mathsf{s}_{id}, \mathsf{s}_{msg})\ .\ \mathsf{p}$ is prioritized over q, which is only considered when the unicast action is not possible ($\neg\mathsf{unicast}\ (\mathsf{s}_{id}\ \xi)$). Which of the actions unicast or $\neg$unicast will occur depends on whether the destination s_{id} is in transmission range of the current node; this is implemented by the first two rules of Figure 7 (described later). In [10,11] the message s_{msg} is an expression with variables that evaluates to a message depending on the current values of those variables. Here, more abstractly, it can be any function of type 'k ⇒ msg that constructs a message from the current data state. The sequential process $\{|\}\mathsf{broadcast}(\mathsf{s}_{msg})\ .\ \mathsf{p}$ broadcasts s_{msg} to the other network nodes within transmission range.[3] The process $\{|\}\mathsf{groupcast}(\mathsf{s}_{ids}, \mathsf{s}_{msg})\ .\ \mathsf{p}$ tries to transmit s_{msg} to all destinations s_{ids}, and proceeds as p regardless of whether any of the transmissions is successful.

The sequential process $\{|\}\mathsf{send}(\mathsf{s}_{msg})\ .\ \mathsf{p}$ synchronously transmits a message to another process running on the same network node; this action can occur only when the other sequential process is able to receive the message. The sequential process $\{|\}\mathsf{receive}(\mathsf{u}_{msg})\ .\ \mathsf{p}$ receives any message u_{msg} either from another node, from another sequential process running on the same node, or from the client[4] connected to the local node. It then proceeds as p, but with an updated data state (the state change is triggered by the message). In the original syntax and semantics of AWN, u_{msg} was a data variable of type msg; here it is an abstract function of type msg ⇒ 'k ⇒ 'k, which changes the data state. The submission of data from a client is modelled by the receipt of a special message (Newpkt d dst), where the function Newpkt generates a message containing the data d and the intended destination dst. Data is delivered to the client by $\{|\}\mathsf{deliver}(\mathsf{s}_{data})\ .\ \mathsf{p}$.

The other constructors are unlabelled and serve to 'glue' processes together: The *choice* construct $\mathsf{p} \oplus \mathsf{q}$ takes the union of two transition sets and hence may act either as p or as q. The procedure call call(pn) affixes a term from the specification (Γ pn). The behaviour of call(pn) is exactly the same as that of the sequential process that Γ associates to the process name pn. In [10, 11], on the other hand, process names pn are explicitly parameterized with a list of data variables which can be defined by arbitrary data expressions at the call site. The semantics of the process call involves running the process Γ pn on an *updated* data state, obtained by evaluating the data expressions in the current state and assigning the resulting values to the corresponding variables, while clearing the values of all variables that do not occur as parameters of pn, effectively making them undefined. In the current treatment, this behaviour is recovered by preceding a call(pn) by an explicit assignment statement. As variables cannot be made undefined, they are cleared by setting them to arbitrary values. This change is the biggest departure from the original definition of AWN; it simplifies the treatment of call, as we show in Section 3.1, and facilitates working with automata where variable locality makes little sense. The drawback is that the atomic 'assign and jump' semantics is lost, which is sometimes inconvenient (an example is given later in Section 2.2).

[3] Whether a node is within transmission range or not is determined later on.

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

```
Γ_Toy PToy = labelled PToy ( receive(λmsg' ξ. ξ (| msg := msg' |)).          {PToy-:0}
                             ⟦λξ. ξ (|nhid := id ξ|)⟧                        {PToy-:1}
                             (   ⟨is-newpkt⟩                                 {PToy-:2}
                                 ⟦λξ. ξ (|no := max (no ξ) (num ξ)|)⟧        {PToy-:3}
                                 broadcast(λξ. Pkt (no ξ) (id ξ)).           {PToy-:4}
                                 ⟦clear-locals⟧ call(PToy)                   {PToy-:5}
                              ⊕ ⟨is-pkt⟩                                     {PToy-:2}
                                 (  ⟨λξ. if num ξ > no ξ then {ξ} else ∅⟩    {PToy-:6}
                                    ⟦λξ. ξ (|no := num ξ|)⟧                  {PToy-:7}
                                    ⟦λξ. ξ (|nhid := sid ξ|)⟧                {PToy-:8}
                                    broadcast(λξ. Pkt (no ξ) (id ξ)).        {PToy-:9}
                                    ⟦clear-locals⟧ call(PToy)                {PToy-:10}
                                 ⊕ ⟨λξ. if num ξ ≤ no ξ then {ξ} else ∅⟩    {PToy-:6}
                                    ⟦clear-locals⟧ call(PToy))))             {PToy-:11}
```

Fig. 3 AWN-specification of a toy protocol.

An example sequential process. We give the specification of a simple 'toy' protocol as a running example. The formal AWN specification is presented in Figure 3. Nodes following the protocol broadcast messages containing an integer no. Each remembers the largest integer it has received and drops messages containing smaller or equal values.

The protocol is defined by a process named PToy that maintains three variables: the integer no; an identifier id—also an integer, which uniquely identifies a node (for example, the node's IP address); and an identifier nhid that stores a node address (either that of the node itself, or the address of another node that supplied the largest number in the last comparison it made).[5] The initial values of nhid and no are id and 0, respectively.

The behaviour of a single node in our toy protocol is given by the recursive specification Γ_{Toy} and an initial state (ξ, p) consisting of a data state ξ—defined above—and a control term p—here the process Γ_{Toy} PToy. The specification Γ_{Toy}, given in Figure 3, assigns a process term to each process name—here only to the name PToy. The process term Γ_{Toy} PToy is defined as the result of applying a function labelled to two arguments: an identifier and the actual process without labels. The labels are supplied by the function labelled: it associates its first argument paired with a number as a label to every prefix construct occurring as a subterm. We show these labels on the right-hand side of Figure 3. Note that the choice construct $\oplus$ and the subterms call(PToy) do not receive a label. Moreover, the function labelled is defined in such a way that both arguments of the $\oplus$ receive the same label; this way labels correspond exactly to states that can be reached during the execution of the process.

A node id running the protocol PToy will wait until it receives a message msg' (line {PToy-:0}). The protocol then updates the local data state ξ by assigning the message msg' to the variable msg ($\lambda\xi.\ \xi$ (| msg := msg' |)). In our scenario, there are two message constructors Pkt d src and Newpkt d dst; both carry an identifier (src and dst) and an integer-payload d. Here, src is, by design, the sender of the message. We require that all messages from the client of a node must have the form Newpkt d dst.

[5] The protocol behaviour regarding nhid is rather arbitrary; it only serves to illustrate some forthcoming concepts.

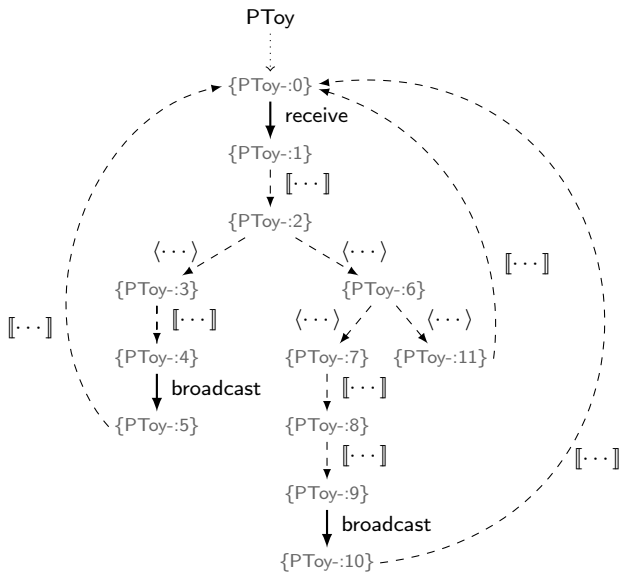

Fig. 4 Control state structure of Γ_{Toy}.

All messages sent by a node have the form Pkt d src. The message type thus uniquely determines whether the message originated from the application layer or from another node.

The choice ({PToy-:2}) makes a case distinction based on whether the message received is a new packet or a 'standard' one. In the former case, the guard/bind statement is-newpkt 'evaluates to true'[6] and copies the message content d to the variable num. Formally, is-newpkt is defined as

$$\begin{aligned}\mathsf{is\text{-}newpkt}\ \xi \quad = \quad & \mathsf{case\ msg}\ \xi\ \mathsf{of} \\ & \quad \mathsf{Pkt\ d\ src} \Rightarrow\ \emptyset \\ & \mid \mathsf{Newpkt\ d\ dst} \Rightarrow \{\xi(\!|\mathsf{num} := \mathsf{d}|\!)\}\,.\end{aligned}$$

Afterwards, the process proceeds to execute the lines labelled {PToy-:3}, {PToy-:4}, and {PToy-:5}. In the case of a 'standard' message, the statement is-pkt evaluates to true, the local state is updated by copying the message contents d into num and src into sid, and the protocol proceeds with lines {PToy-:6}–{PToy-:11}.

In line {PToy-:3} the protocol compares the stored integer no with the integer num that came from the incoming message, determines and stores the larger one into the variable no, and broadcasts this value to all its neighbours with itself listed as sender (line {PToy-:4}). After that, in line {PToy-:5}, the process calls itself recursively, after resetting the local variables msg, num, and sid to arbitrary values.

Depending on the contents of the 'standard' message, the protocol performs two different sequences of actions. (1) If the integer taken from the message and stored in variable num is larger than the stored no (line {PToy-:6}), then it is stored in variable no (line {PToy-:7}) and the sender of the message is stored in nhid (line {PToy-:8}). Before resetting the local variables and returning to the start of the protocol by a

[6] By this we mean that when it is applied to the current data state it returns a non-empty set of updated data states.

```
Γ_QMSG Qmsg = labelled Qmsg (
                receive(λmsg msgs. msgs @ [msg]) . call(Qmsg)                {Qmsg-:0}
              ⊕ ⟨λmsgs. if msgs ≠ [] then {msgs} else ∅⟩                      {Qmsg-:0}
                (   send(λmsgs. hd msgs) .                                    {Qmsg-:1}
                    (     ⟦λmsgs. tl msgs⟧ call(Qmsg)                         {Qmsg-:2}
                        ⊕ receive(λmsg msgs. tl msgs @ [msg]) . call(Qmsg))   {Qmsg-:2}
                  ⊕ receive(λmsg msgs. msgs @ [msg]) . call(Qmsg)))           {Qmsg-:1}
```

Fig. 5 AWN-specification of the queue process.

recursive call (line {PToy-:10}), the node sends out the just updated number no, again identifying itself as sender (line {PToy-:9}). (2) If the integer from the message is smaller than or equal to no (line {PToy-:6}), the node considers the message content outdated, drops the message, and calls itself recursively.

As mentioned before, every sequential process is modelled by an automaton—a record[7] of two fields: a set of initial states and a set of transitions—parameterized by an address i:

$$\mathsf{ptoy\ i} = (\!|\mathsf{init} = \{(\mathsf{toy\text{-}init\ i},\ \Gamma_{\mathsf{Toy}}\ \mathsf{PToy})\},\ \mathsf{trans} = \mathsf{seqp\text{-}sos}\ \Gamma_{\mathsf{Toy}}|\!)\,,$$

where toy-init i yields the initial data state (|id = i, no = 0, nhid = i, msg = SOME x. True, num = SOME x. True, sid = SOME x. True|). The last three variables are initialized to arbitrary values, as they are considered local. A representation of the automaton toy-init i that abstracts from the data state is depicted in Figure 4.

2.2 Local parallel composition

Message sending protocols must nearly always be input enabled, that is, nodes should always be in a state where they can receive messages.[8] To achieve this, and to model asynchronous message transmission, the protocol process is combined with a queue model. A queue can be expressed in AWN as the specification Γ_{QMSG} with a single process Qmsg shown in Figure 5. Unlike the data state of the PToy process, which mapped variable names to values, the data state msgs of Qmsg is simply a list of messages. The control term is always ready to receive a message (lines {Qmsg-:0}, {Qmsg-:1}, and {Qmsg-:2}), in which case it appends (@ concatenates lists) the received message onto the state. When the state is not empty (line {Qmsg-:0}), the first element can be sent (line {Qmsg-:1}: hd returns the head of a list), and, on doing so, removes it from the state (line {Qmsg-:2}: tl returns the tail of a list). A receive command must be repeated at each control location to ensure input enabledness. Compared to the Qmsg process in the original presentation of AWN [11, Process 6], there is an extra receive at {Qmsg-:2}. It is necessary due to the modelling of parameter passing by an assignment followed by a recursive call, which introduces a τ-transition. This is unfortunate, but eliminating

[7] The generic record has type ('s, 'a) automaton, where the type 's is the domain of states, here pairs of data records and control terms, and 'a is the domain of actions.

[8] The semantics of AWN ensures that any message transmitted by a node *will* be received by all intended destinations that are within transmission range—the reasons for this design decision are given in [10, 11]. In this setting, the absence of input enabledness would give rise to the unrealistic phenomenon of *blocking*, the situation where one node is unable to transmit a message simply because another one is not ready to receive it.

$$\frac{(\mathsf{s}, \mathsf{a}, \mathsf{s}') \in \mathsf{T}_A \quad \bigwedge \mathsf{m}.\ \mathsf{a} \neq \mathsf{receive\ m}}{((\mathsf{s}, \mathsf{t}), \mathsf{a}, (\mathsf{s}', \mathsf{t})) \in \mathsf{parp\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B} \qquad \frac{(\mathsf{t}, \mathsf{a}, \mathsf{t}') \in \mathsf{T}_B \quad \bigwedge \mathsf{m}.\ \mathsf{a} \neq \mathsf{send\ m}}{((\mathsf{s}, \mathsf{t}), \mathsf{a}, (\mathsf{s}, \mathsf{t}')) \in \mathsf{parp\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{(\mathsf{s}, \mathsf{receive\ m}, \mathsf{s}') \in \mathsf{T}_A \quad (\mathsf{t}, \mathsf{send\ m}, \mathsf{t}') \in \mathsf{T}_B}{((\mathsf{s}, \mathsf{t}), \tau, (\mathsf{s}', \mathsf{t}')) \in \mathsf{parp\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

Fig. 6 SOS rules for parallel processes: parp-sos.

parameter passing greatly simplifies the constructions presented in Section 3. The corresponding automaton is instantiated with an initially empty list:

$$\mathsf{qmsg} = (\!|\mathsf{init} = \{([], \mathit{\Gamma}_{\mathrm{QMSG}}\ \mathsf{Qmsg})\},\ \mathsf{trans} = \mathsf{seqp\text{-}sos}\ \mathit{\Gamma}_{\mathrm{QMSG}}|\!)\,,$$

The composition of the example protocol with the queue is expressed as

$$\mathsf{ptoy\ i} \langle\langle\ \mathsf{qmsg}\,.$$

This *local parallel* operator is a function over automata:

$$\mathsf{A} \langle\langle\ \mathsf{B} = (\!|\mathsf{init} = \mathsf{init\ A} \times \mathsf{init\ B},\ \mathsf{trans} = \mathsf{parp\text{-}sos\ (trans\ A)\ (trans\ B)}|\!)\,.$$

This is an operator of type ('s, 'a) automaton ⇒ ('t, 'a) automaton ⇒ ('s × 't, 'a) automaton. The process (automaton) A ⟨⟨ B is a parallel composition of A and B, running on the same network node. As formalized in Figure 6, an action receive m of A synchronizes with an action send m of B into an internal action τ. The receive actions of A and send actions of B cannot occur separately. All other actions of A and B, including send actions of A and receive actions of B, occur interleaved in A ⟨⟨ B. A parallel process expression denotes a parallel composition of sequential processes—each with states (ξ, p)—with information flowing from right to left. The variables of different sequential processes running on the same node are maintained separately, and thus cannot be shared.

2.3 Nodes

At the node level, a local (parallel) process A is wrapped in a layer that records its address i and tracks the set of neighbouring node addresses, initially R_0. We define a function from these two parameters and A, an arbitrary automaton, as

$$\langle \mathsf{i} : \mathsf{A} : \mathsf{R}_0 \rangle = (\!|\mathsf{init} = \{\mathsf{s}^{\mathsf{i}}_{\mathsf{R}_0} \mid \mathsf{s} \in \mathsf{init\ A}\},\ \mathsf{trans} = \mathsf{node\text{-}sos\ (trans\ A)}|\!)\,.$$

Node states are triples denoted $\mathsf{s}^{\mathsf{i}}_{\mathsf{R}}$. Figure 7 presents the rules of node-sos. Output network synchronizations, like groupcast or broadcast, are filtered by the list of neighbours to become ∗cast actions. So, an action R:∗cast(m) transmits a message m that can be received by the set R of network nodes. A failed unicast attempt by the process A is modelled as an internal action τ of the node expression.

There is no rule for propagating send m actions from sequential processes to the node level. These actions may only occur locally when paired with a receive action; they then become τ-transitions, which are propagated. The H¬K:arrive(m) action—instantiated in Figure 7 as ∅¬{i}:arrive(m) and {i}¬∅:arrive(m)—is used to model a message m received simultaneously by nodes in H and not by those in K. The rules for arrive m in Figure 7 state that the arrival of a message at a node happens if

$$\frac{(s, \mathsf{unicast\ dst\ m}, s') \in \mathsf{T}_A \quad \mathsf{dst} \in R}{(s_R^i, \{\mathsf{dst}\}{:}{*}\mathsf{cast(m)}, s'^i_R) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A} \qquad \frac{(s, \neg\mathsf{unicast\ dst}, s') \in \mathsf{T}_A \quad \mathsf{dst} \notin R}{(s_R^i, \tau, s'^i_R) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{(s, \mathsf{broadcast\ m}, s') \in \mathsf{T}_A}{(s_R^i, R{:}{*}\mathsf{cast(m)}, s'^i_R) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A} \qquad \frac{(s, \mathsf{groupcast\ D\ m}, s') \in \mathsf{T}_A}{(s_R^i, (R \cap D){:}{*}\mathsf{cast(m)}, s'^i_R) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{(s, \mathsf{receive\ m}, s') \in \mathsf{T}_A}{(s_R^i, \{i\}\neg\emptyset{:}\mathsf{arrive(m)}, s'^i_R) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A} \qquad \frac{(s, \mathsf{deliver\ d}, s') \in \mathsf{T}_A}{(s_R^i, i{:}\mathsf{deliver(d)}, s'^i_R) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A}$$

$$(s_R^i, \emptyset\neg\{i\}{:}\mathsf{arrive(m)}, s_R^i) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A \qquad \frac{(s, \tau, s') \in \mathsf{T}_A}{(s_R^i, \tau, s'^i_R) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A}$$

$$(s_R^i, \mathsf{connect(i, i')}, s_{R \cup \{i'\}}^i) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A \qquad (s_R^i, \mathsf{connect(i', i)}, s_{R \cup \{i'\}}^i) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A$$

$$(s_R^i, \mathsf{disconnect(i, i')}, s_{R\ -\ \{i'\}}^i) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A \qquad (s_R^i, \mathsf{disconnect(i', i)}, s_{R\ -\ \{i'\}}^i) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A$$

$$\frac{i \neq i' \quad i \neq i''}{(s_R^i, \mathsf{connect(i', i'')}, s_R^i) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A} \qquad \frac{i \neq i' \quad i \neq i''}{(s_R^i, \mathsf{disconnect(i', i'')}, s_R^i) \in \mathsf{node\text{-}sos}\ \mathsf{T}_A}$$

Fig. 7 SOS rules for nodes: node-sos.

and only if the node receives it, whereas non-arrival can happen at any time. This embodies the assumption that, at any time, any message that is transmitted to a node within range of the sender is actually received by that node [10, 11].

Internal actions τ and the action {i}:deliver(d) are simply inherited by node expressions from the processes that run on these nodes. Finally, we allow actions connect(i, i') and disconnect(i, i') for nodes i and i'. They model changes in network topology. Each node must synchronize with such an action. These actions can be thought of as occurring nondeterministically or as actions instigated by the environment of the modelled network protocol. In this formalization node i' is in the range of node i, meaning that i' can receive messages sent by i, if and only if i is in the range of i'.

2.4 Partial networks

Partial networks are specified by values of type net-tree. A net-tree is either a node $\langle i;\ R_0\rangle$ with address i and a set of initial neighbours R_0, or a composition of two net-trees $\Psi_1 \parallel \Psi_2$. Hence it denotes a network topology. The net-tree $((\langle 1;\ \{2\}\rangle \parallel \langle 2;\ \{1,\ 3\}\rangle) \parallel \langle 3;\ \{2\}\rangle)$, for instance, puts the three nodes 1, 2, and 3 in a linear topology where 2 is connected to 1 and 3. The name net-tree refers to the parse tree of its syntactic expression; unlike in [10, 11], it is treated as a tree because we do not make use of the associativity of the parallel composition.

The function pnet maps such a value, together with the process np i to execute at each node i, here parameterized by an address, to an automaton:

$$\begin{array}{lcl} \mathsf{pnet\ np}\ \langle i;\ R_0\rangle & = & \langle i : \mathsf{np}\ i : R_0\rangle \\ \mathsf{pnet\ np}\ (\Psi_1 \parallel \Psi_2) & = & (\!|\mathsf{init} = \{s_1 \amalg s_2 \mid s_1 \in \mathsf{init\ (pnet\ np}\ \Psi_1) \wedge s_2 \in \mathsf{init\ (pnet\ np}\ \Psi_2)\}, \\ & & \quad \mathsf{trans} = \mathsf{pnet\text{-}sos\ (trans\ (pnet\ np}\ \Psi_1))\ \mathsf{(trans\ (pnet\ np}\ \Psi_2))|\!)\,, \end{array}$$

$$\frac{(s,\ R{:}{*}\mathsf{cast}(m),\ s') \in \mathsf{T}_A \quad (t,\ H\neg K{:}\mathsf{arrive}(m),\ t') \in \mathsf{T}_B \quad H \subseteq R \quad K \cap R = \emptyset}{(s \between t,\ R{:}{*}\mathsf{cast}(m),\ s' \between t') \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{(s,\ H\neg K{:}\mathsf{arrive}(m),\ s') \in \mathsf{T}_A \quad (t,\ R{:}{*}\mathsf{cast}(m),\ t') \in \mathsf{T}_B \quad H \subseteq R \quad K \cap R = \emptyset}{(s \between t,\ R{:}{*}\mathsf{cast}(m),\ s' \between t') \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{(s,\ H\neg K{:}\mathsf{arrive}(m),\ s') \in \mathsf{T}_A \quad (t,\ H'\neg K'{:}\mathsf{arrive}(m),\ t') \in \mathsf{T}_B}{(s \between t,\ (H \cup H')\neg(K \cup K'){:}\mathsf{arrive}(m),\ s' \between t') \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{(s,\ i{:}\mathsf{deliver}(d),\ s') \in \mathsf{T}_A}{(s \between t,\ i{:}\mathsf{deliver}(d),\ s' \between t) \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B} \qquad \frac{(t,\ i{:}\mathsf{deliver}(d),\ t') \in \mathsf{T}_B}{(s \between t,\ i{:}\mathsf{deliver}(d),\ s \between t') \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{(s,\ \tau,\ s') \in \mathsf{T}_A}{(s \between t,\ \tau,\ s' \between t) \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B} \qquad \frac{(t,\ \tau,\ t') \in \mathsf{T}_B}{(s \between t,\ \tau,\ s \between t') \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{(s,\ \mathsf{connect}(i,\ i'),\ s') \in \mathsf{T}_A \quad (t,\ \mathsf{connect}(i,\ i'),\ t') \in \mathsf{T}_B}{(s \between t,\ \mathsf{connect}(i,\ i'),\ s' \between t') \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{(s,\ \mathsf{disconnect}(i,\ i'),\ s') \in \mathsf{T}_A \quad (t,\ \mathsf{disconnect}(i,\ i'),\ t') \in \mathsf{T}_B}{(s \between t,\ \mathsf{disconnect}(i,\ i'),\ s' \between t') \in \mathsf{pnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

Fig. 8 SOS rules for partial networks pnet-sos.

The states of such automata mirror the tree structure of the network term; we denote composed states by $s_1 \between s_2$. This structure and the node addresses remain constant during an execution.

The preceding definitions for sequential processes, local parallel composition, nodes, and partial networks suffice to model an example three-node network of toy processes:

$$(\mathsf{pnet}\ (\lambda i.\ ((\mathsf{ptoy}\ i)\ \langle\langle\ \mathsf{qmsg}))\ (((\langle 1;\ \{2\}\rangle \parallel \langle 2;\ \{1,\ 3\}\rangle) \parallel \langle 3;\ \{2\}\rangle)))\,.$$

The function pnet is not present in [10, 11], where a partial network is defined simply as a parallel composition of nodes, where in principle a different process could be running on each node. With pnet we ensure that in fact the same process is running on each node, and that this process is specified separately from the network topology.

Figure 8 presents the rules of pnet-sos. An R:∗cast(m) action of one node synchronizes with an action arrive m of all other nodes, where this arrive m amalgamates the arrival of message m at the nodes in the transmission range R of the ∗cast m, and the non-arrival at the other nodes. The third rule of Figure 8, in combination with the rules for arrive in Figure 7 and the fact that qmsg is always ready to receive m, ensures that a partial network can always perform an H¬K:arrive(m) for any combination of H and K consistent with its node addresses. Yet pairing with an R:∗cast(m), through the first two rules in Figure 8, is possible only for those H and K that are consistent with the destinations in R.

Internal actions τ and the action i:deliver(d) are interleaved in the parallel composition of nodes that makes up a network.

$$\frac{(s,\ \mathsf{connect}(i,\ i'),\ s') \in \mathsf{T}_A}{(s,\ \mathsf{connect}(i,\ i'),\ s') \in \mathsf{cnet\text{-}sos}\ \mathsf{T}_A} \qquad \frac{(s,\ \mathsf{disconnect}(i,\ i'),\ s') \in \mathsf{T}_A}{(s,\ \mathsf{disconnect}(i,\ i'),\ s') \in \mathsf{cnet\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{(s,\ R{:}*\mathsf{cast}(m),\ s') \in \mathsf{T}_A}{(s,\ \tau,\ s') \in \mathsf{cnet\text{-}sos}\ \mathsf{T}_A} \qquad \frac{(s,\ \tau,\ s') \in \mathsf{T}_A}{(s,\ \tau,\ s') \in \mathsf{cnet\text{-}sos}\ \mathsf{T}_A} \qquad \frac{(s,\ i{:}\mathsf{deliver}(d),\ s') \in \mathsf{T}_A}{(s,\ i{:}\mathsf{deliver}(d),\ s') \in \mathsf{cnet\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{(s,\ \{i\}\neg K{:}\mathsf{arrive}(\mathsf{Newpkt}\ d\ dst),\ s') \in \mathsf{T}_A}{(s,\ i{:}\mathsf{newpkt}(d,\ dst),\ s') \in \mathsf{cnet\text{-}sos}\ \mathsf{T}_A}$$

Fig. 9 SOS rules for complete networks.

2.5 Complete networks

The last layer closes a network to further interactions with an environment. It ensures that a message cannot be received unless it is sent within the network or it is a Newpkt.

$$\mathsf{closed}\ A = A(\!|\mathsf{trans} := \mathsf{cnet\text{-}sos}\ (\mathsf{trans}\ A)|\!)\,.$$

The rules for cnet-sos are straightforward and presented in Figure 9.

The closed-operator passes through internal actions, as well as the delivery of data to destination nodes, this being an interaction with the outside world. The $*$cast actions are declared internal at this level; they cannot be influenced by the outside world. The connect and disconnect actions are passed through in Figure 9, thereby placing them under the control of the environment. Actions arrive m are simply blocked by the encapsulation—they cannot occur without synchronizing with a $*$cast m—except for {i}¬K:arrive(Newpkt d dst). This action represents new data d that is submitted by a client of the modelled protocol to node i for delivery at destination dst.

3 Basic invariance

This paper only considers proofs of invariance, that is, properties of reachable states and reachable transitions. The basic definitions are classic [27, Part III].

Definition 3.1 (reachability) Given an automaton A and an assumption I over actions, reachable A I is the smallest set defined by the rules:

$$\frac{s \in \mathsf{init}\ A}{s \in \mathsf{reachable}\ A\ I} \qquad \frac{s \in \mathsf{reachable}\ A\ I \quad (s,\ a,\ s') \in \mathsf{trans}\ A \quad I\ a}{s' \in \mathsf{reachable}\ A\ I}\,.$$

As usual, all initial states are reachable, and so is any state that can be reached from a reachable state by a single a-transition that satisfies property I.

Definition 3.2 (invariance) Given an automaton A and an assumption I, a predicate P is *(state) invariant*, denoted $A \models (I \rightarrow)\ P$, iff $\forall s \in \mathsf{reachable}\ A\ I.\ P\ s$.

We define reachability relative to an assumption on (input) actions I. When I is λ-. True, we write simply $A \models P$.

Using this definition of invariance, we can state a basic property of an instance of the toy process:

$$\mathsf{ptoy\ i} \models \mathsf{onl}\ \Gamma_{\mathsf{Toy}}\ (\lambda(\xi, \mathsf{l}).\ \mathsf{l} \in \{\mathsf{PToy\text{-}:2}..\mathsf{PToy\text{-}:8}\} \longrightarrow \mathsf{nhid}\ \xi = \mathsf{id}\ \xi)\,. \tag{1}$$

This invariant states that between the lines labelled PToy-:2 and PToy-:8, that is, after the assignment of PToy-:1 until before the assignment of PToy-:8, the values of nhid and id are equal. Here onl Γ P, defined as $\lambda(\xi, \mathsf{p}).\ \forall \mathsf{l} \in \mathsf{labels}\ \Gamma\ \mathsf{p}.\ \mathsf{P}\ (\xi, \mathsf{l})$, extracts labels from control states, thereby converting a predicate on data states and line numbers into one on data states and control terms.[9] Because a $\oplus$-control term is unlabelled, the function label takes the labels of both of its arguments; for this reason labels Γ p generally yields a set of labels rather than a single label. As a control state call(pn) also is unlabelled, the function label associates labels with it by unwinding the recursion; to enable this, label takes the recursive specification Γ as an extra argument.

The statements of properties that are true of all reachable states (for example, (5), given later) do not depend on the values of control states nor the associated labels, but their proofs will if they involve other invariants (like that of (1)). Technically, the labels then form an integral part of the process model. While this is unfortunate, expressing invariants in terms of the underlying control states is simply impractical: the terms are unwieldy and susceptible to modification.

State invariants concentrate on single states only. It is, however, often useful to characterize properties describing possible changes of the state.

Definition 3.3 (transition invariance) Given an automaton A and an assumption I, a predicate P is *transition invariant*, denoted $\mathsf{A} \models (\mathsf{I} \rightarrow)\ \mathsf{P}$, iff

$$\forall\, \mathsf{a}.\ \mathsf{I\ a} \longrightarrow (\forall\, \mathsf{s} \in \mathsf{reachable\ A\ I}.\ \forall\, \mathsf{s'}.\ (\mathsf{s, a, s'}) \in \mathsf{trans\ A} \longrightarrow \mathsf{P}\ (\mathsf{s, a, s'}))\,.$$

An example for a transition invariant of our running example is that the value of no never decreases over time:

$$\mathsf{ptoy\ i} \models (\lambda((\xi, \text{-}), \text{-}, (\xi', \text{-})).\ \mathsf{no}\ \xi \leq \mathsf{no}\ \xi')\,. \tag{2}$$

Here, the assumption on (input) actions I is λ-. True and hence skipped. In case we want to restrict the statement to specific line numbers, the mechanization provides a function that extracts labels from control states, similar to onl for state invariance:

$$\mathsf{onll}\ \Gamma\ \mathsf{P} \quad = \quad \lambda((\xi, \mathsf{p}), \mathsf{a}, (\xi', \mathsf{p'})).\ \forall \mathsf{l} \in \mathsf{labels}\ \Gamma\ \mathsf{p}.\ \forall \mathsf{l'} \in \mathsf{labels}\ \Gamma\ \mathsf{p'}.\ \mathsf{P}\ ((\xi, \mathsf{l}), \mathsf{a}, (\xi', \mathsf{l'}))\,.$$

Our invariance proofs follow the compositional strategy recommended by de Roever et al. in [34, §1.6.2]. That is, we show properties of sequential process automata using the induction principle of Definition 3.1, and then apply generic proof rules to successively lift such properties over each of the other layers. The inductive assertion method, as stated by Manna and Pnueli in rule INV-B of [23], requires a finite set of transition schemas, which, together with the obligation on initial states yields a set of sufficient verification conditions. We develop this set in Section 3.1 and use it to derive the main proof rule presented in Section 3.2 together with some examples.

[9] Using labels in this way is standard, see, for instance, [23, Chap. 1], or the 'assertion networks' of [34, §2.5.1].

3.1 Control terms

Given a specification Γ over finitely many process names, we can generate a finite set of verification conditions because transitions from ('s, 'p, 'l) seqp terms always yield subterms of terms in Γ. But, rather than simply considering the set of all subterms, we prefer to define a subset of 'control terms' that reduces the number of verification conditions, avoids tedious duplication in proofs, and corresponds with the obligations considered in pen-and-paper proofs. The main idea is that the $\oplus$ and call operators serve only to combine process terms: they are, in a sense, executed recursively by seqp-sos (see Section 2.1) to determine the actions that a term offers to its environment. This is made precise by defining a relation between sequential process terms.

Definition 3.4 ($\rightsquigarrow_\Gamma$) For a (recursive) specification Γ, let $\rightsquigarrow_\Gamma$ be the smallest relation such that $(\mathsf{p} \oplus \mathsf{q}) \rightsquigarrow_\Gamma \mathsf{p}$, $(\mathsf{p} \oplus \mathsf{q}) \rightsquigarrow_\Gamma \mathsf{q}$, and $(\mathsf{call}(\mathsf{pn})) \rightsquigarrow_\Gamma \Gamma\ \mathsf{pn}$.

We write ${\rightsquigarrow_\Gamma}^*$ for its reflexive transitive closure. We consider a specification to be *well formed*, when the inverse of this relation is well founded:

$$\mathsf{wellformed}\ \Gamma = \mathsf{wf}\ \{(\mathsf{q}, \mathsf{p}) \mid \mathsf{p} \rightsquigarrow_\Gamma \mathsf{q}\}.$$ [10]

Most of our lemmas apply only to well-formed specifications, since otherwise functions over the terms they contain cannot be guaranteed to terminate. Neither of these two specifications is well formed: $\Gamma_a(1) = \mathsf{p} \oplus \mathsf{call}(1)$; $\Gamma_b(\mathsf{n}) = \mathsf{call}(\mathsf{n}{+}1)$.

We will also need a set of 'start terms' of a process—the subterms that can act directly.

Definition 3.5 (sterms) Given a wellformed Γ and a sequential process term p, sterms Γ p is the set of maximal elements related to p by the reflexive transitive closure of the $\rightsquigarrow_\Gamma$ relation:[11]

$$\begin{array}{lcl} \mathsf{sterms}\ \Gamma\ (\mathsf{p} \oplus \mathsf{q}) & = & \mathsf{sterms}\ \Gamma\ \mathsf{p} \cup \mathsf{sterms}\ \Gamma\ \mathsf{q}\,, \\ \mathsf{sterms}\ \Gamma\ (\mathsf{call}(\mathsf{pn})) & = & \mathsf{sterms}\ \Gamma\ (\Gamma\ \mathsf{pn})\text{, and,} \\ \mathsf{sterms}\ \Gamma\ \mathsf{p} & = & \{\mathsf{p}\}\ \text{otherwise}. \end{array}$$

As an example, consider the sterms of the Γ_{QMSG} Qmsg process from Figure 5.

$$\mathsf{sterms}\ \Gamma_{\text{QMSG}}\ (\Gamma_{\text{QMSG}}\ \mathsf{Qmsg}) = \left\{ \begin{array}{c} \{\text{Qmsg-:0}\}\mathsf{receive}(\lambda \mathsf{msg}\ \mathsf{msgs}.\ \mathsf{msgs} \mathbin{@} [\mathsf{msg}])\ .\ \mathsf{call}(\mathsf{Qmsg})\,, \\ \{\text{Qmsg-:0}\}\langle \lambda \mathsf{msgs}.\ \mathsf{if}\ \mathsf{msgs} \neq [\,]\ \mathsf{then}\ \{\mathsf{msgs}\}\ \mathsf{else}\ \emptyset \rangle\ (\{\text{Qmsg-:1}\}\mathsf{send}(\lambda \mathsf{msgs}.\ \mathsf{hd}\ \mathsf{msgs})\ \cdots) \end{array} \right\},$$

which contains the two subterms from either side of the initial choice: one that receives and loops, and another that begins by testing the value of msgs. An execution of the Γ_{QMSG} Qmsg process amounts to an execution of one of these two terms.

We also define 'local start terms' by $\mathsf{stermsl}\ (\mathsf{p}_1 \oplus \mathsf{p}_2) = \mathsf{stermsl}\ \mathsf{p}_1 \cup \mathsf{stermsl}\ \mathsf{p}_2$ and otherwise $\mathsf{stermsl}\ \mathsf{p} = \{\mathsf{p}\}$ to permit the sufficient syntactic condition that a specification Γ is well formed if $\mathsf{call}(\mathsf{pn}') \notin \mathsf{stermsl}\ (\Gamma\ \mathsf{pn})$.

[10] A specification is well formed iff it can be converted into one that is *weakly guarded* in the sense of [26].

[11] This characterization is equivalent to $\{\mathsf{q} \mid \mathsf{p} \rightsquigarrow_\Gamma{}^* \mathsf{q} \wedge (\nexists \mathsf{q}'.\ \mathsf{q} \rightsquigarrow_\Gamma \mathsf{q}')\}$. Termination follows from wellformed Γ, that is, $\mathsf{wellformed}\ \Gamma \Longrightarrow \mathsf{sterms\text{-}dom}\ (\Gamma, \mathsf{p})$ for all p.

Since sterms Γ_{QMSG} (Γ_{QMSG} Qmsg) = stermsl (Γ_{QMSG} Qmsg), and Qmsg is the only process in Γ_{QMSG}, we can conclude that Γ_{QMSG} is well formed,

Similarly to the way that start terms act as direct sources of transitions, we define 'derivative terms' giving possible 'active' destinations of transitions.

Definition 3.6 (dterms) Given a wellformed Γ and a sequential process term p, dterms p is defined by:

$$\begin{aligned}
&\text{dterms } \Gamma\ (\mathsf{p} \oplus \mathsf{q}) &&= \text{dterms } \Gamma\ \mathsf{p} \cup \text{dterms } \Gamma\ \mathsf{q},\\
&\text{dterms } \Gamma\ (\mathsf{call(pn)}) &&= \text{dterms } \Gamma\ (\Gamma\ \mathsf{pn}),\\
&\text{dterms } \Gamma\ (\{\mathsf{l}\}\langle \mathsf{g}\rangle\ \mathsf{p}) &&= \text{sterms } \Gamma\ \mathsf{p},\\
&\text{dterms } \Gamma\ (\{\mathsf{l}\}[\![\mathsf{u}]\!]\ \mathsf{p}) &&= \text{sterms } \Gamma\ \mathsf{p},\\
&\text{dterms } \Gamma\ (\{\mathsf{l}\}\mathsf{unicast}(\mathsf{s}_{id}, \mathsf{s}_{msg})\ .\ \mathsf{p} \triangleright \mathsf{q}) &&= \text{sterms } \Gamma\ \mathsf{p} \cup \text{sterms } \Gamma\ \mathsf{q},\\
&\text{dterms } \Gamma\ (\{\mathsf{l}\}\mathsf{broadcast}(\mathsf{s}_{msg})\ .\ \mathsf{p}) &&= \text{sterms } \Gamma\ \mathsf{p},\\
&\text{dterms } \Gamma\ (\{\mathsf{l}\}\mathsf{groupcast}(\mathsf{s}_{ids}, \mathsf{s}_{msg})\ .\ \mathsf{p}) &&= \text{sterms } \Gamma\ \mathsf{p},\\
&\text{dterms } \Gamma\ (\{\mathsf{l}\}\mathsf{send}(\mathsf{s}_{msg})\ .\ \mathsf{p}) &&= \text{sterms } \Gamma\ \mathsf{p},\\
&\text{dterms } \Gamma\ (\{\mathsf{l}\}\mathsf{deliver}(\mathsf{s}_{data})\ .\ \mathsf{p}) &&= \text{sterms } \Gamma\ \mathsf{p}, \text{ and,}\\
&\text{dterms } \Gamma\ (\{\mathsf{l}\}\mathsf{receive}(\mathsf{u}_{msg})\ .\ \mathsf{p}) &&= \text{sterms } \Gamma\ \mathsf{p}.
\end{aligned}$$

For Γ_{QMSG} Qmsg, for example, we calculate dterms Γ_{QMSG} (Γ_{QMSG} Qmsg) =

$$\left\{\begin{array}{c}
\{\text{Qmsg-:0}\}\text{receive}(\lambda\text{msg msgs. msgs @ [msg])} \ .\ \text{call(Qmsg)},\\
\{\text{Qmsg-:0}\}\langle\lambda\text{msgs. if msgs} \neq [] \text{ then } \{\text{msgs}\} \text{ else } \emptyset\rangle\ (\{\text{Qmsg-:1}\}\text{send}(\lambda\text{msgs. hd msgs}) \cdots),\\
\{\text{Qmsg-:1}\}\text{send}(\lambda\text{msgs. hd msgs})\ .\ (\{\text{Qmsg-:2}\}[\![\lambda\text{msgs. tl msgs}]\!]\ \text{call(Qmsg)} \oplus \cdots),\\
\{\text{Qmsg-:1}\}\text{receive}(\lambda\text{msg msgs. msgs @ [msg])} \ .\ \text{call(Qmsg)}
\end{array}\right\}.$$

These derivative terms overapproximate the set of sterms of processes that can be reached in exactly one transition, since they do not consider the truth of guards (like msgs $\neq$ []) nor the willingness of communication partners (like receive(...)).

These auxiliary definitions lead to a succinct definition of the set of control terms of a specification.

Definition 3.7 (cterms) For a specification Γ, cterms is the smallest set where:

$$\frac{\mathsf{p} \in \text{sterms } \Gamma\ (\Gamma\ \mathsf{pn})}{\mathsf{p} \in \text{cterms } \Gamma} \qquad \frac{\mathsf{q} \in \text{cterms } \Gamma \quad \mathsf{p} \in \text{dterms } \Gamma\ \mathsf{q}}{\mathsf{p} \in \text{cterms } \Gamma}$$

There are, for example, six control terms in cterms Γ_{QMSG} =

$$\left\{\begin{array}{c}
\{\text{Qmsg-:0}\}\text{receive}(\lambda\text{msg msgs. msgs @ [msg])} \ .\ \text{call(Qmsg)},\\
\{\text{Qmsg-:0}\}\langle\lambda\text{msgs. if msgs} \neq [] \text{ then } \{\text{msgs}\} \text{ else } \emptyset\rangle\ (\{\text{Qmsg-:1}\}\text{send}(\lambda\text{msgs. hd msgs}) \cdots),\\
\{\text{Qmsg-:1}\}\text{send}(\lambda\text{msgs. hd msgs})\ .\ (\{\text{Qmsg-:2}\}[\![\lambda\text{msgs. tl msgs}]\!]\ \text{call(Qmsg)} \oplus \cdots),\\
\{\text{Qmsg-:2}\}[\![\lambda\text{msgs. tl msgs}]\!]\ \text{call(Qmsg)},\\
\{\text{Qmsg-:2}\}\text{receive}(\lambda\text{msg msgs. tl msgs @ [msg])} \ .\ \text{call(Qmsg)},\\
\{\text{Qmsg-:1}\}\text{receive}(\lambda\text{msg msgs. msgs @ [msg])} \ .\ \text{call(Qmsg)}
\end{array}\right\}.$$

In terms of the main example, the set cterms Γ_{Toy} has fourteen elements; exactly one for each printed line in Figure 3 or each transition in Figure 4.[12]

When proving state or transition invariants of the form onl Γ P or onll Γ P, these are the only control states for which the conditions of Definitions 3.2 and 3.3 need be checked.

As for sterms, it is useful to define a local version independent of any specification.

[12] Of all the control terms, only those beginning with unicast may induce more than one transition.

Definition 3.8 (ctermsl) Let ctermsl be the smallest set defined by:

$$
\begin{array}{lcl}
\mathsf{ctermsl}\ (\mathsf{p} \oplus \mathsf{q}) & = & \mathsf{ctermsl}\ \mathsf{p} \cup \mathsf{ctermsl}\ \mathsf{q}, \\
\mathsf{ctermsl}\ (\mathsf{call(pn)}) & = & \{\mathsf{call(pn)}\}, \\
\mathsf{ctermsl}\ (\{\mathsf{I}\}\langle \mathsf{g} \rangle\ \mathsf{p}) & = & \{\{\mathsf{I}\}\langle \mathsf{g} \rangle\ \mathsf{p}\} \cup \mathsf{ctermsl}\ \mathsf{p}, \\
\mathsf{ctermsl}\ (\{\mathsf{I}\}[\![\mathsf{u}]\!]\ \mathsf{p}) & = & \{\{\mathsf{I}\}[\![\mathsf{u}]\!]\ \mathsf{p}\} \cup \mathsf{ctermsl}\ \mathsf{p}, \\
\mathsf{ctermsl}\ (\{\mathsf{I}\}\mathsf{unicast}(\mathsf{s}_{id}, \mathsf{s}_{msg})\ .\ \mathsf{p} \triangleright \mathsf{q}) & = & \{\{\mathsf{I}\}\mathsf{unicast}(\mathsf{s}_{id}, \mathsf{s}_{msg})\ .\ \mathsf{p} \triangleright \mathsf{q}\} \\
 & & \cup\ (\mathsf{ctermsl}\ \mathsf{p} \cup \mathsf{ctermsl}\ \mathsf{q}), \\
\mathsf{ctermsl}\ (\{\mathsf{I}\}\mathsf{broadcast}(\mathsf{s}_{msg})\ .\ \mathsf{p}) & = & \{\{\mathsf{I}\}\mathsf{broadcast}(\mathsf{s}_{msg})\ .\ \mathsf{p}\} \cup \mathsf{ctermsl}\ \mathsf{p}, \\
\mathsf{ctermsl}\ (\{\mathsf{I}\}\mathsf{groupcast}(\mathsf{s}_{ids}, \mathsf{s}_{msg})\ .\ \mathsf{p}) & = & \{\{\mathsf{I}\}\mathsf{groupcast}(\mathsf{s}_{ids}, \mathsf{s}_{msg})\ .\ \mathsf{p}\} \cup \mathsf{ctermsl}\ \mathsf{p}, \\
\mathsf{ctermsl}\ (\{\mathsf{I}\}\mathsf{send}(\mathsf{s}_{msg})\ .\ \mathsf{p}) & = & \{\{\mathsf{I}\}\mathsf{send}(\mathsf{s}_{msg})\ .\ \mathsf{p}\} \cup \mathsf{ctermsl}\ \mathsf{p}, \\
\mathsf{ctermsl}\ (\{\mathsf{I}\}\mathsf{deliver}(\mathsf{s}_{data})\ .\ \mathsf{p}) & = & \{\{\mathsf{I}\}\mathsf{deliver}(\mathsf{s}_{data})\ .\ \mathsf{p}\} \cup \mathsf{ctermsl}\ \mathsf{p}, \text{ and}, \\
\mathsf{ctermsl}\ (\{\mathsf{I}\}\mathsf{receive}(\mathsf{u}_{msg})\ .\ \mathsf{p}) & = & \{\{\mathsf{I}\}\mathsf{receive}(\mathsf{u}_{msg})\ .\ \mathsf{p}\} \cup \mathsf{ctermsl}\ \mathsf{p}.
\end{array}
$$

For our running example we have $\mathsf{ctermsl}\ (\Gamma_{\mathrm{QMSG}}\ \mathsf{Qmsg}) = \mathsf{cterms}\ \Gamma_{\mathrm{QMSG}} \cup \{\mathsf{call(Qmsg)}\}$. Including call terms ensures that $\mathsf{q} \in \mathsf{stermsl}\ \mathsf{p}$ implies $\mathsf{q} \in \mathsf{ctermsl}\ \mathsf{p}$, which facilitates proofs. For wellformed Γ, ctermsl allows an alternative definition of cterms,

$$\mathsf{cterms}\ \Gamma = \{\mathsf{p} \mid \exists\, \mathsf{pn}.\ \mathsf{p} \in \mathsf{ctermsl}\ (\Gamma\ \mathsf{pn}) \wedge \mathsf{not\text{-}call}\ \mathsf{p}\}. \tag{3}$$

While the original definition is convenient for developing the meta-theory, due to the accompanying induction principle, this one is more useful for systematically generating the set of control terms of a specification, and thus, as we will see, sets of verification conditions. And, for wellformed Γ, we have as a corollary that

$$\mathsf{cterms}\ \Gamma = \{\mathsf{p} \mid \exists\, \mathsf{pn}.\ \mathsf{p} \in \mathsf{subterms}\ (\Gamma\ \mathsf{pn}) \wedge \mathsf{not\text{-}call}\ \mathsf{p} \wedge \mathsf{not\text{-}choice}\ \mathsf{p}\}, \tag{4}$$

where subterms, not-call, and not-choice are defined in the obvious way.

Our example already indicates that cterms over-approximates the set of start terms of reachable control states. Formally we have the following theorem.

Lemma 3.9 *For wellformed Γ and automaton A where control-within Γ (init A) and trans A = seqp-sos Γ, if $(\xi, p) \in$ reachable A I and $q \in$ sterms Γ p then $q \in$ cterms Γ.*

The predicate $\mathsf{control\text{-}within}\ \Gamma\ \mathsf{Z} = \forall\, (\xi, \mathsf{p}) \in \mathsf{Z}.\ \exists\, \mathsf{pn}.\ \mathsf{p} \in \mathsf{subterms}\ (\Gamma\ \mathsf{pn})$ serves to state that the initial control state is within the specification.

3.2 Basic proof rule and invariants

State invariants such as (1) are solved using a procedure whose soundness is justified as a theorem. The proof exploits (3) and Lemma 3.9.

Theorem 3.10 *To prove A $\models$ (I $\rightarrow$) onl Γ P, where wellformed Γ, simple-labels Γ, control-within Γ (init A), and trans A = seqp-sos Γ, it suffices*

(init) *for arbitrary $(\xi, p) \in$ init A and $I \in$ labels Γ p, to show P (ξ, I), and,*

(trans) *for arbitrary $p \in$ ctermsl $(\Gamma$ pn$)$, but not-call p, and $I \in$ labels Γ p, given that $p \in$ sterms Γ pp for some $(\xi, pp) \in$ reachable A I, to assume P (ξ, I), and then for any a with I a and any (ξ', q) such that $((\xi, p), a, (\xi', q)) \in$ seqp-sos Γ and $I' \in$ labels Γ q, to show P (ξ', I').*

Here, simple-labels $\Gamma = \forall$ pn. $\forall$ p $\in$ subterms (Γ pn). $\exists!$ l. labels Γ p = {l}: each subterm must have exactly one label, that is, $\oplus$ terms must be labelled consistently. The specification Γ_c Q = {Q-:1}⟦f⟧ call(Q) $\oplus$ {Q-:2}⟦g⟧ call(Q), for updates f and g, does not satisfy simple-labels. Overlooking the technicalities, Theorem 3.10 defines the expected set of verification conditions: we must show that a property P holds of all initial states and that it is preserved by all transitions from control terms in a specification Γ.

We incorporate this theorem into a generic tactic that (1) applies it as an introduction rule, (2) replaces p $\in$ ctermsl (Γ pn) by a disjunction over the values of pn, (3) applies Definition 3.8 and repeated simplifications of Γs and eliminations on disjunctions to generate one subgoal (verification condition) for each control term, (4) replaces control term derivatives, the subterms in Definition 3.6, by fresh variables, and, finally, (5) tries to solve each subgoal by simplification. Step 4 replaces potentially large control terms by their (labelled) heads, which is important for readability and prover performance. The tactic takes as arguments a list of existing invariants to include after having applied the introduction rule and a list of lemmas for trying to solve any subgoals that survive the final simplification. There are no schematic variables in the subgoals and we benefit greatly from Isabelle's PARALLEL_GOALS tactical [38].

In practice, one states an invariant, applies the tactic, and examines the resulting goals. One may need new lemmas for functions over the data state or explicit proofs for difficult goals. That said, we find that the tactic generally dispatches the uninteresting goals, and the remaining ones typically correspond with the cases treated explicitly in the pen-and-paper proofs [5].

Using the generic tactic, the verification of (1) is fully automatic. Isabelle rapidly dispatches the fourteen cases; one for each element of ctermsl Γ_{Toy}.

For transition invariants, we show a counterpart to Theorem 3.10, and declare it to the tactic described above.

Theorem 3.11 *To prove A $\models$ (I $\rightarrow$) onll Γ P, where* wellformed Γ, simple-labels Γ, control-within Γ (init A), *and* trans A = seqp-sos Γ, *it suffices for arbitrary* p $\in$ ctermsl (Γ pn), *but* not-call p, *and* l $\in$ labels Γ p, *given that* p $\in$ sterms Γ pp *for some* (ξ, pp) $\in$ reachable A I, *for any* a *with* I a, *and for any* (ξ', q) *such that* ((ξ, p), a, (ξ', q)) $\in$ seqp-sos Γ *and* l' $\in$ labels Γ q, *to show* P ((ξ, l), a, (ξ', l')).

Again, stripped of its technicalities, this theorem simply requires checking a predicate P across all transitions from all control terms in a specification Γ.

Using Theorem 3.11 we can prove that, within our toy-protocol, the value of no never decreases (Equation (2)). Isabelle dispatches all cases but one, leaving the goal no $\xi \leq$ no ξ' to be shown after the update ⟦$\lambda\xi$. ξ (|nhid := sid ξ|)⟧ at line {PToy-:7}. In fact, Isabelle determines that no ξ' = num ξ, and hence it suffices to prove no $\xi \leq$ num ξ before the update. A manual inspection shows that neither no ξ nor num ξ change after the guard is evaluated and hence that the statement must be true. However, Isabelle cannot 'inspect' the specification and we must introduce an auxiliary invariant:

$$\mathsf{ptoy\ i} \models \mathsf{onl}\ \Gamma_{\mathsf{Toy}}\ (\lambda(\xi, \mathsf{l}).\ \mathsf{l} \in \{\mathsf{PToy\text{-}:7..PToy\text{-}:8}\} \longrightarrow \mathsf{no}\ \xi \leq \mathsf{num}\ \xi)\,.$$

This state invariant is proven by Isabelle immediately, using our tactic; afterwards the transition invariant ptoy i $\models$ (λ((_, -), -, (_', -)). no $\xi \leq$ no ξ') passes without difficulty.

4 Open invariance

The analysis of network protocols often requires 'inter-node' invariants, like

$$\mathsf{wf\text{-}net\text{-}tree}\ \Psi \Longrightarrow \mathsf{closed}\ (\mathsf{pnet}\ (\lambda \mathsf{i}.\ \mathsf{ptoy}\ \mathsf{i}\ \langle\langle\ \mathsf{qmsg})\ \Psi) \models\!\!\models \\ \mathsf{netglobal}\ (\lambda\sigma.\ \forall \mathsf{i}.\ \mathsf{no}\ (\sigma\ \mathsf{i}) \leq \mathsf{no}\ (\sigma\ (\mathsf{nhid}\ (\sigma\ \mathsf{i})))), \quad (5)$$

which states that, for any network topology, specified as a net-tree with disjoint node addresses (wf-net-tree Ψ), the value of no at a node is never greater than its value at the 'next hop'—the address in nhid. This is a property of a global state σ mapping addresses to corresponding local data states ξ.

We build a global state in two steps. The first step maps a tree of states to a partial function from addresses to the data states of node processes:

$$\begin{array}{lcl} \mathsf{netlift\ ps}\ (\mathsf{s}^{\mathsf{i}}_{\mathsf{R}}) & = & [\mathsf{i} \mapsto \mathsf{fst}\ (\mathsf{ps\ s})]\,, \\ \mathsf{netlift\ ps}\ (\mathsf{s} \shortparallel \mathsf{t}) & = & \mathsf{netlift\ ps\ s} +\!\!+ \mathsf{netlift\ ps\ t}\,. \end{array}$$

The netlift function is parameterized by a 'process selection' function ps that is applied to the state of a node process—that is, a state of the np i of Section 2.4. In typical applications, such a state is the local parallel composition of a protocol process and a message queue (see Section 2.2). In such a case, ps selects just the protocol process, while abstracting from the queue. The first netlift rule associates the node address i with the process data state. The fst elides the local component of the process state. The second rule concatenates the partial maps generated for each branch of the state tree. The assumption of disjoint node addresses is critical for reasoning about the resulting map.

The idea is to treat all (local) data states ξ as a single global state σ and to abstract from local details like the process control state and queue. The local details are important for stating and showing intermediate lemmas, but their inclusion in global invariants would be an unnecessary complication.

The second step in building the global state is to add default elements df for undefined addresses i. We first define the auxiliary function

$$\mathsf{default\ df\ f} = (\lambda \mathsf{i}.\ \mathsf{case\ f\ i\ of\ None} \Rightarrow \mathsf{df\ i} \mid \mathsf{Some\ s} \Rightarrow \mathsf{s})\,,$$

and then apply it to the result of netlift in the definition of netglobal. For our example we set

$$\mathsf{netglobal\ P} = \lambda \mathsf{s}.\ \mathsf{P\ (default\ toy\text{-}init\ (netlift\ fst\ s))}\,.$$

Basically, we associate a state with every node address by setting the state at non-existent addresses to the initial state (here toy-init). The advantage is that invariants and associated proofs need not consider the possibility of an undefined state or, in other words, that σ i could be None. In (5), for example, this convention avoids three guards on address definedness. One must decide, however, whether this convention is appropriate for a given property.

While we can readily state inter-node invariants of a complete model, showing them compositionally is another issue. Sections 4.1 and 4.2 present a way to state and prove such invariants at the level of sequential processes—in our example that is, with only ptoy i left of the turnstile. Sections 4.3 and 4.4 present, respectively, rules for lifting such results to network models and for recovering invariants like (5).

$$\frac{\sigma' \; \mathsf{i} = \mathsf{u} \; (\sigma \; \mathsf{i})}{((\sigma, \; \{\mathsf{l}\}[\![\mathsf{u}]\!] \; \mathsf{p}), \; \tau, \; (\sigma', \; \mathsf{p})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}} \qquad \frac{\sigma' \; \mathsf{i} \in \mathsf{g} \; (\sigma \; \mathsf{i})}{((\sigma, \; \{\mathsf{l}\}\langle \mathsf{g} \rangle \; \mathsf{p}), \; \tau, \; (\sigma', \; \mathsf{p})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{\sigma' \; \mathsf{i} = \sigma \; \mathsf{i}}{((\sigma, \; \{\mathsf{l}\}\mathsf{unicast}(\mathsf{s}_{id}, \; \mathsf{s}_{msg}) \; . \; \mathsf{p} \triangleright \mathsf{q}), \; \mathsf{unicast} \; (\mathsf{s}_{id} \; (\sigma \; \mathsf{i})) \; (\mathsf{s}_{msg} \; (\sigma \; \mathsf{i})), \; (\sigma', \; \mathsf{p})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{\sigma' \; \mathsf{i} = \sigma \; \mathsf{i}}{((\sigma, \; \{\mathsf{l}\}\mathsf{unicast}(\mathsf{s}_{id}, \; \mathsf{s}_{msg}) \; . \; \mathsf{p} \triangleright \mathsf{q}), \; \neg\mathsf{unicast} \; (\mathsf{s}_{id} \; (\sigma \; \mathsf{i})), \; (\sigma', \; \mathsf{q})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{\sigma' \; \mathsf{i} = \sigma \; \mathsf{i}}{((\sigma, \; \{\mathsf{l}\}\mathsf{broadcast}(\mathsf{s}_{msg}) \; . \; \mathsf{p}), \; \mathsf{broadcast} \; (\mathsf{s}_{msg} \; (\sigma \; \mathsf{i})), \; (\sigma', \; \mathsf{p})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{\sigma' \; \mathsf{i} = \sigma \; \mathsf{i}}{((\sigma, \; \{\mathsf{l}\}\mathsf{groupcast}(\mathsf{s}_{ids}, \; \mathsf{s}_{msg}) \; . \; \mathsf{p}), \; \mathsf{groupcast} \; (\mathsf{s}_{ids} \; (\sigma \; \mathsf{i})) \; (\mathsf{s}_{msg} \; (\sigma \; \mathsf{i})), \; (\sigma', \; \mathsf{p})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{\sigma' \; \mathsf{i} = \sigma \; \mathsf{i}}{((\sigma, \; \{\mathsf{l}\}\mathsf{send}(\mathsf{s}_{msg}) \; . \; \mathsf{p}), \; \mathsf{send} \; (\mathsf{s}_{msg} \; (\sigma \; \mathsf{i})), \; (\sigma', \; \mathsf{p})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{\sigma' \; \mathsf{i} = \mathsf{u}_{msg} \; \mathsf{msg} \; (\sigma \; \mathsf{i})}{((\sigma, \; \{\mathsf{l}\}\mathsf{receive}(\mathsf{u}_{msg}) \; . \; \mathsf{p}), \; \mathsf{receive} \; \mathsf{msg}, \; (\sigma', \; \mathsf{p})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{\sigma' \; \mathsf{i} = \sigma \; \mathsf{i}}{((\sigma, \; \{\mathsf{l}\}\mathsf{deliver}(\mathsf{s}_{data}) \; . \; \mathsf{p}), \; \mathsf{deliver} \; (\mathsf{s}_{data} \; (\sigma \; \mathsf{i})), \; (\sigma', \; \mathsf{p})) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{((\sigma, \; \mathsf{p}), \; \mathsf{a}, \; (\sigma', \; \mathsf{p}')) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}{((\sigma, \; \mathsf{p} \oplus \mathsf{q}), \; \mathsf{a}, \; (\sigma', \; \mathsf{p}')) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}} \qquad \frac{((\sigma, \; \mathsf{q}), \; \mathsf{a}, \; (\sigma', \; \mathsf{q}')) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}{((\sigma, \; \mathsf{p} \oplus \mathsf{q}), \; \mathsf{a}, \; (\sigma', \; \mathsf{q}')) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

$$\frac{((\sigma, \; \Gamma \; \mathsf{pn}), \; \mathsf{a}, \; (\sigma', \; \mathsf{p}')) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}{((\sigma, \; \mathsf{call}(\mathsf{pn})), \; \mathsf{a}, \; (\sigma', \; \mathsf{p}')) \in \mathsf{oseqp\text{-}sos} \; \Gamma \; \mathsf{i}}$$

Fig. 10 Sequential processes: oseqp-sos.

4.1 The open model

In (the standard model of) AWN as presented in Section 2, a network state is a closed parallel composition of the states of the nodes in the network, arranged in a tree structure. A state of a node is, in turn, a wrapper around a local parallel composition of states of sequential processes, each consisting of a local data state ξ and a control term p. To reason compositionally about the relations between these local data states, we introduce the *open model* of AWN. This model collects relevant information from the individual local states ξ into a single global state σ. For our applications so far, we have not needed to include *all* local state elements in this global data state; in fact we need only one local data state per node, namely the one stemming from the leftmost component of the (non-commutative) local parallel composition of processes running on that node. The leftmost component is by convention the main protocol process. This type of global state is not only sufficient for our purposes, but also easier to manipulate in Isabelle.

Recall that the data type ip contains identifiers for all nodes that could occur in a network. Since the leftmost parallel process running on a node has a local data state of type 'k, our global data state is of type ip ⇒ 'k. As described previously, identifiers that are not in a given network are mapped to default values.

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{a},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \qquad \bigwedge \mathsf{m}.\ \mathsf{a} \neq \mathsf{receive\ m}}{((\sigma,\ (\mathsf{s},\ \mathsf{t})),\ \mathsf{a},\ (\sigma',\ (\mathsf{s}',\ \mathsf{t}))) \in \mathsf{oparp\text{-}sos\ i\ T}_A\ \mathsf{T}_B}$$

$$\frac{(\mathsf{t},\ \mathsf{a},\ \mathsf{t}') \in \mathsf{T}_B \qquad \bigwedge \mathsf{m}.\ \mathsf{a} \neq \mathsf{send\ m} \qquad \sigma'\ \mathsf{i} = \sigma\ \mathsf{i}}{((\sigma,\ (\mathsf{s},\ \mathsf{t})),\ \mathsf{a},\ (\sigma',\ (\mathsf{s},\ \mathsf{t}'))) \in \mathsf{oparp\text{-}sos\ i\ T}_A\ \mathsf{T}_B}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{receive\ m},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \qquad (\mathsf{t},\ \mathsf{send\ m},\ \mathsf{t}') \in \mathsf{T}_B}{((\sigma,\ (\mathsf{s},\ \mathsf{t})),\ \tau,\ (\sigma',\ (\mathsf{s}',\ \mathsf{t}'))) \in \mathsf{oparp\text{-}sos\ i\ T}_A\ \mathsf{T}_B}$$

Fig. 11 Parallel processes: oparp-sos.

In the open model, a state of a network is described as a pair (σ, s) of such a global state and a closed parallel composition s of the *control* states of the nodes in the network. The control state of a node is a wrapper around a local parallel composition of states of sequential processes, where we take only the control term p from the state (ξ, p) of the leftmost parallel process running on the node, and the entire state from all other components. As a result, a state in the open model contains exactly the same information as a state in the default model, even if it is arranged differently.

Figures 10–14 present the SOS rules for the open model. Many of them are similar to the rules presented in Section 2; for the sake of completeness we list them nevertheless.

4.1.1 Sequential Processes

The rules for the sequential control terms in the open model, oseqp-sos, are presented in Figure 10. They are nearly identical to the ones in the original model, but have to be parameterized by an address i and constrain only that entry of the global state, either to say how it changes (σ' i = u (σ i)) or that it does not change (σ' i = σ i). These rules do not restrict changes in the data state of any other node j (j≠i). In principle, the data states of these nodes can change arbitrarily, so that any state (σ, p) has infinitely many outgoing transitions. However, the composition with other nodes, introduced in a higher layer of the process algebra, will limit the set of outgoing transitions by combining the restrictions imposed by each of the nodes.

4.1.2 Local parallel composition

The states (σ, s) in an automaton of the open model are of type (ip ⇒ 'k) × 's with ip ⇒ 'k the type of global data states and 's the type of control states. Hence, such an automaton has type ((ip ⇒ 'k) × 's, 'a) automaton. The local parallel composition of the open model pairs an open automaton with a standard one, and thus has type ((ip ⇒ 'k) × 's, 'a) automaton ⇒ ('t, 'a) automaton ⇒ ((ip ⇒ 'k) × ('s × 't), 'a) automaton.

The rules for oparp-sos, depicted in Figure 11, only allow the first sub-process to constrain σ: the global data state that appears in the parallel composition is simply taken from its first component. This choice precludes comparing the states of qmsgs (and any other local filters) across a network, but it also simplifies the mechanics and use of this layer of the framework. Since our mechanization aims at

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{unicast\ dst\ m},\ (\sigma',\ \mathsf{s}'))\in \mathsf{T}_A \qquad \mathsf{dst}\in \mathsf{R}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \{\mathsf{dst}\}{:}{*}\mathsf{cast(m)},\ (\sigma',\ \mathsf{s}'^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \neg\mathsf{unicast\ dst},\ (\sigma',\ \mathsf{s}'))\in \mathsf{T}_A \qquad \mathsf{dst}\notin \mathsf{R} \qquad \forall \mathsf{j}.\ \mathsf{j}\neq \mathsf{i}\longrightarrow \sigma'\ \mathsf{j}=\sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \tau,\ (\sigma',\ \mathsf{s}'^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{broadcast\ m},\ (\sigma',\ \mathsf{s}'))\in \mathsf{T}_A}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \mathsf{R}{:}{*}\mathsf{cast(m)},\ (\sigma',\ \mathsf{s}'^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{groupcast\ D\ m},\ (\sigma',\ \mathsf{s}'))\in \mathsf{T}_A}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ (\mathsf{R}\cap \mathsf{D}){:}{*}\mathsf{cast(m)},\ (\sigma',\ \mathsf{s}'^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{receive\ m},\ (\sigma',\ \mathsf{s}'))\in \mathsf{T}_A}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \{\mathsf{i}\}\neg\emptyset{:}\mathsf{arrive(m)},\ (\sigma',\ \mathsf{s}'^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{deliver\ d},\ (\sigma',\ \mathsf{s}'))\in \mathsf{T}_A \qquad \forall \mathsf{j}.\ \mathsf{j}\neq \mathsf{i}\longrightarrow \sigma'\ \mathsf{j}=\sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \mathsf{i{:}deliver(d)},\ (\sigma',\ \mathsf{s}'^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \tau,\ (\sigma',\ \mathsf{s}'))\in \mathsf{T}_A \qquad \forall \mathsf{j}\neq \mathsf{i}.\ \sigma'\ \mathsf{j}=\sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \tau,\ (\sigma',\ \mathsf{s}'^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{\sigma'\ \mathsf{i}=\sigma\ \mathsf{i}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \emptyset\neg\{\mathsf{i}\}{:}\mathsf{arrive(m)},\ (\sigma',\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{\sigma'\ \mathsf{i}=\sigma\ \mathsf{i}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \mathsf{connect(i,\ i')},\ (\sigma',\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}\cup\{\mathsf{i}'\}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{\sigma'\ \mathsf{i}=\sigma\ \mathsf{i}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \mathsf{connect(i',\ i)},\ (\sigma',\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}\cup\{\mathsf{i}'\}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{\sigma'\ \mathsf{i}=\sigma\ \mathsf{i}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \mathsf{disconnect(i,\ i')},\ (\sigma',\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}-\{\mathsf{i}'\}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{\sigma'\ \mathsf{i}=\sigma\ \mathsf{i}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \mathsf{disconnect(i',\ i)},\ (\sigma',\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}-\{\mathsf{i}'\}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{\mathsf{i}\neq \mathsf{i}' \qquad \mathsf{i}\neq \mathsf{i}'' \qquad \sigma'\ \mathsf{i}=\sigma\ \mathsf{i}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \mathsf{connect(i',\ i'')},\ (\sigma',\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{\mathsf{i}\neq \mathsf{i}' \qquad \mathsf{i}\neq \mathsf{i}'' \qquad \sigma'\ \mathsf{i}=\sigma\ \mathsf{i}}{((\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}),\ \mathsf{disconnect(i',\ i'')},\ (\sigma',\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}}))\in \mathsf{onode\text{-}sos}\ \mathsf{T}_A}$$

Fig. 12 Nodes: onode-sos.

the verification of (routing) protocols [5], which nearly always implement a queue, simplifying the mechanization in this way seems reasonable. The treatment of the other layers is independent of this choice. So, if our work were to be applied in another setting where queues are not used, or where data states of more than one parallel control term need to be lifted to a control state, only this layer need be adapted.

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{R{:}{*}cast(m)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \quad ((\sigma,\ \mathsf{t}),\ \mathsf{H}\neg\mathsf{K{:}arrive(m)},\ (\sigma',\ \mathsf{t}')) \in \mathsf{T}_B \quad \mathsf{H} \subseteq \mathsf{R} \quad \mathsf{K} \cap \mathsf{R} = \emptyset}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ \mathsf{R{:}{*}cast(m)},\ (\sigma',\ \mathsf{s}' \shuffle \mathsf{t}')) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{H}\neg\mathsf{K{:}arrive(m)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \quad ((\sigma,\ \mathsf{t}),\ \mathsf{R{:}{*}cast(m)},\ (\sigma',\ \mathsf{t}')) \in \mathsf{T}_B \quad \mathsf{H} \subseteq \mathsf{R} \quad \mathsf{K} \cap \mathsf{R} = \emptyset}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ \mathsf{R{:}{*}cast(m)},\ (\sigma',\ \mathsf{s}' \shuffle \mathsf{t}')) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{H}\neg\mathsf{K{:}arrive(m)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \quad ((\sigma,\ \mathsf{t}),\ \mathsf{H}'\neg\mathsf{K}'\mathsf{:arrive(m)},\ (\sigma',\ \mathsf{t}')) \in \mathsf{T}_B}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ (\mathsf{H} \cup \mathsf{H}')\neg(\mathsf{K} \cup \mathsf{K}')\mathsf{:arrive(m)},\ (\sigma',\ \mathsf{s}' \shuffle \mathsf{t}')) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{i{:}deliver(d)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ \mathsf{i{:}deliver(d)},\ (\sigma',\ \mathsf{s}' \shuffle \mathsf{t})) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{((\sigma,\ \mathsf{t}),\ \mathsf{i{:}deliver(d)},\ (\sigma',\ \mathsf{t}')) \in \mathsf{T}_B}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ \mathsf{i{:}deliver(d)},\ (\sigma',\ \mathsf{s} \shuffle \mathsf{t}')) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \tau,\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ \tau,\ (\sigma',\ \mathsf{s}' \shuffle \mathsf{t})) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B} \qquad \frac{((\sigma,\ \mathsf{t}),\ \tau,\ (\sigma',\ \mathsf{t}')) \in \mathsf{T}_B}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ \tau,\ (\sigma',\ \mathsf{s} \shuffle \mathsf{t}')) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{connect(i,\ i')},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \quad ((\sigma,\ \mathsf{t}),\ \mathsf{connect(i,\ i')},\ (\sigma',\ \mathsf{t}')) \in \mathsf{T}_B}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ \mathsf{connect(i,\ i')},\ (\sigma',\ \mathsf{s}' \shuffle \mathsf{t}')) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{disconnect(i,\ i')},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \quad ((\sigma,\ \mathsf{t}),\ \mathsf{disconnect(i,\ i')},\ (\sigma',\ \mathsf{t}')) \in \mathsf{T}_B}{((\sigma,\ \mathsf{s} \shuffle \mathsf{t}),\ \mathsf{disconnect(i,\ i')},\ (\sigma',\ \mathsf{s}' \shuffle \mathsf{t}')) \in \mathsf{opnet\text{-}sos}\ \mathsf{T}_A\ \mathsf{T}_B}$$

Fig. 13 Partial networks: opnet-sos.

4.1.3 Nodes and partial networks

The sets onode-sos (Figure 12) and opnet-sos (Figure 13) need not be parameterized by an address since they are generated inductively from lower layers. Together they constrain parts of σ. This occurs naturally for rules like those for arrive and $*$cast, where the synchronous communication serves as a conjunction of constraints on different parts of σ. But for others that normally only constrain a single element, like those for τ, assumptions $(\forall \mathsf{j} \neq \mathsf{i}.\ \sigma'\ \mathsf{j} = \sigma\ \mathsf{j})$ are introduced here and dispatched later (Section 4.4). Such assumptions aid later proofs, but they must be justified when transferring results to closed systems.

4.1.4 Complete networks

The rules for ocnet-sos are shown in Figure 14. Each rule includes a precondition that ensures that elements not addressed within a model do not change: net-ips gives the set of node addresses in a state of a partial network.

4.1.5 Application

We now show how to construct an open model. For the running example, a sequential instance of the toy protocol is defined as

$$\mathsf{optoy\ i} = (\!|\mathsf{init} = \{(\mathsf{toy\text{-}init},\ \Gamma_{\mathsf{Toy}}\ \mathsf{PToy})\},\ \mathsf{trans} = \mathsf{oseqp\text{-}sos}\ \Gamma_{\mathsf{Toy}}\ \mathsf{i}|\!)\,,$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{connect(i,\ i')},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \qquad \forall \mathsf{j}.\ \mathsf{j} \notin \mathsf{net\text{-}ips\ s} \longrightarrow \sigma'\ \mathsf{j} = \sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}),\ \mathsf{connect(i,\ i')},\ (\sigma',\ \mathsf{s}')) \in \mathsf{ocnet\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{disconnect(i,\ i')},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \qquad \forall \mathsf{j}.\ \mathsf{j} \notin \mathsf{net\text{-}ips\ s} \longrightarrow \sigma'\ \mathsf{j} = \sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}),\ \mathsf{disconnect(i,\ i')},\ (\sigma',\ \mathsf{s}')) \in \mathsf{ocnet\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{R{:}{*}cast(m)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \qquad \forall \mathsf{j}.\ \mathsf{j} \notin \mathsf{net\text{-}ips\ s} \longrightarrow \sigma'\ \mathsf{j} = \sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}),\ \tau,\ (\sigma',\ \mathsf{s}')) \in \mathsf{ocnet\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \tau,\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \qquad \forall \mathsf{j}.\ \mathsf{j} \notin \mathsf{net\text{-}ips\ s} \longrightarrow \sigma'\ \mathsf{j} = \sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}),\ \tau,\ (\sigma',\ \mathsf{s}')) \in \mathsf{ocnet\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{i{:}deliver(d)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \qquad \forall \mathsf{j}.\ \mathsf{j} \notin \mathsf{net\text{-}ips\ s} \longrightarrow \sigma'\ \mathsf{j} = \sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}),\ \mathsf{i{:}deliver(d)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{ocnet\text{-}sos}\ \mathsf{T}_A}$$

$$\frac{((\sigma,\ \mathsf{s}),\ \mathsf{\{i\}\neg K{:}arrive(Newpkt\ d\ dst)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{T}_A \qquad \forall \mathsf{j}.\ \mathsf{j} \notin \mathsf{net\text{-}ips\ s} \longrightarrow \sigma'\ \mathsf{j} = \sigma\ \mathsf{j}}{((\sigma,\ \mathsf{s}),\ \mathsf{i{:}newpkt(d,\ dst)},\ (\sigma',\ \mathsf{s}')) \in \mathsf{ocnet\text{-}sos}\ \mathsf{T}_A}$$

Fig. 14 Complete networks: ocnet-sos.

combined with the standard qmsg process into

$$\mathsf{onp\ i} = \mathsf{optoy\ i}\ \langle\langle_i\ \mathsf{qmsg}\,,$$

using the operator

$$\mathsf{A}\ \langle\langle_i\ \mathsf{B} = (\!|\mathsf{init} = \{(\sigma,\ (\mathsf{s},\ \mathsf{t}))\ |\ (\sigma,\ \mathsf{s}) \in \mathsf{init\ A} \wedge \mathsf{t} \in \mathsf{init\ B}\},\ \mathsf{trans} = \mathsf{oparp\text{-}sos\ i\ (trans\ A)\ (trans\ B)}|\!)\,,$$

and lifted to the node level via the open node constructor

$$\langle \mathsf{i} : \mathsf{onp} : \mathsf{R}_0\rangle_o = (\!|\mathsf{init} = \{(\sigma,\ \mathsf{s}^{\mathsf{i}}_{\mathsf{R}_0}) \mid (\sigma,\ \mathsf{s}) \in \mathsf{init\ onp}\},\ \mathsf{trans} = \mathsf{onode\text{-}sos\ (trans\ onp)}|\!)\,.$$

Similarly, to map a net-tree term to an open model we define:

$$\begin{aligned}
\mathsf{opnet\ onp}\ \langle \mathsf{i};\ \mathsf{R}_0\rangle &= \langle \mathsf{i} : \mathsf{onp\ i} : \mathsf{R}_0\rangle_o \\
\mathsf{opnet\ onp}\ (\Psi_1 \| \Psi_2) &= (\!|\mathsf{init} = \{(\sigma,\ \mathsf{s}_1 \uplus \mathsf{s}_2) \mid (\sigma,\ \mathsf{s}_1) \in \mathsf{init\ (opnet\ onp}\ \Psi_1) \\
&\qquad \wedge\ (\sigma,\ \mathsf{s}_2) \in \mathsf{init\ (opnet\ onp}\ \Psi_2) \\
&\qquad \wedge\ \mathsf{net\text{-}ips\ s}_1 \cap \mathsf{net\text{-}ips\ s}_2 = \emptyset\}, \\
&\quad \mathsf{trans} = \mathsf{opnet\text{-}sos\ (trans\ (opnet\ onp}\ \Psi_1))\ (\mathsf{trans\ (opnet\ onp}\ \Psi_2))|\!)\,.
\end{aligned}$$

The third requirement on initial states makes the open model non-empty only for net-trees with disjoint node addresses. Including such a constraint within the open model, rather than as a separate assumption like the wf-net-tree n in (5), eliminates an annoying technicality from the inductions described in Section 4.3. As with the extra premises in the open SOS rules, we can freely adjust the open model to facilitate proofs, but each 'encoded assumption' becomes an obligation that must be discharged in the transfer lemma of Section 4.4.

Of course, the above constructs apply to any function onp from addresses to automata in the open model, that is, any onp of type ip ⇒ ((ip ⇒ 'k) × 's, 'a) automaton.

An operator for adding the last layer is also readily defined by

$$\mathsf{oclosed\ A} = \mathsf{A}(\!|\mathsf{trans} := \mathsf{ocnet\text{-}sos\ (trans\ A)}|\!)\,,$$

giving all the definitions necessary to turn a standard model into an open one.

4.2 Open invariants

The basic definitions of reachability, invariance, and transition invariance, Definitions 3.1–3.3, apply to open models since they are given for generic automata, but constructing a compositional proof requires considering the effects of both synchronized and interleaved actions of possible environments. Our automaton A could, for instance, be a partial network, consisting of several nodes, and the environment could be another partial network running in parallel. An action performed by the environment and the automaton together, or indeed, since the distinction is unimportant here, by the automaton alone, is termed *synchronized* and an action made by the environment without the participation of the automaton is termed *interleaved*. We identify the nature of a synchronized action by the environment through the action of A that synchronizes with it. We focus first of all on the case where A is a single node i.

The proper analysis of properties of A, such as (5), often requires assumptions on the behaviour of the environment. We consider assumptions on both synchronized and interleaved actions.

A typical example for an assumption on synchronized actions is

$$(\forall \mathsf{j}.\ \mathsf{j} \neq \mathsf{i} \longrightarrow \mathsf{no}\ (\sigma\ \mathsf{j}) \leq \mathsf{no}\ (\sigma'\ \mathsf{j})) \wedge \mathsf{orecvmsg}\ \mathsf{msg\text{-}ok}\ \sigma\ \mathsf{a}\,, \tag{6}$$

where orecvmsg applies a predicate (here msg-ok) to receive actions and is otherwise true: msg-ok σ (Pkt data src) = (data $\leq$ no (σ src)) and msg-ok σ (Newpkt d dst) = True. So, the assumption manifests two properties of the environment (nodes that are not equal to i) (1) it guarantees that all nodes different from i preserve the property that the value of no cannot be decreased by the protocol; (2) whenever a Pkt message is sent, the value d stored in the message is smaller than or equal to the current value of no, stored at the sender of the message src. The synchronization occurs via the exchange of messages.

A typical example for an interleaved (un-synchronized) action *from the environment* is

$$(\forall \mathsf{j}.\ \mathsf{j} \neq \mathsf{i} \longrightarrow \mathsf{no}\ (\sigma\ \mathsf{j}) \leq \mathsf{no}\ (\sigma'\ \mathsf{j})) \wedge \sigma'\ \mathsf{i} = \sigma\ \mathsf{i}, \tag{7}$$

This assumption states that (1) nodes that are not equal to i do not decrease the value of no—as before—and (2) that the data state at node i does not change. So transitions of the environment may interleave with actions performed by node i, as long as they are 'well-behaved' and do not interfere with the state of i.

Definition 4.1 (open reachability) Given an automaton A and assumptions S and U over, respectively, synchronized and interleaved actions of the environment, oreachable A S U is the smallest set defined by the rules:

$$\frac{(\sigma,\ \mathsf{s}) \in \mathsf{init}\ \mathsf{A}}{(\sigma,\ \mathsf{s}) \in \mathsf{oreachable}\ \mathsf{A}\ \mathsf{S}\ \mathsf{U}} \qquad \frac{(\sigma,\ \mathsf{s}) \in \mathsf{oreachable}\ \mathsf{A}\ \mathsf{S}\ \mathsf{U} \quad \mathsf{U}\ \sigma\ \sigma'}{(\sigma',\ \mathsf{s}) \in \mathsf{oreachable}\ \mathsf{A}\ \mathsf{S}\ \mathsf{U}}$$

$$\frac{(\sigma,\ \mathsf{s}) \in \mathsf{oreachable}\ \mathsf{A}\ \mathsf{S}\ \mathsf{U} \quad ((\sigma,\ \mathsf{s}),\ \mathsf{a},\ (\sigma',\ \mathsf{s}')) \in \mathsf{trans}\ \mathsf{A} \quad \mathsf{S}\ \sigma\ \sigma'\ \mathsf{a}}{(\sigma',\ \mathsf{s}') \in \mathsf{oreachable}\ \mathsf{A}\ \mathsf{S}\ \mathsf{U}}.$$

The first rule declares all initial states reachable. The second declares as reachable all states that result from an interleaving transition of the environment that satisfies U and where the process s does not perform any action. In the third rule process s performs an action that yields a reachable state if s in combination

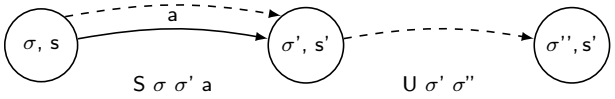

Fig. 15 Open reachability with assumptions on synchronized and interleaved actions.

with the global data state was reachable and if the assumption S is respected by the action and the environment. Figure 15 illustrates the main idea of synchronized and interleaved actions—the solid arrow represents an action a performed by state (σ, s), the dashed arrows indicate transitions taken by other nodes (on the left in synchrony with action a).

In practice, we use restricted forms otherwith E N I and other E N of the assumptions S and U, respectively:

$$\text{otherwith E N I } \sigma\ \sigma' \text{ a} = (\forall \text{i. i} \notin \text{N} \longrightarrow \text{E } (\sigma \text{ i}) \ (\sigma' \text{ i})) \wedge \text{I } \sigma \text{ a}, \text{ and} \tag{8}$$

$$\text{other E N } \sigma\ \sigma' = \forall \text{i. if i} \in \text{N then } \sigma' \text{ i} = \sigma \text{ i else E } (\sigma \text{ i}) \ (\sigma' \text{ i}). \tag{9}$$

The requirements (6) and (7), presented above, have exactly these forms.

The assumptions otherwith and other are parameterized with a set N of type ip set of *scoped* nodes—those that occur in the control states of the automaton. They both restrict the environments under consideration by applying a predicate E of type 's ⇒ 's ⇒ bool to possible changes of (local) data states of nodes i of the environment. In addition, otherwith permits constraints on the information I from shared actions, like broadcast or receive. These constraints refer to the action a and the global data state σ.

In contrast to (8), Equation (9) excludes changes in scoped nodes (σ' i = σ i).

Definition 4.2 (open invariance) Given an automaton A and assumptions S and U over synchronized and interleaved actions, respectively, a predicate P is an *open invariant*, denoted A ⊨ (S, U →) P, iff ∀ s ∈ oreachable A S U. P s.

It follows easily that existing invariants can be made open. In practice, this means that most invariants can be shown in the basic context but still exploited in the more complicated one.

Lemma 4.3 *Given an invariant A ⊨ (I →) P where trans A = seqp-sos Γ, and any predicate F, there is an open invariant A' ⊨ (λ- -. I, other F {i} →) (λ(σ, p). P (σ i, p)) where trans A' = oseqp-sos Γ i, provided that init A = {(σ i, p) | (σ, p) ∈ init A'}.*

Open transition invariance and a similar transfer lemma are defined similarly. The meta theory for basic invariants is also readily adapted, in particular,

Theorem 4.4 *To show A ⊨ (S, U →) onl Γ P, in addition to the conditions and the obligations (init) and (trans) of Theorem 3.10, suitably adjusted, it suffices,*

(env) *for arbitrary (σ, p) ∈ oreachable A S U and I ∈ labels Γ p,*
to assume both P (σ, I) and U σ σ', and then to show P (σ', I).

This theorem (together the counterpart of Theorem 3.11 for open transition invariance) is declared to the tactic described in Section 3.2 and proofs proceed as before, but with the new obligation to show invariance over interleaved transitions.

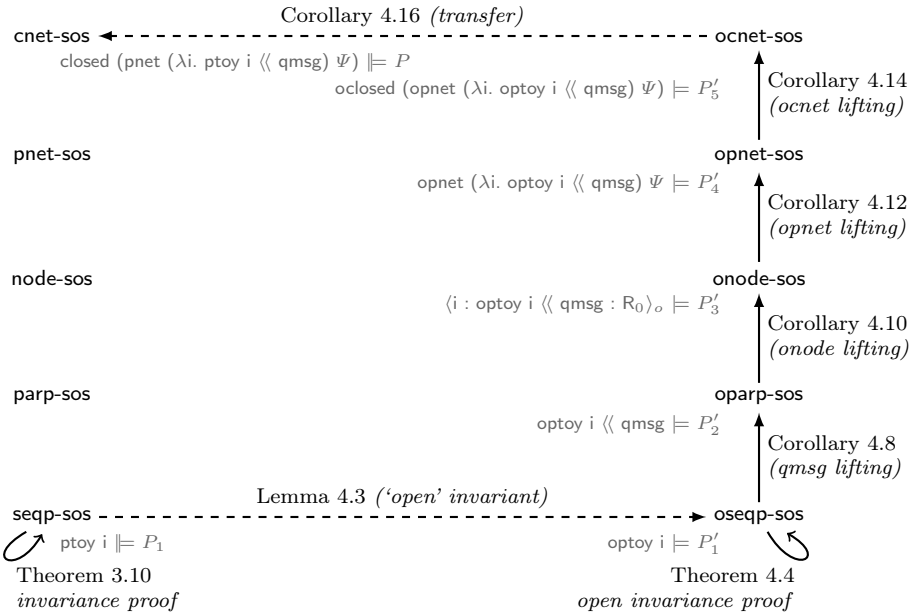

Fig. 16 Schema of the overall proof structure.

We finally have sufficient machinery to state and prove Invariant (5) at the level of a sequential process:

$$\begin{aligned}&\mathsf{optoy\ i} \models (\mathsf{otherwith\ nos\text{-}inc\ \{i\}\ (orecvmsg\ msg\text{-}ok),\ other\ nos\text{-}inc\ \{i\}} \rightarrow)\\ &\qquad(\lambda(\sigma, \text{-}).\ \mathsf{no}\ (\sigma\ \mathsf{i}) \leq \mathsf{no}\ (\sigma\ (\mathsf{nhid}\ (\sigma\ \mathsf{i})))),\end{aligned} \tag{10}$$

where nos-inc ξ ξ' = no $\xi \leq$ no ξ', So, given that the variables no in the environment never decrease and that incoming Pkts reflect the state of the sender, there is a relation between the local node and the next hop. Similar invariants occur in proofs of realistic protocols [5].

4.3 Lifting open invariants

The preceding two sections provide enough machinery to state and show global invariants at the level of sequential processes, that is, over automata like optoy i in Invariant (10). It still remains to extend such results to models of entire networks, and ultimately to re-establish them in the original model of Section 2.

Our approach is sketched in Figure 16. We prove as many invariants as possible in the closed sequential model (seqp-sos) as described in Section 3.2. These invariants are extended to the open sequential model (oseqp-sos) using Lemma 4.3, where they support proofs of the forms of global invariants described in Section 4.2. Invariants that cannot be stated in seqp-sos, because they interrelate the states of multiple nodes, are proved directly in oseqp-sos using Theorem 4.4 and its counter part for open transition invariance. Once established in oseqp-sos, global invariants can be lifted successively over the composition operators of the open model

(oparp-sos, onode-sos, opnet-sos, ocnet-sos), using the lemmas described in this section, and then transferred into the closed complete model (cnet-sos), using the lemma described in the next section. Figure 16 shows, in grey, examples of the forms of invariants at each stage. The goal is to show a property P over an entire arbitrary network in the closed model (at top-left). The property P is proven via a succession of intermediate invariants, starting with P_1, which is expressed relative to a single node, possibly in relation to the rest of the network (P_1'). At each step its form changes slightly (P_2', P_3', P_4', and P_5') to hide technical details introduced at each layer and as its range extends to multiple nodes.

The first lifting rule (Corollary 4.8) treats composition with the qmsg process. It mixes oreachable and reachable predicates: the former for the automaton being lifted, the latter for properties of qmsg. Two main properties of qmsg are required: only received messages are added to the queue and sent messages come from the queue. They are shown using the techniques of Section 3.

Lemma 4.5 *qmsg* ⊨ *(λ((msgs, q), a, (msgs', q')).*
case a of receive m ⇒ set msgs' ⊆ set (msgs @ [m])
| - ⇒ set msgs' ⊆ set msgs).

Lemma 4.6 *qmsg* ⊨ *(λ((msgs, q), a, -). sendmsg (λm. m ∈ set msgs) a).*

These two properties of qmsg are used to prove a lemma that decomposes open reachability of A ⟨|ᵢ qmsg into open reachability of A and reachability of qmsg.

Lemma 4.7 (qmsg reachability)
Given (σ, (s, (msgs, q))) ∈ oreachable (A ⟨|ᵢ qmsg) S U, with assumptions on synchronizing and interleaved transitions S = otherwith E {i} (orecvmsg M) and U = other F {i}, and provided

1. *F is reflexive,*
2. *for all ξ, ξ', E ξ ξ' implies F ξ ξ',*
3. *A ⊨ (S, U →) (λ((σ, -), -, (σ', -)). F (σ i) (σ' i)), and,*
4. *for all σ, σ', m, ∀ j. F (σ j) (σ' j) and M σ m imply M σ' m,*

then (σ, s) ∈ oreachable A S U and (msgs, q) ∈ reachable qmsg (recvmsg (M σ)), and furthermore ∀ m ∈ set msgs. M σ m.

In the qmsg part of the local state (msgs, q), msgs is a list of messages and q is the control state of the queue. We write set msgs to generate a set of messages from the list of messages. The key intuition behind the four clauses is that every message m received, queued, and sent by qmsg must satisfy M σ m. That is, the properties of messages received into the queue continue to hold—even as the environment and thus the original senders act—until those messages are transmitted from the queue to the automaton A. The proof is by induction over oreachable. The M's are preserved when the external environment acts independently (1, 4), when it acts synchronously (2, 4), and when the local process acts (3, 4).

The preceding lemma allows open reachability of the parallel composition to be decomposed into (open) reachability of its components. It follows easily that an invariant of the principle component (A) is also an invariant of the composition.

Corollary 4.8 (qmsg lifting) *Given* $A \models (S, U \rightarrow)\ (\lambda(\sigma, \text{-}).\ P\ \sigma)$*, with the predicates* $S =$ otherwith E {i} (orecvmsg M) *and* $U =$ other F {i}*, and provided*

1. F *is reflexive,*
2. *for all* ξ, ξ', $E\ \xi\ \xi'$ *implies* $F\ \xi\ \xi'$*,*
3. $A \models (S, U \rightarrow)\ (\lambda((\sigma, \text{-}), \text{-}, (\sigma', \text{-})).\ F\ (\sigma\ i)\ (\sigma'\ i))$*, and,*
4. *for all* σ, σ', m, $\forall j.\ F\ (\sigma\ j)\ (\sigma'\ j)$ *and* $M\ \sigma\ m$ *imply* $M\ \sigma'\ m$*,*

then $A\ \langle\langle_i$ qmsg $\models (S, U \rightarrow)\ (\lambda(\sigma, \text{-}).\ P\ \sigma)$.

This lifting rule is specific to the queue model presented in Figure 5. Changes to the model may necessitate changes to the rule or its proof, or indeed, a different parallel composition would require a new rule and proof. No assumptions are made, however, on the structure of A, the automaton being lifted.

The rule for lifting to the node level is straightforward since a node simply encapsulates the state of the underlying model. It is necessary, though, to adapt assumptions on receive actions (orecvmsg) to arrive actions (oarrivemsg), but this is essentially a minor technicality.

Lemma 4.9 (onode reachability) *If, for all* ξ *and* ξ', $E\ \xi\ \xi'$ *implies* $F\ \xi\ \xi'$*, then given* $(\sigma, s_R^i) \in$ oreachable $(\langle i : A : R_0 \rangle_o)$ (otherwith E {i} (oarrivemsg M)) (other F {i}) *it follows that* $(\sigma, s) \in$ oreachable A (otherwith E {i} (orecvmsg M)) (other F {i})*.*

The sole condition $E\ \xi\ \xi' \Rightarrow F\ \xi\ \xi'$ is needed because certain node-level actions—namely connect, disconnect, and $\emptyset\neg$\{i\}:arrive(m)—synchronize with the environment (giving $E\ \xi\ \xi'$) but appear to 'stutter' (requiring $F\ \xi\ \xi'$) relative to the underlying process. That is, showing oreachable by induction involves the three cases in Definition 4.1. The first two, initial states and interleaved actions, follow directly from the induction hypothesis at the node level. For the third, synchronized actions where A participates also follow directly, but when the node layer acts alone we only know that the state component of A is unchanged ($\sigma'\ i = \sigma\ i$) and that changes in the environment components ($\sigma'\ j$ for all $j \neq i$) satisfy the synchronizing assumption (E). The implication between E and F gives other F {i} and thus open reachability in A is preserved by applying the rule for an interleaved transition.

Corollary 4.10 (onode lifting) *If, for all* ξ *and* ξ', $E\ \xi\ \xi'$ *implies* $F\ \xi\ \xi'$*, then given* $A \models$ (otherwith E {i} (orecvmsg M), other F {i} $\rightarrow$) $(\lambda(\sigma, \text{-}).\ P\ \sigma)$ *it follows that*

$$\langle i : A : R_0 \rangle_o \models \text{(otherwith } E \text{ \{i\} (oarrivemsg } M\text{), other } F \text{ \{i\} } \rightarrow\text{)}\ (\lambda(\sigma, \text{-}).\ P\ \sigma).$$

The lifting rule for partial networks is the most demanding to state and prove. We require the function net-tree-ips, which gives the set of addresses in a net-tree. It is defined in the obvious way:

$$\begin{array}{lcl} \text{net-tree-ips } \langle \mathsf{i};\ R_0 \rangle & = & \{\mathsf{i}\}, \text{ and} \\ \text{net-tree-ips } (\Psi_1 \parallel \Psi_2) & = & \text{net-tree-ips } \Psi_1 \cup \text{net-tree-ips } \Psi_2\,. \end{array}$$

This function is important for bookkeeping in inductions over net-trees since it allows the identification of the nodes on either side of a partial network composition.[13] In turn, this identification determines which nodes are scoped to each side of the composition, and whose properties are assured by the induction hypothesis,

[13] Recall that the wf-net-tree condition on the disjointness of such sets is encoded in opnet.

and which are in the environment, and whose properties are thus assumed by the induction hypothesis.

Similarly to the treatment of parallel composition with qmsg, it is necessary to break the open reachability of a composition of partial networks into open reachability of both components. For this, we require transition invariants guaranteeing that the messages sent by nodes in one partial network satisfy the assumptions made by nodes in the other partial network on messages arriving from their environment. We therefore introduce the castmsg predicate: castmsg M σ (R:∗cast(m)) iff M σ m, while castmsg M σ a is true for all other a.

Lemma 4.11 (opnet reachability) *If* $(\sigma, s \| t) \in$ oreachable (opnet onp $(\Psi_1 \| \Psi_2)$) S U, *with* S = otherwith E (net-tree-ips $(\Psi_1 \| \Psi_2)$) (oarrivemsg M), U = other F (net-tree-ips $(\Psi_1 \| \Psi_2)$), *and* E *and* F *reflexive, and given*

1. $\langle i$: onp i : $R_0 \rangle_o \models (\lambda\sigma$ -. oarrivemsg M σ, other F {i} $\rightarrow)$
   $(\lambda((\sigma, -), a, (\sigma', -)).$ castmsg M σ a),
2. $\langle i$: onp i : $R_0 \rangle_o \models (\lambda\sigma$ -. oarrivemsg M σ, other F {i} $\rightarrow)$
   $(\lambda((\sigma, -), a, (\sigma', -)).\ a \neq \tau \wedge (\forall d.\ a \neq i{:}deliver(d)) \longrightarrow E\ (\sigma\ i)\ (\sigma'\ i))$, *and*
3. $\langle i$: onp i : $R_0 \rangle_o \models (\lambda\sigma$ -. oarrivemsg M σ, other F {i} $\rightarrow)$
   $(\lambda((\sigma, -), a, (\sigma', -)).\ a = \tau \vee (\exists d.\ a = i{:}deliver(d)) \longrightarrow F\ (\sigma\ i)\ (\sigma'\ i))$,

then it follows that both

1. $(\sigma, s) \in$ oreachable (opnet onp Ψ_1) S_1 U_1 *and*
2. $(\sigma, t) \in$ oreachable (opnet onp Ψ_2) S_2 U_2,

where S_1 = otherwith E (net-tree-ips Ψ_1) (oarrivemsg M), U_1 = other F (net-tree-ips Ψ_1), S_2 = otherwith E (net-tree-ips Ψ_2) (oarrivemsg M), *and* U_2 = other F (net-tree-ips Ψ_2).

The proof is by induction over oreachable. The initial and interleaved cases are trivial. For the local case, given open reachability of (σ, s) and (σ, t) for Ψ_1 and Ψ_2, respectively, and $((\sigma, s \| t), a, (\sigma', s' \| t')) \in$ trans (opnet onp $(\Psi_1 \| \Psi_2)$), we must show open reachability of (σ', s') and (σ', t'). The proof proceeds by a case distinction on actions a. The key step is to have stated the lemma without introducing cyclic dependencies between (synchronizing) assumptions and (transition-invariant) guarantees: that is, each partial network Ψ_i assumes that the other partial network Ψ_j satisfies S_j and U_j, while itself guaranteeing S_i and U_i thanks to the lifting of Conditions (2) and (3). For a synchronizing action like arrive, Definition 4.1 requires satisfaction of S_1 in order to advance in Ψ_1 and of S_2 to advance in Ψ_2, but the assumption S only guarantees that E holds for addresses j $\notin$ net-tree-ips $(\Psi_1 \| \Psi_2)$—the gap is filled by Assumption (2). This is why the transition invariants required of nodes (Conditions (1–3)) may not assume otherwith E {i}. This is not unduly restrictive, since the transition invariants provide guarantees for individual local state elements and not between network nodes. The assumption oarrivemsg M σ is never cyclic: it is either assumed of the environment for paired arrives, or trivially satisfied for the side that ∗casts.

The transition invariants are lifted from nodes to networks by induction over net-trees, using the above decomposition of open reachability. For non-synchronizing actions, we exploit the extra guarantees built into the open SOS rules.

Corollary 4.12 (opnet lifting) *Given*

$$\langle i : \mathsf{onp}\ i : R_0\rangle_o \models (\mathsf{otherwith}\ E\ \{i\}\ (\mathsf{oarrivemsg}\ M), \mathsf{other}\ F\ \{i\} \rightarrow)\ (\lambda(\sigma, \text{-}).\ P\ i\ \sigma)$$

and provided that,

1. $\langle i : \mathsf{onp}\ i : R_0\rangle_o \models (\lambda(\sigma, \text{-}).\ \mathsf{oarrivemsg}\ M\ \sigma, \mathsf{other}\ F\ \{i\} \rightarrow)$
   $(\lambda((\sigma, \text{-}), a, (\sigma', \text{-})).\ \mathsf{castmsg}\ M\ \sigma\ a)$,
2. $\langle i : \mathsf{onp}\ i : R_0\rangle_o \models (\lambda(\sigma, \text{-}).\ \mathsf{oarrivemsg}\ M\ \sigma, \mathsf{other}\ F\ \{i\} \rightarrow)$
   $(\lambda((\sigma, \text{-}), a, (\sigma', \text{-})).\ a \neq \tau \wedge (\forall d.\ a \neq i{:}\mathsf{deliver}(d)) \longrightarrow E\ (\sigma\ i)\ (\sigma'\ i))$, *and*
3. $\langle i : \mathsf{onp}\ i : R_0\rangle_o \models (\lambda(\sigma, \text{-}).\ \mathsf{oarrivemsg}\ M\ \sigma, \mathsf{other}\ F\ \{i\} \rightarrow)$
   $(\lambda((\sigma, \text{-}), a, (\sigma', \text{-})).\ a = \tau \vee (\exists d.\ a = i{:}\mathsf{deliver}(d)) \longrightarrow F\ (\sigma\ i)\ (\sigma'\ i))$,

for all i *and* R_0, *with* E *and* F *reflexive, then*

$$\mathsf{opnet\ onp}\ \Psi \models (\mathsf{otherwith}\ E\ (\mathsf{net\text{-}tree\text{-}ips}\ \Psi)\ (\mathsf{oarrivemsg}\ M), \mathsf{other}\ F\ (\mathsf{net\text{-}tree\text{-}ips}\ \Psi) \rightarrow)\ (\lambda(\sigma, \text{-}).\ \forall i \in \mathsf{net\text{-}tree\text{-}ips}\ \Psi.\ P\ i\ \sigma).$$

For transition invariants, we obtain results similar to Corollaries 4.8, 4.10, and 4.12. They are essential for discharging the three conditions of Corollary 4.12.

The rule for closed networks is similar to the others. Its important function is to eliminate the synchronizing assumption (S in the lemmas above), since messages no longer arrive from the environment. The conclusion of this rule has the form required by the transfer lemma of the next section.

Lemma 4.13 (ocnet reachability)
From $(\sigma, s) \in \mathsf{oreachable}\ (\mathsf{oclosed}\ (\mathsf{opnet\ onp}\ \Psi))\ (\lambda\text{- - -}.\ \mathsf{True})\ U$, *it follows that* $(\sigma, s) \in \mathsf{oreachable}\ (\mathsf{opnet\ onp}\ \Psi)\ (\mathsf{otherwith}\ (\mathsf{op} =)\ (\mathsf{net\text{-}tree\text{-}ips}\ \Psi)\ \mathsf{inoclosed})\ U$,[14] *where* $\mathsf{inoclosed}\ \sigma\ (H \neg K{:}\mathsf{arrive}(\mathsf{Newpkt}\ d\ dst))$, *but* $\neg\ \mathsf{inoclosed}\ \sigma\ (H \neg K{:}\mathsf{arrive}(m))$, *for all other* m, $\neg\ \mathsf{inoclosed}\ \sigma\ (i{:}\mathsf{newpkt}(d, dst))$, *and otherwise, for all other* a, $\mathsf{inoclosed}\ \sigma\ a$.

That is, reachability in opnet onp p prior to closing need not consider transitions with the action arrive for any message other than a Newpkt, nor with the action newpkt.

Corollary 4.14 (ocnet lifting)
From $\mathsf{opnet\ np}\ \Psi \models (\mathsf{otherwith}\ (\mathsf{op} =)\ (\mathsf{net\text{-}tree\text{-}ips}\ \Psi)\ \mathsf{inoclosed}, U \rightarrow)\ P$, *it follows that* $\mathsf{oclosed}\ (\mathsf{opnet\ np}\ \Psi) \models (\lambda\text{- - -}.\ \mathsf{True}, U \rightarrow)\ P$.

4.4 Transferring open invariants

The rules in the last section extend invariants over sequential processes, like (10), for example, to arbitrary, open network models. All that remains is to transfer the extended invariants to the standard model. Our approach is to define a relation between a standard automaton and an open automaton, for instance at the level of local parallel processes, and then to show that this relation implies the desired transfer property between the respective network models at the closed level.

We construct our proofs using Isabelle's *locale* feature [20], which allows one to fix a set of constants and their properties, and then to derive lemmas about them. The constants can later be instantiated with any terms that satisfy the assumed properties and the system automatically specializes the associated lemmas. Specifically, we define the locale openproc np onp sr, which relates the three constants

[14] The predicate (op =) simply compares its two arguments for equality.

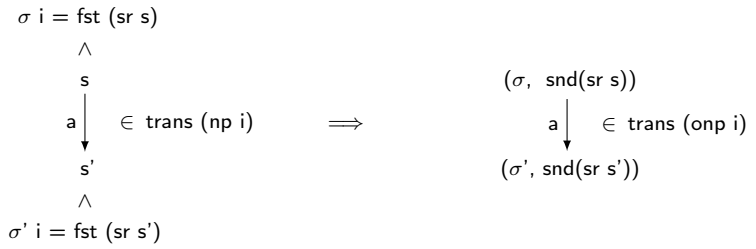

Fig. 17 Schema of the openproc np onp sr relation.

1. np of type ip ⇒ ('s, proc-action) automaton,
2. onp of type ip ⇒ ((ip ⇒ 'k) × 't, proc-action) automaton, and
3. sr of type 's ⇒ 'k × 't,

where proc-action stands for the actions on transitions in Figures 2 and 6, and where sr is a simulation relation that effectively divides the states of np i into data and control states. Unlike for the process selection function ps described in Section 4, we cannot simply discard control state elements because they are critical to formalizing and reasoning about the relationship between the two automata. The three constants must satisfy two technical conditions that guarantee that the initial states of np i are correctly 'embedded' into the (global) initial states of onp i, which we do not detail here, and a condition relating transitions across the two models. The condition on transitions is illustrated in Figure 17: for every transition (s, a, s') ∈ trans (np i), and given σ i = fst (sr s) and σ' i = fst (sr s'), it must be the case that ((σ, snd (sr s)), a, (σ', snd (sr s'))) ∈ trans (onp i). In other words, openproc np onp sr holds if onp simulates np for each component i of σ.

The simulation requirement ensures that any step of the standard model is taken into account by the corresponding open model. Indeed, for any state reachable in the standard model, a corresponding state is reachable in the open model.

Lemma 4.15 (transfer reachability) *Given np, onp, and sr such that openproc np onp sr, then for any wf-net-tree Ψ and s ∈ reachable (closed (pnet np Ψ)) (λ-. True), it follows that*
(default (someinit np sr) (netlift sr s), netliftc sr s)
∈ oreachable (oclosed (opnet onp Ψ)) (λ- - -. True) U.

This lemma uses two openproc constants: someinit np sr i chooses an arbitrary initial data state from np i,[15] with which default completes missing state elements, and netliftc lifts the control part of a process state to nodes and partial networks:

$$\begin{array}{lcl} \mathsf{netliftc\ sr\ } (\mathsf{s}^{\mathsf{i}}_{\mathsf{R}}) & = & (\mathsf{snd\ (sr\ s)})^{\mathsf{i}}_{\mathsf{R}} \\ \mathsf{netliftc\ sr\ } (\mathsf{s} \parallel \mathsf{t}) & = & (\mathsf{netliftc\ sr\ s}) \parallel (\mathsf{netliftc\ sr\ t})\,. \end{array}$$

Lemma 4.15 is shown by lifting the simulation relation to nodes and partial networks by an induction on Ψ. A separate subproof is required for each type of action and the assumptions incorporated into the corresponding open SOS rules (see Sections 4.1.3 and 4.1.4) are 'discharged' using contextual information from

[15] SOME x. x ∈ (fst ∘ sr) ' init (np i), where the '' is the image operator.

the transition in the standard model. The result is that every transition in the standard model (closed (pnet np Ψ)) is simulated by a transition in the open model (oclosed (opnet onp Ψ)). An implication from an open invariant on an open model to an invariant on the corresponding standard model follows directly.

Corollary 4.16 (transfer) *Given np, onp, and sr such that openproc np onp sr, and provided wf-net-tree Ψ, then from oclosed (opnet onp Ψ) $\models$ (λ- - -. True, U $\rightarrow$) ($\lambda(\sigma$,-). P σ), it follows that closed (pnet np Ψ) $\Vdash$ netglobal np sr P.*

In terms of our running example, we first show openproc ptoy optoy id. We then apply a generic sublocale relation for parallel composition with qmsg to obtain openproc (λi. ptoy i $\langle\langle$ qmsg) (λi. optoy i $\langle\langle_i$ qmsg) ($\lambda((\xi, p), q).\ (\xi, (p, q))$), to which we can apply Corollary 4.16 to obtain an appropriate transfer lemma. Compared to the netglobal constant in Invariant (5), the one in Corollary 4.16 is defined generically within the openproc locale and is therefore parameterized by np and sr. The former is obtained from the latter by a simple instantiation.

Summary. The technicalities of the lemmas in this and the preceding section are essential for the underlying proofs to succeed. The key idea is that through an open version of AWN where automaton states are segregated into data and control components, one can reason locally about global properties, but still, using the transfer and lifting results, obtain a result over the original model (c.f. Figure 16).

5 Concluding remarks

We present a mechanization of AWN, a modelling language for MANET and WMN protocols, including a streamlined adaptation of standard theory for showing invariants of individual reactive processes, and a novel and compositional framework for lifting such results to network models. The framework allows the statement and proof of inter-node properties. We think that many elements of our approach would apply to similarly structured models in other formalisms.

It is reasonable to ask whether the basic model presented in Section 2 could not simply be abandoned in favour of the open model of Section 4.1. We believe, however, that the basic model is the most natural way of describing what AWN means, proving semantic properties of the language, showing 'node-only' invariants, and, potentially, for showing refinement relations. Having such a reference model allows us to freely incorporate assumptions into the open SOS rules, knowing that their soundness will later be justified.

The AODV case study. The framework we present in this paper was successfully applied in the mechanization of a proof of loop freedom [11, §7] of the AODV protocol [31], a widely-used routing protocol designed for MANETs, and one of the four protocols currently standardized by the IETF MANET working group. The model has about 100 control locations across 6 different processes, and uses about 40 functions to manipulate the data state. The main property (loop freedom) roughly states that 'a data packet is never sent round in circles without being delivered'. To establish this property, we proved around 400 lemmas. Due to the complexity of the protocol logic and the length of the proof, we present the details elsewhere [5]. The case study shows that the presented framework can be applied to verification tasks of industrial relevance.

Verifying implementations. We argue in the introduction that AWN is well-adapted for modelling MANET and WMN protocols due to its support for their data structures and specialized communication primitives, and also because of its operational style. In the rest of the paper, we present techniques for the machine-assisted and compositional verification of safety properties of networks of cooperating nodes; and we claim that the AODV case study is testament to the effectiveness of this approach. An important question remains: *are AWN models suitable specifications for protocol implementations?* For instance, is it feasible to prove that a program written in C or a similar programming language correctly implements a sequential AWN process? Would it be better to try to refine or transform an AWN process into an executable form? Or simply to analyse network traces against an instantiation of the model [2]? In any case, all of these challenges require precise, and ideally mechanized, protocol models, and proofs that they satisfy given properties.

Acknowledgements We thank G. Klein and M. Pouzet for their support and complaisance, M. Daum for his participation in early discussions, and T. Sewell for sharing his talent with Isabelle. The tools Isabelle/jEdit [37], Sledgehammer [3], parallel processing [38], and the TPTP project [35] were invaluable.

References

1. Bengtson, J., Parrow, J.: Psi-calculi in Isabelle. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2009), *LNCS*, vol. 5674, pp. 99–114. Springer (2009)
2. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In: Principles of Programming Languages (POPL'06), pp. 55–66. ACM (2006)
3. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. In: N. Bjørner, V. Sofronie-Stokkermans (eds.) Conference on Automated Deduction (CADE-23), *LNCS*, vol. 6803, pp. 116–130. Springer (2011)
4. Bourke, T.: Mechanization of the Algebra for Wireless Networks (AWN). Archive of Formal Proofs (2014). `http://afp.sf.net/entries/AWN.shtml`
5. Bourke, T., van Glabbeek, R.J., Höfner, P.: A mechanized proof of loop freedom of the (untimed) AODV routing protocol. In: F. Cassez, J.F. Raskin (eds.) Automated Technology for Verification and Analysis (ATVA'14), *LNCS*, vol. 8837, pp. 47–63. Springer (2014)
6. Bourke, T., van Glabbeek, R.J., Höfner, P.: Showing invariance compositionally for a process algebra for network protocols. In: G. Klein, R. Gamboa (eds.) Interactive Theorem Proving (ITP'14), *LNCS*, vol. 8558, pp. 144–159. Springer (2014)
7. Bourke, T., Höfner, P.: Loop freedom of the (untimed) aodv routing protocol. Archive of Formal Proofs (2014). `http://afp.sf.net/entries/AODV.shtml`
8. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison Wesley (1988)
9. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA$^+$ proof system. In: J. Giesl, R. Hähnle (eds.) Int. Joint Conference on Automated Reasoning (IJCAR'10), *LNCS*, vol. 6173, pp. 142–148. Springer (2010)
10. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: H. Seidl (ed.) European Symposium on Programming (ESOP '12), *LNCS*, vol. 7211, pp. 295–315. Springer (2012)
11. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Technical Report 5513, NICTA (2013). URL `http://arxiv.org/abs/1312.7645`
12. Feliachi, A., Gaudel, M.C., Wolff, B.: Isabelle/Circus: A process specification and verification environment. In: R. Joshi, P. Müller, A. Podelski (eds.) Verified Software: Theories, Tools, and Experiments (VSTTE'12), *LNCS*, vol. 7152, pp. 243–260. Springer (2012)
13. Floyd, R.W.: Assigning meanings to programs. Proceedings of the Symposia in Applied Mathematics **19**, 19–32 (1967)

14. Fokkink, W., Groote, J.F., Reniers, M.: Process algebra needs proof methodology. In: EATCS Bulletin 82, pp. 109–125 (2004)
15. Ghassemi, F., Fokkink, W., Movaghar, A.: Restricted broadcast process theory. In: A. Cerone, S. Gruner (eds.) Software Engineering and Formal Methods (SEFM '08), pp. 345–354. IEEE Computer Society (2008)
16. Godskesen, J.C.: A calculus for mobile ad hoc networks. In: A.L. Murphy, J. Vitek (eds.) Coordination Models and Languages (COORDINATION '07), *LNCS*, vol. 4467, pp. 132–150. Springer (2007)
17. Göthel, T., Glesner, S.: An approach for machine-assisted verification of Timed CSP specifications. Innovations in Systems and Software Engineering **6**(3), 181–193 (2010)
18. Heyd, B., Crégut, P.: A modular coding of UNITY in COQ. In: G. Goos, J. Hartmanis, J. Leeuwen, J. Wright, J. Grundy, J. Harrison (eds.) Theorem Proving in Higher Order Logics (TPHOLs 1996), *LNCS*, vol. 1125, pp. 251–266. Springer (1996)
19. Hirschkoff, D.: A full formalisation of π-calculus theory in the Calculus of Constructions. In: K. Schneider, J. Brandt (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2007), *LNCS*, vol. 4732, pp. 153–169. Springer (2007)
20. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales: A sectioning concept for Isabelle. In: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Théry (eds.) Theorem Proving in Higher Order Logics (TPHOLs 1999), *LNCS*, vol. 1690, pp. 149–165. Springer (1999)
21. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison Wesley (2002)
22. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2**(3), 219–246 (1989). URL `http://theory.lcs.mit.edu/tds/papers/Lynch/CWI89.html`
23. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer (1995)
24. Merro, M.: An observational theory for mobile ad hoc networks (full version). Information and Computation **207**(2), 194–208 (2009)
25. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electronic Notes in Theoretical Computer Science (ENTCS) **158**, 331–353 (2006)
26. Milner, R.: Operational and algebraic semantics of concurrent processes. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, chap. 19, pp. 1201–1242. Elsevier Science Publishers B.V. (North-Holland) (1990). Alternatively see *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, 1989, of which an earlier version appeared as *A Calculus of Communicating Systems*, LNCS 92, Springer, 1980
27. Müller, O.: A verification environment for I/O Automata based on formalized meta-theory. Ph.D. thesis, TU München (1998)
28. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theoretical Computer Science **367**, 203–227 (2006)
29. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
30. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security **6**(1–2), 85–128 (1998)
31. Perkins, C.E., Belding-Royer, E.M., Das, S.R.: Ad hoc on-demand distance vector (AODV) routing. RFC 3561 (Experimental), Network Working Group (2003)
32. Plotkin, G.: A structural approach to operational semantics. The Journal of Logic and Algebraic Programming **60–61**, 17–139 (2004). Originally appeared in 1981
33. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Science of Computer Programming **75**, 440–469 (2010)
34. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science 54. CUP (2001)
35. Sutcliffe, G.: The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. Journal of Automated Reasoning **43**(4), 337–362 (2009)
36. Tej, H., Wolff, B.: A corrected failure divergence model for CSP in Isabelle/HOL. In: J.S. Fitzgerald, C.B. Jones, P. Lucas (eds.) Industrial Applications and Strengthened Foundations of Formal Methods (FME'97), *LNCS*, vol. 1313, pp. 318–337. Springer (1997)
37. Wenzel, M.: Isabelle/jEdit—a prover IDE within the PIDE framework. In: J. Jeuring, J.A. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, V. Sorge (eds.) Intelligent Computer Mathematics, *LNCS*, vol. 7362, pp. 468–471. Springer (2012)
38. Wenzel, M.: Shared-memory multiprocessing for interactive theorem proving. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) Interactive Theorem Proving (ITP'13), *LNCS*, vol. 7998, pp. 418–434. Springer (2013)
39. Zhou, M., Yang, H., Zhang, X., Wang, J.: The proof of AODV loop freedom. In: Int. Conf. on Wireless Communications and Signal Processing (WCSP'09). IEEE (2009)