

Mechanization of the Algebra for Wireless Networks (AWN)

Timothy Bourke*

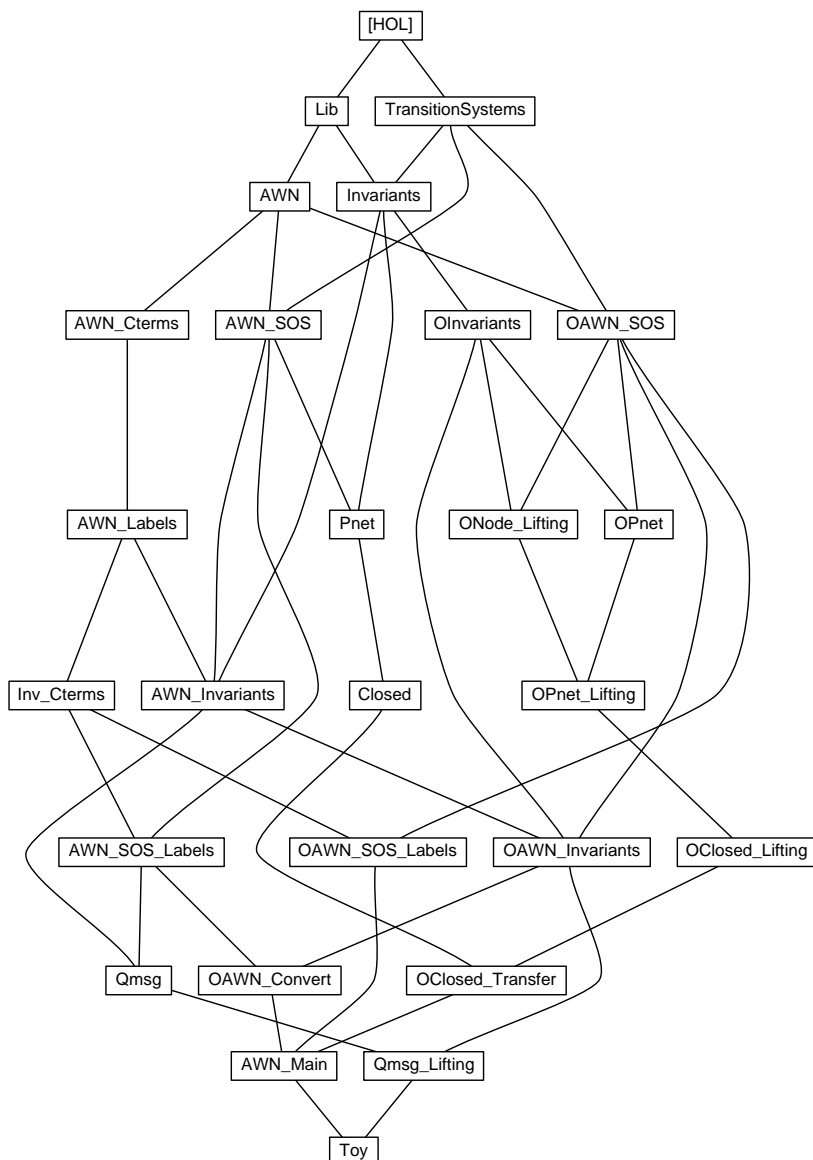
December 17, 2016

Abstract

AWN is a process algebra developed for modelling and analysing protocols for Mobile Ad hoc Networks (MANETs) and Wireless Mesh Networks (WMNs) [2, §4]. AWN models comprise five distinct layers: sequential processes, local parallel compositions, nodes, partial networks, and complete networks.

This development mechanises the original operational semantics of AWN and introduces a variant ‘open’ operational semantics that enables the compositional statement and proof of invariants across distinct network nodes. It supports labels (for weakening invariants) and (abstract) data state manipulations. A framework for compositional invariant proofs is developed, including a tactic (`inv_cterms`) for inductive invariant proofs of sequential processes, lifting rules for the open versions of the higher layers, and a rule for transferring lifted properties back to the standard semantics. A notion of ‘control terms’ reduces proof obligations to the subset of subterms that act directly (in contrast to operators for combining terms and joining processes).

Further documentation is available in [1].



*Inria, École normale supérieure, and NICTA

Contents

1	Generic functions and lemmas	4
2	Transition systems (automata)	4
3	Reachability and Invariance	4
3.1	Reachability	4
3.2	Invariance	5
4	Open reachability and invariance	7
4.1	Open reachability	7
4.2	Open Invariance	8
4.3	Standard assumption predicates	12
5	Terms of the Algebra for Wireless Networks	13
5.1	Sequential Processes	14
5.2	Actions	17
5.2.1	Sequential Actions (and related predicates)	17
5.2.2	Node Actions (and related predicates)	19
5.3	Networks	20
6	Semantics of the Algebra of Wireless Networks	23
6.1	Table 1: Structural operational semantics for sequential process expressions	24
6.2	Table 2: Structural operational semantics for parallel process expressions	25
6.3	Table 3: Structural operational semantics for node expressions	26
6.4	Table 4: Structural operational semantics for partial network expressions	28
6.5	Table 5: Structural operational semantics for complete network expressions	29
7	Control terms and well-definedness of sequential processes	30
7.1	Microsteps	30
7.2	Wellformed process specifications	32
7.3	Start terms (sterms)	32
7.4	Start terms	33
7.5	Derivative terms	35
7.6	Control terms	37
7.7	Local control terms	38
7.8	Local derivative terms	39
7.9	More properties of control terms	40
8	Labelling sequential processes	41
8.1	Labels	41
9	A custom tactic for showing invariants via control terms	44
10	Configure the inv-cterm tactic for sequential processes	44
11	Lemmas for partial networks	46
12	Lemmas for closed networks	48
13	Open semantics of the Algebra of Wireless Networks	49
13.1	Open structural operational semantics for sequential process expressions	49
13.2	Open structural operational semantics for parallel process expressions	51
13.3	Open structural operational semantics for node expressions	52
13.4	Open structural operational semantics for partial network expressions	55
13.5	Open structural operational semantics for complete network expressions	56
14	Configure the inv-cterm tactic for open sequential processes	57

15	Lemmas for open partial networks	58
16	Lifting rules for (open) nodes	59
17	Lifting rules for (open) partial networks	61
18	Lifting rules for (open) closed networks	63
19	Generic invariants on sequential AWN processes	64
19.1	Invariants via labelled control terms	64
19.2	Step invariants via labelled control terms	67
20	Generic open invariants on sequential AWN processes	70
20.1	Open invariants via labelled control terms	70
20.2	Open step invariants via labelled control terms	73
21	Transfer standard invariants into open invariants	75
22	Model the standard queuing model	77
23	Lifting rules for parallel compositions with QMSG	78
24	Transfer open results onto closed models	80
25	Import all AWN-related theories	83
26	Simple toy example	84
26.1	Messages used in the protocol	84
26.2	Protocol model	84
26.3	Define an open version of the protocol	87
26.4	Predicates	87
26.5	Sequential Invariants	88
26.6	Global Invariants	88
26.7	Lifting	90
26.8	Transfer	90
26.9	Final result	91
27	Acknowledgements	91

1 Generic functions and lemmas

```
theory Lib
imports Main
begin
```

definition

```
TT :: "'a ⇒ bool"
```

where

```
"TT = (λ_. True)"
```

lemma *TT_True* [intro, simp]: "TT a"

```
<proof>
```

lemma *in_set_tl*: "x ∈ set (tl xs) ⇒ x ∈ set xs"

```
<proof>
```

lemma *nat_le_eq_or_lt* [elim]:

```
fixes x :: nat
```

```
assumes "x ≤ y"
```

```
and eq: "x = y ⇒ P x y"
```

```
and lt: "x < y ⇒ P x y"
```

```
shows "P x y"
```

```
<proof>
```

lemma *disjoint_commute*:

```
"(A ∩ B = {}) ⇒ (B ∩ A = {})"
```

```
<proof>
```

definition

```
default :: "('i ⇒ 's) ⇒ ('i ⇒ 's option) ⇒ ('i ⇒ 's)"
```

where

```
"default df f = (λi. case f i of None ⇒ df i | Some s ⇒ s)"
```

end

2 Transition systems (automata)

```
theory TransitionSystems
```

```
imports Main
```

```
begin
```

```
type_synonym ('s, 'a) transition = "'s × 'a × 's"
```

```
record ('s, 'a) automaton =
```

```
init :: "'s set"
```

```
trans :: "('s, 'a) transition set"
```

end

3 Reachability and Invariance

```
theory Invariants
```

```
imports Lib TransitionSystems
```

```
begin
```

3.1 Reachability

A state is ‘reachable’ under I if either it is the initial state, or it is the destination of a transition whose action satisfies I from a reachable state. The ‘standard’ definition of reachability is recovered by setting I to TT .

```
inductive_set reachable
```

```
for A :: "('s, 'a) automaton"
```

```

and I :: "'a ⇒ bool"
where
  reachable_init: "s ∈ init A ⇒ s ∈ reachable A I"
  | reachable_step: "[[ s ∈ reachable A I; (s, a, s') ∈ trans A; I a ]] ⇒ s' ∈ reachable A I"

inductive_cases reachable_icas: "s ∈ reachable A I"

lemma reachable_pair_induct [consumes, case_names init step]:
  assumes "(ξ, p) ∈ reachable A I"
    and "∧ξ p. (ξ, p) ∈ init A ⇒ P ξ p"
    and "(∧ξ p ξ' p' a. [[ (ξ, p) ∈ reachable A I; P ξ p;
                          ((ξ, p), a, (ξ', p')) ∈ trans A; I a ]] ⇒ P ξ' p'"
  shows "P ξ p"
  ⟨proof⟩

lemma reachable_weakenE [elim]:
  assumes "s ∈ reachable A P"
    and PQ: "∧a. P a ⇒ Q a"
  shows "s ∈ reachable A Q"
  ⟨proof⟩

lemma reachable_weaken_TT [elim]:
  assumes "s ∈ reachable A I"
  shows "s ∈ reachable A TT"
  ⟨proof⟩

lemma init_empty_reachable_empty:
  assumes "init A = {}"
  shows "reachable A I = {}"
  ⟨proof⟩

```

3.2 Invariance

```

definition invariant
  :: "('s, 'a) automaton ⇒ ('a ⇒ bool) ⇒ ('s ⇒ bool) ⇒ bool"
  ("_ ⊨ (1'(_ →'))/_" [100, 0, 9] 8)

```

```

where
  "(A ⊨ (I →) P) = (∀s∈reachable A I. P s)"

```

abbreviation

```

any_invariant
  :: "('s, 'a) automaton ⇒ ('s ⇒ bool) ⇒ bool"
  ("_ ⊨ _" [100, 9] 8)

```

```

where
  "(A ⊨ P) ≡ (A ⊨ (TT →) P)"

```

```

lemma invariantI [intro]:
  assumes init: "∧s. s ∈ init A ⇒ P s"
    and step: "∧s a s'. [[ s ∈ reachable A I; P s; (s, a, s') ∈ trans A; I a ]] ⇒ P s'"
  shows "A ⊨ (I →) P"
  ⟨proof⟩

```

```

lemma invariant_pairI [intro]:
  assumes init: "∧ξ p. (ξ, p) ∈ init A ⇒ P (ξ, p)"
    and step: "∧ξ p ξ' p' a.
              [[ (ξ, p) ∈ reachable A I; P (ξ, p); ((ξ, p), a, (ξ', p')) ∈ trans A; I a ]]
              ⇒ P (ξ', p'"
  shows "A ⊨ (I →) P"
  ⟨proof⟩

```

```

lemma invariant_arbitraryI:
  assumes "∧s. s ∈ reachable A I ⇒ P s"
  shows "A ⊨ (I →) P"
  ⟨proof⟩

```

```

lemma invariantD [dest]:
  assumes "A  $\models$  (I  $\rightarrow$ ) P"
    and "s  $\in$  reachable A I"
  shows "P s"
  <proof>

lemma invariant_initE [elim]:
  assumes invP: "A  $\models$  (I  $\rightarrow$ ) P"
    and init: "s  $\in$  init A"
  shows "P s"
  <proof>

lemma invariant_weakenE [elim]:
  fixes T  $\sigma$  P Q
  assumes invP: "A  $\models$  (PI  $\rightarrow$ ) P"
    and PQ: " $\bigwedge$ s. P s  $\implies$  Q s"
    and QUIPI: " $\bigwedge$ a. QI a  $\implies$  PI a"
  shows "A  $\models$  (QI  $\rightarrow$ ) Q"
  <proof>

definition
  step_invariant
  :: "('s, 'a) automaton  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  (('s, 'a) transition  $\Rightarrow$  bool)  $\Rightarrow$  bool"
  ("_  $\models_A$  (1'(_  $\rightarrow$ '))/_)" [100, 0, 0] 8)
where
  "(A  $\models_A$  (I  $\rightarrow$ ) P) = ( $\forall$ a. I a  $\longrightarrow$  ( $\forall$ s $\in$ reachable A I. ( $\forall$ s'.(s, a, s')  $\in$  trans A  $\longrightarrow$  P (s, a, s'))))"\models (TT  $\rightarrow$ ) P"
  shows "A  $\models$  (QI  $\rightarrow$ ) P"
  <proof>

abbreviation
  any_step_invariant
  :: "('s, 'a) automaton  $\Rightarrow$  (('s, 'a) transition  $\Rightarrow$  bool)  $\Rightarrow$  bool"
  ("_  $\models_A$  _" [100, 9] 8)
where
  "(A  $\models_A$  P)  $\equiv$  (A  $\models_A$  (TT  $\rightarrow$ ) P)"

lemma step_invariant_true:
  "P  $\models_A$  ( $\lambda$ (s, a, s'). True)"
  <proof>

lemma step_invariantI [intro]:
  assumes *: " $\bigwedge$ s a s'. [ $s \in$ reachable A I; (s, a, s') $\in$ trans A; I a]  $\implies$  P (s, a, s')"
  shows "A  $\models_A$  (I  $\rightarrow$ ) P"
  <proof>

lemma step_invariantD [dest]:
  assumes "A  $\models_A$  (I  $\rightarrow$ ) P"
    and "s $\in$ reachable A I"
    and "(s, a, s')  $\in$  trans A"
    and "I a"
  shows "P (s, a, s')"
  <proof>

lemma step_invariantE [elim]:
  fixes T  $\sigma$  P I s a s'
  assumes "A  $\models_A$  (I  $\rightarrow$ ) P"
    and "s $\in$ reachable A I"
    and "(s, a, s')  $\in$  trans A"
    and "I a"
    and "P (s, a, s')  $\implies$  Q"

```

```

  shows "Q"
  ⟨proof⟩

```

```

lemma step_invariant_pairI [intro]:

```

```

  assumes *: "∧ξ p ξ' p' a.
             [ (ξ, p) ∈ reachable A I; ((ξ, p), a, (ξ', p')) ∈ trans A; I a ]
             ⇒ P ((ξ, p), a, (ξ', p'))"

```

```

  shows "A ⊨A (I →) P"
  ⟨proof⟩

```

```

lemma step_invariant_arbitraryI:

```

```

  assumes "∧ξ p a ξ' p'. [ (ξ, p) ∈ reachable A I; ((ξ, p), a, (ξ', p')) ∈ trans A; I a ]
          ⇒ P ((ξ, p), a, (ξ', p'))"

```

```

  shows "A ⊨A (I →) P"
  ⟨proof⟩

```

```

lemma step_invariant_weakenE [elim!]:

```

```

  fixes T σ P Q
  assumes invP: "A ⊨A (PI →) P"
    and PQ: "∧t. P t ⇒ Q t"
    and QUIPI: "∧a. QI a ⇒ PI a"
  shows "A ⊨A (QI →) Q"

```

```

  ⟨proof⟩

```

```

lemma step_invariant_weaken_with_invariantE [elim]:

```

```

  assumes pinv: "A ⊨ (I →) P"
    and qinv: "A ⊨A (I →) Q"
    and wr: "∧s a s'. [ P s; Q (s, a, s'); I a ] ⇒ R (s, a, s')"
  shows "A ⊨A (I →) R"

```

```

  ⟨proof⟩

```

```

lemma step_to_invariantI:

```

```

  assumes sinv: "A ⊨A (I →) Q"
    and init: "∧s. s ∈ init A ⇒ P s"
    and step: "∧s s' a.
              [ s ∈ reachable A I;
                P s;
                Q (s, a, s');
                I a ] ⇒ P s'"

```

```

  shows "A ⊨ (I →) P"

```

```

  ⟨proof⟩

```

```

end

```

4 Open reachability and invariance

```

theory OInvariants

```

```

imports Invariants

```

```

begin

```

4.1 Open reachability

By convention, the states of an open automaton are pairs. The first component is considered to be the global state and the second is the local state.

A state is ‘open reachable’ under S and U if it is the initial state, or it is the destination of a transition—where the global components satisfy S —from an open reachable state, or it is the destination of an interleaved environment step where the global components satisfy U .

```

inductive_set oreachable

```

```

  :: "('g × 'l, 'a) automaton
     ⇒ ('g ⇒ 'g ⇒ 'a ⇒ bool)
     ⇒ ('g ⇒ 'g ⇒ bool)
     ⇒ ('g × 'l) set"

```

```

for A :: "('g × 'l, 'a) automaton"
and S :: "'g ⇒ 'g ⇒ 'a ⇒ bool"
and U :: "'g ⇒ 'g ⇒ bool"
where
  oreachable_init: "s ∈ init A ⇒ s ∈ oreachable A S U"
  / oreachable_local: "[[ s ∈ oreachable A S U; (s, a, s') ∈ trans A; S (fst s) (fst s') a ]]
    ⇒ s' ∈ oreachable A S U"
  / oreachable_other: "[[ s ∈ oreachable A S U; U (fst s) σ' ]]
    ⇒ (σ', snd s) ∈ oreachable A S U"

lemma oreachable_local' [elim]:
  assumes "(σ, p) ∈ oreachable A S U"
    and "((σ, p), a, (σ', p')) ∈ trans A"
    and "S σ σ' a"
  shows "(σ', p') ∈ oreachable A S U"
  ⟨proof⟩

lemma oreachable_other' [elim]:
  assumes "(σ, p) ∈ oreachable A S U"
    and "U σ σ'"
  shows "(σ', p) ∈ oreachable A S U"
  ⟨proof⟩

lemma oreachable_pair_induct [consumes, case_names init other local]:
  assumes "(σ, p) ∈ oreachable A S U"
    and "∧σ p. (σ, p) ∈ init A ⇒ P σ p"
    and "(∧σ p σ'. [[ (σ, p) ∈ oreachable A S U; P σ p; U σ σ' ]] ⇒ P σ' p)"
    and "(∧σ p σ' p' a. [[ (σ, p) ∈ oreachable A S U; P σ p;
      ((σ, p), a, (σ', p')) ∈ trans A; S σ σ' a ]] ⇒ P σ' p)"
  shows "P σ p"
  ⟨proof⟩

lemma oreachable_weakenE [elim]:
  assumes "s ∈ oreachable A PS PU"
    and PSQS: "∧s s' a. PS s s' a ⇒ QS s s' a"
    and PUQU: "∧s s'. PU s s' ⇒ QU s s'"
  shows "s ∈ oreachable A QS QU"
  ⟨proof⟩

definition
  act :: "('a ⇒ bool) ⇒ 's ⇒ 's ⇒ 'a ⇒ bool"
where
  "act I ≡ (λ_ _ . I)"

lemma act_simp [iff]: "act I s s' a = I a"
  ⟨proof⟩

lemma reachable_in_oreachable [elim]:
  fixes s
  assumes "s ∈ reachable A I"
  shows "s ∈ oreachable A (act I) U"
  ⟨proof⟩



## 4.2 Open Invariance



definition oinvariant
  :: "('g × 'l, 'a) automaton
    ⇒ ('g ⇒ 'g ⇒ 'a ⇒ bool) ⇒ ('g ⇒ 'g ⇒ bool)
    ⇒ (('g × 'l) ⇒ bool) ⇒ bool"
  ("_ ⊨ (I'((1_)/ (1_ → ')/ _)" [100, 0, 0, 9] 8)
where
  "(A ⊨ (S, U →) P) = (∀s∈oreachable A S U. P s)"

lemma oinvariantI [intro]:

```



```

fixes T TI S U P
assumes init: " $\bigwedge s. s \in \text{init } A \implies P \ s$ "
and other: " $\bigwedge g \ g' \ l. \llbracket (g, l) \in \text{oreachable } A \ S \ U; P \ (g, l); U \ g \ g' \rrbracket \implies P \ (g', l)$ "
and local: " $\bigwedge s \ a \ s'. \llbracket s \in \text{oreachable } A \ S \ U; P \ s; (s, a, s') \in \text{trans } A; S \ (\text{fst } s) \ (\text{fst } s') \ a \rrbracket \implies P \ s'$ "
shows " $A \models (S, U \rightarrow) P$ "
<proof>

lemma oinvariant_oreachableI:
assumes " $\bigwedge \sigma \ s. (\sigma, s) \in \text{oreachable } A \ S \ U \implies P \ (\sigma, s)$ "
shows " $A \models (S, U \rightarrow) P$ "
<proof>

lemma oinvariant_pairI [intro]:
assumes init: " $\bigwedge \sigma \ p. (\sigma, p) \in \text{init } A \implies P \ (\sigma, p)$ "
and local: " $\bigwedge \sigma \ p \ \sigma' \ p' \ a. \llbracket (\sigma, p) \in \text{oreachable } A \ S \ U; P \ (\sigma, p); ((\sigma, p), a, (\sigma', p')) \in \text{trans } A; S \ \sigma \ \sigma' \ a \rrbracket \implies P \ (\sigma', p')$ "
and other: " $\bigwedge \sigma \ \sigma' \ p. \llbracket (\sigma, p) \in \text{oreachable } A \ S \ U; P \ (\sigma, p); U \ \sigma \ \sigma' \rrbracket \implies P \ (\sigma', p)$ "
shows " $A \models (S, U \rightarrow) P$ "
<proof>

lemma oinvariantD [dest]:
assumes " $A \models (S, U \rightarrow) P$ "
and " $s \in \text{oreachable } A \ S \ U$ "
shows " $P \ s$ "
<proof>

lemma oinvariant_initD [dest, elim]:
assumes invP: " $A \models (S, U \rightarrow) P$ "
and init: " $s \in \text{init } A$ "
shows " $P \ s$ "
<proof>

lemma oinvariant_weakenE [elim!]:
assumes invP: " $A \models (PS, PU \rightarrow) P$ "
and PQ: " $\bigwedge s. P \ s \implies Q \ s$ "
and QSPS: " $\bigwedge s \ s' \ a. QS \ s \ s' \ a \implies PS \ s \ s' \ a$ "
and QUPU: " $\bigwedge s \ s'. QU \ s \ s' \implies PU \ s \ s'$ "
shows " $A \models (QS, QU \rightarrow) Q$ "
<proof>

lemma oinvariant_weakenD [dest]:
assumes " $A \models (S', U' \rightarrow) P$ "
and " $(\sigma, p) \in \text{oreachable } A \ S \ U$ "
and weakenS: " $\bigwedge s \ s' \ a. S \ s \ s' \ a \implies S' \ s \ s' \ a$ "
and weakenU: " $\bigwedge s \ s'. U \ s \ s' \implies U' \ s \ s'$ "
shows " $P \ (\sigma, p)$ "
<proof>

lemma close_open_invariant:
assumes oinv: " $A \models (\text{act } I, U \rightarrow) P$ "
shows " $A \models (I \rightarrow) P$ "
<proof>

definition local_steps :: " $((('i \Rightarrow 's1) \times 'l1) \times 'a \times ('i \Rightarrow 's2) \times 'l2) \text{ set} \Rightarrow 'i \text{ set} \Rightarrow \text{bool}$ "
where "local_steps T J  $\equiv$ 
 $(\forall \sigma \ \zeta \ s \ a \ \sigma' \ s'. ((\sigma, s), a, (\sigma', s')) \in T \wedge (\forall j \in J. \zeta \ j = \sigma \ j) \rightarrow (\exists \zeta'. (\forall j \in J. \zeta' \ j = \sigma' \ j) \wedge ((\zeta, s), a, (\zeta', s')) \in T))$ "

lemma local_stepsI [intro!]:
assumes " $\bigwedge \sigma \ \zeta \ s \ a \ \sigma' \ \zeta' \ s'. \llbracket ((\sigma, s), a, (\sigma', s')) \in T; \forall j \in J. \zeta \ j = \sigma \ j \rrbracket$ "

```

$$\implies (\exists \zeta'. (\forall j \in J. \zeta' j = \sigma' j) \wedge ((\zeta, s), a, (\zeta', s')) \in T)$$

shows "local_steps T J"
 <proof>

lemma local_stepsE [elim, dest]:

assumes "local_steps T J"
 and " $((\sigma, s), a, (\sigma', s')) \in T$ "
 and " $\forall j \in J. \zeta j = \sigma j$ "
 shows " $\exists \zeta'. (\forall j \in J. \zeta' j = \sigma' j) \wedge ((\zeta, s), a, (\zeta', s')) \in T$ "
 <proof>

definition other_steps :: " $((i \Rightarrow 's) \Rightarrow (i \Rightarrow 's) \Rightarrow \text{bool}) \Rightarrow 'i \text{ set} \Rightarrow \text{bool}$ "
 where "other_steps U J $\equiv \forall \sigma \sigma'. U \sigma \sigma' \longrightarrow (\forall j \in J. \sigma' j = \sigma j)$ "

lemma other_stepsI [intro!]:

assumes " $\bigwedge \sigma \sigma' j. [\![U \sigma \sigma'; j \in J]\!] \implies \sigma' j = \sigma j$ "
 shows "other_steps U J"
 <proof>

lemma other_stepsE [elim]:

assumes "other_steps U J"
 and "U $\sigma \sigma'$ "
 shows " $\forall j \in J. \sigma' j = \sigma j$ "
 <proof>

definition subreachable

where "subreachable A U J $\equiv \forall I. \forall s \in \text{oreachable A } (\lambda s s'. I) U.$
 ($\exists \sigma. (\forall j \in J. \sigma j = (\text{fst } s) j) \wedge (\sigma, \text{snd } s) \in \text{reachable A I}$)"

lemma subreachableI [intro]:

assumes "local_steps (trans A) J"
 and "other_steps U J"
 shows "subreachable A U J"
 <proof>

lemma subreachableE [elim]:

assumes "subreachable A U J"
 and " $s \in \text{oreachable A } (\lambda s s'. I) U$ "
 shows " $\exists \sigma. (\forall j \in J. \sigma j = (\text{fst } s) j) \wedge (\sigma, \text{snd } s) \in \text{reachable A I}$ "
 <proof>

lemma subreachableE_pair [elim]:

assumes "subreachable A U J"
 and " $(\sigma, s) \in \text{oreachable A } (\lambda s s'. I) U$ "
 shows " $\exists \zeta. (\forall j \in J. \zeta j = \sigma j) \wedge (\zeta, s) \in \text{reachable A I}$ "
 <proof>

lemma subreachable_otherE [elim]:

assumes "subreachable A U J"
 and " $(\sigma, l) \in \text{oreachable A } (\lambda s s'. I) U$ "
 and "U $\sigma \sigma'$ "
 shows " $\exists \zeta'. (\forall j \in J. \zeta' j = \sigma' j) \wedge (\zeta', l) \in \text{reachable A I}$ "
 <proof>

lemma open_closed_invariant:

fixes J
 assumes "A $\Vdash (I \rightarrow) P$ "
 and "subreachable A U J"
 and localp: " $\bigwedge \sigma \sigma' s. [\![\forall j \in J. \sigma' j = \sigma j; P(\sigma', s)]\!] \implies P(\sigma, s)$ "
 shows "A $\models (\text{act } I, U \rightarrow) P$ "
 <proof>

lemma oinvariant_anyact:

assumes "A $\models (\text{act } TT, U \rightarrow) P$ "

shows "A \models (S, U \rightarrow) P"
 <proof>

definition

ostep_invariant
 :: "('g \times 'l, 'a) automaton
 \Rightarrow ('g \Rightarrow 'a \Rightarrow bool) \Rightarrow ('g \Rightarrow 'g \Rightarrow bool)
 \Rightarrow (('g \times 'l, 'a) transition \Rightarrow bool) \Rightarrow bool"
 ("_ \models_A (1'((1_)/ (1_) \rightarrow)/ _) [100, 0, 0, 9] 8)

where

"(A \models_A (S, U \rightarrow) P) =
 ($\forall s \in \text{oreachable } A \ S \ U. (\forall a \ s'. (s, a, s') \in \text{trans } A \wedge S \ (\text{fst } s) \ (\text{fst } s') \ a \ \longrightarrow P \ (s, a, s'))$)"

lemma ostep_invariant_def':

"(A \models_A (S, U \rightarrow) P) = ($\forall s \in \text{oreachable } A \ S \ U. (\forall a \ s'. (s, a, s') \in \text{trans } A \wedge S \ (\text{fst } s) \ (\text{fst } s') \ a \ \longrightarrow P \ (s, a, s'))$)"

<proof>

lemma ostep_invariantI [intro]:

assumes *: " $\bigwedge \sigma \ s \ a \ \sigma' \ s'. \llbracket (\sigma, s) \in \text{oreachable } A \ S \ U; ((\sigma, s), a, (\sigma', s')) \in \text{trans } A; S \ \sigma \ \sigma' \ a \rrbracket \Longrightarrow P \ ((\sigma, s), a, (\sigma', s'))$ "

shows "A \models_A (S, U \rightarrow) P"
 <proof>

lemma ostep_invariantD [dest]:

assumes "A \models_A (S, U \rightarrow) P"
 and " $(\sigma, s) \in \text{oreachable } A \ S \ U$ "
 and " $((\sigma, s), a, (\sigma', s')) \in \text{trans } A$ "
 and "S $\sigma \ \sigma' \ a$ "

shows "P $((\sigma, s), a, (\sigma', s'))$ "
 <proof>

lemma ostep_invariantE [elim]:

assumes "A \models_A (S, U \rightarrow) P"
 and " $(\sigma, s) \in \text{oreachable } A \ S \ U$ "
 and " $((\sigma, s), a, (\sigma', s')) \in \text{trans } A$ "
 and "S $\sigma \ \sigma' \ a$ "
 and "P $((\sigma, s), a, (\sigma', s')) \Longrightarrow Q$ "

shows "Q"
 <proof>

lemma ostep_invariant_weakenE [elim!]:

assumes invP: "A \models_A (PS, PU \rightarrow) P"
 and PQ: " $\bigwedge t. P \ t \Longrightarrow Q \ t$ "
 and QSPS: " $\bigwedge \sigma \ \sigma' \ a. QS \ \sigma \ \sigma' \ a \Longrightarrow PS \ \sigma \ \sigma' \ a$ "
 and QUPU: " $\bigwedge \sigma \ \sigma'. QU \ \sigma \ \sigma' \Longrightarrow PU \ \sigma \ \sigma'$ "

shows "A \models_A (QS, QU \rightarrow) Q"
 <proof>

lemma ostep_invariant_weaken_with_invariantE [elim]:

assumes pinv: "A \models (S, U \rightarrow) P"
 and qinv: "A \models_A (S, U \rightarrow) Q"
 and wr: " $\bigwedge \sigma \ s \ a \ \sigma' \ s'. \llbracket P \ (\sigma, s); P \ (\sigma', s'); Q \ ((\sigma, s), a, (\sigma', s')); S \ \sigma \ \sigma' \ a \rrbracket \Longrightarrow R \ ((\sigma, s), a, (\sigma', s'))$ "

shows "A \models_A (S, U \rightarrow) R"
 <proof>

lemma ostep_to_invariantI:

assumes sinv: "A \models_A (S, U \rightarrow) Q"
 and init: " $\bigwedge \sigma \ s. (\sigma, s) \in \text{init } A \Longrightarrow P \ (\sigma, s)$ "
 and local: " $\bigwedge \sigma \ s \ \sigma' \ s' \ a. \llbracket (\sigma, s) \in \text{oreachable } A \ S \ U; P \ (\sigma, s); Q \ ((\sigma, s), a, (\sigma', s')) \rrbracket$ "

$S \sigma \sigma' a \Longrightarrow P (\sigma', s')$
and other: " $\bigwedge \sigma \sigma' s. \llbracket (\sigma, s) \in \text{oreachable } A \ S \ U; U \sigma \sigma'; P (\sigma, s) \rrbracket \Longrightarrow P (\sigma', s)$ "
shows " $A \models (S, U \rightarrow) P$ "
 <proof>

lemma open_closed_step_invariant:

assumes " $A \models_A (I \rightarrow) P$ "
and " $\text{local_steps } (\text{trans } A) \ J$ "
and " $\text{other_steps } U \ J$ "
and localp: " $\bigwedge \sigma \zeta a \sigma' \zeta' s s'.$
 $\llbracket \forall j \in J. \sigma \ j = \zeta \ j; \forall j \in J. \sigma' \ j = \zeta' \ j; P ((\sigma, s), a, (\sigma', s')) \rrbracket$
 $\Longrightarrow P ((\zeta, s), a, (\zeta', s'))$ "
shows " $A \models_A (\text{act } I, U \rightarrow) P$ "
 <proof>

lemma oinvariant_step_anyact:

assumes " $p \models_A (\text{act } TT, U \rightarrow) P$ "
shows " $p \models_A (S, U \rightarrow) P$ "
 <proof>

4.3 Standard assumption predicates

otherwith

definition otherwith :: " $('s \Rightarrow 's \Rightarrow \text{bool})$
 $\Rightarrow 'i \ \text{set}$
 $\Rightarrow (('i \Rightarrow 's) \Rightarrow 'a \Rightarrow \text{bool})$
 $\Rightarrow ('i \Rightarrow 's) \Rightarrow ('i \Rightarrow 's) \Rightarrow 'a \Rightarrow \text{bool}$ "
where " $\text{otherwith } Q \ I \ P \ \sigma \ \sigma' \ a \equiv (\forall i. i \notin I \longrightarrow Q (\sigma \ i) (\sigma' \ i)) \wedge P \ \sigma \ a$ "

lemma otherwithI [intro]:

assumes other: " $\bigwedge j. j \notin I \Longrightarrow Q (\sigma \ j) (\sigma' \ j)$ "
and sync: " $P \ \sigma \ a$ "
shows " $\text{otherwith } Q \ I \ P \ \sigma \ \sigma' \ a$ "
 <proof>

lemma otherwithE [elim]:

assumes " $\text{otherwith } Q \ I \ P \ \sigma \ \sigma' \ a$ "
and " $\llbracket P \ \sigma \ a; \forall j. j \notin I \longrightarrow Q (\sigma \ j) (\sigma' \ j) \rrbracket \Longrightarrow R \ \sigma \ \sigma' \ a$ "
shows " $R \ \sigma \ \sigma' \ a$ "
 <proof>

lemma otherwith_actionD [dest]:

assumes " $\text{otherwith } Q \ I \ P \ \sigma \ \sigma' \ a$ "
shows " $P \ \sigma \ a$ "
 <proof>

lemma otherwith_syncD [dest]:

assumes " $\text{otherwith } Q \ I \ P \ \sigma \ \sigma' \ a$ "
shows " $\forall j. j \notin I \longrightarrow Q (\sigma \ j) (\sigma' \ j)$ "
 <proof>

lemma otherwithEI [elim]:

assumes " $\text{otherwith } P \ I \ PO \ \sigma \ \sigma' \ a$ "
and " $\bigwedge \sigma \ a. PO \ \sigma \ a \Longrightarrow QO \ \sigma \ a$ "
shows " $\text{otherwith } P \ I \ QO \ \sigma \ \sigma' \ a$ "
 <proof>

lemma all_but:

assumes " $\bigwedge \xi. S \ \xi \ \xi$ "
and " $\sigma' \ i = \sigma \ i$ "
and " $\forall j. j \neq i \longrightarrow S (\sigma \ j) (\sigma' \ j)$ "
shows " $\forall j. S (\sigma \ j) (\sigma' \ j)$ "
 <proof>

```

lemma all_but_eq [dest]:
  assumes " $\sigma' i = \sigma i$ "
    and " $\forall j. j \neq i \longrightarrow \sigma j = \sigma' j$ "
  shows " $\sigma = \sigma'$ "
  <proof>

other

definition other :: " $('s \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow 'i \text{ set} \Rightarrow ('i \Rightarrow 's) \Rightarrow ('i \Rightarrow 's) \Rightarrow \text{bool}$ "
where "other P I  $\sigma \sigma' \equiv \forall i. \text{if } i \in I \text{ then } \sigma' i = \sigma i \text{ else } P (\sigma i) (\sigma' i)$ "

lemma otherI [intro]:
  assumes local: " $\bigwedge i. i \in I \implies \sigma' i = \sigma i$ "
    and other: " $\bigwedge j. j \notin I \implies P (\sigma j) (\sigma' j)$ "
  shows "other P I  $\sigma \sigma'$ "
  <proof>

lemma otherE [elim]:
  assumes "other P I  $\sigma \sigma'$ "
    and "[ $\forall i \in I. \sigma' i = \sigma i; \forall j. j \notin I \longrightarrow P (\sigma j) (\sigma' j)$ ]  $\implies R \sigma \sigma'$ "
  shows "R  $\sigma \sigma'$ "
  <proof>

lemma other_localD [dest]:
  "other P {i}  $\sigma \sigma' \implies \sigma' i = \sigma i$ "
  <proof>

lemma other_otherD [dest]:
  "other P {i}  $\sigma \sigma' \implies \forall j. j \neq i \longrightarrow P (\sigma j) (\sigma' j)$ "
  <proof>

lemma other_bothE [elim]:
  assumes "other P {i}  $\sigma \sigma'$ "
  obtains " $\sigma' i = \sigma i$ " and " $\forall j. j \neq i \longrightarrow P (\sigma j) (\sigma' j)$ "
  <proof>

lemma weaken_local [elim]:
  assumes "other P I  $\sigma \sigma'$ "
    and PQ: " $\bigwedge \xi \xi'. P \xi \xi' \implies Q \xi \xi'$ "
  shows "other Q I  $\sigma \sigma'$ "
  <proof>

definition global :: " $((\text{nat} \Rightarrow 's) \Rightarrow \text{bool}) \Rightarrow (\text{nat} \Rightarrow 's) \times 'local \Rightarrow \text{bool}$ "
where "global P  $\equiv (\lambda(\sigma, \_). P \sigma)$ "

lemma globalsimp [simp]: "global P s = P (fst s)"
  <proof>

definition globala :: " $((\text{nat} \Rightarrow 's, 'action) \text{ transition} \Rightarrow \text{bool})$   

 $\implies ((\text{nat} \Rightarrow 's) \times 'local, 'action) \text{ transition} \Rightarrow \text{bool}$ "
where "globala P  $\equiv (\lambda((\sigma, \_), a, (\sigma', \_)). P (\sigma, a, \sigma'))$ "

lemma globalasimp [simp]: "globala P s = P (fst (fst s), fst (snd s), fst (snd (snd s)))"
  <proof>

end

```

5 Terms of the Algebra for Wireless Networks

```

theory AWW
imports Lib
begin

```

5.1 Sequential Processes

```
type_synonym ip = nat
type_synonym data = nat
```

Most of AWN is independent of the type of messages, but the closed layer turns newpkt actions into the arrival of newpkt messages. We use a type class to maintain some abstraction (and independence from the definition of particular protocols).

```
class msg =
  fixes newpkt :: "data × ip ⇒ 'a"
  and eq_newpkt :: "'a ⇒ bool"
  assumes eq_newpkt_eq [simp]: "eq_newpkt (newpkt (d, i))"
```

Sequential process terms abstract over the types of data states ('s), messages ('m), process names ('p), and labels ('l).

```
datatype (dead 's, dead 'm, dead 'p, 'l) seqp =
  GUARD "'l" "'s ⇒ 's set" "('s, 'm, 'p, 'l) seqp"
| ASSIGN "'l" "'s ⇒ 's" "('s, 'm, 'p, 'l) seqp"
| CHOICE "('s, 'm, 'p, 'l) seqp" "('s, 'm, 'p, 'l) seqp"
| UCAST "'l" "'s ⇒ ip" "'s ⇒ 'm" "('s, 'm, 'p, 'l) seqp" "('s, 'm, 'p, 'l) seqp"
| BCAST "'l" "'s ⇒ 'm" "('s, 'm, 'p, 'l) seqp"
| GCAST "'l" "'s ⇒ ip set" "'s ⇒ 'm" "('s, 'm, 'p, 'l) seqp"
| SEND "'l" "'s ⇒ 'm" "('s, 'm, 'p, 'l) seqp"
| DELIVER "'l" "'s ⇒ data" "('s, 'm, 'p, 'l) seqp"
| RECEIVE "'l" "'m ⇒ 's ⇒ 's" "('s, 'm, 'p, 'l) seqp"
| CALL 'p
for map: labelmap
```

syntax

```
"_guard"    :: "[ 'a, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨⟨unbreakable⟩⟨_⟩⟩//_" [0, 60] 60)
"_lguard"   :: "[ 'a, 'a, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨{ }⟨unbreakable⟩⟨_⟩⟩//_" [0, 0, 60] 60)
"_ifguard"  :: "[ pptrn, bool, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨⟨unbreakable⟩⟨_ . _⟩⟩//_" [0, 0, 60] 60)

"_bassign"  :: "[ pptrn, 'a, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨⟨unbreakable⟩⟨[_ . _]⟩⟩//_" [0, 0, 60] 60)
"_lbassign" :: "[ 'a, pptrn, 'a, ('s, 'm, 'p, 'a) seqp ] ⇒ ('s, 'm, 'p, 'a) seqp"
              ("⟨{ }⟨unbreakable⟩⟨[_ . _]⟩⟩//_" [0, 0, 0, 60] 60)

"_assign"   :: "[ 'a, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨⟨unbreakable⟩⟨[_]⟩⟩//_" [0, 60] 60)
"_lassign"  :: "[ 'a, 'a, ('s, 'm, 'p, 'a) seqp ] ⇒ ('s, 'm, 'p, 'a) seqp"
              ("⟨{ }⟨unbreakable⟩⟨[_]⟩⟩//_" [0, 0, 60] 60)

"_unicast"  :: "[ 'a, 'a, ('s, 'm, 'p, unit) seqp, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨3unicast'((1(3_)/ (3_))') .//(_)/ (2> _))" [0, 0, 60, 60] 60)
"_lunicast" :: "[ 'a, 'a, 'a, ('s, 'm, 'p, 'a) seqp, ('s, 'm, 'p, 'a) seqp ] ⇒ ('s, 'm, 'p, 'a) seqp"
              ("⟨3{ }unicast'((1(3_)/ (3_))') .//(_)/ (2> _))" [0, 0, 0, 60, 60] 60)

"_bcast"    :: "[ 'a, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨3broadcast'((1(_))') .) //_" [0, 60] 60)
"_lbcast"   :: "[ 'a, 'a, ('s, 'm, 'p, 'a) seqp ] ⇒ ('s, 'm, 'p, 'a) seqp"
              ("⟨3{ }broadcast'((1(_))') .) //_" [0, 0, 60] 60)

"_gcast"    :: "[ 'a, 'a, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨3groupcast'((1(_)/ (_))') .) //_" [0, 0, 60] 60)
"_lgcast"   :: "[ 'a, 'a, 'a, ('s, 'm, 'p, 'a) seqp ] ⇒ ('s, 'm, 'p, 'a) seqp"
              ("⟨3{ }groupcast'((1(_)/ (_))') .) //_" [0, 0, 0, 60] 60)

"_send"     :: "[ 'a, ('s, 'm, 'p, unit) seqp ] ⇒ ('s, 'm, 'p, unit) seqp"
              ("⟨3send'((_)') .) //_" [0, 60] 60)
"_lsend"    :: "[ 'a, 'a, ('s, 'm, 'p, 'a) seqp ] ⇒ ('s, 'm, 'p, 'a) seqp"
```

"(3{_{send}'()')//_" [0, 0, 60] 60)

"_deliver" :: "['a, ('s, 'm, 'p, unit) seqp] ⇒ ('s, 'm, 'p, unit) seqp"
("3deliver'()')//_" [0, 60] 60)
"_ldeliver" :: "['a, 'a, ('s, 'm, 'p, 'a) seqp] ⇒ ('s, 'm, 'p, 'a) seqp"
("3{_{deliver}'()')//_" [0, 0, 60] 60)
"_receive" :: "['a, ('s, 'm, 'p, unit) seqp] ⇒ ('s, 'm, 'p, unit) seqp"
("3receive'()')//_" [0, 60] 60)
"_lreceive" :: "['a, 'a, ('s, 'm, 'p, 'a) seqp] ⇒ ('s, 'm, 'p, 'a) seqp"
("3{_{receive}'()')//_" [0, 0, 60] 60)

translations

"_guard f p" ⇒ "CONST GUARD () f p"
"_lguard l f p" ⇒ "CONST GUARD l f p"
"_ifguard ξ e p" → "CONST GUARD () (λξ. if e then {ξ} else {}) p"
"
"_assign f p" ⇒ "CONST ASSIGN () f p"
"_lassign l f p" ⇒ "CONST ASSIGN l f p"
"
"_bassign ξ e p" ⇒ "CONST ASSIGN () (λξ. e) p"
"_lbassign l ξ e p" ⇒ "CONST ASSIGN l (λξ. e) p"
"
"_unicast fip fmsg p q" ⇒ "CONST UCAST () fip fmsg p q"
"_lunicast l fip fmsg p q" ⇒ "CONST UCAST l fip fmsg p q"
"
"_bcast fmsg p" ⇒ "CONST BCAST () fmsg p"
"_lbcast l fmsg p" ⇒ "CONST BCAST l fmsg p"
"
"_gcast fipset fmsg p" ⇒ "CONST GCAST () fipset fmsg p"
"_lgcast l fipset fmsg p" ⇒ "CONST GCAST l fipset fmsg p"
"
"_send fmsg p" ⇒ "CONST SEND () fmsg p"
"_lsend l fmsg p" ⇒ "CONST SEND l fmsg p"
"
"_deliver fdata p" ⇒ "CONST DELIVER () fdata p"
"_ldeliver l fdata p" ⇒ "CONST DELIVER l fdata p"
"
"_receive fmsg p" ⇒ "CONST RECEIVE () fmsg p"
"_lreceive l fmsg p" ⇒ "CONST RECEIVE l fmsg p"

notation "CHOICE" ("()//⊕//()") [56, 55] 55)
and "CALL" ("(3call'(3_)')") [0] 60)

definition not_call :: "('s, 'm, 'p, 'l) seqp ⇒ bool"
where "not_call p ≡ ∀pn. p ≠ call(pn)"

lemma not_call_simps [simp]:

"∧ l fg p. not_call (f g) p"
"∧ l fa p. not_call (f a) p"
"∧ p1 p2. not_call (p1 ⊕ p2)"
"∧ l fip fmsg p q. not_call (unicast(fip, fmsg).p ▷ q)"
"∧ l fmsg p. not_call (broadcast(fmsg).p)"
"∧ l fips fmsg p. not_call (groupcast(fips, fmsg).p)"
"∧ l fmsg p. not_call (send(fmsg).p)"
"∧ l fdata p. not_call (deliver(fdata).p)"
"∧ l fmsg p. not_call (receive(fmsg).p)"
"∧ l pn. ¬(not_call (call(pn)))"
<proof>

definition not_choice :: "('s, 'm, 'p, 'l) seqp ⇒ bool"
where "not_choice p ≡ ∀p1 p2. p ≠ p1 ⊕ p2"

lemma not_choice_simps [simp]:

```

"∧1 fg p.      not_choice (⟦fg⟧ p)"
"∧1 fa p.      not_choice (⟦fa⟧ p)"
"∧p1 p2.      ¬(not_choice (p1 ⊕ p2))"
"∧1 fip fmsg p q. not_choice (⟦unicast(fip, fmsg).p ▷ q⟧)"
"∧1 fmsg p.     not_choice (⟦broadcast(fmsg).p⟧)"
"∧1 fips fmsg p. not_choice (⟦groupcast(fips, fmsg).p⟧)"
"∧1 fmsg p.     not_choice (⟦send(fmsg).p⟧)"
"∧1 fdata p.    not_choice (⟦deliver(fdata).p⟧)"
"∧1 fmsg p.     not_choice (⟦receive(fmsg).p⟧)"
"∧1 pn.         not_choice (call(pn))"
⟨proof⟩

```

lemma seqp_congs:

```

"∧1 fg p.  ⟦fg⟧ p = ⟦fg⟧ p"
"∧1 fa p.  ⟦fa⟧ p = ⟦fa⟧ p"
"∧p1 p2.  p1 ⊕ p2 = p1 ⊕ p2"
"∧1 fip fmsg p q.  ⟦unicast(fip, fmsg).p ▷ q⟧ = ⟦unicast(fip, fmsg).p ▷ q⟧"
"∧1 fmsg p.  ⟦broadcast(fmsg).p⟧ = ⟦broadcast(fmsg).p⟧"
"∧1 fips fmsg p.  ⟦groupcast(fips, fmsg).p⟧ = ⟦groupcast(fips, fmsg).p⟧"
"∧1 fmsg p.  ⟦send(fmsg).p⟧ = ⟦send(fmsg).p⟧"
"∧1 fdata p.  ⟦deliver(fdata).p⟧ = ⟦deliver(fdata).p⟧"
"∧1 fmsg p.  ⟦receive(fmsg).p⟧ = ⟦receive(fmsg).p⟧"
"∧1 pn.  call(pn) = call(pn)"
⟨proof⟩

```

Remove data expressions from process terms.

```
fun seqp_skeleton :: "(s, m, p, l) seqp ⇒ (unit, unit, p, l) seqp"
```

where

```

"seqp_skeleton (⟦_⟧ p)          = ⟦λ_. {}⟧ (seqp_skeleton p)"
| "seqp_skeleton (⟦_⟧ p)      = ⟦λ_. ()⟧ (seqp_skeleton p)"
| "seqp_skeleton (p ⊕ q)      = (seqp_skeleton p) ⊕ (seqp_skeleton q)"
| "seqp_skeleton (⟦unicast(_, _) . p ▷ q⟧) = ⟦unicast(λ_. 0, λ_. ()) . (seqp_skeleton p) ▷ (seqp_skeleton q)⟧"
| "seqp_skeleton (⟦broadcast(_). p⟧) = ⟦broadcast(λ_. ()) . (seqp_skeleton p)⟧"
| "seqp_skeleton (⟦groupcast(_, _) . p⟧) = ⟦groupcast(λ_. {}, λ_. ()) . (seqp_skeleton p)⟧"
| "seqp_skeleton (⟦send(_). p⟧) = ⟦send(λ_. ()) . (seqp_skeleton p)⟧"
| "seqp_skeleton (⟦deliver(_). p⟧) = ⟦deliver(λ_. 0) . (seqp_skeleton p)⟧"
| "seqp_skeleton (⟦receive(_). p⟧) = ⟦receive(λ_. ()) . (seqp_skeleton p)⟧"
| "seqp_skeleton (call(pn)) = call(pn)"

```

Calculate the subterms of a term.

```
fun subterms :: "(s, m, p, l) seqp ⇒ (s, m, p, l) seqp set"
```

where

```

"subterms (⟦fg⟧ p) = {⟦fg⟧ p} ∪ subterms p"
| "subterms (⟦fa⟧ p) = {⟦fa⟧ p} ∪ subterms p"
| "subterms (p1 ⊕ p2) = {p1 ⊕ p2} ∪ subterms p1 ∪ subterms p2"
| "subterms (⟦unicast(fip, fmsg). p ▷ q⟧) =
  {⟦unicast(fip, fmsg). p ▷ q⟧} ∪ subterms p ∪ subterms q"
| "subterms (⟦broadcast(fmsg). p⟧) = {⟦broadcast(fmsg). p⟧} ∪ subterms p"
| "subterms (⟦groupcast(fips, fmsg). p⟧) = {⟦groupcast(fips, fmsg). p⟧} ∪ subterms p"
| "subterms (⟦send(fmsg). p⟧) = {⟦send(fmsg). p⟧} ∪ subterms p"
| "subterms (⟦deliver(fdata). p⟧) = {⟦deliver(fdata). p⟧} ∪ subterms p"
| "subterms (⟦receive(fmsg). p⟧) = {⟦receive(fmsg). p⟧} ∪ subterms p"
| "subterms (call(pn)) = {call(pn)}"

```

```
lemma subterms_refl [simp]: "p ∈ subterms p"
```

⟨proof⟩

```
lemma subterms_trans [elim]:
```

```

  assumes "q ∈ subterms p"
  and "r ∈ subterms q"
  shows "r ∈ subterms p"

```

⟨proof⟩


```

lemma root_in_subterms [simp]:
  " $\bigwedge \Gamma pn. \exists pn'. \Gamma pn \in \text{subterms } (\Gamma pn')$ "
  <proof>

```

```

lemma deriv_in_subterms [elim, dest]:
  " $\bigwedge l f p q. \{l\}(f) q \in \text{subterms } p \implies q \in \text{subterms } p$ "
  " $\bigwedge l fa p q. \{l\}[[fa]] q \in \text{subterms } p \implies q \in \text{subterms } p$ "
  " $\bigwedge p1 p2 p. p1 \oplus p2 \in \text{subterms } p \implies p1 \in \text{subterms } p$ "
  " $\bigwedge p1 p2 p. p1 \oplus p2 \in \text{subterms } p \implies p2 \in \text{subterms } p$ "
  " $\bigwedge l fip fmsg p q r. \{l\}\text{unicast}(fip, fmsg). q \triangleright r \in \text{subterms } p \implies q \in \text{subterms } p$ "
  " $\bigwedge l fip fmsg p q r. \{l\}\text{unicast}(fip, fmsg). q \triangleright r \in \text{subterms } p \implies r \in \text{subterms } p$ "
  " $\bigwedge l fmsg p q. \{l\}\text{broadcast}(fmsg). q \in \text{subterms } p \implies q \in \text{subterms } p$ "
  " $\bigwedge l fips fmsg p q. \{l\}\text{groupcast}(fips, fmsg). q \in \text{subterms } p \implies q \in \text{subterms } p$ "
  " $\bigwedge l fmsg p q. \{l\}\text{send}(fmsg). q \in \text{subterms } p \implies q \in \text{subterms } p$ "
  " $\bigwedge l fdata p q. \{l\}\text{deliver}(fdata). q \in \text{subterms } p \implies q \in \text{subterms } p$ "
  " $\bigwedge l fmsg p q. \{l\}\text{receive}(fmsg). q \in \text{subterms } p \implies q \in \text{subterms } p$ "
  <proof>

```

5.2 Actions

There are two sorts of τ actions in AWN: one at the level of individual processes (within nodes), and one at the network level (outside nodes). We define a class so that we can ignore this distinction whenever it is not critical.

```

class tau =
  fixes tau :: "'a" ("τ")

```

5.2.1 Sequential Actions (and related predicates)

```

datatype 'm seq_action =
  broadcast 'm
  | groupcast "ip set" 'm
  | unicast ip 'm
  | notunicast ip          ("¬unicast _" [1000] 60)
  | send 'm
  | deliver data
  | receive 'm
  | seq_tau                ("τs")

```

```

instantiation "seq_action" :: (type) tau
begin
definition step_seq_tau [simp]: "τ ≡ τs"
instance <proof>
end

```

```

definition recvmsg :: "('m ⇒ bool) ⇒ 'm seq_action ⇒ bool"
where "recvmsg P a ≡ case a of receive m ⇒ P m
      | _ ⇒ True"

```

```

lemma recvmsg_simps[simp]:
  " $\bigwedge m. \text{recvmsg } P (\text{broadcast } m) = \text{True}$ "
  " $\bigwedge ips m. \text{recvmsg } P (\text{groupcast } ips m) = \text{True}$ "
  " $\bigwedge ip m. \text{recvmsg } P (\text{unicast } ip m) = \text{True}$ "
  " $\bigwedge ip. \text{recvmsg } P (\text{notunicast } ip) = \text{True}$ "
  " $\bigwedge m. \text{recvmsg } P (\text{send } m) = \text{True}$ "
  " $\bigwedge d. \text{recvmsg } P (\text{deliver } d) = \text{True}$ "
  " $\bigwedge m. \text{recvmsg } P (\text{receive } m) = P m$ "
  " $\text{recvmsg } P \tau_s = \text{True}$ "
  <proof>

```

```

lemma recvmsgTT [simp]: "recvmsg TT a"
  <proof>

```

```

lemma recvmsgE [elim]:
  assumes "recvmsg (R σ) a"
  and " $\bigwedge m. R \sigma m \implies R \sigma' m$ "

```

shows "recvmsg (R σ') a"
 ⟨proof⟩

definition anycast :: "('m ⇒ bool) ⇒ 'm seq_action ⇒ bool"
 where "anycast P a ≡ case a of broadcast m ⇒ P m
 | groupcast _ m ⇒ P m
 | unicast _ m ⇒ P m
 | _ ⇒ True"

lemma anycast_simps [simp]:
 "∧m. anycast P (broadcast m) = P m"
 "∧ips m. anycast P (groupcast ips m) = P m"
 "∧ip m. anycast P (unicast ip m) = P m"
 "∧ip. anycast P (notunicast ip) = True"
 "∧m. anycast P (send m) = True"
 "∧d. anycast P (deliver d) = True"
 "∧m. anycast P (receive m) = True"
 " anycast P τ_s = True"
 ⟨proof⟩

definition orecvmsg :: "((ip ⇒ 's) ⇒ 'm ⇒ bool) ⇒ (ip ⇒ 's) ⇒ 'm seq_action ⇒ bool"
 where "orecvmsg P σ a ≡ (case a of receive m ⇒ P σ m
 | _ ⇒ True)"

lemma orecvmsg_simps [simp]:
 "∧m. orecvmsg P σ (broadcast m) = True"
 "∧ips m. orecvmsg P σ (groupcast ips m) = True"
 "∧ip m. orecvmsg P σ (unicast ip m) = True"
 "∧ip. orecvmsg P σ (notunicast ip) = True"
 "∧m. orecvmsg P σ (send m) = True"
 "∧d. orecvmsg P σ (deliver d) = True"
 "∧m. orecvmsg P σ (receive m) = P σ m"
 " orecvmsg P σ τ_s = True"
 ⟨proof⟩

lemma orecvmsgEI [elim]:
 "⌈ orecvmsg P σ a; ∧σ a. P σ a ⇒ Q σ a ⌋ ⇒ orecvmsg Q σ a"
 ⟨proof⟩

lemma orecvmsg_stateless_recvmsg [elim]:
 "orecvmsg (λ_. P) σ a ⇒ recvmsg P a"
 ⟨proof⟩

lemma orecvmsg_recv_weaken [elim]:
 "⌈ orecvmsg P σ a; ∧σ a. P σ a ⇒ Q a ⌋ ⇒ recvmsg Q a"
 ⟨proof⟩

lemma orecvmsg_recvmsg [elim]:
 "orecvmsg P σ a ⇒ recvmsg (P σ) a"
 ⟨proof⟩

definition sendmsg :: "('m ⇒ bool) ⇒ 'm seq_action ⇒ bool"
 where "sendmsg P a ≡ case a of send m ⇒ P m | _ ⇒ True"

lemma sendmsg_simps [simp]:
 "∧m. sendmsg P (broadcast m) = True"
 "∧ips m. sendmsg P (groupcast ips m) = True"
 "∧ip m. sendmsg P (unicast ip m) = True"
 "∧ip. sendmsg P (notunicast ip) = True"
 "∧m. sendmsg P (send m) = P m"
 "∧d. sendmsg P (deliver d) = True"
 "∧m. sendmsg P (receive m) = True"
 " sendmsg P τ_s = True"
 ⟨proof⟩

type_synonym ('s, 'm, 'p, 'l) seqp_env = "'p \Rightarrow ('s, 'm, 'p, 'l) seqp"

5.2.2 Node Actions (and related predicates)

```
datatype 'm node_action =
  node_cast "ip set" 'm          ("_*cast'(_)")      [200, 200] 200
| node_deliver ip data          ("_:_deliver'(_)")    [200, 200] 200
| node_arrive "ip set" "ip set" 'm ("_~_:_arrive'(_)") [200, 200, 200] 200
| node_connect ip ip           ("connect'(_, _)")    [200, 200] 200
| node_disconnect ip ip       ("disconnect'(_, _)")  [200, 200] 200
| node_newpkt ip data ip      ("_:_newpkt'(_, _)") [200, 200, 200] 200
| node_tau                     ("τn")
```

instantiation "node_action" :: (type) tau

begin

definition step_node_tau [simp]: "τ ≡ τ_n"

instance <proof>

end

definition arrivemsg :: "ip \Rightarrow ('m \Rightarrow bool) \Rightarrow 'm node_action \Rightarrow bool"

where "arrivemsg i P a \equiv case a of node_arrive ii ni m \Rightarrow ((ii = {i} \longrightarrow P m) | _ \Rightarrow True"

lemma arrivemsg_simps[simp]:

```
" $\bigwedge$ R m.      arrivemsg i P (R:*cast(m))      = True"
" $\bigwedge$ d m.      arrivemsg i P (d:deliver(m))    = True"
" $\bigwedge$ i ii ni m. arrivemsg i P (ii~ni:arrive(m)) = (ii = {i}  $\longrightarrow$  P m)"
" $\bigwedge$ i1 i2.    arrivemsg i P (connect(i1, i2))  = True"
" $\bigwedge$ i1 i2.    arrivemsg i P (disconnect(i1, i2)) = True"
" $\bigwedge$ i i' d di. arrivemsg i P (i':newpkt(d, di)) = True"
"             arrivemsg i P τn                = True"
```

<proof>

lemma arrivemsgTT [simp]: "arrivemsg i TT = TT"

<proof>

definition oarrivemsg :: "((ip \Rightarrow 's) \Rightarrow 'm \Rightarrow bool) \Rightarrow (ip \Rightarrow 's) \Rightarrow 'm node_action \Rightarrow bool"

where "oarrivemsg P σ a \equiv case a of node_arrive ii ni m \Rightarrow P σ m | _ \Rightarrow True"

lemma oarrivemsg_simps[simp]:

```
" $\bigwedge$ R m.      oarrivemsg P σ (R:*cast(m))      = True"
" $\bigwedge$ d m.      oarrivemsg P σ (d:deliver(m))    = True"
" $\bigwedge$ i ii ni m. oarrivemsg P σ (ii~ni:arrive(m)) = P σ m"
" $\bigwedge$ i1 i2.    oarrivemsg P σ (connect(i1, i2))  = True"
" $\bigwedge$ i1 i2.    oarrivemsg P σ (disconnect(i1, i2)) = True"
" $\bigwedge$ i i' d di. oarrivemsg P σ (i':newpkt(d, di)) = True"
"             oarrivemsg P σ τn                = True"
```

<proof>

lemma oarrivemsg_True [simp, intro]: "oarrivemsg (λ_ _ . True) σ a"

<proof>

definition castmsg :: "('m \Rightarrow bool) \Rightarrow 'm node_action \Rightarrow bool"

where "castmsg P a \equiv case a of _:*cast(m) \Rightarrow P m | _ \Rightarrow True"

lemma castmsg_simps[simp]:

```
" $\bigwedge$ R m.      castmsg P (R:*cast(m))      = P m"
" $\bigwedge$ d m.      castmsg P (d:deliver(m))    = True"
" $\bigwedge$ i ii ni m. castmsg P (ii~ni:arrive(m))    = True"
" $\bigwedge$ i1 i2.    castmsg P (connect(i1, i2))    = True"
" $\bigwedge$ i1 i2.    castmsg P (disconnect(i1, i2)) = True"
" $\bigwedge$ i i' d di. castmsg P (i':newpkt(d, di)) = True"
```

```

"                                castmsg P  $\tau_n$                                 = True"
⟨proof⟩

```

5.3 Networks

```

datatype net_tree =
  Node ip "ip set"                ("⟨_; _⟩")
  | Subnet net_tree net_tree      (infixl "||" 90)

declare net_tree.induct [[induct del]]
lemmas net_tree_induct [induct type: net_tree] = net_tree.induct [rename_abs i R p1 p2]

datatype 's net_state =
  NodeS ip 's "ip set"
  | SubnetS "'s net_state" "'s net_state"

fun net_ips :: "'s net_state ⇒ ip set"
where
  "net_ips (NodeS i s R) = {i}"
  | "net_ips (SubnetS n1 n2) = net_ips n1 ∪ net_ips n2"

fun net_tree_ips :: "net_tree ⇒ ip set"
where
  "net_tree_ips (p1 || p2) = net_tree_ips p1 ∪ net_tree_ips p2"
  | "net_tree_ips (⟨i; R⟩) = {i}"

lemma net_tree_ips_commute:
  "net_tree_ips (p1 || p2) = net_tree_ips (p2 || p1)"
⟨proof⟩

fun wf_net_tree :: "net_tree ⇒ bool"
where
  "wf_net_tree (p1 || p2) = (net_tree_ips p1 ∩ net_tree_ips p2 = {})
    ∧ wf_net_tree p1 ∧ wf_net_tree p2"
  | "wf_net_tree (⟨i; R⟩) = True"

lemma wf_net_tree_children [elim]:
  assumes "wf_net_tree (p1 || p2)"
  obtains "wf_net_tree p1"
    and "wf_net_tree p2"
⟨proof⟩

fun netmap :: "'s net_state ⇒ ip ⇒ 's option"
where
  "netmap (NodeS i p Ri) = [i ↦ p]"
  | "netmap (SubnetS s t) = netmap s ++ netmap t"

lemma not_in_netmap [simp]:
  assumes "i ∉ net_ips ns"
  shows "netmap ns i = None"
⟨proof⟩

lemma netmap_none_not_in_net_ips:
  assumes "netmap ns i = None"
  shows "i ∉ net_ips ns"
⟨proof⟩

lemma net_ips_is_dom_netmap: "net_ips s = dom(netmap s)"
⟨proof⟩

lemma in_netmap [simp]:
  assumes "i ∈ net_ips ns"
  shows "netmap ns i ≠ None"
⟨proof⟩

```

```

lemma netmap_subnets_same:
  assumes "netmap s1 i = x"
    and "netmap s2 i = x"
  shows "netmap (SubnetS s1 s2) i = x"
  <proof>

lemma netmap_subnets_samef:
  assumes "netmap s1 = f"
    and "netmap s2 = f"
  shows "netmap (SubnetS s1 s2) = f"
  <proof>

lemma netmap_add_disjoint [elim]:
  assumes " $\forall i \in \text{net\_ips } s1 \cup \text{net\_ips } s2. \text{the } ((\text{netmap } s1 ++ \text{netmap } s2) i) = \sigma i$ "
    and " $\text{net\_ips } s1 \cap \text{net\_ips } s2 = \{\}$ "
  shows " $\forall i \in \text{net\_ips } s1. \text{the } (\text{netmap } s1 i) = \sigma i$ "
  <proof>

lemma netmap_add_disjoint2 [elim]:
  assumes " $\forall i \in \text{net\_ips } s1 \cup \text{net\_ips } s2. \text{the } ((\text{netmap } s1 ++ \text{netmap } s2) i) = \sigma i$ "
  shows " $\forall i \in \text{net\_ips } s2. \text{the } (\text{netmap } s2 i) = \sigma i$ "
  <proof>

lemma net_ips_netmap_subnet [elim]:
  assumes "net_ips s1  $\cap$  net_ips s2 =  $\{\}$ "
    and " $\forall i \in \text{net\_ips } (\text{SubnetS } s1 s2). \text{the } (\text{netmap } (\text{SubnetS } s1 s2) i) = \sigma i$ "
  shows " $\forall i \in \text{net\_ips } s1. \text{the } (\text{netmap } s1 i) = \sigma i$ "
    and " $\forall i \in \text{net\_ips } s2. \text{the } (\text{netmap } s2 i) = \sigma i$ "
  <proof>

fun inclosed :: "'s  $\Rightarrow$  'm::msg node_action  $\Rightarrow$  bool"
where
  "inclosed _ (node_arrive ii ni m) = eq_newpkt m"
| "inclosed _ (node_newpkt i d di) = False"
| "inclosed _ _ = True"

lemma inclosed_simps [simp]:
  " $\bigwedge \sigma ii ni. \text{inclosed } \sigma (ii \neg ni : \text{arrive}(m)) = \text{eq\_newpkt } m$ "
  " $\bigwedge \sigma d di. \text{inclosed } \sigma (i : \text{newpkt}(d, di)) = \text{False}$ "
  " $\bigwedge \sigma R m. \text{inclosed } \sigma (R : * \text{cast}(m)) = \text{True}$ "
  " $\bigwedge \sigma i d. \text{inclosed } \sigma (i : \text{deliver}(d)) = \text{True}$ "
  " $\bigwedge \sigma i i'. \text{inclosed } \sigma (\text{connect}(i, i')) = \text{True}$ "
  " $\bigwedge \sigma i i'. \text{inclosed } \sigma (\text{disconnect}(i, i')) = \text{True}$ "
  " $\bigwedge \sigma. \text{inclosed } \sigma (\tau) = \text{True}$ "
  <proof>

definition
  netmask :: "ip set  $\Rightarrow$  ((ip  $\Rightarrow$  's)  $\times$  'l)  $\Rightarrow$  ((ip  $\Rightarrow$  's option)  $\times$  'l)"
where
  "netmask I s  $\equiv$  ( $\lambda i. \text{if } i \in I \text{ then Some } (\text{fst } s i) \text{ else None, snd } s)$ "

lemma netmask_def' [simp]:
  "netmask I ( $\sigma, \zeta$ ) = ( $\lambda i. \text{if } i \in I \text{ then Some } (\sigma i) \text{ else None, } \zeta)$ "
  <proof>

fun netgmap :: "('s  $\Rightarrow$  'g  $\times$  'l)  $\Rightarrow$  's net_state  $\Rightarrow$  (nat  $\Rightarrow$  'g option)  $\times$  'l net_state"
where
  "netgmap sr (NodeS i s R) = ([i  $\mapsto$  fst (sr s)], NodeS i (snd (sr s)) R)"
| "netgmap sr (SubnetS s1 s2) = (let ( $\sigma_1, ss$ ) = netgmap sr s1 in
  let ( $\sigma_2, tt$ ) = netgmap sr s2 in
  ( $\sigma_1 ++ \sigma_2, \text{SubnetS } ss tt)$ )"

lemma dom_fst_netgmap [simp, intro]: "dom (fst (netgmap sr n)) = net_ips n"

```

```

⟨proof⟩

lemma netgmap_pair_dom [elim]:
  obtains  $\sigma \zeta$  where "netgmap sr n = ( $\sigma$ ,  $\zeta$ )"
    and "dom  $\sigma$  = net_ips n"
  ⟨proof⟩

lemma net_ips_netgmap [simp]:
  "net_ips (snd (netgmap sr s)) = net_ips s"
  ⟨proof⟩

lemma some_the_fst_netgmap:
  assumes "i ∈ net_ips s"
  shows "Some (the (fst (netgmap sr s) i)) = fst (netgmap sr s) i"
  ⟨proof⟩

lemma fst_netgmap_none [simp]:
  assumes "i ∉ net_ips s"
  shows "fst (netgmap sr s) i = None"
  ⟨proof⟩

lemma fst_netgmap_subnet [simp]:
  "fst (case netgmap sr s1 of ( $\sigma_1$ , ss) ⇒
    case netgmap sr s2 of ( $\sigma_2$ , tt) ⇒
      ( $\sigma_1$  ++  $\sigma_2$ , SubnetS ss tt)) = (fst (netgmap sr s1) ++ fst (netgmap sr s2))"
  ⟨proof⟩

lemma snd_netgmap_subnet [simp]:
  "snd (case netgmap sr s1 of ( $\sigma_1$ , ss) ⇒
    case netgmap sr s2 of ( $\sigma_2$ , tt) ⇒
      ( $\sigma_1$  ++  $\sigma_2$ , SubnetS ss tt)) = (SubnetS (snd (netgmap sr s1)) (snd (netgmap sr s2)))"
  ⟨proof⟩

lemma fst_netgmap_not_none [simp]:
  assumes "i ∈ net_ips s"
  shows "fst (netgmap sr s) i ≠ None"
  ⟨proof⟩

lemma netgmap_netgmap_not_rhs [simp]:
  assumes "i ∉ net_ips s2"
  shows "(fst (netgmap sr s1) ++ fst (netgmap sr s2)) i = (fst (netgmap sr s1)) i"
  ⟨proof⟩

lemma netgmap_netgmap_rhs [simp]:
  assumes "i ∈ net_ips s2"
  shows "(fst (netgmap sr s1) ++ fst (netgmap sr s2)) i = (fst (netgmap sr s2)) i"
  ⟨proof⟩

lemma netgmap_netmask_subnets [elim]:
  assumes "netgmap sr s1 = netmask (net_tree_ips n1) ( $\sigma$ , snd (netgmap sr s1))"
    and "netgmap sr s2 = netmask (net_tree_ips n2) ( $\sigma$ , snd (netgmap sr s2))"
  shows "fst (netgmap sr (SubnetS s1 s2))
    = fst (netmask (net_tree_ips (n1 || n2)) ( $\sigma$ , snd (netgmap sr (SubnetS s1 s2))))"
  ⟨proof⟩

lemma netgmap_netmask_subnets' [elim]:
  assumes "netgmap sr s1 = netmask (net_tree_ips n1) ( $\sigma$ , snd (netgmap sr s1))"
    and "netgmap sr s2 = netmask (net_tree_ips n2) ( $\sigma$ , snd (netgmap sr s2))"
    and "s = SubnetS s1 s2"
  shows "netgmap sr s = netmask (net_tree_ips (n1 || n2)) ( $\sigma$ , snd (netgmap sr s))"
  ⟨proof⟩

lemma netgmap_subnet_split1:

```

```

assumes "netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 || n2)) (σ, ζ)"
  and "net_tree_ips n1 ∩ net_tree_ips n2 = {}"
  and "net_ips s1 = net_tree_ips n1"
  and "net_ips s2 = net_tree_ips n2"
shows "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
⟨proof⟩

lemma netgmap_subnet_split2:
assumes "netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 || n2)) (σ, ζ)"
  and "net_ips s1 = net_tree_ips n1"
  and "net_ips s2 = net_tree_ips n2"
shows "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
⟨proof⟩

lemma netmap_fst_netgmap_rel:
shows "(λi. map_option (fst o sr) (netmap s i)) = fst (netgmap sr s)"
⟨proof⟩

lemma netmap_is_fst_netgmap:
assumes "netmap s' = netmap s"
shows "fst (netgmap sr s') = fst (netgmap sr s)"
⟨proof⟩

lemma netmap_is_fst_netgmap':
assumes "netmap s' i = netmap s i"
shows "fst (netgmap sr s') i = fst (netgmap sr s) i"
⟨proof⟩

lemma fst_netgmap_pair_fst [simp]:
"fst (netgmap (λ(p, q). (fst p, snd p, q)) s) = fst (netgmap fst s)"
⟨proof⟩

Introduce streamlined alternatives to netgmap to simplify certain property statements and thus make them easier
to understand and to present.

fun netlift :: "('s ⇒ 'g × 'l) ⇒ 's net_state ⇒ (nat ⇒ 'g option)"
  where
    "netlift sr (NodeS i s R) = [i ↦ fst (sr s)]"
  | "netlift sr (SubnetS s t) = (netlift sr s) ++ (netlift sr t)"

lemma fst_netgmap_netlift:
"fst (netgmap sr s) = netlift sr s"
⟨proof⟩

fun netliftl :: "('s ⇒ 'g × 'l) ⇒ 's net_state ⇒ 'l net_state"
  where
    "netliftl sr (NodeS i s R) = NodeS i (snd (sr s)) R"
  | "netliftl sr (SubnetS s t) = SubnetS (netliftl sr s) (netliftl sr t)"

lemma snd_netgmap_netliftl:
"snd (netgmap sr s) = netliftl sr s"
⟨proof⟩

lemma netgmap_netlift_netliftl: "netgmap sr s = (netlift sr s, netliftl sr s)"
⟨proof⟩

end

```

6 Semantics of the Algebra of Wireless Networks

```

theory Awn_SOS
imports TransitionSystems Awn
begin

```

6.1 Table 1: Structural operational semantics for sequential process expressions

inductive_set

seqp_sos

:: "('s, 'm, 'p, 'l) seqp_env \Rightarrow ('s \times ('s, 'm, 'p, 'l) seqp, 'm seq_action) transition set"

for Γ :: "('s, 'm, 'p, 'l) seqp_env"

where

broadcastT: " $((\xi, \{1\}\text{broadcast}(s_{msg}).p), \text{broadcast } (s_{msg} \xi), (\xi, p)) \in \text{seqp_sos } \Gamma$ "
 / groupcastT: " $((\xi, \{1\}\text{groupcast}(s_{ips}, s_{msg}).p), \text{groupcast } (s_{ips} \xi) (s_{msg} \xi), (\xi, p)) \in \text{seqp_sos } \Gamma$ "
 / unicastT: " $((\xi, \{1\}\text{unicast}(s_{ip}, s_{msg}).p \triangleright q), \text{unicast } (s_{ip} \xi) (s_{msg} \xi), (\xi, p)) \in \text{seqp_sos } \Gamma$ "
 / notunicastT: " $((\xi, \{1\}\text{unicast}(s_{ip}, s_{msg}).p \triangleright q), \neg \text{unicast } (s_{ip} \xi), (\xi, q)) \in \text{seqp_sos } \Gamma$ "
 / sendT: " $((\xi, \{1\}\text{send}(s_{msg}).p), \text{send } (s_{msg} \xi), (\xi, p)) \in \text{seqp_sos } \Gamma$ "
 / deliverT: " $((\xi, \{1\}\text{deliver}(s_{data}).p), \text{deliver } (s_{data} \xi), (\xi, p)) \in \text{seqp_sos } \Gamma$ "
 / receiveT: " $((\xi, \{1\}\text{receive}(u_{msg}).p), \text{receive } \text{msg}, (u_{msg} \text{msg } \xi, p)) \in \text{seqp_sos } \Gamma$ "
 / assignT: " $((\xi, \{1\}\llbracket u \rrbracket p), \tau, (u \xi, p)) \in \text{seqp_sos } \Gamma$ "

/ callT: " $\llbracket ((\xi, \Gamma \text{pn}), a, (\xi', p')) \in \text{seqp_sos } \Gamma \rrbracket \Longrightarrow ((\xi, \text{call}(\text{pn})), a, (\xi', p')) \in \text{seqp_sos } \Gamma$ "

/ choiceT1: " $((\xi, p), a, (\xi', p')) \in \text{seqp_sos } \Gamma \Longrightarrow ((\xi, p \oplus q), a, (\xi', p')) \in \text{seqp_sos } \Gamma$ "

/ choiceT2: " $((\xi, q), a, (\xi', q')) \in \text{seqp_sos } \Gamma \Longrightarrow ((\xi, p \oplus q), a, (\xi', q')) \in \text{seqp_sos } \Gamma$ "

/ guardT: " $\xi' \in g \xi \Longrightarrow ((\xi, \{1\}\langle g \rangle p), \tau, (\xi', p)) \in \text{seqp_sos } \Gamma$ "

inductive_cases

seqp_callTE [elim]: " $((\xi, \text{call}(\text{pn})), a, (\xi', q)) \in \text{seqp_sos } \Gamma$ "

and *seqp_choiceTE [elim]*: " $((\xi, p1 \oplus p2), a, (\xi', q)) \in \text{seqp_sos } \Gamma$ "

lemma seqp_broadcastTE [elim]:

" $\llbracket ((\xi, \{1\}\text{broadcast}(s_{msg}).p), a, (\xi', q)) \in \text{seqp_sos } \Gamma; \llbracket a = \text{broadcast } (s_{msg} \xi); \xi' = \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

" $\llbracket a = \text{broadcast } (s_{msg} \xi); \xi' = \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

<proof>

lemma seqp_groupcastTE [elim]:

" $\llbracket ((\xi, \{1\}\text{groupcast}(s_{ips}, s_{msg}).p), a, (\xi', q)) \in \text{seqp_sos } \Gamma; \llbracket a = \text{groupcast } (s_{ips} \xi) (s_{msg} \xi); \xi' = \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

" $\llbracket a = \text{groupcast } (s_{ips} \xi) (s_{msg} \xi); \xi' = \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

<proof>

lemma seqp_unicastTE [elim]:

" $\llbracket ((\xi, \{1\}\text{unicast}(s_{ip}, s_{msg}).p \triangleright q), a, (\xi', r)) \in \text{seqp_sos } \Gamma; \llbracket a = \text{unicast } (s_{ip} \xi) (s_{msg} \xi); \xi' = \xi; r = p \rrbracket \Longrightarrow P; \llbracket a = \neg \text{unicast } (s_{ip} \xi); \xi' = \xi; r = q \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

" $\llbracket a = \text{unicast } (s_{ip} \xi) (s_{msg} \xi); \xi' = \xi; r = p \rrbracket \Longrightarrow P; \llbracket a = \neg \text{unicast } (s_{ip} \xi); \xi' = \xi; r = q \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

" $\llbracket a = \neg \text{unicast } (s_{ip} \xi); \xi' = \xi; r = q \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

<proof>

lemma seqp_sendTE [elim]:

" $\llbracket ((\xi, \{1\}\text{send}(s_{msg}).p), a, (\xi', q)) \in \text{seqp_sos } \Gamma; \llbracket a = \text{send } (s_{msg} \xi); \xi' = \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

" $\llbracket a = \text{send } (s_{msg} \xi); \xi' = \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

<proof>

lemma seqp_deliverTE [elim]:

" $\llbracket ((\xi, \{1\}\text{deliver}(s_{data}).p), a, (\xi', q)) \in \text{seqp_sos } \Gamma; \llbracket a = \text{deliver } (s_{data} \xi); \xi' = \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

" $\llbracket a = \text{deliver } (s_{data} \xi); \xi' = \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

<proof>

lemma seqp_receiveTE [elim]:

" $\llbracket ((\xi, \{1\}\text{receive}(u_{msg}).p), a, (\xi', q)) \in \text{seqp_sos } \Gamma; \llbracket \bigwedge \text{msg}. \llbracket a = \text{receive } \text{msg}; \xi' = u_{msg} \text{msg } \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

" $\llbracket \bigwedge \text{msg}. \llbracket a = \text{receive } \text{msg}; \xi' = u_{msg} \text{msg } \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

<proof>

lemma seqp_assignTE [elim]:

" $\llbracket ((\xi, \{1\}\llbracket u \rrbracket p), a, (\xi', q)) \in \text{seqp_sos } \Gamma; \llbracket a = \tau; \xi' = u \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

<proof>

lemma seqp_guardTE [elim]:

" $\llbracket ((\xi, \{1\}\langle g \rangle p), a, (\xi', q)) \in \text{seqp_sos } \Gamma; \llbracket a = \tau; \xi' \in g \xi; q = p \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ "

$\langle proof \rangle$

```
lemmas seqpTEs =  
  seqp_broadcastTE  
  seqp_groupcastTE  
  seqp_unicastTE  
  seqp_sendTE  
  seqp_deliverTE  
  seqp_receiveTE  
  seqp_assignTE  
  seqp_callTE  
  seqp_choiceTE  
  seqp_guardTE
```

```
declare seqp_sos.intros [intro]
```

6.2 Table 2: Structural operational semantics for parallel process expressions

inductive_set

```
parp_sos :: "('s1, 'm seq_action) transition set  
            $\Rightarrow$  ('s2, 'm seq_action) transition set  
            $\Rightarrow$  ('s1  $\times$  's2, 'm seq_action) transition set"  
for S :: "('s1, 'm seq_action) transition set"  
and T :: "('s2, 'm seq_action) transition set"
```

where

```
parleft: "[ (s, a, s')  $\in$  S;  $\bigwedge m. a \neq \text{receive } m$  ]  $\Rightarrow$  ((s, t), a, (s', t))  $\in$  parp_sos S T"  
| parright: "[ (t, a, t')  $\in$  T;  $\bigwedge m. a \neq \text{send } m$  ]  $\Rightarrow$  ((s, t), a, (s, t'))  $\in$  parp_sos S T"  
| parboth: "[ (s, receive m, s')  $\in$  S; (t, send m, t')  $\in$  T ]  
             $\Rightarrow$  ((s, t),  $\tau$ , (s', t'))  $\in$  parp_sos S T"
```

lemma par_broadcastTE [elim]:

```
"[ ((s, t), broadcast m, (s', t'))  $\in$  parp_sos S T;  
  [(s, broadcast m, s')  $\in$  S; t' = t]  $\Rightarrow$  P;  
  [(t, broadcast m, t')  $\in$  T; s' = s]  $\Rightarrow$  P]  $\Rightarrow$  P"  
 $\langle proof \rangle$ 
```

lemma par_groupcastTE [elim]:

```
"[ ((s, t), groupcast ips m, (s', t'))  $\in$  parp_sos S T;  
  [(s, groupcast ips m, s')  $\in$  S; t' = t]  $\Rightarrow$  P;  
  [(t, groupcast ips m, t')  $\in$  T; s' = s]  $\Rightarrow$  P]  $\Rightarrow$  P"  
 $\langle proof \rangle$ 
```

lemma par_unicastTE [elim]:

```
"[ ((s, t), unicast i m, (s', t'))  $\in$  parp_sos S T;  
  [(s, unicast i m, s')  $\in$  S; t' = t]  $\Rightarrow$  P;  
  [(t, unicast i m, t')  $\in$  T; s' = s]  $\Rightarrow$  P]  $\Rightarrow$  P"  
 $\langle proof \rangle$ 
```

lemma par_notunicastTE [elim]:

```
"[ ((s, t), notunicast i, (s', t'))  $\in$  parp_sos S T;  
  [(s, notunicast i, s')  $\in$  S; t' = t]  $\Rightarrow$  P;  
  [(t, notunicast i, t')  $\in$  T; s' = s]  $\Rightarrow$  P]  $\Rightarrow$  P"  
 $\langle proof \rangle$ 
```

lemma par_sendTE [elim]:

```
"[ ((s, t), send m, (s', t'))  $\in$  parp_sos S T;  
  [(s, send m, s')  $\in$  S; t' = t]  $\Rightarrow$  P]  $\Rightarrow$  P"  
 $\langle proof \rangle$ 
```

lemma par_deliverTE [elim]:

```
"[ ((s, t), deliver d, (s', t'))  $\in$  parp_sos S T;  
  [(s, deliver d, s')  $\in$  S; t' = t]  $\Rightarrow$  P;  
  [(t, deliver d, t')  $\in$  T; s' = s]  $\Rightarrow$  P]  $\Rightarrow$  P"  
 $\langle proof \rangle$ 
```

```

lemma par_receiveTE [elim]:
  "[[(s, t), receive m, (s', t')] ∈ parp_sos S T;
   [(t, receive m, t') ∈ T; s' = s] ⇒ P] ⇒ P"
  ⟨proof⟩

```

```

inductive_cases par_tauTE: "(s, t), τ, (s', t') ∈ parp_sos S T"

```

```

lemmas parpTEs =
  par_broadcastTE
  par_groupcastTE
  par_unicastTE
  par_notunicastTE
  par_sendTE
  par_deliverTE
  par_receiveTE

```

```

lemma parp_sos_cases [elim]:
  assumes "(s, t), a, (s', t') ∈ parp_sos S T"
    and "[[(s, a, s') ∈ S; ∧m. a ≠ receive m; t' = t] ⇒ P"
    and "[[(t, a, t') ∈ T; ∧m. a ≠ send m; s' = s] ⇒ P"
    and "∧m. [(s, receive m, s') ∈ S; (t, send m, t') ∈ T] ⇒ P"
  shows "P"
  ⟨proof⟩

```

definition

```

par_comp :: "('s1, 'm seq_action) automaton
           ⇒ ('s2, 'm seq_action) automaton
           ⇒ ('s1 × 's2, 'm seq_action) automaton"
("⟨_⟩") [102, 103] 102

```

where

```

"s ⟨⟨ t ≡ (| init = init s × init t, trans = parp_sos (trans s) (trans t) |) ⟩"

```

```

lemma trans_par_comp [simp]:
  "trans (s ⟨⟨ t) = parp_sos (trans s) (trans t)"
  ⟨proof⟩

```

```

lemma init_par_comp [simp]:
  "init (s ⟨⟨ t) = init s × init t"
  ⟨proof⟩

```

6.3 Table 3: Structural operational semantics for node expressions

inductive_set

```

node_sos :: "('s, 'm seq_action) transition set ⇒ ('s net_state, 'm node_action) transition set"
for S :: "('s, 'm seq_action) transition set"

```

where

```

node_bcast:
  "(s, broadcast m, s') ∈ S ⇒ (NodeS i s R, R:*cast(m), NodeS i s' R) ∈ node_sos S"
/ node_gcast:
  "(s, groupcast D m, s') ∈ S ⇒ (NodeS i s R, (R∩D):*cast(m), NodeS i s' R) ∈ node_sos S"
/ node_ucast:
  "[[(s, unicast d m, s') ∈ S; d∈R] ⇒ (NodeS i s R, {d}:*cast(m), NodeS i s' R) ∈ node_sos S"
/ node_notucast:
  "[[(s, ¬unicast d, s') ∈ S; d∉R] ⇒ (NodeS i s R, τ, NodeS i s' R) ∈ node_sos S"
/ node_deliver:
  "(s, deliver d, s') ∈ S ⇒ (NodeS i s R, i:deliver(d), NodeS i s' R) ∈ node_sos S"
/ node_receive:
  "(s, receive m, s') ∈ S ⇒ (NodeS i s R, {i}¬{i}:arrive(m), NodeS i s' R) ∈ node_sos S"
/ node_tau:
  "(s, τ, s') ∈ S ⇒ (NodeS i s R, τ, NodeS i s' R) ∈ node_sos S"
/ node_arrive:
  "(NodeS i s R, {i}¬{i}:arrive(m), NodeS i s R) ∈ node_sos S"
/ node_connect1:

```

```

(NodeS i s R, connect(i, i'), NodeS i s (R ∪ {i'})) ∈ node_sos S"
/ node_connect2:
(NodeS i s R, connect(i', i), NodeS i s (R ∪ {i'})) ∈ node_sos S"
/ node_disconnect1:
(NodeS i s R, disconnect(i, i'), NodeS i s (R - {i'})) ∈ node_sos S"
/ node_disconnect2:
(NodeS i s R, disconnect(i', i), NodeS i s (R - {i'})) ∈ node_sos S"
/ node_connect_other:
"[ i ≠ i'; i ≠ i'' ] ⇒ (NodeS i s R, connect(i', i''), NodeS i s R) ∈ node_sos S"
/ node_disconnect_other:
"[ i ≠ i'; i ≠ i'' ] ⇒ (NodeS i s R, disconnect(i', i''), NodeS i s R) ∈ node_sos S"

inductive_cases node_arriveTE: "(NodeS i s R, ii¬ni:arrive(m), NodeS i s' R) ∈ node_sos S"
and node_arriveTE': "(NodeS i s R, H¬K:arrive(m), s') ∈ node_sos S"
and node_castTE: "(NodeS i s R, RM:*cast(m), NodeS i s' R') ∈ node_sos S"
and node_castTE': "(NodeS i s R, RM:*cast(m), s') ∈ node_sos S"
and node_deliverTE: "(NodeS i s R, i:deliver(d), NodeS i s' R) ∈ node_sos S"
and node_deliverTE': "(s, i:deliver(d), s') ∈ node_sos S"
and node_deliverTE'': "(NodeS ii s R, i:deliver(d), s') ∈ node_sos S"
and node_tauTE: "(NodeS i s R, τ, NodeS i s' R) ∈ node_sos S"
and node_tauTE': "(NodeS i s R, τ, s') ∈ node_sos S"
and node_connectTE: "(NodeS ii s R, connect(i, i'), NodeS ii s' R') ∈ node_sos S"
and node_connectTE': "(NodeS ii s R, connect(i, i'), s') ∈ node_sos S"
and node_disconnectTE: "(NodeS ii s R, disconnect(i, i'), NodeS ii s' R') ∈ node_sos S"
and node_disconnectTE': "(NodeS ii s R, disconnect(i, i'), s') ∈ node_sos S"

lemma node_sos_never_newpkt [simp]:
  assumes "(s, a, s') ∈ node_sos S"
  shows "a ≠ i:newpkt(d, di)"
  <proof>

lemma arrives_or_not:
  assumes "(NodeS i s R, ii¬ni:arrive(m), NodeS i' s' R') ∈ node_sos S"
  shows "(ii = {i} ∧ ni = {}) ∨ (ii = {} ∧ ni = {i})"
  <proof>

definition
  node_comp :: "ip ⇒ ('s, 'm seq_action) automaton ⇒ ip set
    ⇒ ('s net_state, 'm node_action) automaton"
  ("⟨_ : (_ : _)⟩" [0, 0, 0] 104)

where
  "⟨i : np : Ri⟩ ≡ (| init = {NodeS i s Ri | s. s ∈ init np}, trans = node_sos (trans np) |)"

lemma trans_node_comp:
  "trans (⟨i : np : Ri⟩) = node_sos (trans np)"
  <proof>

lemma init_node_comp:
  "init (⟨i : np : Ri⟩) = {NodeS i s Ri | s. s ∈ init np}"
  <proof>

lemmas node_comps = trans_node_comp init_node_comp

lemma trans_par_node_comp [simp]:
  "trans (⟨i : s ⟨⟨ t : R ⟩⟩) = node_sos (parp_sos (trans s) (trans t))"
  <proof>

lemma snd_par_node_comp [simp]:
  "init (⟨i : s ⟨⟨ t : R ⟩⟩) = {NodeS i st R | st. st ∈ init s × init t}"
  <proof>

lemma node_sos_dest_is_net_state:
  assumes "(s, a, s') ∈ node_sos S"
  shows "∃ i' P' R'. s' = NodeS i' P' R'"

```

$\langle \text{proof} \rangle$

lemma *node_sos_dest*:

assumes "(NodeS i p R, a, s') ∈ node_sos S"
shows "∃ P' R'. s' = NodeS i P' R'"

$\langle \text{proof} \rangle$

lemma *node_sos_states [elim]*:

assumes "(ns, a, ns') ∈ node_sos S"
obtains i s R s' R' where "ns = NodeS i s R"
and "ns' = NodeS i s' R'"

$\langle \text{proof} \rangle$

lemma *node_sos_cases [elim]*:

"(NodeS i p R, a, NodeS i p' R') ∈ node_sos S ⇒
($\bigwedge m$. $\llbracket a = R:*cast(m); R' = R; (p, broadcast\ m, p') \in S \rrbracket \implies P \rrbracket \implies$
($\bigwedge m D$. $\llbracket a = (R \cap D):*cast(m); R' = R; (p, groupcast\ D\ m, p') \in S \rrbracket \implies P \rrbracket \implies$
($\bigwedge d m$. $\llbracket a = \{d\}:*cast(m); R' = R; (p, unicast\ d\ m, p') \in S; d \in R \rrbracket \implies P \rrbracket \implies$
($\bigwedge d$. $\llbracket a = \tau; R' = R; (p, \neg unicast\ d, p') \in S; d \notin R \rrbracket \implies P \rrbracket \implies$
($\bigwedge d$. $\llbracket a = i:deliver(d); R' = R; (p, deliver\ d, p') \in S \rrbracket \implies P \rrbracket \implies$
($\bigwedge m$. $\llbracket a = \{i\}\neg\{i\}:arrive(m); R' = R; (p, receive\ m, p') \in S \rrbracket \implies P \rrbracket \implies$
($\llbracket a = \tau; R' = R; (p, \tau, p') \in S \rrbracket \implies P \rrbracket \implies$
($\bigwedge m$. $\llbracket a = \{i\}\neg\{i\}:arrive(m); R' = R; p = p' \rrbracket \implies P \rrbracket \implies$
($\bigwedge i i'$. $\llbracket a = connect(i, i'); R' = R \cup \{i'\}; p = p' \rrbracket \implies P \rrbracket \implies$
($\bigwedge i i'$. $\llbracket a = connect(i', i); R' = R \cup \{i'\}; p = p' \rrbracket \implies P \rrbracket \implies$
($\bigwedge i i'$. $\llbracket a = disconnect(i, i'); R' = R - \{i'\}; p = p' \rrbracket \implies P \rrbracket \implies$
($\bigwedge i i'$. $\llbracket a = disconnect(i', i); R' = R - \{i'\}; p = p' \rrbracket \implies P \rrbracket \implies$
($\bigwedge i i' i''$. $\llbracket a = connect(i', i''); R' = R; p = p'; i \neq i'; i \neq i'' \rrbracket \implies P \rrbracket \implies$
($\bigwedge i i' i''$. $\llbracket a = disconnect(i', i''); R' = R; p = p'; i \neq i'; i \neq i'' \rrbracket \implies P \rrbracket \implies$
P"

$\langle \text{proof} \rangle$

6.4 Table 4: Structural operational semantics for partial network expressions

inductive_set

pnet_sos :: "('s net_state, 'm node_action) transition set
⇒ ('s net_state, 'm node_action) transition set
⇒ ('s net_state, 'm node_action) transition set"

for *S* :: "('s net_state, 'm node_action) transition set"

and *T* :: "('s net_state, 'm node_action) transition set"

where

pnet_cast1: " $\llbracket (s, R:*cast(m), s') \in S; (t, H\neg K:arrive(m), t') \in T; H \subseteq R; K \cap R = \{\} \rrbracket$
⇒ (SubnetS s t, R:*cast(m), SubnetS s' t') ∈ *pnet_sos* S T"

| *pnet_cast2*: " $\llbracket (s, H\neg K:arrive(m), s') \in S; (t, R:*cast(m), t') \in T; H \subseteq R; K \cap R = \{\} \rrbracket$
⇒ (SubnetS s t, R:*cast(m), SubnetS s' t') ∈ *pnet_sos* S T"

| *pnet_arrive*: " $\llbracket (s, H\neg K:arrive(m), s') \in S; (t, H'\neg K':arrive(m), t') \in T \rrbracket$
⇒ (SubnetS s t, (H ∪ H')\neg(K ∪ K'):arrive(m), SubnetS s' t') ∈ *pnet_sos* S T"

| *pnet_deliver1*: "(s, i:deliver(d), s') ∈ S
⇒ (SubnetS s t, i:deliver(d), SubnetS s' t) ∈ *pnet_sos* S T"

| *pnet_deliver2*: " $\llbracket (t, i:deliver(d), t') \in T \rrbracket$
⇒ (SubnetS s t, i:deliver(d), SubnetS s t') ∈ *pnet_sos* S T"

| *pnet_tau1*: "(s, τ, s') ∈ S ⇒ (SubnetS s t, τ, SubnetS s' t) ∈ *pnet_sos* S T"

| *pnet_tau2*: "(t, τ, t') ∈ T ⇒ (SubnetS s t, τ, SubnetS s t') ∈ *pnet_sos* S T"

| *pnet_connect*: " $\llbracket (s, connect(i, i'), s') \in S; (t, connect(i, i'), t') \in T \rrbracket$
⇒ (SubnetS s t, connect(i, i'), SubnetS s' t') ∈ *pnet_sos* S T"

| *pnet_disconnect*: " $\llbracket (s, disconnect(i, i'), s') \in S; (t, disconnect(i, i'), t') \in T \rrbracket$
⇒ (SubnetS s t, disconnect(i, i'), SubnetS s' t') ∈ *pnet_sos* S T"

```

inductive_cases partial_castTE [elim]: "(s, R:*cast(m), s') ∈ pnet_sos S T"
and partial_arriveTE [elim]: "(s, H-K:arrive(m), s') ∈ pnet_sos S T"
and partial_deliverTE [elim]: "(s, i:deliver(d), s') ∈ pnet_sos S T"
and partial_tauTE [elim]: "(s, τ, s') ∈ pnet_sos S T"
and partial_connectTE [elim]: "(s, connect(i, i'), s') ∈ pnet_sos S T"
and partial_disconnectTE [elim]: "(s, disconnect(i, i'), s') ∈ pnet_sos S T"

```

lemma pnet_sos_never_newpkt:

```

assumes "(st, a, st') ∈ pnet_sos S T"
and "∧ i d di a s s'. (s, a, s') ∈ S ⇒ a ≠ i:newpkt(d, di)"
and "∧ i d di a t t'. (t, a, t') ∈ T ⇒ a ≠ i:newpkt(d, di)"
shows "a ≠ i:newpkt(d, di)"
⟨proof⟩

```

```

fun pnet :: "('s, 'm seq_action) automaton
⇒ net_tree ⇒ ('s net_state, 'm node_action) automaton"

```

where

```

"pnet np (<i; R_i>) = <i : np i : R_i>"
| "pnet np (p_1 || p_2) = (| init = {SubnetS s_1 s_2 | s_1 s_2. s_1 ∈ init (pnet np p_1)
∧ s_2 ∈ init (pnet np p_2)},
trans = pnet_sos (trans (pnet np p_1)) (trans (pnet np p_2)) |)"

```

lemma pnet_node_init [elim, simp]:

```

assumes "s ∈ init (pnet np <i; R>)"
shows "s ∈ {NodeS i s R | s. s ∈ init (np i)}"
⟨proof⟩

```

lemma pnet_node_init' [elim]:

```

assumes "s ∈ init (pnet np <i; R>)"
obtains ns where "s = NodeS i ns R"
and "ns ∈ init (np i)"
⟨proof⟩

```

lemma pnet_node_trans [elim, simp]:

```

assumes "(s, a, s') ∈ trans (pnet np <i; R>)"
shows "(s, a, s') ∈ node_sos (trans (np i))"
⟨proof⟩

```

lemma pnet_never_newpkt':

```

assumes "(s, a, s') ∈ trans (pnet np n)"
shows "∀ i d di. a ≠ i:newpkt(d, di)"
⟨proof⟩

```

lemma pnet_never_newpkt:

```

assumes "(s, a, s') ∈ trans (pnet np n)"
shows "a ≠ i:newpkt(d, di)"
⟨proof⟩

```

6.5 Table 5: Structural operational semantics for complete network expressions

inductive_set

```

cnet_sos :: "('s, ('m::msg) node_action) transition set
⇒ ('s, 'm node_action) transition set"

```

```

for S :: "('s, 'm node_action) transition set"

```

where

```

cnet_connect: "(s, connect(i, i'), s') ∈ S ⇒ (s, connect(i, i'), s') ∈ cnet_sos S"
| cnet_disconnect: "(s, disconnect(i, i'), s') ∈ S ⇒ (s, disconnect(i, i'), s') ∈ cnet_sos S"
| cnet_cast: "(s, R:*cast(m), s') ∈ S ⇒ (s, τ, s') ∈ cnet_sos S"
| cnet_tau: "(s, τ, s') ∈ S ⇒ (s, τ, s') ∈ cnet_sos S"
| cnet_deliver: "(s, i:deliver(d), s') ∈ S ⇒ (s, i:deliver(d), s') ∈ cnet_sos S"
| cnet_newpkt: "(s, {i}~K:arrive(newpkt(d, di)), s') ∈ S ⇒ (s, i:newpkt(d, di), s') ∈ cnet_sos S"

```

inductive_cases connect_completeTE: "(s, connect(i, i'), s') ∈ cnet_sos S"

and disconnect_completeTE: "(s, disconnect(i, i'), s') ∈ cnet_sos S"

```

and tau_completeTE: "(s,  $\tau$ , s')  $\in$  cnet_sos S"
and deliver_completeTE: "(s, i:deliver(d), s')  $\in$  cnet_sos S"
and newpkt_completeTE: "(s, i:newpkt(d, di), s')  $\in$  cnet_sos S"

```

```

lemmas completeTEs = connect_completeTE
                    disconnect_completeTE
                    tau_completeTE
                    deliver_completeTE
                    newpkt_completeTE

```

```

lemma complete_no_cast [simp]:
  "(s, R:*cast(m), s')  $\notin$  cnet_sos T"
  <proof>

```

```

lemma complete_no_arrive [simp]:
  "(s, ii $\neg$ ni:arrive(m), s')  $\notin$  cnet_sos T"
  <proof>

```

abbreviation

```

closed :: "('s net_state, ('m::msg) node_action) automaton  $\Rightarrow$  ('s net_state, 'm node_action) automaton"
where
  "closed  $\equiv$  ( $\lambda$ A. A ( $\lfloor$  trans := cnet_sos (trans A)  $\rfloor$ )")

```

end

7 Control terms and well-definedness of sequential processes

```

theory AWN_Cterms
imports AWN
begin

```

7.1 Microsteps

We distinguish microsteps from ‘external’ transitions (observable or not). Here, they are a kind of ‘hypothetical computation’, since, unlike τ -transitions, they do not make choices but rather ‘compute’ which choices are possible.

inductive

```

microstep :: "('s, 'm, 'p, 'l) seqp_env
             $\Rightarrow$  ('s, 'm, 'p, 'l) seqp
             $\Rightarrow$  ('s, 'm, 'p, 'l) seqp
             $\Rightarrow$  bool"

```

```

for  $\Gamma$  :: "('s, 'm, 'p, 'l) seqp_env"

```

where

```

  microstep_choiceI1 [intro, simp]: "microstep  $\Gamma$  (p1  $\oplus$  p2) p1"
  | microstep_choiceI2 [intro, simp]: "microstep  $\Gamma$  (p1  $\oplus$  p2) p2"
  | microstep_callI [intro, simp]: "microstep  $\Gamma$  (call(pn)) ( $\Gamma$  pn)"

```

abbreviation microstep_rtcl

```

where "microstep_rtcl  $\Gamma$  p q  $\equiv$  (microstep  $\Gamma$ )** p q"

```

abbreviation microstep_tcl

```

where "microstep_tcl  $\Gamma$  p q  $\equiv$  (microstep  $\Gamma$ )++ p q"

```

syntax

```

"_microstep"
  :: "[('s, 'm, 'p, 'l) seqp, ('s, 'm, 'p, 'l) seqp_env, ('s, 'm, 'p, 'l) seqp]  $\Rightarrow$  bool"
  ("( $\_$ )  $\rightsquigarrow$   $\_$ " [61, 0, 61] 50)
"_microstep_rtcl"
  :: "[('s, 'm, 'p, 'l) seqp, ('s, 'm, 'p, 'l) seqp_env, ('s, 'm, 'p, 'l) seqp]  $\Rightarrow$  bool"
  ("( $\_$ )  $\rightsquigarrow$ * ( $\_$ )" [61, 0, 61] 50)
"_microstep_tcl"
  :: "[('s, 'm, 'p, 'l) seqp, ('s, 'm, 'p, 'l) seqp_env, ('s, 'm, 'p, 'l) seqp]  $\Rightarrow$  bool"
  ("( $\_$ )  $\rightsquigarrow$ + ( $\_$ )" [61, 0, 61] 50)

```

translations

" $p1 \rightsquigarrow_{\Gamma} p2$ " \Leftrightarrow "CONST microstep Γ $p1$ $p2$ "
" $p1 \rightsquigarrow_{\Gamma}^* p2$ " \Leftrightarrow "CONST microstep_rtcl Γ $p1$ $p2$ "
" $p1 \rightsquigarrow_{\Gamma}^+ p2$ " \Leftrightarrow "CONST microstep_tcl Γ $p1$ $p2$ "

lemma microstep_choiceD [dest]:

" $(p1 \oplus p2) \rightsquigarrow_{\Gamma} p \Rightarrow p = p1 \vee p = p2$ "
<proof>

lemma microstep_choiceE [elim]:

" $\llbracket (p1 \oplus p2) \rightsquigarrow_{\Gamma} p;$
 $(p1 \oplus p2) \rightsquigarrow_{\Gamma} p1 \Rightarrow P;$
 $(p1 \oplus p2) \rightsquigarrow_{\Gamma} p2 \Rightarrow P \rrbracket \Rightarrow P$ "
<proof>

lemma microstep_callD [dest]:

" $(\text{call}(pn)) \rightsquigarrow_{\Gamma} p \Rightarrow p = \Gamma pn$ "
<proof>

lemma microstep_callE [elim]:

" $\llbracket (\text{call}(pn)) \rightsquigarrow_{\Gamma} p; p = \Gamma(pn) \Rightarrow P \rrbracket \Rightarrow P$ "
<proof>

lemma no_microstep_guard: " $\neg ((\{l\}g) p \rightsquigarrow_{\Gamma} q)$ "
<proof>

lemma no_microstep_assign: " $\neg (\{l\}f p \rightsquigarrow_{\Gamma} q)$ "
<proof>

lemma no_microstep_unicast: " $\neg ((\{l\}\text{unicast}(s_{ip}, s_{msg}).p \triangleright q) \rightsquigarrow_{\Gamma} r)$ "
<proof>

lemma no_microstep_broadcast: " $\neg ((\{l\}\text{broadcast}(s_{msg}).p) \rightsquigarrow_{\Gamma} q)$ "
<proof>

lemma no_microstep_groupcast: " $\neg ((\{l\}\text{groupcast}(s_{ips}, s_{msg}).p) \rightsquigarrow_{\Gamma} q)$ "
<proof>

lemma no_microstep_send: " $\neg ((\{l\}\text{send}(s_{msg}).p) \rightsquigarrow_{\Gamma} q)$ "
<proof>

lemma no_microstep_deliver: " $\neg ((\{l\}\text{deliver}(s_{data}).p) \rightsquigarrow_{\Gamma} q)$ "
<proof>

lemma no_microstep_receive: " $\neg ((\{l\}\text{receive}(u_{msg}).p) \rightsquigarrow_{\Gamma} q)$ "
<proof>

lemma microstep_call_or_choice [dest]:

assumes " $p \rightsquigarrow_{\Gamma} q$ "
shows " $(\exists pn. p = \text{call}(pn)) \vee (\exists p1 p2. p = p1 \oplus p2)$ "
<proof>

lemmas no_microstep [intro,simp] =

no_microstep_guard
no_microstep_assign
no_microstep_unicast
no_microstep_broadcast
no_microstep_groupcast
no_microstep_send
no_microstep_deliver
no_microstep_receive

7.2 Wellformed process specifications

A process specification Γ is wellformed if its *microstep* Γ relation is free of loops and infinite chains.

For example, these specifications are not wellformed: $\Gamma_1 \ p1 = \text{call}(p1)$

$\Gamma_2 \ p1 = \text{send}(msg) . \text{call}(p1) \oplus \text{call}(p1)$

$\Gamma_3 \ p1 = \text{send}(msg) . \text{call}(p2) \ \Gamma_3 \ p2 = \text{call}(p3) \ \Gamma_3 \ p3 = \text{call}(p4) \ \Gamma_3 \ p4 = \text{call}(p5) \dots$

definition

```
wellformed :: "('s, 'm, 'p, 'l) seqp_env  $\Rightarrow$  bool"
```

where

```
"wellformed  $\Gamma = \text{wf} \{(q, p). p \rightsquigarrow_{\Gamma} q\}$ "
```

lemma wellformed_defP: "wellformed $\Gamma = \text{wfP} (\lambda q \ p. p \rightsquigarrow_{\Gamma} q)$ "

<proof>

The induction rule for *wellformed* Γ is stronger than $\llbracket \bigwedge x1 \ x2 \ x3. ?P \ x3 \Longrightarrow ?P \ (\{x1\}\langle x2 \rangle \ x3); \bigwedge x1 \ x2 \ x3. ?P \ x3 \Longrightarrow ?P \ (\{x1\}\llbracket x2 \rrbracket \ x3); \bigwedge x1 \ x2. \llbracket ?P \ x1; ?P \ x2 \rrbracket \Longrightarrow ?P \ (x1 \oplus x2); \bigwedge x1 \ x2 \ x3 \ x4 \ x5. \llbracket ?P \ x4; ?P \ x5 \rrbracket \Longrightarrow ?P \ (\{x1\}\text{unicast}(x2, x3) . x4 \triangleright x5); \bigwedge x1 \ x2 \ x3. ?P \ x3 \Longrightarrow ?P \ (\{x1\}\text{broadcast}(x2) . x3); \bigwedge x1 \ x2 \ x3 \ x4. ?P \ x4 \Longrightarrow ?P \ (\{x1\}\text{groupcast}(x2, x3) . x4); \bigwedge x1 \ x2 \ x3. ?P \ x3 \Longrightarrow ?P \ (\{x1\}\text{send}(x2) . x3); \bigwedge x1 \ x2 \ x3. ?P \ x3 \Longrightarrow ?P \ (\{x1\}\text{deliver}(x2) . x3); \bigwedge x1 \ x2 \ x3. ?P \ x3 \Longrightarrow ?P \ (\{x1\}\text{receive}(x2) . x3); \bigwedge x. ?P \ (\text{call}(x)) \rrbracket \Longrightarrow ?P \ ?\text{seqp}$ because the case for *call*(*pn*) can be shown with the assumption on $\Gamma \ pn$.

lemma wellformed_induct

```
[consumes 1, case_names ASSIGN CHOICE CALL GUARD UCAST BCAST GCAST SEND DELIVER RECEIVE,
 induct set: wellformed]:
```

```
assumes "wellformed  $\Gamma$ "
```

and ASSIGN:	" $\bigwedge l \ f \ p.$	wellformed $\Gamma \Longrightarrow P \ (\{l\}\llbracket f \rrbracket \ p)$ "
and GUARD:	" $\bigwedge l \ f \ p.$	wellformed $\Gamma \Longrightarrow P \ (\{l\}\langle f \rangle \ p)$ "
and UCAST:	" $\bigwedge l \ \text{fip} \ \text{fmsg} \ p \ q.$	wellformed $\Gamma \Longrightarrow P \ (\{l\}\text{unicast}(\text{fip}, \text{fmsg}). p \triangleright q)$ "
and BCAST:	" $\bigwedge l \ \text{fmsg} \ p.$	wellformed $\Gamma \Longrightarrow P \ (\{l\}\text{broadcast}(\text{fmsg}). p)$ "
and GCAST:	" $\bigwedge l \ \text{fips} \ \text{fmsg} \ p.$	wellformed $\Gamma \Longrightarrow P \ (\{l\}\text{groupcast}(\text{fips}, \text{fmsg}). p)$ "
and SEND:	" $\bigwedge l \ \text{fmsg} \ p.$	wellformed $\Gamma \Longrightarrow P \ (\{l\}\text{send}(\text{fmsg}). p)$ "
and DELIVER:	" $\bigwedge l \ \text{fdata} \ p.$	wellformed $\Gamma \Longrightarrow P \ (\{l\}\text{deliver}(\text{fdata}). p)$ "
and RECEIVE:	" $\bigwedge l \ \text{fmsg} \ p.$	wellformed $\Gamma \Longrightarrow P \ (\{l\}\text{receive}(\text{fmsg}). p)$ "
and CHOICE:	" $\bigwedge p1 \ p2.$	$\llbracket \text{wellformed } \Gamma; P \ p1; P \ p2 \rrbracket \Longrightarrow P \ (p1 \oplus p2)$ "
and CALL:	" $\bigwedge pn.$	$\llbracket \text{wellformed } \Gamma; P \ (\Gamma \ pn) \rrbracket \Longrightarrow P \ (\text{call}(pn))$ "

shows "P a"

<proof>

7.3 Start terms (sterms)

Formulate sets of local subterms from which an action is directly possible. Since the process specification Γ is not considered, only choice terms $p1 \oplus p2$ are traversed, and not *call*(*p*) terms.

```
fun stermsl :: "('s, 'm, 'p, 'l) seqp  $\Rightarrow$  ('s, 'm, 'p, 'l) seqp set"
```

where

```
"stermsl (p1  $\oplus$  p2) = stermsl p1  $\cup$  stermsl p2"
| "stermsl p = {p}"
```

lemma stermsl_nobigger: " $q \in \text{stermsl } p \Longrightarrow \text{size } q \leq \text{size } p$ "

<proof>

lemma stermsl_no_choice[simp]: " $p1 \oplus p2 \notin \text{stermsl } p$ "

<proof>

lemma stermsl_choice_disj[simp]:

```
"p  $\in$  stermsl (p1  $\oplus$  p2) = (p  $\in$  stermsl p1  $\vee$  p  $\in$  stermsl p2)"
```

<proof>

lemma stermsl_in_branch[elim]:

```
" $\llbracket p \in \text{stermsl } (p1 \oplus p2); p \in \text{stermsl } p1 \Longrightarrow P; p \in \text{stermsl } p2 \Longrightarrow P \rrbracket \Longrightarrow P$ "
```

<proof>

lemma stermsl_commute:


```

"stermsl (p1 ⊕ p2) = stermsl (p2 ⊕ p1)"
⟨proof⟩

lemma stermsl_not_empty:
"stermsl p ≠ {}"
⟨proof⟩

lemma stermsl_idem [simp]:
"(⋃ q ∈ stermsl p. stermsl q) = stermsl p"
⟨proof⟩

lemma stermsl_in_wfpf:
assumes AA: "A ⊆ {(q, p). p ~>_Γ q}" "A"
and *: "p ∈ A"
shows "∃ r ∈ stermsl p. r ∈ A"
⟨proof⟩

lemma nocall_stermsl_max:
assumes "r ∈ stermsl p"
and "not_call r"
shows "¬ (r ~>_Γ q)"
⟨proof⟩

theorem wf_no_direct_calls[intro]:
fixes Γ :: "('s, 'm, 'p, 'l) seqp_env"
assumes no_calls: "∧pn. ∀pn'. call(pn') ∉ stermsl(Γ(pn))"
shows "wellformed Γ"
⟨proof⟩

7.4 Start terms

The start terms are those terms, relative to a wellformed process specification  $\Gamma$ , from which transitions can occur directly.

function (domintros, sequential) sterms
:: "('s, 'm, 'p, 'l) seqp_env ⇒ ('s, 'm, 'p, 'l) seqp ⇒ ('s, 'm, 'p, 'l) seqp set"
where
sterms_choice: "sterms Γ (p1 ⊕ p2) = sterms Γ p1 ∪ sterms Γ p2"
| sterms_call: "sterms Γ (call(pn)) = sterms Γ (Γ pn)"
| sterms_other: "sterms Γ p = {p}"
⟨proof⟩

lemma sterms_dom_basic[simp]:
assumes "not_call p"
and "not_choice p"
shows "sterms_dom (Γ, p)"
⟨proof⟩

lemma sterms_termination:
assumes "wellformed Γ"
shows "sterms_dom (Γ, p)"
⟨proof⟩

declare sterms.psimps [simp]

lemmas sterms.psimps[simp] = sterms.psimps [OF sterms_termination]
and sterms.pinduct = sterms.pinduct [OF sterms_termination]

lemma sterms_reflD [dest]:
assumes "q ∈ sterms Γ p"
and "not_choice p" "not_call p"
shows "q = p"
⟨proof⟩

```

```

lemma sterms_choice_disj [simp]:
  assumes "wellformed  $\Gamma$ "
  shows " $p \in \text{sterms } \Gamma (p1 \oplus p2) = (p \in \text{sterms } \Gamma p1 \vee p \in \text{sterms } \Gamma p2)$ "
  <proof>

lemma sterms_no_choice [simp]:
  assumes "wellformed  $\Gamma$ "
  shows " $p1 \oplus p2 \notin \text{sterms } \Gamma p$ "
  <proof>

lemma sterms_not_choice [simp]:
  assumes "wellformed  $\Gamma$ "
  and "q  $\in \text{sterms } \Gamma p$ "
  shows "not_choice q"
  <proof>

lemma sterms_no_call [simp]:
  assumes "wellformed  $\Gamma$ "
  shows "call(pn)  $\notin \text{sterms } \Gamma p$ "
  <proof>

lemma sterms_not_call [simp]:
  assumes "wellformed  $\Gamma$ "
  and "q  $\in \text{sterms } \Gamma p$ "
  shows "not_call q"
  <proof>

lemma sterms_in_branch:
  assumes "wellformed  $\Gamma$ "
  and "p  $\in \text{sterms } \Gamma (p1 \oplus p2)$ "
  and "p  $\in \text{sterms } \Gamma p1 \implies P$ "
  and "p  $\in \text{sterms } \Gamma p2 \implies P$ "
  shows "P"
  <proof>

lemma sterms_commute:
  assumes "wellformed  $\Gamma$ "
  shows "sterms  $\Gamma (p1 \oplus p2) = \text{sterms } \Gamma (p2 \oplus p1)$ "
  <proof>

lemma sterms_not_empty:
  assumes "wellformed  $\Gamma$ "
  shows "sterms  $\Gamma p \neq \{\}$ "
  <proof>

lemma sterms_sterms [simp]:
  assumes "wellformed  $\Gamma$ "
  shows " $(\bigcup x \in \text{sterms } \Gamma p. \text{sterms } \Gamma x) = \text{sterms } \Gamma p$ "
  <proof>

lemma sterms_stermsl:
  assumes "ps  $\in \text{sterms } \Gamma p$ "
  and "wellformed  $\Gamma$ "
  shows "ps  $\in \text{stermsl } p \vee (\exists pn. ps \in \text{stermsl } (\Gamma pn))$ "
  <proof>

lemma stermsl_sterms [elim]:
  assumes "q  $\in \text{stermsl } p$ "
  and "not_call q"
  and "wellformed  $\Gamma$ "
  shows "q  $\in \text{sterms } \Gamma p$ "
  <proof>

lemma sterms_stermsl_heads:

```

```

assumes "ps ∈ sterms Γ (Γ pn)"
  and "wellformed Γ"
  shows "∃pn. ps ∈ stermsl (Γ pn)"
⟨proof⟩

```

```

lemma sterms_subterms [dest]:
  assumes "wellformed Γ"
    and "∃pn. p ∈ subterms (Γ pn)"
    and "q ∈ sterms Γ p"
  shows "∃pn. q ∈ subterms (Γ pn)"
⟨proof⟩

```

```

lemma no_microsteps_sterms_refl:
  assumes "wellformed Γ"
  shows "(¬(∃q. p ⇨Γ q)) = (sterms Γ p = {p})"
⟨proof⟩

```

```

lemma sterms_maximal [elim]:
  assumes "wellformed Γ"
    and "q ∈ sterms Γ p"
  shows "sterms Γ q = {q}"
⟨proof⟩

```

```

lemma microstep_rtranscl_equal:
  assumes "not_call p"
    and "not_choice p"
    and "p ⇨Γ* q"
  shows "q = p"
⟨proof⟩

```

```

lemma microstep_rtranscl_singleton [simp]:
  assumes "not_call p"
    and "not_choice p"
  shows "{q. p ⇨Γ* q ∧ sterms Γ q = {q}} = {p}"
⟨proof⟩

```

```

theorem sterms_maximal_microstep:
  assumes "wellformed Γ"
  shows "sterms Γ p = {q. p ⇨Γ* q ∧ ¬(∃q'. q ⇨Γ q')}"
⟨proof⟩

```

7.5 Derivative terms

The derivatives of a term are those *sterms* potentially reachable by taking a transition, relative to a wellformed process specification Γ . These terms overapproximate the reachable terms, since the truth of guards is not considered.

```

function (domintros) dterms
:: "('s, 'm, 'p, 'l) seqp_env ⇒ ('s, 'm, 'p, 'l) seqp ⇒ ('s, 'm, 'p, 'l) seqp set"
where
  "dterms Γ ({l}⟨g⟩ p) = sterms Γ p"
| "dterms Γ ({l}⟦u⟧ p) = sterms Γ p"
| "dterms Γ (p1 ⊕ p2) = dterms Γ p1 ∪ dterms Γ p2"
| "dterms Γ ({l}unicast(sip, smsg).p ▷ q) = sterms Γ p ∪ sterms Γ q"
| "dterms Γ ({l}broadcast(smsg). p) = sterms Γ p"
| "dterms Γ ({l}groupcast(sips, smsg). p) = sterms Γ p"
| "dterms Γ ({l}send(smsg).p) = sterms Γ p"
| "dterms Γ ({l}deliver(sdata).p) = sterms Γ p"
| "dterms Γ ({l}receive(umsg).p) = sterms Γ p"
| "dterms Γ (call(pn)) = dterms Γ (Γ pn)"
⟨proof⟩

```

```

lemma dterms_dom_basic [simp]:
  assumes "not_call p"

```

```

    and "not_choice p"
    shows "dterms_dom ( $\Gamma$ , p)"
  <proof>

lemma dterms_termination:
  assumes "wellformed  $\Gamma$ "
  shows "dterms_dom ( $\Gamma$ , p)"
  <proof>

lemmas dterms_psimps [simp] = dterms.psimps [OF dterms_termination]
  and dterms_pinduct = dterms.pinduct [OF dterms_termination]

lemma sterms_after_dterms [simp]:
  assumes "wellformed  $\Gamma$ "
  shows " $(\bigcup x \in \text{dterms } \Gamma \text{ p. sterms } \Gamma \text{ x}) = \text{dterms } \Gamma \text{ p}$ "
  <proof>

lemma sterms_before_dterms [simp]:
  assumes "wellformed  $\Gamma$ "
  shows " $(\bigcup x \in \text{sterms } \Gamma \text{ p. dterms } \Gamma \text{ x}) = \text{dterms } \Gamma \text{ p}$ "
  <proof>

lemma dterms_choice_disj [simp]:
  assumes "wellformed  $\Gamma$ "
  shows " $p \in \text{dterms } \Gamma \text{ (p1 } \oplus \text{ p2)} = (p \in \text{dterms } \Gamma \text{ p1} \vee p \in \text{dterms } \Gamma \text{ p2})$ "
  <proof>

lemma dterms_in_branch:
  assumes "wellformed  $\Gamma$ "
  and "p  $\in$  dterms  $\Gamma$  (p1  $\oplus$  p2)"
  and "p  $\in$  dterms  $\Gamma$  p1  $\implies$  P"
  and "p  $\in$  dterms  $\Gamma$  p2  $\implies$  P"
  shows "P"
  <proof>

lemma dterms_no_choice:
  assumes "wellformed  $\Gamma$ "
  shows "p1  $\oplus$  p2  $\notin$  dterms  $\Gamma$  p"
  <proof>

lemma dterms_not_choice [simp]:
  assumes "wellformed  $\Gamma$ "
  and "q  $\in$  dterms  $\Gamma$  p"
  shows "not_choice q"
  <proof>

lemma dterms_no_call:
  assumes "wellformed  $\Gamma$ "
  shows "call(pn)  $\notin$  dterms  $\Gamma$  p"
  <proof>

lemma dterms_not_call [simp]:
  assumes "wellformed  $\Gamma$ "
  and "q  $\in$  dterms  $\Gamma$  p"
  shows "not_call q"
  <proof>

lemma dterms_subterms:
  assumes wf: "wellformed  $\Gamma$ "
  and " $\exists pn. p \in \text{subterms } (\Gamma \text{ pn})$ "
  and "q  $\in$  dterms  $\Gamma$  p"
  shows " $\exists pn. q \in \text{subterms } (\Gamma \text{ pn})$ "
  <proof>

```

Note that the converse of $\llbracket \text{wellformed } ?\Gamma; \exists pn. ?p \in \text{subterms } (?\Gamma pn); ?q \in \text{dterms } ?\Gamma ?p \rrbracket \implies \exists pn. ?q \in \text{subterms } (?\Gamma pn)$ is not true because *dterms* are an over-approximation; i.e., we cannot show, in general, that guards return a non-empty set of post-states.

7.6 Control terms

The control terms of a process specification Γ are those subterms from which transitions are directly possible. We can omit $\text{call}(pn)$ terms, since the root terms of all processes are considered, and also $p1 \oplus p2$ terms since they effectively combine the transitions of the subterms $p1$ and $p2$.

It will be shown that only the control terms, rather than all subterms, need be considered in invariant proofs.

inductive_set

```
cterm :: "('s, 'm, 'p, 'l) seqp_env  $\Rightarrow$  ('s, 'm, 'p, 'l) seqp set"
for  $\Gamma$  :: "('s, 'm, 'p, 'l) seqp_env"
```

where

```
ctermSI[intro]: "p  $\in$  sterm  $\Gamma$  ( $\Gamma$  pn)  $\implies$  p  $\in$  cterm  $\Gamma$ "
| ctermDI[intro]: " $\llbracket$  pp  $\in$  cterm  $\Gamma$ ; p  $\in$  dterm  $\Gamma$  pp  $\rrbracket \implies$  p  $\in$  cterm  $\Gamma$ "
```

lemma *cterm_not_choice* [*simp*]:

```
assumes "wellformed  $\Gamma$ "
and "p  $\in$  cterm  $\Gamma$ "
shows "not_choice p"
```

<proof>

lemma *cterm_no_choice* [*simp*]:

```
assumes "wellformed  $\Gamma$ "
shows "p1  $\oplus$  p2  $\notin$  cterm  $\Gamma$ "
```

<proof>

lemma *cterm_not_call* [*simp*]:

```
assumes "wellformed  $\Gamma$ "
and "p  $\in$  cterm  $\Gamma$ "
shows "not_call p"
```

<proof>

lemma *cterm_no_call* [*simp*]:

```
assumes "wellformed  $\Gamma$ "
shows "call(pn)  $\notin$  cterm  $\Gamma$ "
```

<proof>

lemma *sterm_cterm* [*elim*]:

```
assumes "p  $\in$  cterm  $\Gamma$ "
and "q  $\in$  sterm  $\Gamma$  p"
and "wellformed  $\Gamma$ "
shows "q  $\in$  cterm  $\Gamma$ "
```

<proof>

lemma *dterm_cterm* [*elim*]:

```
assumes "p  $\in$  cterm  $\Gamma$ "
and "q  $\in$  dterm  $\Gamma$  p"
and "wellformed  $\Gamma$ "
shows "q  $\in$  cterm  $\Gamma$ "
```

<proof>

lemma *derivs_in_cterm* [*simp*]:

```
" $\bigwedge$  l f p. {l}<f> p  $\in$  cterm  $\Gamma$   $\implies$  sterm  $\Gamma$  p  $\subseteq$  cterm  $\Gamma$ "
" $\bigwedge$  l f p. {l}[f] p  $\in$  cterm  $\Gamma$   $\implies$  sterm  $\Gamma$  p  $\subseteq$  cterm  $\Gamma$ "
" $\bigwedge$  l fip fmsg q p. {l}unicast(fip, fmsg). p  $\triangleright$  q  $\in$  cterm  $\Gamma$ 
 $\implies$  sterm  $\Gamma$  p  $\subseteq$  cterm  $\Gamma$   $\wedge$  sterm  $\Gamma$  q  $\subseteq$  cterm  $\Gamma$ "
" $\bigwedge$  l fmsg p. {l}broadcast(fmsg).p  $\in$  cterm  $\Gamma$   $\implies$  sterm  $\Gamma$  p  $\subseteq$  cterm  $\Gamma$ "
" $\bigwedge$  l fips fmsg p. {l}groupcast(fips, fmsg).p  $\in$  cterm  $\Gamma$   $\implies$  sterm  $\Gamma$  p  $\subseteq$  cterm  $\Gamma$ "
" $\bigwedge$  l fmsg p. {l}send(fmsg).p  $\in$  cterm  $\Gamma$   $\implies$  sterm  $\Gamma$  p  $\subseteq$  cterm  $\Gamma$ "
" $\bigwedge$  l fdata p. {l}deliver(fdata).p  $\in$  cterm  $\Gamma$   $\implies$  sterm  $\Gamma$  p  $\subseteq$  cterm  $\Gamma$ "
```

" $\wedge l$ fmsg p. $\{l\}receive(fmsg).p \in cterms \Gamma \implies sterm s \Gamma p \subseteq cterms \Gamma$ "
 <proof>

7.7 Local control terms

We introduce a ‘local’ version of *cterms* that does not step through calls and, thus, that is defined independently of a process specification Γ . This allows an alternative, terminating characterisation of *cterms* as a set of subterms. Including *call(pn)*s in the set makes for a simpler relation with *stermsl*, even if they must be filtered out for the desired characterisation.

function

ctermsl :: "('s, 'm, 'p, 'l) seqp \Rightarrow ('s, 'm, 'p, 'l) seqp set"

where

"*ctermsl* ($\{l\}\langle g \rangle p$) = insert ($\{l\}\langle g \rangle p$) (*ctermsl* p)"
 | "*ctermsl* ($\{l\}\llbracket u \rrbracket p$) = insert ($\{l\}\llbracket u \rrbracket p$) (*ctermsl* p)"
 | "*ctermsl* ($\{l\}unicast(s_{ip}, s_{msg}). p \triangleright q$) = insert ($\{l\}unicast(s_{ip}, s_{msg}). p \triangleright q$) (*ctermsl* p \cup *ctermsl* q)"
 | "*ctermsl* ($\{l\}broadcast(s_{msg}). p$) = insert ($\{l\}broadcast(s_{msg}). p$) (*ctermsl* p)"
 | "*ctermsl* ($\{l\}groupcast(s_{ips}, s_{msg}). p$) = insert ($\{l\}groupcast(s_{ips}, s_{msg}). p$) (*ctermsl* p)"
 | "*ctermsl* ($\{l\}send(s_{msg}). p$) = insert ($\{l\}send(s_{msg}). p$) (*ctermsl* p)"
 | "*ctermsl* ($\{l\}deliver(s_{data}). p$) = insert ($\{l\}deliver(s_{data}). p$) (*ctermsl* p)"
 | "*ctermsl* ($\{l\}receive(u_{msg}). p$) = insert ($\{l\}receive(u_{msg}). p$) (*ctermsl* p)"
 | "*ctermsl* ($p_1 \oplus p_2$) = *ctermsl* p₁ \cup *ctermsl* p₂"
 | "*ctermsl* (*call*(pn)) = {*call*(pn)}"

<proof>

termination <proof>

lemmas *ctermsl_induct* =

ctermsl.induct [case_names GUARD ASSIGN UCAST BCAST GCAST
 SEND DELIVER RECEIVE CHOICE CALL]

lemma *ctermsl_refl* [intro]: "*not_choice* p $\implies p \in ctermsl p$ "

<proof>

lemma *ctermsl_subterms*:

"*ctermsl* p = {q. q \in subterms p \wedge *not_choice* q}" (is "?lhs = ?rhs")

<proof>

lemma *ctermsl_trans* [elim]:

assumes "q \in *ctermsl* p"

and "r \in *ctermsl* q"

shows "r \in *ctermsl* p"

<proof>

lemma *ctermsl_ex_trans* [elim]:

assumes " $\exists q \in ctermsl p. r \in ctermsl q$ "

shows "r \in *ctermsl* p"

<proof>

lemma *call_ctermsl_empty* [elim]:

" $\llbracket p \in ctermsl p' ; not_call p \rrbracket \implies not_call p'$ "

<proof>

lemma *stermsl_ctermsl_choice1* [simp]:

assumes "q \in *stermsl* p₁"

shows "q \in *ctermsl* (p₁ \oplus p₂)"

<proof>

lemma *stermsl_ctermsl_choice2* [simp]:

assumes "q \in *stermsl* p₂"

shows "q \in *ctermsl* (p₁ \oplus p₂)"

<proof>

lemma *stermsl_ctermsl* [elim]:

```

assumes "q ∈ stermsl p"
shows "q ∈ ctermsl p"
⟨proof⟩

```

```

lemma stermsl_after_ctermsl [simp]:
"(⋃x∈ctermsl p. stermsl x) = ctermsl p"
⟨proof⟩

```

```

lemma stermsl_before_ctermsl [simp]:
"(⋃x∈stermsl p. ctermsl x) = ctermsl p"
⟨proof⟩

```

```

lemma ctermsl_no_choice: "p1 ⊕ p2 ∉ ctermsl p"
⟨proof⟩

```

```

lemma ctermsl_ex_stermsl: "q ∈ ctermsl p ⇒ ∃ps∈stermsl p. q ∈ ctermsl ps"
⟨proof⟩

```

```

lemma dterms_ctermsl [intro]:
assumes "q ∈ dterms Γ p"
and "wellformed Γ"
shows "q ∈ ctermsl p ∨ (∃pn. q ∈ ctermsl (Γ pn))"
⟨proof⟩

```

```

lemma ctermsl_cterms [elim]:
assumes "q ∈ ctermsl p"
and "not_call q"
and "sterms Γ p ⊆ cterms Γ"
and "wellformed Γ"
shows "q ∈ cterms Γ"
⟨proof⟩

```

7.8 Local deriviative terms

We define local *dterms* for use in the theorem that relates *cterms* and sets of *ctermsl*.

```

function dtermsl
:: "('s, 'm, 'p, 'l) seqp ⇒ ('s, 'm, 'p, 'l) seqp set"
where
  "dtermsl ({}⟨fg⟩ p) = stermsl p"
| "dtermsl ({}[[fa]] p) = stermsl p"
| "dtermsl (p1 ⊕ p2) = dtermsl p1 ∪ dtermsl p2"
| "dtermsl ({}unicast(fip, fmsg).p ▷ q) = stermsl p ∪ stermsl q"
| "dtermsl ({}broadcast(fmsg). p) = stermsl p"
| "dtermsl ({}groupcast(fips, fmsg). p) = stermsl p"
| "dtermsl ({}send(fmsg).p) = stermsl p"
| "dtermsl ({}deliver(fdata).p) = stermsl p"
| "dtermsl ({}receive(fmsg).p) = stermsl p"
| "dtermsl (call(pn)) = {}"
⟨proof⟩
termination ⟨proof⟩

```

```

lemma stermsl_after_dtermsl [simp]:
shows "(⋃x∈dtermsl p. stermsl x) = dtermsl p"
⟨proof⟩

```

```

lemma stermsl_before_dtermsl [simp]:
"(⋃x∈stermsl p. dtermsl x) = dtermsl p"
⟨proof⟩

```

```

lemma dtermsl_no_choice [simp]: "p1 ⊕ p2 ∉ dtermsl p"
⟨proof⟩

```

```

lemma dtermsl_choice_disj [simp]:
"p ∈ dtermsl (p1 ⊕ p2) = (p ∈ dtermsl p1 ∨ p ∈ dtermsl p2)"

```

<proof>

lemma *dtermsl_in_branch* [elim]:

" $\llbracket p \in \text{dtermsl } (p1 \oplus p2); p \in \text{dtermsl } p1 \implies P; p \in \text{dtermsl } p2 \implies P \rrbracket \implies P$ "
<proof>

lemma *ctermssl_dtermsl* [elim]:

assumes "q \in dtermsl p"
shows "q \in ctermssl p"
<proof>

lemma *dtermsl_dterms* [elim]:

assumes "q \in dtermsl p"
and "not_call q"
and "wellformed Γ "
shows "q \in dterms Γ p"
<proof>

lemma *ctermssl_stermssl_or_dtermssl*:

assumes "q \in ctermssl p"
shows "q \in stermssl p \vee ($\exists p' \in \text{dtermsl } p. q \in \text{ctermssl } p'$)"
<proof>

lemma *dtermsl_add_stermssl_beforeD*:

assumes "q \in dtermsl p"
shows " $\exists ps \in \text{stermssl } p. q \in \text{dtermsl } ps$ "
<proof>

lemma *call_dtermsl_empty* [elim]:

"q \in dtermsl p \implies not_call p"
<proof>

7.9 More properties of control terms

We now show an alternative definition of *ctermssl* based on sets of local control terms. While the original definition has convenient induction and simplification rules, useful for proving properties like *ctermssl_includes_stermssl_of_seq_reachable*, this definition makes it easier to systematically generate the set of control terms of a process specification.

theorem *ctermssl_def'*:

assumes wfg: "wellformed Γ "
shows "ctermssl $\Gamma = \{ p \mid p \text{ pn. } p \in \text{ctermssl } (\Gamma \text{ pn}) \wedge \text{not_call } p \}$ "
(is " $_ = ?\text{ctermssl_set}$ ")
<proof>

lemma *ctermsslE* [elim]:

assumes "wellformed Γ "
and "p \in ctermssl Γ "
obtains pn where "p \in ctermssl (Γ pn)"
and "not_call p"
<proof>

corollary *ctermssl_subtermssl*:

assumes "wellformed Γ "
shows "ctermssl $\Gamma = \{ p \mid p \text{ pn. } p \in \text{subtermssl } (\Gamma \text{ pn}) \wedge \text{not_call } p \wedge \text{not_choice } p \}$ "
<proof>

lemma *subtermssl_in_ctermssl* [elim]:

assumes "wellformed Γ "
and "p \in subtermssl (Γ pn)"
and "not_call p"
and "not_choice p"
shows "p \in ctermssl Γ "
<proof>


```

lemma subterms_stermsl_ctermsl:
  assumes "q ∈ subterms p"
    and "r ∈ stermsl q"
  shows "r ∈ ctermsl p"
  ⟨proof⟩

```

```

lemma subterms_sterms_cterms:
  assumes wf: "wellformed Γ"
    and "p ∈ subterms (Γ pn)"
  shows "sterms Γ p ⊆ cterms Γ"
  ⟨proof⟩

```

```

lemma subterms_sterms_in_cterms:
  assumes "wellformed Γ"
    and "p ∈ subterms (Γ pn)"
    and "q ∈ sterms Γ p"
  shows "q ∈ cterms Γ"
  ⟨proof⟩

```

end

8 Labelling sequential processes

```

theory AWN_Labels
imports AWN AWN_Cterms
begin

```

8.1 Labels

Labels serve two main purposes. They allow the substitution of *sterms* in *invariant* proofs. They also allow the strengthening (control state dependent) of invariants.

```

function (domintros) labels
:: "('s, 'm, 'p, 'l) seqp_env ⇒ ('s, 'm, 'p, 'l) seqp ⇒ 'l set"
where
  "labels Γ ({l}⟨fg⟩ p) = {l}"
| "labels Γ ({l}[[fa] p) = {l}"
| "labels Γ (p1 ⊕ p2) = labels Γ p1 ∪ labels Γ p2"
| "labels Γ ({l}unicast(fip, fmsg).p ▷ q) = {l}"
| "labels Γ ({l}broadcast(fmsg). p) = {l}"
| "labels Γ ({l}groupcast(fips, fmsg). p) = {l}"
| "labels Γ ({l}send(fmsg).p) = {l}"
| "labels Γ ({l}deliver(fdata).p) = {l}"
| "labels Γ ({l}receive(fmsg).p) = {l}"
| "labels Γ (call(pn)) = labels Γ (Γ pn)"
  ⟨proof⟩

```

```

lemma labels_dom_basic [simp]:
  assumes "not_call p"
    and "not_choice p"
  shows "labels_dom (Γ, p)"
  ⟨proof⟩

```

```

lemma labels_termination:
  fixes Γ p
  assumes "wellformed(Γ)"
  shows "labels_dom (Γ, p)"
  ⟨proof⟩

```

```

declare labels.psimps[simp]

```

```

lemmas labels_pinduct = labels.pinduct [OF labels_termination]
and labels_psimps[simp] = labels.psimps [OF labels_termination]

```

lemma labels_not_empty:

```

  fixes  $\Gamma$  p
  assumes "wellformed  $\Gamma$ "
  shows "labels  $\Gamma$  p  $\neq$  {}"
  <proof>

```

lemma has_label [dest]:

```

  fixes  $\Gamma$  p
  assumes "wellformed  $\Gamma$ "
  shows " $\exists l. l \in \text{labels } \Gamma p$ "
  <proof>

```

lemma singleton_labels [simp]:

```

" $\wedge \Gamma l l' f p. l \in \text{labels } \Gamma (\{l'\}\langle f \rangle p) = (l = l')$ "
" $\wedge \Gamma l l' f p. l \in \text{labels } \Gamma (\{l'\}\llbracket f \rrbracket p) = (l = l')$ "
" $\wedge \Gamma l l' \text{fip } fmsg p q. l \in \text{labels } \Gamma (\{l'\}\text{unicast}(fip, fmsg).p \triangleright q) = (l = l')$ "
" $\wedge \Gamma l l' fmsg p. l \in \text{labels } \Gamma (\{l'\}\text{broadcast}(fmsg).p) = (l = l')$ "
" $\wedge \Gamma l l' \text{fips } fmsg p. l \in \text{labels } \Gamma (\{l'\}\text{groupcast}(fips, fmsg).p) = (l = l')$ "
" $\wedge \Gamma l l' fmsg p. l \in \text{labels } \Gamma (\{l'\}\text{send}(fmsg).p) = (l = l')$ "
" $\wedge \Gamma l l' fdata p. l \in \text{labels } \Gamma (\{l'\}\text{deliver}(fdata).p) = (l = l')$ "
" $\wedge \Gamma l l' fmsg p. l \in \text{labels } \Gamma (\{l'\}\text{receive}(fmsg).p) = (l = l')$ "
<proof>

```

lemma in_labels_singletons [dest!]:

```

" $\wedge \Gamma l l' f p. l \in \text{labels } \Gamma (\{l'\}\langle f \rangle p) \implies l = l'$ "
" $\wedge \Gamma l l' f p. l \in \text{labels } \Gamma (\{l'\}\llbracket f \rrbracket p) \implies l = l'$ "
" $\wedge \Gamma l l' \text{fip } fmsg p q. l \in \text{labels } \Gamma (\{l'\}\text{unicast}(fip, fmsg).p \triangleright q) \implies l = l'$ "
" $\wedge \Gamma l l' fmsg p. l \in \text{labels } \Gamma (\{l'\}\text{broadcast}(fmsg).p) \implies l = l'$ "
" $\wedge \Gamma l l' \text{fips } fmsg p. l \in \text{labels } \Gamma (\{l'\}\text{groupcast}(fips, fmsg).p) \implies l = l'$ "
" $\wedge \Gamma l l' fmsg p. l \in \text{labels } \Gamma (\{l'\}\text{send}(fmsg).p) \implies l = l'$ "
" $\wedge \Gamma l l' fdata p. l \in \text{labels } \Gamma (\{l'\}\text{deliver}(fdata).p) \implies l = l'$ "
" $\wedge \Gamma l l' fmsg p. l \in \text{labels } \Gamma (\{l'\}\text{receive}(fmsg).p) \implies l = l'$ "
<proof>

```

definition

```

simple_labels :: "( $'s, 'm, 'p, 'l$ ) seqp_env  $\Rightarrow$  bool"

```

where

```

"simple_labels  $\Gamma \equiv \forall pn. \forall p \in \text{subterms } (\Gamma pn). (\exists !l. \text{labels } \Gamma p = \{l\})"$ 

```

lemma simple_labelsI [intro]:

```

assumes " $\wedge pn p. p \in \text{subterms } (\Gamma pn) \implies \exists !l. \text{labels } \Gamma p = \{l\}$ "
shows "simple_labels  $\Gamma$ "
<proof>

```

The *simple_labels* Γ property is necessary to transfer results shown over the *cterm*s of a process specification Γ to the reachable actions of that process.

Consider the process $\{l_1\}\text{send}(m1) . p1 \oplus \{l_2\}\text{send}(m2) . p2$. The iteration over *cterm*s Γ will cover the two transitions $(l_1, \text{send } m1, p1)$ and $(l_2, \text{send } m2, p2)$, but reachability requires the four transitions $(l_1, \text{send } m1, p1)$, $(l_1, \text{send } m2, p2)$, $(l_2, \text{send } m1, p1)$, and $(l_2, \text{send } m2, p2)$.

In a simply labelled process, the former is sufficient to show the latter, since $l_1 = l_2$.

This requirement seems really only to be restrictive for processes where a *call*(pn) occurs as a direct subterm of a choice operator. Consider, for instance, $\{l_1\}\llbracket e \rrbracket p \oplus \text{call}(pn)$. Here l_1 must equal the label of Γpn , which can then not be distinguished from any other subterm that calls pn in any other process.

This limitation stems from the fact that the "call points" of a process are effectively treated as the root of the called process. This is by design; we try to treat call sites as "syntactic pastings" of process terms, giving rise, conceptually, to an infinite tree structure. But this prejudices the alternative view that process calls are used as "join points" of "process threads", in complement to the "fork points" of the $p1 \oplus p2$ operator.

lemma simple_labels_in_sterms:

```

  fixes  $\Gamma l p$ 
  assumes "simple_labels  $\Gamma$ "
  and "wellformed  $\Gamma$ "
  and " $\exists pn. p \in \text{subterms } (\Gamma pn)$ "

```

```

    and "l ∈ labels Γ p"
    shows "∀ p' ∈ sterm s Γ p. l ∈ labels Γ p'"
  <proof>

```

lemma labels_in_sterm s:

```

  fixes Γ l p
  assumes "wellformed Γ"
    and "l ∈ labels Γ p"
  shows "∃ p' ∈ sterm s Γ p. l ∈ labels Γ p'"
  <proof>

```

lemma labels_sterm s_labels:

```

  fixes Γ p p' l
  assumes "wellformed Γ"
    and "p' ∈ sterm s Γ p"
    and "l ∈ labels Γ p'"
  shows "l ∈ labels Γ p"
  <proof>

```

primrec labelfrom :: "int ⇒ int ⇒ ('s, 'm, 'p, 'a) seqp ⇒ int × ('s, 'm, 'p, int) seqp"
where

```

  "labelfrom n nn (f) p =
    (let (nn', p') = labelfrom nn (nn + 1) p in
     (nn', {n} f p'))"
| "labelfrom n nn (f) p =
    (let (nn', p') = labelfrom nn (nn + 1) p in
     (nn', {n} f p'))"
| "labelfrom n nn (p ⊕ q) =
    (let (nn', p') = labelfrom n nn p in
     let (nn'', q') = labelfrom n nn' q in
     (nn'', p' ⊕ q'))"
| "labelfrom n nn (unicast(fip, fmsg). p ▷ q) =
    (let (nn', p') = labelfrom nn (nn + 1) p in
     let (nn'', q') = labelfrom nn' (nn' + 1) q in
     (nn'', {n} unicast(fip, fmsg). p ▷ q'))"
| "labelfrom n nn (broadcast(fmsg). p) =
    (let (nn', p') = labelfrom nn (nn + 1) p in
     (nn', {n} broadcast(fmsg). p'))"
| "labelfrom n nn (groupcast(fipset, fmsg). p) =
    (let (nn', p') = labelfrom nn (nn + 1) p in
     (nn', {n} groupcast(fipset, fmsg). p'))"
| "labelfrom n nn (send(fmsg). p) =
    (let (nn', p') = labelfrom nn (nn + 1) p in
     (nn', {n} send(fmsg). p'))"
| "labelfrom n nn (deliver(fdata). p) =
    (let (nn', p') = labelfrom nn (nn + 1) p in
     (nn', {n} deliver(fdata). p'))"
| "labelfrom n nn (receive(fmsg). p) =
    (let (nn', p') = labelfrom nn (nn + 1) p in
     (nn', {n} receive(fmsg). p'))"
| "labelfrom n nn (call(fargs)) = (nn - 1, call(fargs))"

```

datatype 'pn label =

```

  LABEL 'pn int ("_:_:" [1000, 1000] 999)

```

instantiation "label" :: (ord) ord

begin

```

fun less_eq_label :: "'a label ⇒ 'a label ⇒ bool"

```

```

where "(l1:_:n1) ≤ (l2:_:n2) = (l1 = l2 ∧ n1 ≤ n2)"

```

```

definition less_label: "(l1:_: 'a label) < l2 ⇔ l1 ≤ l2 ∧ ¬ (l1 ≤ l2)"

```

instance <proof>

```

end

abbreviation labelled :: "'p ⇒ ('s, 'm, 'p, 'a) seqp ⇒ ('s, 'm, 'p, 'p label) seqp"
where "labelled pn p ≡ labelmap (λl. LABEL pn l) (snd (labelfrom 0 1 p))"

end

```

9 A custom tactic for showing invariants via control terms

```

theory Inv_Cterms
imports AWN_Labels
begin

```

This tactic tries to solve a goal by reducing it to a problem over (local) cterms (using one of the cterms_intros intro rules); expanding those to consider all process names (using one of the ctermsl_cases destruction rules); simplifying each (using the cterms_env simplification rules); splitting them up into separate subgoals; replacing the derivative term with a variable; ‘executing’ a transition of each term; and then simplifying.

The tactic can stop after applying introduction rule (“inv_cterms (intro_only)”), or after having generated the verification condition subgoals and before having simplified them (“inv_cterms (vcs_only)”). It takes arguments to add or remove simplification rules (“simp add: lemmanames”), to add forward rules on assumptions (to introduce previously proved invariants; “inv add: lemmanames”), or to add elimination rules that solve any remaining subgoals (“solve: lemmanames”).

To configure the tactic for a set of transition rules:

1. add elimination rules: declare seqpTEs [cterms_seqte]
2. add rules to replace derivative terms: declare elimders [cterms_elimders]

To configure the tactic for a process environment (Γ):

1. add simp rules: declare Γ .simps [cterms_env]
2. add case rules: declare aadv_proc_cases [ctermsl_cases]
3. add invariant intros declare seq_invariant_ctermsI [OF aadv_wf aadv_control_within aadv_simple_labels, cterms_intro]
 seq_step_invariant_ctermsI [OF aadv_wf aadv_control_within aadv_simple_labels, cterms_intros]

```

lemma has_ctermsl: "p ∈ ctermsl  $\Gamma$  ⇒ p ∈ ctermsl  $\Gamma$ " <proof>

```

```

named.theorems cterms_elimders "rules for truncating sequential process terms"
named.theorems cterms_seqte "elimination rules for sequential process terms"
named.theorems cterms_env "simplification rules for sequential process environments"
named.theorems ctermsl_cases "destruction rules for case splitting ctermsl"
named.theorems cterms_intros "introduction rules from cterms"
named.theorems cterms_invs "invariants to try to apply at each vc"
named.theorems cterms_final "elimination rules to try on each vc after simplification"

```

```

<ML>

```

```

declare
  insert_iff [cterms_env]
  Un_insert_right [cterms_env]
  sup_bot_right [cterms_env]
  Product_Type.prod_cases [cterms_env]
  ctermsl_simps [cterms_env]

```

```

end

```

10 Configure the inv-cterms tactic for sequential processes

```

theory AWN_SOS_Labels
imports AWN_SOS Inv_Cterms

```

begin

lemma elimder_guard:

assumes "p = {l}<fg> qq"
and "l' ∈ labels Γ q"
and "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
obtains p' where "p = {l}<fg> p'"
and "l' ∈ labels Γ qq"

<proof>

lemma elimder_assign:

assumes "p = {l}[[fa]] qq"
and "l' ∈ labels Γ q"
and "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
obtains p' where "p = {l}[[fa]] p'"
and "l' ∈ labels Γ qq"

<proof>

lemma elimder_ucast:

assumes "p = {l}unicast(fip, fmsg).q1 ▷ q2"
and "l' ∈ labels Γ q"
and "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
obtains p' pp' where "p = {l}unicast(fip, fmsg).p' ▷ pp'"
and "case a of unicast _ _ ⇒ l' ∈ labels Γ q1
| _ ⇒ l' ∈ labels Γ q2"

<proof>

lemma elimder_bcast:

assumes "p = {l}broadcast(fmsg).qq"
and "l' ∈ labels Γ q"
and "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
obtains p' where "p = {l}broadcast(fmsg). p'"
and "l' ∈ labels Γ qq"

<proof>

lemma elimder_gcast:

assumes "p = {l}groupcast(fips, fmsg).qq"
and "l' ∈ labels Γ q"
and "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
obtains p' where "p = {l}groupcast(fips, fmsg). p'"
and "l' ∈ labels Γ qq"

<proof>

lemma elimder_send:

assumes "p = {l}send(fmsg).qq"
and "l' ∈ labels Γ q"
and "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
obtains p' where "p = {l}send(fmsg). p'"
and "l' ∈ labels Γ qq"

<proof>

lemma elimder_deliver:

assumes "p = {l}deliver(fdata).qq"
and "l' ∈ labels Γ q"
and "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
obtains p' where "p = {l}deliver(fdata). p'"
and "l' ∈ labels Γ qq"

<proof>

lemma elimder_receive:

assumes "p = {l}receive(fmsg).qq"
and "l' ∈ labels Γ q"
and "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
obtains p' where "p = {l}receive(fmsg). p'"

```

    and "l' ∈ labels Γ qq"
  ⟨proof⟩

lemmas elimders =
  elimder_guard
  elimder_assign
  elimder_ucast
  elimder_bcast
  elimder_gcast
  elimder_send
  elimder_deliver
  elimder_receive

declare
  seqpTEs [cterms_seqte]
  elimders [cterms_elimders]

end

```

11 Lemmas for partial networks

```

theory Pnet
imports AWN_SOS Invariants
begin

```

These lemmas mostly concern the preservation of node structure by `pnet_sos` transitions.

```

lemma pnet_maintains_dom:
  assumes "(s, a, s') ∈ trans (pnet np p)"
  shows "net_ips s = net_ips s'"
  ⟨proof⟩

lemma pnet_net_ips_net_tree_ips [elim]:
  assumes "s ∈ reachable (pnet np p) I"
  shows "net_ips s = net_tree_ips p"
  ⟨proof⟩

lemma pnet_init_net_ips_net_tree_ips:
  assumes "s ∈ init (pnet np p)"
  shows "net_ips s = net_tree_ips p"
  ⟨proof⟩

lemma pnet_init_in_net_ips_in_net_tree_ips [elim]:
  assumes "s ∈ init (pnet np p)"
  and "i ∈ net_ips s"
  shows "i ∈ net_tree_ips p"
  ⟨proof⟩

lemma pnet_init_in_net_tree_ips_in_net_ips [elim]:
  assumes "s ∈ init (pnet np p)"
  and "i ∈ net_tree_ips p"
  shows "i ∈ net_ips s"
  ⟨proof⟩

lemma pnet_init_not_in_net_tree_ips_not_in_net_ips [elim]:
  assumes "s ∈ init (pnet np p)"
  and "i ∉ net_tree_ips p"
  shows "i ∉ net_ips s"
  ⟨proof⟩

lemma net_node_reachable_is_node:
  assumes "st ∈ reachable (pnet np ⟨ii; R_i⟩) I"
  shows "∃ ns R. st = NodeS ii ns R"
  ⟨proof⟩

```

```

lemma partial_net_preserves_subnets:
  assumes "(SubnetS s t, a, st') ∈ pnet_sos (trans (pnet np p1)) (trans (pnet np p2))"
  shows "∃ s' t'. st' = SubnetS s' t'"
  ⟨proof⟩

lemma net_par_reachable_is_subnet:
  assumes "st ∈ reachable (pnet np (p1 || p2)) I"
  shows "∃ s t. st = SubnetS s t"
  ⟨proof⟩

lemma reachable_par_subnet_induct [consumes, case_names init step]:
  assumes "SubnetS s t ∈ reachable (pnet np (p1 || p2)) I"
  and init: "∧ s t. SubnetS s t ∈ init (pnet np (p1 || p2)) ⇒ P s t"
  and step: "∧ s t s' t' a. [
    SubnetS s t ∈ reachable (pnet np (p1 || p2)) I;
    P s t; (SubnetS s t, a, SubnetS s' t') ∈ (trans (pnet np (p1 || p2))); I a ]
    ⇒ P s' t'"
  shows "P s t"
  ⟨proof⟩

lemma subnet_reachable:
  assumes "SubnetS s1 s2 ∈ reachable (pnet np (p1 || p2)) TT"
  shows "s1 ∈ reachable (pnet np p1) TT"
  "s2 ∈ reachable (pnet np p2) TT"
  ⟨proof⟩

lemma delivered_to_node [elim]:
  assumes "s ∈ reachable (pnet np ⟨ii; Ri⟩) TT"
  and "(s, i:deliver(d), s') ∈ trans (pnet np ⟨ii; Ri⟩)"
  shows "i = ii"
  ⟨proof⟩

lemma delivered_to_net_ips:
  assumes "s ∈ reachable (pnet np p) TT"
  and "(s, i:deliver(d), s') ∈ trans (pnet np p)"
  shows "i ∈ net_ips s"
  ⟨proof⟩

lemma wf_net_tree_net_ips_disjoint [elim]:
  assumes "wf_net_tree (p1 || p2)"
  and "s1 ∈ reachable (pnet np p1) S"
  and "s2 ∈ reachable (pnet np p2) S"
  shows "net_ips s1 ∩ net_ips s2 = {}"
  ⟨proof⟩

lemma init_mapstate_Some_aadv_init [elim]:
  assumes "s ∈ init (pnet np p)"
  and "netmap s i = Some v"
  shows "v ∈ init (np i)"
  ⟨proof⟩

lemma reachable_connect_netmap [elim]:
  assumes "s ∈ reachable (pnet np n) TT"
  and "(s, connect(i, i'), s') ∈ trans (pnet np n)"
  shows "netmap s' = netmap s"
  ⟨proof⟩

lemma reachable_disconnect_netmap [elim]:
  assumes "s ∈ reachable (pnet np n) TT"
  and "(s, disconnect(i, i'), s') ∈ trans (pnet np n)"
  shows "netmap s' = netmap s"
  ⟨proof⟩

fun net_ip_action :: "(ip ⇒ ('s, 'm seq_action) automaton)

```

$\Rightarrow 'm \text{ node_action} \Rightarrow ip \Rightarrow \text{net_tree} \Rightarrow 's \text{ net_state} \Rightarrow 's \text{ net_state} \Rightarrow \text{bool}$ "

where

```
"net_ip_action np a i (p1 || p2) (SubnetS s1 s2) (SubnetS s1' s2') =
  ((i ∈ net_ips s1 → ((s1, a, s1') ∈ trans (pnet np p1)
    ∧ s2' = s2 ∧ net_ip_action np a i p1 s1 s1'))
  ∧ (i ∈ net_ips s2 → ((s2, a, s2') ∈ trans (pnet np p2))
    ∧ s1' = s1 ∧ net_ip_action np a i p2 s2 s2'))"
| "net_ip_action np a i p s s' = True"
```

lemma pnet_tau_single_node [elim]:

```
assumes "wf_net_tree p"
  and "s ∈ reachable (pnet np p) TT"
  and "(s, τ, s') ∈ trans (pnet np p)"
shows "∃i∈net_ips s. ((∀j. j≠i → netmap s' j = netmap s j)
  ∧ net_ip_action np τ i p s s')"
```

<proof>

lemma pnet_deliver_single_node [elim]:

```
assumes "wf_net_tree p"
  and "s ∈ reachable (pnet np p) TT"
  and "(s, i:deliver(d), s') ∈ trans (pnet np p)"
shows "(∀j. j≠i → netmap s' j = netmap s j) ∧ net_ip_action np (i:deliver(d)) i p s s'"
(is "?P p s s'")
```

<proof>

end

12 Lemmas for closed networks

theory Closed

imports Pnet

begin

lemma complete_net_preserves_subnets:

```
assumes "(SubnetS s t, a, st') ∈ cnet_sos (pnet_sos (trans (pnet np p1)) (trans (pnet np p2)))"
shows "∃s' t'. st' = SubnetS s' t'"
```

<proof>

lemma complete_net_reachable_is_subnet:

```
assumes "st ∈ reachable (closed (pnet np (p1 || p2))) I"
shows "∃s t. st = SubnetS s t"
```

<proof>

lemma closed_reachable_par_subnet_induct [consumes, case_names init step]:

```
assumes "SubnetS s t ∈ reachable (closed (pnet np (p1 || p2))) I"
  and init: "∧s t. SubnetS s t ∈ init (closed (pnet np (p1 || p2))) ⇒ P s t"
  and step: "∧s t s' t' a. [
    SubnetS s t ∈ reachable (closed (pnet np (p1 || p2))) I;
    P s t; (SubnetS s t, a, SubnetS s' t') ∈ trans (closed (pnet np (p1 || p2))); I a ]
  ⇒ P s' t'"
```

shows "P s t"

<proof>

lemma reachable_closed_reachable_pnet [elim]:

```
assumes "s ∈ reachable (closed (pnet np n)) TT"
shows "s ∈ reachable (pnet np n) TT"
```

<proof>

lemma closed_node_net_state [elim]:

```
assumes "st ∈ reachable (closed (pnet np ⟨ii; Ri⟩)) TT"
obtains ξ p q R where "st = NodeS ii ((ξ, p), q) R"
```

<proof>

lemma closed_subnet_net_state [elim]:


```

assumes "st ∈ reachable (closed (pnet np (p1 || p2))) TT"
obtains s t where "st = SubnetS s t"
⟨proof⟩

```

```

lemma closed_imp_pnet_trans [elim, dest]:
  assumes "(s, a, s') ∈ trans (closed (pnet np n))"
  shows "∃ a'. (s, a', s') ∈ trans (pnet np n)"
⟨proof⟩

```

```

lemma reachable_not_in_net_tree_ips [elim]:
  assumes "s ∈ reachable (closed (pnet np n)) TT"
  and "i ∉ net_tree_ips n"
  shows "netmap s i = None"
⟨proof⟩

```

```

lemma closed_pnet_aadv_init [elim]:
  assumes "s ∈ init (closed (pnet np n))"
  and "i ∈ net_tree_ips n"
  shows "the (netmap s i) ∈ init (np i)"
⟨proof⟩

```

end

13 Open semantics of the Algebra of Wireless Networks

```

theory OAWN_SOS
imports TransitionSystems AWN
begin

```

These are variants of the SOS rules that work against a mixed global/local context, where the global context is represented by a function σ mapping ip addresses to states.

13.1 Open structural operational semantics for sequential process expressions

```

inductive_set

```

```

  oseqp_sos
  :: "('s, 'm, 'p, 'l) seqp_env ⇒ ip
     ⇒ ((ip ⇒ 's) × ('s, 'm, 'p, 'l) seqp, 'm seq_action) transition set"
  for  $\Gamma$  :: "('s, 'm, 'p, 'l) seqp_env"
  and  $i$  :: ip

```

```

where

```

```

  obroadcastT: " $\sigma' i = \sigma i \implies$ 
                 $((\sigma, \{l\}broadcast(s_{msg}).p), \quad broadcast (s_{msg} (\sigma i)), \quad (\sigma', p)) \in oseqp\_T$ "
 $\Gamma i$ "
  | ogroupcastT: " $\sigma' i = \sigma i \implies$ 
                   $((\sigma, \{l\}groupcast(s_{ips}, s_{msg}).p), \quad groupcast (s_{ips} (\sigma i)) (s_{msg} (\sigma i)), (\sigma', p)) \in oseqp\_T$ "
 $\Gamma i$ "
  | ounicastT: " $\sigma' i = \sigma i \implies$ 
                 $((\sigma, \{l\}unicast(s_{ip}, s_{msg}).p \triangleright q), \quad unicast (s_{ip} (\sigma i)) (s_{msg} (\sigma i)), \quad (\sigma', p)) \in oseqp\_T$ "
 $\Gamma i$ "
  | onotunicastT: " $\sigma' i = \sigma i \implies$ 
                   $((\sigma, \{l\}unicast(s_{ip}, s_{msg}).p \triangleright q), \quad \neg unicast (s_{ip} (\sigma i)), \quad (\sigma', q)) \in oseqp\_T$ "
 $\Gamma i$ "
  | osendT: " $\sigma' i = \sigma i \implies$ 
              $((\sigma, \{l\}send(s_{msg}).p), \quad send (s_{msg} (\sigma i)), \quad (\sigma', p)) \in oseqp\_T$ "
 $\Gamma i$ "
  | odeliverT: " $\sigma' i = \sigma i \implies$ 
                 $((\sigma, \{l\}deliver(s_{data}).p), \quad deliver (s_{data} (\sigma i)), \quad (\sigma', p)) \in oseqp\_T$ "
 $\Gamma i$ "
  | oreceiveT: " $\sigma' i = u_{msg} msg (\sigma i) \implies$ 
                 $((\sigma, \{l\}receive(u_{msg}).p), \quad receive msg, \quad (\sigma', p)) \in oseqp\_T$ "
 $\Gamma i$ "
  | oassignT: " $\sigma' i = u (\sigma i) \implies$ "

```

$\Gamma \ i''$ $((\sigma, \{1\}[u] \ p), \tau, (\sigma', p)) \in \text{oseqp_sos}$

*o*callT: " $((\sigma, \Gamma \ pn), a, (\sigma', p')) \in \text{oseqp_sos } \Gamma \ i \implies$
 $((\sigma, \text{call}(pn)), a, (\sigma', p')) \in \text{oseqp_sos } \Gamma \ i''$ "

*o*choiceT1: " $((\sigma, p), a, (\sigma', p')) \in \text{oseqp_sos } \Gamma \ i \implies$
 $((\sigma, p \oplus q), a, (\sigma', p')) \in \text{oseqp_sos } \Gamma \ i''$ "

*o*choiceT2: " $((\sigma, q), a, (\sigma', q')) \in \text{oseqp_sos } \Gamma \ i \implies$
 $((\sigma, p \oplus q), a, (\sigma', q')) \in \text{oseqp_sos } \Gamma \ i''$ "

*o*guardT: " $\sigma' \ i \in g \ (\sigma \ i) \implies ((\sigma, \{1\}\langle g \rangle \ p), \tau, (\sigma', p)) \in \text{oseqp_sos } \Gamma \ i''$ "

inductive_cases

oseq_callTE [elim]: " $((\sigma, \text{call}(pn)), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i''$
and *oseq_choiceTE* [elim]: " $((\sigma, p_1 \oplus p_2), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i''$ "

lemma *oseq_broadcastTE* [elim]:

" $\llbracket ((\sigma, \{1\}\text{broadcast}(s_{msg}). p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i;$
 $\llbracket a = \text{broadcast } (s_{msg} \ (\sigma \ i)); \sigma' \ i = \sigma \ i; q = p \rrbracket \implies P \rrbracket \implies P''$ "
 $\langle \text{proof} \rangle$

lemma *oseq_groupcastTE* [elim]:

" $\llbracket ((\sigma, \{1\}\text{groupcast}(s_{ips}, s_{msg}). p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i;$
 $\llbracket a = \text{groupcast } (s_{ips} \ (\sigma \ i)) \ (s_{msg} \ (\sigma \ i)); \sigma' \ i = \sigma \ i; q = p \rrbracket \implies P \rrbracket \implies P''$ "
 $\langle \text{proof} \rangle$

lemma *oseq_unicastTE* [elim]:

" $\llbracket ((\sigma, \{1\}\text{unicast}(s_{ip}, s_{msg}). p \triangleright q), a, (\sigma', r)) \in \text{oseqp_sos } \Gamma \ i;$
 $\llbracket a = \text{unicast } (s_{ip} \ (\sigma \ i)) \ (s_{msg} \ (\sigma \ i)); \sigma' \ i = \sigma \ i; r = p \rrbracket \implies P;$
 $\llbracket a = \neg \text{unicast } (s_{ip} \ (\sigma \ i)); \sigma' \ i = \sigma \ i; r = q \rrbracket \implies P \rrbracket \implies P''$ "
 $\langle \text{proof} \rangle$

lemma *oseq_sendTE* [elim]:

" $\llbracket ((\sigma, \{1\}\text{send}(s_{msg}). p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i;$
 $\llbracket a = \text{send } (s_{msg} \ (\sigma \ i)); \sigma' \ i = \sigma \ i; q = p \rrbracket \implies P \rrbracket \implies P''$ "
 $\langle \text{proof} \rangle$

lemma *oseq_deliverTE* [elim]:

" $\llbracket ((\sigma, \{1\}\text{deliver}(s_{data}). p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i;$
 $\llbracket a = \text{deliver } (s_{data} \ (\sigma \ i)); \sigma' \ i = \sigma \ i; q = p \rrbracket \implies P \rrbracket \implies P''$ "
 $\langle \text{proof} \rangle$

lemma *oseq_receiveTE* [elim]:

" $\llbracket ((\sigma, \{1\}\text{receive}(u_{msg}). p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i;$
 $\bigwedge \text{msg}. \llbracket a = \text{receive } \text{msg}; \sigma' \ i = u_{msg} \ \text{msg} \ (\sigma \ i); q = p \rrbracket \implies P \rrbracket \implies P''$ "
 $\langle \text{proof} \rangle$

lemma *oseq_assignTE* [elim]:

" $\llbracket ((\sigma, \{1\}[u] \ p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i; \llbracket a = \tau; \sigma' \ i = u \ (\sigma \ i); q = p \rrbracket \implies P \rrbracket \implies P''$ "
 $\langle \text{proof} \rangle$

lemma *oseq_guardTE* [elim]:

" $\llbracket ((\sigma, \{1\}\langle g \rangle \ p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma \ i; \llbracket a = \tau; \sigma' \ i \in g \ (\sigma \ i); q = p \rrbracket \implies P \rrbracket \implies P''$ "
 $\langle \text{proof} \rangle$

lemmas *oseqpTEs* =

oseq_broadcastTE
oseq_groupcastTE
oseq_unicastTE
oseq_sendTE
oseq_deliverTE
oseq_receiveTE
oseq_assignTE

```

oseq_callTE
oseq_choiceTE
oseq_guardTE

```

```

declare oseqp_sos.intros [intro]

```

13.2 Open structural operational semantics for parallel process expressions

```

inductive_set

```

```

oparp_sos :: "ip
  ⇒ ((ip ⇒ 's) × 's1, 'm seq_action) transition set
  ⇒ ('s2, 'm seq_action) transition set
  ⇒ ((ip ⇒ 's) × ('s1 × 's2), 'm seq_action) transition set"

```

```

for i :: ip

```

```

and S :: "(ip ⇒ 's) × 's1, 'm seq_action) transition set"

```

```

and T :: "('s2, 'm seq_action) transition set"

```

```

where

```

```

oparleft:  "[ ((σ, s), a, (σ', s')) ∈ S; ∧m. a ≠ receive m ] ⇒
  ((σ, (s, t)), a, (σ', (s', t))) ∈ oparp_sos i S T"
| oparright: "[ (t, a, t') ∈ T; ∧m. a ≠ send m; σ' i = σ i ] ⇒
  ((σ, (s, t)), a, (σ', (s, t'))) ∈ oparp_sos i S T"
| oparboth: "[ ((σ, s), receive m, (σ', s')) ∈ S; (t, send m, t') ∈ T ] ⇒
  ((σ, (s, t)), τ, (σ', (s', t'))) ∈ oparp_sos i S T"

```

```

lemma opar_broadcastTE [elim]:

```

```

"[[((σ, (s, t)), broadcast m, (σ', (s', t')))) ∈ oparp_sos i S T;
  [[((σ, s), broadcast m, (σ', s')) ∈ S; t' = t] ⇒ P;
  [[(t, broadcast m, t') ∈ T; s' = s; σ' i = σ i] ⇒ P] ⇒ P"
⟨proof⟩

```

```

lemma opar_groupcastTE [elim]:

```

```

"[[((σ, (s, t)), groupcast ips m, (σ', (s', t')))) ∈ oparp_sos i S T;
  [[((σ, s), groupcast ips m, (σ', s')) ∈ S; t' = t] ⇒ P;
  [[(t, groupcast ips m, t') ∈ T; s' = s; σ' i = σ i] ⇒ P] ⇒ P"
⟨proof⟩

```

```

lemma opar_unicastTE [elim]:

```

```

"[[((σ, (s, t)), unicast i m, (σ', (s', t')))) ∈ oparp_sos i S T;
  [[((σ, s), unicast i m, (σ', s')) ∈ S; t' = t] ⇒ P;
  [[(t, unicast i m, t') ∈ T; s' = s; σ' i = σ i] ⇒ P] ⇒ P"
⟨proof⟩

```

```

lemma opar_notunicastTE [elim]:

```

```

"[[((σ, (s, t)), notunicast i, (σ', (s', t')))) ∈ oparp_sos i S T;
  [[((σ, s), notunicast i, (σ', s')) ∈ S; t' = t] ⇒ P;
  [[(t, notunicast i, t') ∈ T; s' = s; σ' i = σ i] ⇒ P] ⇒ P"
⟨proof⟩

```

```

lemma opar_sendTE [elim]:

```

```

"[[((σ, (s, t)), send m, (σ', (s', t')))) ∈ oparp_sos i S T;
  [[((σ, s), send m, (σ', s')) ∈ S; t' = t] ⇒ P] ⇒ P"
⟨proof⟩

```

```

lemma opar_deliverTE [elim]:

```

```

"[[((σ, (s, t)), deliver d, (σ', (s', t')))) ∈ oparp_sos i S T;
  [[((σ, s), deliver d, (σ', s')) ∈ S; t' = t] ⇒ P;
  [[(t, deliver d, t') ∈ T; s' = s; σ' i = σ i] ⇒ P] ⇒ P"
⟨proof⟩

```

```

lemma opar_receiveTE [elim]:

```

```

"[[((σ, (s, t)), receive m, (σ', (s', t')))) ∈ oparp_sos i S T;
  [[(t, receive m, t') ∈ T; s' = s; σ' i = σ i] ⇒ P] ⇒ P"
⟨proof⟩

```

inductive_cases opar_tauTE: " $((\sigma, (s, t)), \tau, (\sigma', (s', t')))) \in \text{oparp_sos } i \ S \ T$ "

lemmas oparpTEs =
 opar_broadcastTE
 opar_groupcastTE
 opar_unicastTE
 opar_notunicastTE
 opar_sendTE
 opar_deliverTE
 opar_receiveTE

lemma oparp_sos_cases [elim]:

assumes " $((\sigma, (s, t)), a, (\sigma', (s', t')))) \in \text{oparp_sos } i \ S \ T$ "
 and " $\llbracket ((\sigma, s), a, (\sigma', s')) \in S; \bigwedge m. a \neq \text{receive } m; t' = t \rrbracket \implies P$ "
 and " $\llbracket (t, a, t') \in T; \bigwedge m. a \neq \text{send } m; s' = s; \sigma' \ i = \sigma \ i \rrbracket \implies P$ "
 and " $\bigwedge m. \llbracket a = \tau; ((\sigma, s), \text{receive } m, (\sigma', s')) \in S; (t, \text{send } m, t') \in T \rrbracket \implies P$ "
 shows "P"
 <proof>

definition extg :: " $'a \times 'b) \times 'c \Rightarrow 'a \times 'b \times 'c$ "
 where "extg $\equiv \lambda((\sigma, l1), l2). (\sigma, (l1, l2))$ "

lemma extgsimp [simp]:

"extg $((\sigma, l1), l2) = (\sigma, (l1, l2))$ "
 <proof>

lemma extg_range_prod: "extg ' $(i1 \times i2) = \{(\sigma, (s1, s2)) \mid \sigma \ s1 \ s2. (\sigma, s1) \in i1 \wedge s2 \in i2\}$ "
 <proof>

definition

opar_comp :: " $((ip \Rightarrow 's) \times 's1, 'm \ \text{seq_action}) \ \text{automaton}$
 $\Rightarrow ip$
 $\Rightarrow ('s2, 'm \ \text{seq_action}) \ \text{automaton}$
 $\Rightarrow ((ip \Rightarrow 's) \times 's1 \times 's2, 'm \ \text{seq_action}) \ \text{automaton}$ "
 (" $_ \llbracket _ \rrbracket$ " [102, 0, 103] 102)

where

"s $\llbracket_i \ t \equiv \llbracket \text{init} = \text{extg ' } (\text{init } s \times \text{init } t), \text{trans} = \text{oparp_sos } i \ (\text{trans } s) \ (\text{trans } t) \rrbracket$ "

lemma opar_comp_def':

"s $\llbracket_i \ t = \llbracket \text{init} = \{(\sigma, (s_l, t_l)) \mid \sigma \ s_l \ t_l. (\sigma, s_l) \in \text{init } s \wedge t_l \in \text{init } t\},$
 $\text{trans} = \text{oparp_sos } i \ (\text{trans } s) \ (\text{trans } t) \rrbracket$ "
 <proof>

lemma trans_opar_comp [simp]:

"trans $(s \llbracket_i \ t) = \text{oparp_sos } i \ (\text{trans } s) \ (\text{trans } t)$ "
 <proof>

lemma init_opar_comp [simp]:

"init $(s \llbracket_i \ t) = \text{extg ' } (\text{init } s \times \text{init } t)$ "
 <proof>

13.3 Open structural operational semantics for node expressions

inductive_set

onode_sos :: " $((ip \Rightarrow 's) \times 'l, 'm \ \text{seq_action}) \ \text{transition set}$
 $\Rightarrow ((ip \Rightarrow 's) \times 'l \ \text{net_state}, 'm \ \text{node_action}) \ \text{transition set}$ "
 for S :: " $((ip \Rightarrow 's) \times 'l, 'm \ \text{seq_action}) \ \text{transition set}$ "

where

onode_bcast:

" $((\sigma, s), \text{broadcast } m, (\sigma', s')) \in S \implies ((\sigma, \text{NodeS } i \ s \ R), R:*cast(m), (\sigma', \text{NodeS } i \ s' \ R)) \in \text{onode_sos } S$ "

| onode_gcast:

" $((\sigma, s), \text{groupcast } D \ m, (\sigma', s')) \in S \implies ((\sigma, \text{NodeS } i \ s \ R), (R \cap D):*cast(m), (\sigma', \text{NodeS } i \ s' \ R))$

∈ onode_sos S"

| onode_ucast:

"[((σ, s), unicast d m, (σ', s')) ∈ S; d ∈ R] ⇒ ((σ, NodeS i s R), {d}:*cast(m), (σ', NodeS i s' R)) ∈ onode_sos S"

| onode_notucast: "[((σ, s), ¬unicast d, (σ', s')) ∈ S; d ∉ R; ∀j. j ≠ i → σ' j = σ j]
⇒ ((σ, NodeS i s R), τ, (σ', NodeS i s' R)) ∈ onode_sos S"

| onode_deliver: "[((σ, s), deliver d, (σ', s')) ∈ S; ∀j. j ≠ i → σ' j = σ j]
⇒ ((σ, NodeS i s R), i:deliver(d), (σ', NodeS i s' R)) ∈ onode_sos S"

| onode_tau: "[((σ, s), τ, (σ', s')) ∈ S; ∀j. j ≠ i → σ' j = σ j]
⇒ ((σ, NodeS i s R), τ, (σ', NodeS i s' R)) ∈ onode_sos S"

| onode_receive:

"((σ, s), receive m, (σ', s')) ∈ S ⇒ ((σ, NodeS i s R), {i}¬{j}:arrive(m), (σ', NodeS i s' R)) ∈ onode_sos S"

| onode_arrive:

"σ' i = σ i ⇒ ((σ, NodeS i s R), {j}¬{i}:arrive(m), (σ', NodeS i s R)) ∈ onode_sos S"

| onode_connect1:

"σ' i = σ i ⇒ ((σ, NodeS i s R), connect(i, i'), (σ', NodeS i s (R ∪ {i'}))) ∈ onode_sos S"

| onode_connect2:

"σ' i = σ i ⇒ ((σ, NodeS i s R), connect(i', i), (σ', NodeS i s (R ∪ {i'}))) ∈ onode_sos S"

| onode_disconnect1:

"σ' i = σ i ⇒ ((σ, NodeS i s R), disconnect(i, i'), (σ', NodeS i s (R - {i'}))) ∈ onode_sos S"

| onode_disconnect2:

"σ' i = σ i ⇒ ((σ, NodeS i s R), disconnect(i', i), (σ', NodeS i s (R - {i'}))) ∈ onode_sos S"

| onode_connect_other:

"[i ≠ i'; i ≠ i''; σ' i = σ i] ⇒ ((σ, NodeS i s R), connect(i', i''), (σ', NodeS i s R)) ∈ onode_sos S"

| onode_disconnect_other:

"[i ≠ i'; i ≠ i''; σ' i = σ i] ⇒ ((σ, NodeS i s R), disconnect(i', i''), (σ', NodeS i s R)) ∈ onode_sos S"

inductive_cases

onode_arriveTE [elim]: "((σ, NodeS i s R), ii¬ni:arrive(m), (σ', NodeS i' s' R')) ∈ onode_sos S"
and onode_castTE [elim]: "((σ, NodeS i s R), RR:*cast(m), (σ', NodeS i' s' R')) ∈ onode_sos S"
and onode_deliverTE [elim]: "((σ, NodeS i s R), ii:deliver(d), (σ', NodeS i' s' R')) ∈ onode_sos S"
and onode_connectTE [elim]: "((σ, NodeS i s R), connect(ii, ii'), (σ', NodeS i' s' R')) ∈ onode_sos S"
and onode_disconnectTE [elim]: "((σ, NodeS i s R), disconnect(ii, ii'), (σ', NodeS i' s' R')) ∈ onode_sos S"
and onode_newpktTE [elim]: "((σ, NodeS i s R), ii:newpkt(d, di), (σ', NodeS i' s' R')) ∈ onode_sos S"
and onode_tauTE [elim]: "((σ, NodeS i s R), τ, (σ', NodeS i' s' R')) ∈ onode_sos S"

lemma oarrives_or_not:

assumes "((σ, NodeS i s R), ii¬ni:arrive(m), (σ', NodeS i' s' R')) ∈ onode_sos S"

shows "(ii = {i} ∧ ni = {}) ∨ (ii = {} ∧ ni = {i})"

{proof}

definition

```

onode_comp :: "ip
  ⇒ ((ip ⇒ 's) × 'l, 'm seq_action) automaton
  ⇒ ip set
  ⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) automaton"
("⟨_ : (_) : _⟩_o)" [0, 0, 0] 104

```

where

```

"⟨i : onp : R_i⟩_o ≡ (| init = {(σ, NodeS i s R_i) | σ s. (σ, s) ∈ init onp},
  trans = onode_sos (trans onp) |)"

```

lemma trans_onode_comp:

```

"trans (⟨i : S : R⟩_o) = onode_sos (trans S)"
⟨proof⟩

```

lemma init_onode_comp:

```

"init (⟨i : S : R⟩_o) = {(σ, NodeS i s R) | σ s. (σ, s) ∈ init S}"
⟨proof⟩

```

lemmas onode_comps = trans_onode_comp init_onode_comp

lemma fst_par_onode_comp [simp]:

```

"trans (⟨i : s ⟨⟨I t : R⟩_o⟩ = onode_sos (oparp_sos I (trans s) (trans t)))"
⟨proof⟩

```

lemma init_par_onode_comp [simp]:

```

"init (⟨i : s ⟨⟨I t : R⟩_o⟩ = {(σ, NodeS i (s1, s2) R) | σ s1 s2. ((σ, s1), s2) ∈ init s × init t}"
⟨proof⟩

```

lemma onode_sos_dest_is_net_state:

```

assumes "(⟨σ, p⟩, a, s') ∈ onode_sos S"
shows "∃σ' i' ζ' R'. s' = (σ', NodeS i' ζ' R)"
⟨proof⟩

```

lemma onode_sos_dest_is_net_state':

```

assumes "(⟨σ, NodeS i p R⟩, a, s') ∈ onode_sos S"
shows "∃σ' ζ' R'. s' = (σ', NodeS i ζ' R)"
⟨proof⟩

```

lemma onode_sos_dest_is_net_state'':

```

assumes "(⟨σ, NodeS i p R⟩, a, (σ', s')) ∈ onode_sos S"
shows "∃ζ' R'. s' = NodeS i ζ' R"
⟨proof⟩

```

lemma onode_sos_src_is_net_state:

```

assumes "(⟨σ, p⟩, a, s') ∈ onode_sos S"
shows "∃i ζ R. p = NodeS i ζ R"
⟨proof⟩

```

lemma onode_sos_net_states:

```

assumes "(⟨σ, s⟩, a, (σ', s')) ∈ onode_sos S"
shows "∃i ζ R ζ' R'. s = NodeS i ζ R ∧ s' = NodeS i ζ' R'"
⟨proof⟩

```

lemma node_sos_cases [elim]:

```

"((σ, NodeS i p R), a, (σ', NodeS i p' R')) ∈ onode_sos S ⇒
(∧m .      [ a = R:*cast(m);          R' = R; ((σ, p), broadcast m, (σ', p')) ∈ S ] ⇒ P) ⇒
(∧m D.     [ a = (R ∩ D):*cast(m);    R' = R; ((σ, p), groupcast D m, (σ', p')) ∈ S ] ⇒ P) ⇒
(∧d m.     [ a = {d}:*cast(m);        R' = R; ((σ, p), unicast d m, (σ', p')) ∈ S; d ∈ R ] ⇒ P)
⇒
(∧d.       [ a = τ;                    R' = R; ((σ, p), ¬unicast d, (σ', p')) ∈ S; d ∉ R ] ⇒ P)
⇒
(∧d.       [ a = i:deliver(d);         R' = R; ((σ, p), deliver d, (σ', p')) ∈ S ] ⇒ P) ⇒
(∧m.       [ a = {i}¬{j}:arrive(m);    R' = R; ((σ, p), receive m, (σ', p')) ∈ S ] ⇒ P) ⇒
(          [ a = τ;                    R' = R; ((σ, p), τ, (σ', p')) ∈ S ] ⇒ P) ⇒

```

```

(∧m.    [ a = { }¬{i}:arrive(m);    R' = R; p = p'; σ' i = σ i ] ⇒ P) ⇒
(∧i i'.  [ a = connect(i, i');        R' = R ∪ {i'}; p = p'; σ' i = σ i ] ⇒ P) ⇒
(∧i i'.  [ a = connect(i', i);        R' = R ∪ {i'}; p = p'; σ' i = σ i ] ⇒ P) ⇒
(∧i i'.  [ a = disconnect(i, i');     R' = R - {i'}; p = p'; σ' i = σ i ] ⇒ P) ⇒
(∧i i'.  [ a = disconnect(i', i);     R' = R - {i'}; p = p'; σ' i = σ i ] ⇒ P) ⇒
(∧i i' i''. [ a = connect(i', i'');   R' = R; p = p'; i ≠ i'; i ≠ i''; σ' i = σ i ] ⇒ P) ⇒
(∧i i' i''. [ a = disconnect(i', i''); R' = R; p = p'; i ≠ i'; i ≠ i''; σ' i = σ i ] ⇒ P) ⇒
P"
⟨proof⟩

```

13.4 Open structural operational semantics for partial network expressions

inductive_set

```

opnet_sos :: "(ip ⇒ 's) × 'l net_state, 'm node_action) transition set
            ⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) transition set
            ⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) transition set"

```

```
for S :: "(ip ⇒ 's) × 'l net_state, 'm node_action) transition set"
```

```
and T :: "(ip ⇒ 's) × 'l net_state, 'm node_action) transition set"
```

where

```
opnet_cast1:
```

```

" [ ((σ, s), R:*cast(m), (σ', s')) ∈ S; ((σ, t), H¬K:arrive(m), (σ', t')) ∈ T; H ⊆ R; K ∩ R = {} ]
  ⇒ ((σ, SubnetS s t), R:*cast(m), (σ', SubnetS s' t')) ∈ opnet_sos S T"

```

```
/ opnet_cast2:
```

```

" [ ((σ, s), H¬K:arrive(m), (σ', s')) ∈ S; ((σ, t), R:*cast(m), (σ', t')) ∈ T; H ⊆ R; K ∩ R = {} ]
  ⇒ ((σ, SubnetS s t), R:*cast(m), (σ', SubnetS s' t')) ∈ opnet_sos S T"

```

```
/ opnet_arrive:
```

```

" [ ((σ, s), H¬K:arrive(m), (σ', s')) ∈ S; ((σ, t), H'¬K':arrive(m), (σ', t')) ∈ T ]
  ⇒ ((σ, SubnetS s t), (H ∪ H')¬(K ∪ K'):arrive(m), (σ', SubnetS s' t')) ∈ opnet_sos S T"

```

```
/ opnet_deliver1:
```

```

"((σ, s), i:deliver(d), (σ', s')) ∈ S
  ⇒ ((σ, SubnetS s t), i:deliver(d), (σ', SubnetS s' t)) ∈ opnet_sos S T"

```

```
/ opnet_deliver2:
```

```

" [ ((σ, t), i:deliver(d), (σ', t')) ∈ T ]
  ⇒ ((σ, SubnetS s t), i:deliver(d), (σ', SubnetS s t')) ∈ opnet_sos S T"

```

```
/ opnet_tau1:
```

```

"((σ, s), τ, (σ', s')) ∈ S ⇒ ((σ, SubnetS s t), τ, (σ', SubnetS s' t)) ∈ opnet_sos S T"

```

```
/ opnet_tau2:
```

```

"((σ, t), τ, (σ', t')) ∈ T ⇒ ((σ, SubnetS s t), τ, (σ', SubnetS s t')) ∈ opnet_sos S T"

```

```
/ opnet_connect:
```

```

" [ ((σ, s), connect(i, i'), (σ', s')) ∈ S; ((σ, t), connect(i, i'), (σ', t')) ∈ T ]
  ⇒ ((σ, SubnetS s t), connect(i, i'), (σ', SubnetS s' t')) ∈ opnet_sos S T"

```

```
/ opnet_disconnect:
```

```

" [ ((σ, s), disconnect(i, i'), (σ', s')) ∈ S; ((σ, t), disconnect(i, i'), (σ', t')) ∈ T ]
  ⇒ ((σ, SubnetS s t), disconnect(i, i'), (σ', SubnetS s' t')) ∈ opnet_sos S T"

```

```

inductive_cases opartial_castTE [elim]:      "((σ, s), R:*cast(m), (σ', s')) ∈ opnet_sos S T"
and opartial_arriveTE [elim]:               "((σ, s), H¬K:arrive(m), (σ', s')) ∈ opnet_sos S T"
and opartial_deliverTE [elim]:             "((σ, s), i:deliver(d), (σ', s')) ∈ opnet_sos S T"
and opartial_tauTE [elim]:                 "((σ, s), τ, (σ', s')) ∈ opnet_sos S T"
and opartial_connectTE [elim]:             "((σ, s), connect(i, i'), (σ', s')) ∈ opnet_sos S T"
and opartial_disconnectTE [elim]:         "((σ, s), disconnect(i, i'), (σ', s')) ∈ opnet_sos S T"
and opartial_newpktTE [elim]:             "((σ, s), i:newpkt(d, di), (σ', s')) ∈ opnet_sos S T"

```

```
fun opnet :: "(ip ⇒ ((ip ⇒ 's) × 'l, 'm seq_action) automaton)"
```

$\Rightarrow \text{net_tree} \Rightarrow ((\text{ip} \Rightarrow 's) \times 'l \text{ net_state}, 'm \text{ node_action}) \text{ automaton''}$

where

```
"opnet onp ⟨i; Ri⟩ = ⟨i : onp i : Ri⟩o"
| "opnet onp (p1 || p2) = ( [ init = { (σ, SubnetS s1 s2) | σ s1 s2.
    (σ, s1) ∈ init (opnet onp p1)
    ∧ (σ, s2) ∈ init (opnet onp p2)
    ∧ net_ips s1 ∩ net_ips s2 = {} },
  trans = opnet_sos (trans (opnet onp p1)) (trans (opnet onp p2)) ] )"
```

lemma opnet_node_init [elim, simp]:

```
assumes "(σ, s) ∈ init (opnet onp ⟨i; R⟩)"
shows "(σ, s) ∈ { (σ, NodeS i ns R) | σ ns. (σ, ns) ∈ init (onp i) }"
⟨proof⟩
```

lemma opnet_node_init' [elim]:

```
assumes "(σ, s) ∈ init (opnet onp ⟨i; R⟩)"
obtains ns where "s = NodeS i ns R"
and "(σ, ns) ∈ init (onp i)"
⟨proof⟩
```

lemma opnet_node_trans [elim, simp]:

```
assumes "(s, a, s') ∈ trans (opnet onp ⟨i; R⟩)"
shows "(s, a, s') ∈ onode_sos (trans (onp i))"
⟨proof⟩
```

13.5 Open structural operational semantics for complete network expressions

inductive_set

```
ocnet_sos :: "((ip ⇒ 's) × 'l net_state, 'm::msg node_action) transition set
⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) transition set"
for S :: "((ip ⇒ 's) × 'l net_state, 'm node_action) transition set"
```

where

ocnet_connect:

```
"[ ((σ, s), connect(i, i'), (σ', s')) ∈ S; ∀j. j ∉ net_ips s → (σ' j = σ j) ]
⇒ ((σ, s), connect(i, i'), (σ', s')) ∈ ocnet_sos S"
```

| ocnet_disconnect:

```
"[ ((σ, s), disconnect(i, i'), (σ', s')) ∈ S; ∀j. j ∉ net_ips s → (σ' j = σ j) ]
⇒ ((σ, s), disconnect(i, i'), (σ', s')) ∈ ocnet_sos S"
```

| ocnet_cast:

```
"[ ((σ, s), R:*cast(m), (σ', s')) ∈ S; ∀j. j ∉ net_ips s → (σ' j = σ j) ]
⇒ ((σ, s), τ, (σ', s')) ∈ ocnet_sos S"
```

| ocnet_tau:

```
"[ ((σ, s), τ, (σ', s')) ∈ S; ∀j. j ∉ net_ips s → (σ' j = σ j) ]
⇒ ((σ, s), τ, (σ', s')) ∈ ocnet_sos S"
```

| ocnet_deliver:

```
"[ ((σ, s), i:deliver(d), (σ', s')) ∈ S; ∀j. j ∉ net_ips s → (σ' j = σ j) ]
⇒ ((σ, s), i:deliver(d), (σ', s')) ∈ ocnet_sos S"
```

| ocnet_newpkt:

```
"[ ((σ, s), {i}¬K:arrive(newpkt(d, di)), (σ', s')) ∈ S; ∀j. j ∉ net_ips s → (σ' j = σ j) ]
⇒ ((σ, s), i:newpkt(d, di), (σ', s')) ∈ ocnet_sos S"
```

inductive_cases oconnect_completeTE: "((σ, s), connect(i, i'), (σ', s')) ∈ ocnet_sos S"

and odisconnect_completeTE: "((σ, s), disconnect(i, i'), (σ', s')) ∈ ocnet_sos S"

and otau_completeTE: "((σ, s), τ, (σ', s')) ∈ ocnet_sos S"

and odeliver_completeTE: "((σ, s), i:deliver(d), (σ', s')) ∈ ocnet_sos S"

and onewpkt_completeTE: "((σ, s), i:newpkt(d, di), (σ', s')) ∈ ocnet_sos S"

lemmas ocompleteTEs = oconnect_completeTE

odisconnect_completeTE


```

otau_completeTE
odeliver_completeTE
onewpkt_completeTE

```

```

lemma ocomplete_no_cast [simp]:
  "(( $\sigma$ , s), R:*cast(m), ( $\sigma'$ , s'))  $\notin$  ocnet_sos T"
  <proof>

```

```

lemma ocomplete_no_arrive [simp]:
  "(( $\sigma$ , s), ii $\neg$ ni:arrive(m), ( $\sigma'$ , s'))  $\notin$  ocnet_sos T"
  <proof>

```

```

lemma ocomplete_no_change [elim]:
  assumes "(( $\sigma$ , s), a, ( $\sigma'$ , s'))  $\in$  ocnet_sos T"
    and "j  $\notin$  net_ips s"
  shows " $\sigma'$  j =  $\sigma$  j"
  <proof>

```

```

lemma ocomplete_transE [elim]:
  assumes "(( $\sigma$ ,  $\zeta$ ), a, ( $\sigma'$ ,  $\zeta'$ ))  $\in$  ocnet_sos (trans (opnet onp n))"
  obtains a' where "(( $\sigma$ ,  $\zeta$ ), a', ( $\sigma'$ ,  $\zeta'$ ))  $\in$  trans (opnet onp n)"
  <proof>

```

```

abbreviation
  oclosed :: "((ip  $\Rightarrow$  's)  $\times$  'l net_state, ('m::msg) node_action) automaton
             $\Rightarrow$  ((ip  $\Rightarrow$  's)  $\times$  'l net_state, 'm node_action) automaton"

```

```

where
  "oclosed  $\equiv$  ( $\lambda$ A. A ( $\mid$  trans := ocnet_sos (trans A)  $\mid$ ))"

```

end

14 Configure the inv-cterms tactic for open sequential processes

```

theory OAWN_SOS_Labels
imports OAWN_SOS Inv_Cterms
begin

```

```

lemma oelinder_guard:
  assumes "p = {l}<fg> qq"
    and "l'  $\in$  labels  $\Gamma$  q"
    and "(( $\sigma$ , p), a, ( $\sigma'$ , q))  $\in$  oseqp_sos  $\Gamma$  i"
  obtains p' where "p = {l}<fg> p'"
    and "l'  $\in$  labels  $\Gamma$  qq"
  <proof>

```

```

lemma oelinder_assign:
  assumes "p = {l}[[fa]] qq"
    and "l'  $\in$  labels  $\Gamma$  q"
    and "(( $\sigma$ , p), a, ( $\sigma'$ , q))  $\in$  oseqp_sos  $\Gamma$  i"
  obtains p' where "p = {l}[[fa]] p'"
    and "l'  $\in$  labels  $\Gamma$  qq"
  <proof>

```

```

lemma oelinder_ucast:
  assumes "p = {l}unicast(fip, fmsg).q1  $\triangleright$  q2"
    and "l'  $\in$  labels  $\Gamma$  q"
    and "(( $\sigma$ , p), a, ( $\sigma'$ , q))  $\in$  oseqp_sos  $\Gamma$  i"
  obtains p' pp' where "p = {l}unicast(fip, fmsg).p'  $\triangleright$  pp'"
    and "case a of unicast _ _  $\Rightarrow$  l'  $\in$  labels  $\Gamma$  q1
      | _  $\Rightarrow$  l'  $\in$  labels  $\Gamma$  q2"
  <proof>

```

```

lemma oelinder_bcast:
  assumes "p = {l}broadcast(fmsg).qq"

```

```

    and "l' ∈ labels Γ q"
    and "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
obtains p' where "p = {l}broadcast(fmsg). p'"
    and "l' ∈ labels Γ qq"
⟨proof⟩

```

```

lemma oelimder_gcast:
  assumes "p = {l}groupcast(fips, fmsg).qq"
    and "l' ∈ labels Γ q"
    and "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
  obtains p' where "p = {l}groupcast(fips, fmsg). p'"
    and "l' ∈ labels Γ qq"
⟨proof⟩

```

```

lemma oelimder_send:
  assumes "p = {l}send(fmsg).qq"
    and "l' ∈ labels Γ q"
    and "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
  obtains p' where "p = {l}send(fmsg). p'"
    and "l' ∈ labels Γ qq"
⟨proof⟩

```

```

lemma oelimder_deliver:
  assumes "p = {l}deliver(fdata).qq"
    and "l' ∈ labels Γ q"
    and "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
  obtains p' where "p = {l}deliver(fdata).p'"
    and "l' ∈ labels Γ qq"
⟨proof⟩

```

```

lemma oelimder_receive:
  assumes "p = {l}receive(fmsg).qq"
    and "l' ∈ labels Γ q"
    and "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
  obtains p' where "p = {l}receive(fmsg).p'"
    and "l' ∈ labels Γ qq"
⟨proof⟩

```

```

lemmas oelimders =
  oelimder_guard
  oelimder_assign
  oelimder_ucast
  oelimder_bcast
  oelimder_gcast
  oelimder_send
  oelimder_deliver
  oelimder_receive

```

```

declare
  oseqpTEs [cterms_seqte]
  oelimders [cterms_elimders]

```

end

15 Lemmas for open partial networks

```

theory OPnet
imports OAWN_SOS OInvariants
begin

```

These lemmas mostly concern the preservation of node structure by `opnet_sos` transitions.

```

lemma opnet_maintains_dom:
  assumes "((σ, ns), a, (σ', ns')) ∈ trans (opnet np p)"
  shows "net_ips ns = net_ips ns'"

```

⟨proof⟩

lemma *opnet_net_ips_net_tree_ips*:
 assumes " $(\sigma, ns) \in \text{oreachable} (\text{opnet } np \ p) \ S \ U$ "
 shows " $\text{net_ips } ns = \text{net_tree_ips } p$ "
 ⟨proof⟩

lemma *opnet_net_ips_net_tree_ips_init*:
 assumes " $(\sigma, ns) \in \text{init} (\text{opnet } np \ p)$ "
 shows " $\text{net_ips } ns = \text{net_tree_ips } p$ "
 ⟨proof⟩

lemma *opartial_net_preserves_subnets*:
 assumes " $((\sigma, \text{SubnetS } s \ t), a, (\sigma', st')) \in \text{opnet_sos} (\text{trans} (\text{opnet } np \ p1)) (\text{trans} (\text{opnet } np \ p2))$ "
 shows " $\exists s' \ t'. st' = \text{SubnetS } s' \ t'$ "
 ⟨proof⟩

lemma *net_par_oreachable_is_subnet*:
 assumes " $(\sigma, st) \in \text{oreachable} (\text{opnet } np \ (p1 \parallel p2)) \ S \ U$ "
 shows " $\exists s \ t. st = \text{SubnetS } s \ t$ "
 ⟨proof⟩

end

16 Lifting rules for (open) nodes

theory *ONode_Lifting*
imports *AWN OAWN_SOS OInvariants*
begin

lemma *node_net_state'*:
 assumes " $s \in \text{oreachable} (\langle i : T : R_i \rangle_o) \ S \ U$ "
 shows " $\exists \sigma \ \zeta \ R. s = (\sigma, \text{NodeS } i \ \zeta \ R)$ "
 ⟨proof⟩

lemma *node_net_state*:
 assumes " $(\sigma, s) \in \text{oreachable} (\langle i : T : R_i \rangle_o) \ S \ U$ "
 shows " $\exists \zeta \ R. s = \text{NodeS } i \ \zeta \ R$ "
 ⟨proof⟩

lemma *node_net_state_trans [elim]*:
 assumes *sor*: " $(\sigma, s) \in \text{oreachable} (\langle i : \zeta_i : R_i \rangle_o) \ S \ U$ "
 and *str*: " $((\sigma, s), a, (\sigma', s')) \in \text{trans} (\langle i : \zeta_i : R_i \rangle_o)$ "
 obtains $\zeta \ R \ \zeta' \ R'$
 where " $s = \text{NodeS } i \ \zeta \ R$ "
 and " $s' = \text{NodeS } i \ \zeta' \ R'$ "
 ⟨proof⟩

lemma *nodemap_induct' [consumes, case_names init other local]*:
 assumes " $(\sigma, \text{NodeS } ii \ \zeta \ R) \in \text{oreachable} (\langle ii : T : R_i \rangle_o) \ S \ U$ "
 and *init*: " $\bigwedge \sigma \ \zeta. (\sigma, \text{NodeS } ii \ \zeta \ R_i) \in \text{init} (\langle ii : T : R_i \rangle_o) \implies P (\sigma, \text{NodeS } ii \ \zeta \ R_i)$ "
 and *other*: " $\bigwedge \sigma \ \zeta \ R \ \sigma' \ a.$
 $\llbracket (\sigma, \text{NodeS } ii \ \zeta \ R) \in \text{oreachable} (\langle ii : T : R_i \rangle_o) \ S \ U;$
 $U \ \sigma \ \sigma'; P (\sigma, \text{NodeS } ii \ \zeta \ R) \rrbracket \implies P (\sigma', \text{NodeS } ii \ \zeta \ R)$ "
 and *local*: " $\bigwedge \sigma \ \zeta \ R \ \sigma' \ \zeta' \ R' \ a.$
 $\llbracket (\sigma, \text{NodeS } ii \ \zeta \ R) \in \text{oreachable} (\langle ii : T : R_i \rangle_o) \ S \ U;$
 $((\sigma, \text{NodeS } ii \ \zeta \ R), a, (\sigma', \text{NodeS } ii \ \zeta' \ R')) \in \text{trans} (\langle ii : T : R_i \rangle_o);$
 $S \ \sigma \ \sigma' \ a; P (\sigma, \text{NodeS } ii \ \zeta \ R) \rrbracket \implies P (\sigma', \text{NodeS } ii \ \zeta' \ R')$ "
 shows " $P (\sigma, \text{NodeS } ii \ \zeta \ R)$ "
 ⟨proof⟩

lemma *nodemap_induct [consumes, case_names init step]*:
 assumes " $(\sigma, \text{NodeS } ii \ \zeta \ R) \in \text{oreachable} (\langle ii : T : R_i \rangle_o) \ S \ U$ "
 and *init*: " $\bigwedge \sigma \ \zeta. (\sigma, \text{NodeS } ii \ \zeta \ R_i) \in \text{init} (\langle ii : T : R_i \rangle_o) \implies P \ \sigma \ \zeta \ R_i$ "

and other: " $\bigwedge \sigma \zeta R \sigma' a.$
 $\llbracket (\sigma, \text{NodeS } ii \zeta R) \in \text{oreachable } (\langle ii : T : R_i \rangle_o) S U;$
 $U \sigma \sigma'; P \sigma \zeta R \rrbracket \implies P \sigma' \zeta R"$

and local: " $\bigwedge \sigma \zeta R \sigma' \zeta' R' a.$
 $\llbracket (\sigma, \text{NodeS } ii \zeta R) \in \text{oreachable } (\langle ii : T : R_i \rangle_o) S U;$
 $((\sigma, \text{NodeS } ii \zeta R), a, (\sigma', \text{NodeS } ii \zeta' R')) \in \text{trans } (\langle ii : T : R_i \rangle_o);$
 $S \sigma \sigma' a; P \sigma \zeta R \rrbracket \implies P \sigma' \zeta' R'"$

shows " $P \sigma \zeta R"$
 $\langle \text{proof} \rangle$

lemma node_addressD [dest, simp]:
assumes " $(\sigma, \text{NodeS } i \zeta R) \in \text{oreachable } (\langle ii : T : R_i \rangle_o) S U"$
shows " $i = ii"$
 $\langle \text{proof} \rangle$

lemma node_proc_reachable [dest]:
assumes " $(\sigma, \text{NodeS } i \zeta R) \in \text{oreachable } (\langle ii : T : R_i \rangle_o)$
 $(\text{otherwith } S \{ii\} (\text{oarrivemsg } I)) (\text{other } U \{ii\})"$
and sgivesu: " $\bigwedge \xi \xi'. S \xi \xi' \implies U \xi \xi'"$
shows " $(\sigma, \zeta) \in \text{oreachable } T (\text{otherwith } S \{ii\} (\text{orecvmsg } I)) (\text{other } U \{ii\})"$
 $\langle \text{proof} \rangle$

lemma node_proc_reachable_statelessassm [dest]:
assumes " $(\sigma, \text{NodeS } i \zeta R) \in \text{oreachable } (\langle ii : T : R_i \rangle_o)$
 $(\text{otherwith } (\lambda_ _ . \text{True}) \{ii\} (\text{oarrivemsg } I))$
 $(\text{other } (\lambda_ _ . \text{True}) \{ii\})"$
shows " $(\sigma, \zeta) \in \text{oreachable } T$
 $(\text{otherwith } (\lambda_ _ . \text{True}) \{ii\} (\text{orecvmsg } I)) (\text{other } (\lambda_ _ . \text{True}) \{ii\})"$
 $\langle \text{proof} \rangle$

lemma node_lift:
assumes " $T \models (\text{otherwith } S \{ii\} (\text{orecvmsg } I), \text{other } U \{ii\} \rightarrow) \text{global } P"$
and " $\bigwedge \xi \xi'. S \xi \xi' \implies U \xi \xi'"$
shows " $\langle ii : T : R_i \rangle_o \models (\text{otherwith } S \{ii\} (\text{oarrivemsg } I), \text{other } U \{ii\} \rightarrow) \text{global } P"$
 $\langle \text{proof} \rangle$

lemma node_lift_step [intro]:
assumes $\text{pinv: } "T \models_A (\text{otherwith } S \{i\} (\text{orecvmsg } I), \text{other } U \{i\} \rightarrow) \text{globala } (\lambda(\sigma, _, \sigma'). Q \sigma \sigma)'"$
and other: " $\bigwedge \sigma \sigma'. \text{other } U \{i\} \sigma \sigma' \implies Q \sigma \sigma'"$
and sgivesu: " $\bigwedge \xi \xi'. S \xi \xi' \implies U \xi \xi'"$
shows " $\langle i : T : R_i \rangle_o \models_A (\text{otherwith } S \{i\} (\text{oarrivemsg } I), \text{other } U \{i\} \rightarrow)$
 $\text{globala } (\lambda(\sigma, _, \sigma'). Q \sigma \sigma)'"$
 $(\text{is } _ \models_A (?S, ?U \rightarrow) _)"$
 $\langle \text{proof} \rangle$

lemma node_lift_step_statelessassm [intro]:
assumes " $T \models_A (\lambda _ . \text{orecvmsg } I \sigma, \text{other } (\lambda _ _ . \text{True}) \{i\} \rightarrow)$
 $\text{globala } (\lambda(\sigma, _, \sigma'). Q (\sigma \ i) (\sigma' \ i))"$
and " $\bigwedge \xi. Q \xi \xi"$
shows " $\langle i : T : R_i \rangle_o \models_A (\lambda _ . \text{oarrivemsg } I \sigma, \text{other } (\lambda _ _ . \text{True}) \{i\} \rightarrow)$
 $\text{globala } (\lambda(\sigma, _, \sigma'). Q (\sigma \ i) (\sigma' \ i))"$
 $\langle \text{proof} \rangle$

lemma node_lift_anycast [intro]:
assumes $\text{pinv: } "T \models_A (\text{otherwith } S \{i\} (\text{orecvmsg } I), \text{other } U \{i\} \rightarrow)$
 $\text{globala } (\lambda(\sigma, a, \sigma'). \text{anycast } (Q \sigma \sigma') a)"$
and " $\bigwedge \xi \xi'. S \xi \xi' \implies U \xi \xi'"$
shows " $\langle i : T : R_i \rangle_o \models_A (\text{otherwith } S \{i\} (\text{oarrivemsg } I), \text{other } U \{i\} \rightarrow)$
 $\text{globala } (\lambda(\sigma, a, \sigma'). \text{castmsg } (Q \sigma \sigma') a)"$
 $(\text{is } _ \models_A (?S, ?U \rightarrow) _)"$
 $\langle \text{proof} \rangle$

lemma node_lift_anycast_statelessassm [intro]:
assumes $\text{pinv: } "T \models_A (\lambda _ . \text{orecvmsg } I \sigma, \text{other } (\lambda _ _ . \text{True}) \{i\} \rightarrow)"$

```

      globala (λ(σ, a, σ'). anycast (Q σ σ') a)"
shows "⟨i : T : R_i⟩_o ⊨_A (λσ -. oarrivemsg I σ, other (λ_ -. True) {i} →)
      globala (λ(σ, a, σ'). castmsg (Q σ σ') a)"
(is "_ ⊨_A (?S, _ →) _")
⟨proof⟩

```

lemma node_local_deliver:

```

"⟨i : ζ_i : R_i⟩_o ⊨_A (S, U →) globala (λ(_, a, _). ∀j. j≠i → (∀d. a ≠ j:deliver(d)))"
⟨proof⟩

```

lemma node_tau_deliver_unchanged:

```

"⟨i : ζ_i : R_i⟩_o ⊨_A (S, U →) globala (λ(σ, a, σ'). a = τ ∨ (∃i d. a = i:deliver(d))
      → (∀j. j≠i → σ' j = σ j))"
⟨proof⟩

```

end

17 Lifting rules for (open) partial networks

theory OPnet_Lifting

imports ONode_Lifting OAWN_SOS OPnet

begin

lemma oreachable_par_subnet_induct [consumes, case_names init other local]:

```

assumes "(σ, SubnetS s t) ∈ oreachable (opnet onp (p_1 || p_2)) S U"
and init: "∧σ s t. (σ, SubnetS s t) ∈ init (opnet onp (p_1 || p_2)) ⇒ P σ s t"
and other: "∧σ s t σ'. [ (σ, SubnetS s t) ∈ oreachable (opnet onp (p_1 || p_2)) S U;
      U σ σ'; P σ s t ] ⇒ P σ' s t"
and local: "∧σ s t σ' s' t' a. [ (σ, SubnetS s t) ∈ oreachable (opnet onp (p_1 || p_2)) S U;
      ((σ, SubnetS s t), a, (σ', SubnetS s' t')) ∈ trans (opnet onp (p_1 || p_2));
      S σ σ' a; P σ s t ] ⇒ P σ' s' t'"
shows "P σ s t"
⟨proof⟩

```

lemma other_net_tree_ips_par_left:

```

assumes "other U (net_tree_ips (p_1 || p_2)) σ σ'"
and "∧ξ. U ξ ξ"
shows "other U (net_tree_ips p_1) σ σ'"
⟨proof⟩

```

lemma other_net_tree_ips_par_right:

```

assumes "other U (net_tree_ips (p_1 || p_2)) σ σ'"
and "∧ξ. U ξ ξ"
shows "other U (net_tree_ips p_2) σ σ'"
⟨proof⟩

```

lemma ostep_arrive_invariantD [elim]:

```

assumes "p ⊨_A (λσ -. oarrivemsg I σ, U →) P"
and "(σ, s) ∈ oreachable p (otherwith S IPS (oarrivemsg I)) U"
and "((σ, s), a, (σ', s')) ∈ trans p"
and "oarrivemsg I σ a"
shows "P ((σ, s), a, (σ', s'))"
⟨proof⟩

```

lemma opnet_sync_action_subnet_oreachable:

```

assumes "(σ, SubnetS s t) ∈ oreachable (opnet onp (p_1 || p_2))
      (λσ -. oarrivemsg I σ) (other U (net_tree_ips (p_1 || p_2)))"
(is "_ ∈ oreachable _ (?S (p_1 || p_2)) (?U (p_1 || p_2)))"

and "∧ξ. U ξ ξ"

and act1: "opnet onp p_1 ⊨_A (λσ -. oarrivemsg I σ, other U (net_tree_ips p_1) →)
      globala (λ(σ, a, σ'). castmsg (I σ) a
      ∧ (a = τ ∨ (∃i d. a = i:deliver(d)) →)

```

$$((\forall i \in \text{net_tree_ips } p_1. U (\sigma \ i) (\sigma' \ i)) \wedge (\forall i. i \notin \text{net_tree_ips } p_1 \longrightarrow \sigma' \ i = \sigma \ i)))"$$

and act2: "opnet onp p₂ \models_A ($\lambda \sigma _ . \text{oarrivemsg } I \ \sigma, \text{ other } U (\text{net_tree_ips } p_2) \rightarrow$
 globala ($\lambda(\sigma, a, \sigma'). \text{castmsg } (I \ \sigma) \ a$
 $\wedge (a = \tau \vee (\exists i \ d. a = i:\text{deliver}(d)) \rightarrow$
 $((\forall i \in \text{net_tree_ips } p_2. U (\sigma \ i) (\sigma' \ i))$
 $\wedge (\forall i. i \notin \text{net_tree_ips } p_2 \longrightarrow \sigma' \ i = \sigma \ i))))"$

shows " $(\sigma, s) \in \text{oreachable } (\text{opnet onp } p_1) (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma) (\text{other } U (\text{net_tree_ips } p_1))$
 $\wedge (\sigma, t) \in \text{oreachable } (\text{opnet onp } p_2) (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma) (\text{other } U (\text{net_tree_ips } p_2))$
 $\wedge \text{net_tree_ips } p_1 \cap \text{net_tree_ips } p_2 = \{\}$ "

<proof>

'Splitting' reachability is trivial when there are no assumptions on interleavings, but this is useless for showing non-trivial properties, since the interleaving steps can do anything at all. This lemma is too weak.

lemma subnet_oreachable_true_true:

assumes " $(\sigma, \text{SubnetS } s_1 \ s_2) \in \text{oreachable } (\text{opnet onp } (p_1 \parallel p_2)) (\lambda _ _ . \text{True}) (\lambda _ _ . \text{True})"$
shows " $(\sigma, s_1) \in \text{oreachable } (\text{opnet onp } p_1) (\lambda _ _ . \text{True}) (\lambda _ _ . \text{True})"$
 $(\sigma, s_2) \in \text{oreachable } (\text{opnet onp } p_2) (\lambda _ _ . \text{True}) (\lambda _ _ . \text{True})"$
 (is " $_ \in ?\text{oreachable } p_2$ ")

<proof>

It may also be tempting to try splitting from the assumption $(\sigma, \text{SubnetS } s_1 \ s_2) \in \text{oreachable } (\text{opnet onp } (p_1 \parallel p_2)) (\lambda _ _ . \text{True}) (\lambda _ _ . \text{False})$, where the environment step would be trivially true (since the assumption is false), but the lemma cannot be shown when only one side acts, since it must guarantee the assumption for the other side.

lemma lift_opnet_sync_action:

assumes " $\bigwedge \xi. U \ \xi \ \xi"$
and act1: " $\bigwedge i \ R. \langle i : \text{onp } i : R \rangle_o \models_A (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma, \text{ other } U \{i\} \rightarrow$
 globala ($\lambda(\sigma, a, _). \text{castmsg } (I \ \sigma) \ a$)"
and act2: " $\bigwedge i \ R. \langle i : \text{onp } i : R \rangle_o \models_A (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma, \text{ other } U \{i\} \rightarrow$
 globala ($\lambda(\sigma, a, \sigma'). (a \neq \tau \wedge (\forall d. a \neq i:\text{deliver}(d)) \rightarrow S (\sigma \ i) (\sigma' \ i)))"$
and act3: " $\bigwedge i \ R. \langle i : \text{onp } i : R \rangle_o \models_A (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma, \text{ other } U \{i\} \rightarrow$
 globala ($\lambda(\sigma, a, \sigma'). (a = \tau \vee (\exists d. a = i:\text{deliver}(d)) \rightarrow U (\sigma \ i) (\sigma' \ i)))"$
shows " $\text{opnet onp } p \models_A (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma, \text{ other } U (\text{net_tree_ips } p) \rightarrow$
 globala ($\lambda(\sigma, a, \sigma'). \text{castmsg } (I \ \sigma) \ a$
 $\wedge (a \neq \tau \wedge (\forall i \ d. a \neq i:\text{deliver}(d)) \rightarrow$
 $(\forall i \in \text{net_tree_ips } p. S (\sigma \ i) (\sigma' \ i)))$
 $\wedge (a = \tau \vee (\exists i \ d. a = i:\text{deliver}(d)) \rightarrow$
 $((\forall i \in \text{net_tree_ips } p. U (\sigma \ i) (\sigma' \ i))$
 $\wedge (\forall i. i \notin \text{net_tree_ips } p \longrightarrow \sigma' \ i = \sigma \ i))))"$
 (is " $\text{opnet onp } p \models_A (?I, ?U \ p \rightarrow) ?\text{inv } (\text{net_tree_ips } p)$ ")

<proof>

theorem subnet_oreachable:

assumes " $(\sigma, \text{SubnetS } s \ t) \in \text{oreachable } (\text{opnet onp } (p_1 \parallel p_2))$
 $(\text{otherwith } S (\text{net_tree_ips } (p_1 \parallel p_2)) (\text{oarrivemsg } I))$
 $(\text{other } U (\text{net_tree_ips } (p_1 \parallel p_2)))"$
 (is " $_ \in \text{oreachable } _ (?S (p_1 \parallel p_2)) (?U (p_1 \parallel p_2))"$)

and " $\bigwedge \xi. S \ \xi \ \xi"$

and " $\bigwedge \xi. U \ \xi \ \xi"$

and node1: " $\bigwedge i \ R. \langle i : \text{onp } i : R \rangle_o \models_A (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma, \text{ other } U \{i\} \rightarrow$
 globala ($\lambda(\sigma, a, _). \text{castmsg } (I \ \sigma) \ a$)"

and node2: " $\bigwedge i \ R. \langle i : \text{onp } i : R \rangle_o \models_A (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma, \text{ other } U \{i\} \rightarrow$
 globala ($\lambda(\sigma, a, \sigma'). (a \neq \tau \wedge (\forall d. a \neq i:\text{deliver}(d)) \rightarrow S (\sigma \ i) (\sigma' \ i)))"$

and node3: " $\bigwedge i \ R. \langle i : \text{onp } i : R \rangle_o \models_A (\lambda \sigma _ . \text{oarrivemsg } I \ \sigma, \text{ other } U \{i\} \rightarrow$
 globala ($\lambda(\sigma, a, \sigma'). (a = \tau \vee (\exists d. a = i:\text{deliver}(d)) \rightarrow U (\sigma \ i) (\sigma' \ i)))"$

shows " $(\sigma, s) \in \text{oreachable } (\text{opnet onp } p_1)$
 $(\text{otherwith } S (\text{net_tree_ips } p_1) (\text{oarrivemsg } I))"$

```

      (other U (net_tree_ips p1))
    ∧ (σ, t) ∈ oreachable (opnet onp p2)
      (otherwith S (net_tree_ips p2) (oarrivmsg I))
      (other U (net_tree_ips p2))
    ∧ net_tree_ips p1 ∩ net_tree_ips p2 = {}"
  ⟨proof⟩

```

```

lemmas subnet_reachable1 [dest] = subnet_reachable [THEN conjunct1, rotated 1]
lemmas subnet_reachable2 [dest] = subnet_reachable [THEN conjunct2, THEN conjunct1, rotated 1]
lemmas subnet_reachable_disjoint [dest] = subnet_reachable
      [THEN conjunct2, THEN conjunct2, rotated 1]

```

corollary pnet_lift:

```

  assumes "∧ii Ri. ⟨ii : onp ii : Ri⟩o
    ⊨ (otherwith S {ii} (oarrivmsg I), other U {ii} →) global (P ii)"

```

```

  and "∧ξ. S ξ ξ"
  and "∧ξ. U ξ ξ"

```

```

  and node1: "∧i R. ⟨i : onp i : R⟩o ⊨A (λσ _. oarrivmsg I σ, other U {i} →)
    globala (λ(σ, a, _). castmsg (I σ) a)"
  and node2: "∧i R. ⟨i : onp i : R⟩o ⊨A (λσ _. oarrivmsg I σ, other U {i} →)
    globala (λ(σ, a, σ'). (a ≠ τ ∧ (∀d. a ≠ i:deliver(d)) → S (σ i) (σ' i)))"
  and node3: "∧i R. ⟨i : onp i : R⟩o ⊨A (λσ _. oarrivmsg I σ, other U {i} →)
    globala (λ(σ, a, σ'). (a = τ ∨ (∃d. a = i:deliver(d)) → U (σ i) (σ' i)))"

```

```

  shows "opnet onp p ⊨ (otherwith S (net_tree_ips p) (oarrivmsg I),
    other U (net_tree_ips p) →) global (λσ. ∀i∈net_tree_ips p. P i σ)"
  (is "_ ⊨ (?owS p, ?U p →) _")

```

⟨proof⟩

end

18 Lifting rules for (open) closed networks

```

theory OClosed_Lifting
imports OPnet_Lifting
begin

```

lemma trans_fst_oclosed_fst1 [dest]:

```

  "(s, connect(i, i'), s') ∈ ocnet_sos (trans p) ⇒ (s, connect(i, i'), s') ∈ trans p"
  ⟨proof⟩

```

lemma trans_fst_oclosed_fst2 [dest]:

```

  "(s, disconnect(i, i'), s') ∈ ocnet_sos (trans p) ⇒ (s, disconnect(i, i'), s') ∈ trans p"
  ⟨proof⟩

```

lemma trans_fst_oclosed_fst3 [dest]:

```

  "(s, i:deliver(d), s') ∈ ocnet_sos (trans p) ⇒ (s, i:deliver(d), s') ∈ trans p"
  ⟨proof⟩

```

lemma oclosed_reachable_inclosed:

```

  assumes "(σ, ζ) ∈ oreachable (oclosed (opnet np p)) (λ_ _ . True) U"
  shows "(σ, ζ) ∈ oreachable (opnet np p) (otherwith (op=) (net_tree_ips p) inclosed) U"
  (is "_ ∈ oreachable _ ?owS _")
  ⟨proof⟩

```

lemma oclosed_reachable_reachable [elim]:

```

  assumes "(σ, ζ) ∈ oreachable (oclosed (opnet onp p)) (λ_ _ . True) U"
  shows "(σ, ζ) ∈ oreachable (opnet onp p) (λ_ _ . True) U"
  ⟨proof⟩

```

lemma inclosed_closed [intro]:

```

  assumes cinv: "opnet np p ⊨ (otherwith (op=) (net_tree_ips p) inclosed, U →) P"

```

```

  shows "oclosed (opnet np p)  $\models$  ( $\lambda\_ \_ \_ . \text{True}, U \rightarrow$ ) P"
  <proof>

```

end

19 Generic invariants on sequential AWN processes

```

theory AWN_Invariants
imports Invariants AWN_SOS AWN_Labels
begin

```

19.1 Invariants via labelled control terms

Used to state that the initial control-state of an automaton appears within a process specification Γ , meaning that its transitions, and those of its subterms, are subsumed by those of Γ .

definition

```

control_within :: "('s, 'm, 'p, 'l) seqp_env  $\Rightarrow$  ('z  $\times$  ('s, 'm, 'p, 'l) seqp) set  $\Rightarrow$  bool"

```

where

```

"control_within  $\Gamma$   $\sigma$   $\equiv$   $\forall$  ( $\xi, p$ )  $\in$   $\sigma$ .  $\exists$  pn.  $p \in$  subterms ( $\Gamma$  pn)"

```

lemma control_withinI [intro]:

```

assumes " $\bigwedge p. p \in$  Range  $\sigma \implies \exists$  pn.  $p \in$  subterms ( $\Gamma$  pn)"

```

```

  shows "control_within  $\Gamma$   $\sigma$ "

```

<proof>

lemma control_withinD [dest]:

```

assumes "control_within  $\Gamma$   $\sigma$ "

```

```

  and " $(\xi, p) \in \sigma$ "

```

```

  shows " $\exists$  pn.  $p \in$  subterms ( $\Gamma$  pn)"

```

<proof>

lemma control_within_topI [intro]:

```

assumes " $\bigwedge p. p \in$  Range  $\sigma \implies \exists$  pn.  $p = \Gamma$  pn"

```

```

  shows "control_within  $\Gamma$   $\sigma$ "

```

<proof>

lemma seqp_sos_subterms:

```

assumes "wellformed  $\Gamma$ "

```

```

  and " $\exists$  pn.  $p \in$  subterms ( $\Gamma$  pn)"

```

```

  and " $((\xi, p), a, (\xi', p')) \in$  seqp_sos  $\Gamma$ "

```

```

  shows " $\exists$  pn.  $p' \in$  subterms ( $\Gamma$  pn)"

```

<proof>

lemma reachable_subterms:

```

assumes "wellformed  $\Gamma$ "

```

```

  and "control_within  $\Gamma$  (init A)"

```

```

  and "trans A = seqp_sos  $\Gamma$ "

```

```

  and " $(\xi, p) \in$  reachable A I"

```

```

  shows " $\exists$  pn.  $p \in$  subterms ( $\Gamma$  pn)"

```

<proof>

definition

```

onl :: "('s, 'm, 'p, 'l) seqp_env
 $\Rightarrow$  ('z  $\times$  'l  $\Rightarrow$  bool)
 $\Rightarrow$  'z  $\times$  ('s, 'm, 'p, 'l) seqp
 $\Rightarrow$  bool"

```

where

```

"onl  $\Gamma$  P  $\equiv$  ( $\lambda$ ( $\xi, p$ ).  $\forall l \in$  labels  $\Gamma$  p. P ( $\xi, l$ ))"

```

lemma onlI [intro]:

```

assumes " $\bigwedge l. l \in$  labels  $\Gamma$  p  $\implies$  P ( $\xi, l$ )"

```

```

  shows "onl  $\Gamma$  P ( $\xi, p$ )"

```

<proof>

lemmas onlI' [intro] = onlI [simplified atomize_ball]

lemma onlD [dest]:

assumes "onl Γ P (ξ , p)"
shows " $\forall l \in \text{labels } \Gamma p. P (\xi, l)$ "
<proof>

lemma onl_invariantI [intro]:

assumes init: " $\bigwedge \xi p l. [(\xi, p) \in \text{init } A; l \in \text{labels } \Gamma p] \implies P (\xi, l)$ "
and step: " $\bigwedge \xi p a \xi' p' l'. [(\xi, p) \in \text{reachable } A I;$
 $\forall l \in \text{labels } \Gamma p. P (\xi, l);$
 $((\xi, p), a, (\xi', p')) \in \text{trans } A;$
 $l' \in \text{labels } \Gamma p';$
 $I a] \implies P (\xi', l')$ "
shows " $A \models (I \rightarrow) \text{onl } \Gamma P$ "
<proof>

lemma onl_invariantD [dest]:

assumes " $A \models (I \rightarrow) \text{onl } \Gamma P$ "
and " $(\xi, p) \in \text{reachable } A I$ "
and " $l \in \text{labels } \Gamma p$ "
shows " $P (\xi, l)$ "
<proof>

lemma onl_invariant_initD [dest]:

assumes invP: " $A \models (I \rightarrow) \text{onl } \Gamma P$ "
and init: " $(\xi, p) \in \text{init } A$ "
and pnl: " $l \in \text{labels } \Gamma p$ "
shows " $P (\xi, l)$ "
<proof>

lemma onl_invariant_sterms:

assumes wf: "wellformed Γ "
and il: " $A \models (I \rightarrow) \text{onl } \Gamma P$ "
and rp: " $(\xi, p) \in \text{reachable } A I$ "
and "p' \in sterms Γp "
and " $l \in \text{labels } \Gamma p'$ "
shows " $P (\xi, l)$ "
<proof>

lemma onl_invariant_sterms_weaken:

assumes wf: "wellformed Γ "
and il: " $A \models (I \rightarrow) \text{onl } \Gamma P$ "
and rp: " $(\xi, p) \in \text{reachable } A I$ "
and "p' \in sterms Γp "
and " $l \in \text{labels } \Gamma p'$ "
and weaken: " $\bigwedge a. I' a \implies I a$ "
shows " $P (\xi, l)$ "
<proof>

lemma onl_invariant_sterms_TT:

assumes wf: "wellformed Γ "
and il: " $A \models \text{onl } \Gamma P$ "
and rp: " $(\xi, p) \in \text{reachable } A I$ "
and "p' \in sterms Γp "
and " $l \in \text{labels } \Gamma p'$ "
shows " $P (\xi, l)$ "
<proof>

lemma trans_from_sterms:

assumes " $((\xi, p), a, (\xi', q)) \in \text{seqp_sos } \Gamma$ "
and "wellformed Γ "

shows " $\exists p' \in \text{sterms } \Gamma p. ((\xi, p'), a, (\xi', q)) \in \text{seqp_sos } \Gamma$ "
 <proof>

lemma *trans_from_sterms'*:

assumes " $((\xi, p'), a, (\xi', q)) \in \text{seqp_sos } \Gamma$ "
 and "wellformed Γ "
 and " $p' \in \text{sterms } \Gamma p$ "
 shows " $((\xi, p), a, (\xi', q)) \in \text{seqp_sos } \Gamma$ "
 <proof>

lemma *trans_to_dterms*:

assumes " $((\xi, p), a, (\xi', q)) \in \text{seqp_sos } \Gamma$ "
 and "wellformed Γ "
 shows " $\forall r \in \text{sterms } \Gamma q. r \in \text{dterms } \Gamma p$ "
 <proof>

theorem *cterms_includes_sterms_of_seq_reachable*:

assumes "wellformed Γ "
 and "control_within Γ (init A)"
 and "trans A = seqp_sos Γ "
 shows " $\bigcup (\text{sterms } \Gamma \text{ 'snd ' reachable A I}) \subseteq \text{cterms } \Gamma$ "
 <proof>

corollary *seq_reachable_in_cterms*:

assumes "wellformed Γ "
 and "control_within Γ (init A)"
 and "trans A = seqp_sos Γ "
 and " $(\xi, p) \in \text{reachable A I}$ "
 and " $p' \in \text{sterms } \Gamma p$ "
 shows " $p' \in \text{cterms } \Gamma$ "
 <proof>

lemma *seq_invariant_ctermI*:

assumes wf: "wellformed Γ "
 and cw: "control_within Γ (init A)"
 and sl: "simple_labels Γ "
 and sp: "trans A = seqp_sos Γ "
 and init: " $\bigwedge \xi p l. \llbracket$
 $(\xi, p) \in \text{init A};$
 $l \in \text{labels } \Gamma p$
 $\rrbracket \implies P (\xi, l)$ "
 and step: " $\bigwedge p l \xi a q l' \xi' pp. \llbracket$
 $p \in \text{cterms } \Gamma;$
 $l \in \text{labels } \Gamma p;$
 $P (\xi, l);$
 $((\xi, p), a, (\xi', q)) \in \text{seqp_sos } \Gamma;$
 $((\xi, p), a, (\xi', q)) \in \text{trans A};$
 $l' \in \text{labels } \Gamma q;$
 $(\xi, pp) \in \text{reachable A I};$
 $p \in \text{sterms } \Gamma pp;$
 $(\xi', q) \in \text{reachable A I};$
 I a
 $\rrbracket \implies P (\xi', l')$ "
 shows "A $\models (I \rightarrow) \text{onl } \Gamma P$ "
 <proof>

lemma *seq_invariant_ctermsI*:

assumes wf: "wellformed Γ "
 and "control_within Γ (init A)"
 and "simple_labels Γ "
 and "trans A = seqp_sos Γ "
 and init: " $\bigwedge \xi p l. \llbracket$
 $(\xi, p) \in \text{init A};$
 $l \in \text{labels } \Gamma p$

```

    ]  $\implies P (\xi, l)$ "
  and step: " $\bigwedge p \ l \ \xi \ a \ q \ l' \ \xi' \ pp \ pn. \llbracket$ 
    wellformed  $\Gamma$ ;
    p $\in$ ctermstl ( $\Gamma \ pn$ );
    not_call p;
    l $\in$ labels  $\Gamma \ p$ ;
    P ( $\xi, l$ );
    (( $\xi, p$ ), a, ( $\xi', q$ ))  $\in$  seqp_sos  $\Gamma$ ;
    (( $\xi, p$ ), a, ( $\xi', q$ ))  $\in$  trans A;
    l' $\in$ labels  $\Gamma \ q$ ;
    ( $\xi, pp$ ) $\in$ reachable A I;
    p $\in$ sterms  $\Gamma \ pp$ ;
    ( $\xi', q$ ) $\in$ reachable A I;
    I a
    ]  $\implies P (\xi', l')$ "
  shows "A  $\models (I \rightarrow) \text{onl } \Gamma \ P$ "
<proof>

```

19.2 Step invariants via labelled control terms

definition

```

onll :: "('s, 'm, 'p, 'l) seqp_env
       $\Rightarrow$  (('z  $\times$  'l, 'a) transition  $\Rightarrow$  bool)
       $\Rightarrow$  ('z  $\times$  ('s, 'm, 'p, 'l) seqp, 'a) transition  $\Rightarrow$  bool"

```

where

```

"onll  $\Gamma \ P \equiv (\lambda((\xi, p), a, (\xi', p')). \forall l \in \text{labels } \Gamma \ p. \forall l' \in \text{labels } \Gamma \ p'. P ((\xi, l), a, (\xi', l')))"$ 
```

lemma onllI [intro]:

```

assumes " $\bigwedge l \ l'. \llbracket l \in \text{labels } \Gamma \ p; l' \in \text{labels } \Gamma \ p' \rrbracket \implies P ((\xi, l), a, (\xi', l'))"$ 
  shows "onll  $\Gamma \ P ((\xi, p), a, (\xi', p'))"$ 
<proof>

```

lemma onllII [intro]:

```

assumes " $\forall l \in \text{labels } \Gamma \ p. \forall l' \in \text{labels } \Gamma \ p'. P ((\xi, l), a, (\xi', l'))"$ 
  shows "onll  $\Gamma \ P ((\xi, p), a, (\xi', p'))"$ 
<proof>

```

lemma onllD [dest]:

```

assumes "onll  $\Gamma \ P ((\xi, p), a, (\xi', p'))"$ 
  shows " $\forall l \in \text{labels } \Gamma \ p. \forall l' \in \text{labels } \Gamma \ p'. P ((\xi, l), a, (\xi', l'))"$ 
<proof>

```

lemma onl_weaken [elim!]: " $\bigwedge \Gamma \ P \ Q \ s. \llbracket \text{onl } \Gamma \ P \ s; \bigwedge s. P \ s \implies Q \ s \rrbracket \implies \text{onl } \Gamma \ Q \ s"$

<proof>

lemma onll_weaken [elim!]: " $\bigwedge \Gamma \ P \ Q \ s. \llbracket \text{onll } \Gamma \ P \ s; \bigwedge s. P \ s \implies Q \ s \rrbracket \implies \text{onll } \Gamma \ Q \ s"$

<proof>

lemma onll_weaken' [elim!]: " $\bigwedge \Gamma \ P \ Q \ s. \llbracket \text{onll } \Gamma \ P ((\xi, p), a, (\xi', p'));$

```

 $\bigwedge l \ l'. P ((\xi, l), a, (\xi', l')) \implies Q ((\xi, l), a, (\xi', l')) \rrbracket$ 
 $\implies \text{onll } \Gamma \ Q ((\xi, p), a, (\xi', p'))"$ 

```

<proof>

lemma onll_step_invariantI [intro]:

```

assumes *: " $\bigwedge \xi \ p \ l \ a \ \xi' \ p' \ l'. \llbracket (\xi, p) \in \text{reachable } A \ I;$ 
  (( $\xi, p$ ), a, ( $\xi', p'$ ))  $\in$  trans A;
  I a;
  l  $\in$  labels  $\Gamma \ p$ ;
  l'  $\in$  labels  $\Gamma \ p' \rrbracket$ 
 $\implies P ((\xi, l), a, (\xi', l'))"$ 

```

shows "A $\models_A (I \rightarrow) \text{onll } \Gamma \ P$ "

<proof>

lemma onll_step_invariantE [elim]:

```

assumes "A  $\models_A$  (I  $\rightarrow$ ) onll  $\Gamma$  P"
  and "( $\xi$ , p)  $\in$  reachable A I"
  and "(( $\xi$ , p), a, ( $\xi'$ , p'))  $\in$  trans A"
  and "I a"
  and lp: "l  $\in$  labels  $\Gamma$  p"
  and lp': "l'  $\in$  labels  $\Gamma$  p'"
shows "P (( $\xi$ , l), a, ( $\xi'$ , l'))"
<proof>

lemma onll_step_invariantD [dest]:
assumes "A  $\models_A$  (I  $\rightarrow$ ) onll  $\Gamma$  P"
  and "( $\xi$ , p)  $\in$  reachable A I"
  and "(( $\xi$ , p), a, ( $\xi'$ , p'))  $\in$  trans A"
  and "I a"
shows " $\forall l \in$  labels  $\Gamma$  p.  $\forall l' \in$  labels  $\Gamma$  p'. P (( $\xi$ , l), a, ( $\xi'$ , l'))"
<proof>

lemma onll_step_to_invariantI [intro]:
assumes sinv: "A  $\models_A$  (I  $\rightarrow$ ) onll  $\Gamma$  Q"
  and wf: "wellformed  $\Gamma$ "
  and init: " $\bigwedge \xi$  l p.  $\llbracket$  ( $\xi$ , p)  $\in$  init A; l  $\in$  labels  $\Gamma$  p  $\rrbracket \implies$  P ( $\xi$ , l)"
  and step: " $\bigwedge \xi$  p l  $\xi'$  l' a.
     $\llbracket$  ( $\xi$ , p)  $\in$  reachable A I;
      l  $\in$  labels  $\Gamma$  p;
      P ( $\xi$ , l);
      Q (( $\xi$ , l), a, ( $\xi'$ , l'));
      I a  $\rrbracket \implies$  P ( $\xi'$ , l'"
shows "A  $\models$  (I  $\rightarrow$ ) onl  $\Gamma$  P"
<proof>

lemma onll_step_invariant_sterms:
assumes wf: "wellformed  $\Gamma$ "
  and si: "A  $\models_A$  (I  $\rightarrow$ ) onll  $\Gamma$  P"
  and sr: "( $\xi$ , p)  $\in$  reachable A I"
  and sos: "(( $\xi$ , p), a, ( $\xi'$ , q))  $\in$  trans A"
  and "I a"
  and "l'  $\in$  labels  $\Gamma$  q"
  and "p'  $\in$  sterms  $\Gamma$  p"
  and "l  $\in$  labels  $\Gamma$  p'"
shows "P (( $\xi$ , l), a, ( $\xi'$ , l'))"
<proof>

lemma seq_step_invariant_sterms:
assumes inv: "A  $\models_A$  (I  $\rightarrow$ ) onll  $\Gamma$  P"
  and wf: "wellformed  $\Gamma$ "
  and sp: "trans A = seqp_sos  $\Gamma$ "
  and "l'  $\in$  labels  $\Gamma$  q"
  and sr: "( $\xi$ , p)  $\in$  reachable A I"
  and tr: "(( $\xi$ , p'), a, ( $\xi'$ , q))  $\in$  trans A"
  and "I a"
  and "p'  $\in$  sterms  $\Gamma$  p"
shows " $\forall l \in$  labels  $\Gamma$  p'. P (( $\xi$ , l), a, ( $\xi'$ , l'))"
<proof>

lemma seq_step_invariant_sterms_weaken:
assumes "A  $\models_A$  (I  $\rightarrow$ ) onll  $\Gamma$  P"
  and "wellformed  $\Gamma$ "
  and "trans A = seqp_sos  $\Gamma$ "
  and "l'  $\in$  labels  $\Gamma$  q"
  and "( $\xi$ , p)  $\in$  reachable A I'"
  and "(( $\xi$ , p'), a, ( $\xi'$ , q))  $\in$  trans A"
  and "I' a"
  and "p'  $\in$  sterms  $\Gamma$  p"
  and weaken: " $\bigwedge a. I' a \implies I a$ "

```

shows " $\forall l \in \text{labels } \Gamma \ p'. \ P \ ((\xi, l), a, (\xi', l'))$ "
 <proof>

lemma seq_step_invariant_sterms_TT:

assumes " $A \Vdash_A \text{onll } \Gamma \ P$ "
 and "wellformed Γ "
 and " $\text{trans } A = \text{seqp_sos } \Gamma$ "
 and " $l' \in \text{labels } \Gamma \ q$ "
 and " $(\xi, p) \in \text{reachable } A \ I$ "
 and " $((\xi, p'), a, (\xi', q)) \in \text{trans } A$ "
 and " $I \ a$ "
 and " $p' \in \text{sterms } \Gamma \ p$ "
 shows " $\forall l \in \text{labels } \Gamma \ p'. \ P \ ((\xi, l), a, (\xi', l'))$ "
 <proof>

lemma onll_step_invariant_any_sterms:

assumes "wellformed Γ "
 and " $A \Vdash_A (I \rightarrow) \text{onll } \Gamma \ P$ "
 and " $(\xi, p) \in \text{reachable } A \ I$ "
 and " $((\xi, p), a, (\xi', q)) \in \text{trans } A$ "
 and " $I \ a$ "
 and " $l' \in \text{labels } \Gamma \ q$ "
 shows " $\forall p' \in \text{sterms } \Gamma \ p. \ \forall l \in \text{labels } \Gamma \ p'. \ P \ ((\xi, l), a, (\xi', l'))$ "
 <proof>

lemma seq_step_invariant_ctermI [intro]:

assumes wf: "wellformed Γ "
 and cw: "control_within $\Gamma \ (\text{init } A)$ "
 and sl: "simple_labels Γ "
 and sp: " $\text{trans } A = \text{seqp_sos } \Gamma$ "
 and step: " $\bigwedge p \ pp \ l \ \xi \ a \ q \ l' \ \xi'. \ [$
 $p \in \text{cterms } \Gamma;$
 $l \in \text{labels } \Gamma \ p;$
 $((\xi, p), a, (\xi', q)) \in \text{seqp_sos } \Gamma;$
 $((\xi, p), a, (\xi', q)) \in \text{trans } A;$
 $l' \in \text{labels } \Gamma \ q;$
 $(\xi, pp) \in \text{reachable } A \ I;$
 $p \in \text{sterms } \Gamma \ pp;$
 $(\xi', q) \in \text{reachable } A \ I;$
 $I \ a$
 $]\implies P \ ((\xi, l), a, (\xi', l'))$ "
 shows " $A \Vdash_A (I \rightarrow) \text{onll } \Gamma \ P$ "
 <proof>

lemma seq_step_invariant_ctermsI [intro]:

assumes wf: "wellformed Γ "
 and cw: "control_within $\Gamma \ (\text{init } A)$ "
 and sl: "simple_labels Γ "
 and sp: " $\text{trans } A = \text{seqp_sos } \Gamma$ "
 and step: " $\bigwedge p \ l \ \xi \ a \ q \ l' \ \xi' \ pp \ pn. \ [$
 wellformed $\Gamma;$
 $p \in \text{ctermsl } (\Gamma \ pn);$
 not_call $p;$
 $l \in \text{labels } \Gamma \ p;$
 $((\xi, p), a, (\xi', q)) \in \text{seqp_sos } \Gamma;$
 $((\xi, p), a, (\xi', q)) \in \text{trans } A;$
 $l' \in \text{labels } \Gamma \ q;$
 $(\xi, pp) \in \text{reachable } A \ I;$
 $p \in \text{sterms } \Gamma \ pp;$
 $(\xi', q) \in \text{reachable } A \ I;$
 $I \ a$
 $]\implies P \ ((\xi, l), a, (\xi', l'))$ "
 shows " $A \Vdash_A (I \rightarrow) \text{onll } \Gamma \ P$ "
 <proof>

end

20 Generic open invariants on sequential AWN processes

```
theory OAWN_Invariants
imports Invariants OInvariants
        AWN_Cterms AWN_Labels AWN_Invariants
        OAWN_SOS
```

begin

20.1 Open invariants via labelled control terms

```
lemma oseqp_sos_subterms:
  assumes "wellformed  $\Gamma$ "
    and " $\exists pn. p \in \text{subterms } (\Gamma pn)$ "
    and " $((\sigma, p), a, (\sigma', p')) \in \text{oseqp\_sos } \Gamma i$ "
  shows " $\exists pn. p' \in \text{subterms } (\Gamma pn)$ "
  <proof>

lemma oreachable_subterms:
  assumes "wellformed  $\Gamma$ "
    and "control_within  $\Gamma$  (init A)"
    and "trans A = oseqp_sos  $\Gamma i$ "
    and " $(\sigma, p) \in \text{oreachable } A S U$ "
  shows " $\exists pn. p \in \text{subterms } (\Gamma pn)$ "
  <proof>

lemma onl_oinvariantI [intro]:
  assumes init: " $\bigwedge \sigma p l. \llbracket (\sigma, p) \in \text{init } A; l \in \text{labels } \Gamma p \rrbracket \implies P (\sigma, l)$ "
    and other: " $\bigwedge \sigma \sigma' p l. \llbracket (\sigma, p) \in \text{oreachable } A S U;$   

 $\forall l \in \text{labels } \Gamma p. P (\sigma, l);$   

 $U \sigma \sigma' \rrbracket \implies \forall l \in \text{labels } \Gamma p. P (\sigma', l)$ "
    and step: " $\bigwedge \sigma p a \sigma' p' l'.$   

 $\llbracket (\sigma, p) \in \text{oreachable } A S U;$   

 $\forall l \in \text{labels } \Gamma p. P (\sigma, l);$   

 $((\sigma, p), a, (\sigma', p')) \in \text{trans } A;$   

 $l' \in \text{labels } \Gamma p';$   

 $S \sigma \sigma' a \rrbracket \implies P (\sigma', l')$ "
  shows " $A \models (S, U \rightarrow) \text{onl } \Gamma P$ "
  <proof>

lemma global_oinvariantI [intro]:
  assumes init: " $\bigwedge \sigma p. (\sigma, p) \in \text{init } A \implies P \sigma$ "
    and other: " $\bigwedge \sigma \sigma' p l. \llbracket (\sigma, p) \in \text{oreachable } A S U; P \sigma; U \sigma \sigma' \rrbracket \implies P \sigma'$ "
    and step: " $\bigwedge \sigma p a \sigma' p'.$   

 $\llbracket (\sigma, p) \in \text{oreachable } A S U;$   

 $P \sigma;$   

 $((\sigma, p), a, (\sigma', p')) \in \text{trans } A;$   

 $S \sigma \sigma' a \rrbracket \implies P \sigma'$ "
  shows " $A \models (S, U \rightarrow) (\lambda(\sigma, _). P \sigma)$ "
  <proof>

lemma onl_oinvariantD [dest]:
  assumes " $A \models (S, U \rightarrow) \text{onl } \Gamma P$ "
    and " $(\sigma, p) \in \text{oreachable } A S U$ "
    and " $l \in \text{labels } \Gamma p$ "
  shows " $P (\sigma, l)$ "
  <proof>

lemma onl_oinvariant_weakenD [dest]:
  assumes " $A \models (S', U' \rightarrow) \text{onl } \Gamma P$ "
    and " $(\sigma, p) \in \text{oreachable } A S U$ "
    and " $l \in \text{labels } \Gamma p$ "
```

```

    and weakenS: " $\bigwedge s s' a. S s s' a \implies S' s s' a$ "
    and weakenU: " $\bigwedge s s'. U s s' \implies U' s s'$ "
  shows "P ( $\sigma$ , l)"
  <proof>

lemma onl_oInvariant_initD [dest]:
  assumes invP: "A  $\models$  (S, U  $\rightarrow$ ) onl  $\Gamma$  P"
    and init: " $(\sigma, p) \in \text{init } A$ "
    and pnl: "l  $\in$  labels  $\Gamma$  p"
  shows "P ( $\sigma$ , l)"
  <proof>

lemma onl_oInvariant_sterms:
  assumes wf: "wellformed  $\Gamma$ "
    and il: "A  $\models$  (S, U  $\rightarrow$ ) onl  $\Gamma$  P"
    and rp: " $(\sigma, p) \in \text{oreachable } A S U$ "
    and "p'  $\in$  sterms  $\Gamma$  p"
    and "l  $\in$  labels  $\Gamma$  p'"
  shows "P ( $\sigma$ , l)"
  <proof>

lemma onl_oInvariant_sterms_weaken:
  assumes wf: "wellformed  $\Gamma$ "
    and il: "A  $\models$  (S', U'  $\rightarrow$ ) onl  $\Gamma$  P"
    and rp: " $(\sigma, p) \in \text{oreachable } A S U$ "
    and "p'  $\in$  sterms  $\Gamma$  p"
    and "l  $\in$  labels  $\Gamma$  p'"
    and weakenS: " $\bigwedge \sigma \sigma' a. S \sigma \sigma' a \implies S' \sigma \sigma' a$ "
    and weakenU: " $\bigwedge \sigma \sigma'. U \sigma \sigma' \implies U' \sigma \sigma'$ "
  shows "P ( $\sigma$ , l)"
  <proof>

lemma oTrans_from_sterms:
  assumes " $((\sigma, p), a, (\sigma', q)) \in \text{oseqp\_sos } \Gamma i$ "
    and "wellformed  $\Gamma$ "
  shows " $\exists p' \in \text{sterms } \Gamma p. ((\sigma, p'), a, (\sigma', q)) \in \text{oseqp\_sos } \Gamma i$ "
  <proof>

lemma oTrans_from_sterms':
  assumes " $((\sigma, p'), a, (\sigma', q)) \in \text{oseqp\_sos } \Gamma i$ "
    and "wellformed  $\Gamma$ "
    and "p'  $\in$  sterms  $\Gamma$  p"
  shows " $((\sigma, p), a, (\sigma', q)) \in \text{oseqp\_sos } \Gamma i$ "
  <proof>

lemma oTrans_to_dterms:
  assumes " $((\sigma, p), a, (\sigma', q)) \in \text{oseqp\_sos } \Gamma i$ "
    and "wellformed  $\Gamma$ "
  shows " $\forall r \in \text{sterms } \Gamma q. r \in \text{dterms } \Gamma p$ "
  <proof>

theorem cterms_includes_sterms_of_oseq_reachable:
  assumes "wellformed  $\Gamma$ "
    and "control_within  $\Gamma$  (init A)"
    and "trans A = oseqp_sos  $\Gamma$  i"
  shows " $\bigcup (\text{sterms } \Gamma \text{ 'snd ' oreachable } A S U) \subseteq \text{cterms } \Gamma$ "
  <proof>

corollary oseq_reachable_in_cterms:
  assumes "wellformed  $\Gamma$ "
    and "control_within  $\Gamma$  (init A)"
    and "trans A = oseqp_sos  $\Gamma$  i"
    and " $(\sigma, p) \in \text{oreachable } A S U$ "
    and "p'  $\in$  sterms  $\Gamma$  p"

```

shows "p' ∈ cterms Γ"
 ⟨proof⟩

lemma oseq_invariant_ctermI:

assumes wf: "wellformed Γ"
 and cw: "control_within Γ (init A)"
 and sl: "simple_labels Γ"
 and sp: "trans A = oseqp_sos Γ i"
 and init: " $\bigwedge \sigma p l. \llbracket$
 $(\sigma, p) \in \text{init } A;$
 $l \in \text{labels } \Gamma p$
 $\rrbracket \implies P (\sigma, l)$ "
 and other: " $\bigwedge \sigma \sigma' p l. \llbracket$
 $(\sigma, p) \in \text{oreachable } A S U;$
 $l \in \text{labels } \Gamma p;$
 $P (\sigma, l);$
 $U \sigma \sigma' \rrbracket \implies P (\sigma', l)$ "
 and local: " $\bigwedge p l \sigma a q l' \sigma' pp. \llbracket$
 $p \in \text{cterm } \Gamma;$
 $l \in \text{labels } \Gamma p;$
 $P (\sigma, l);$
 $((\sigma, p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma i;$
 $((\sigma, p), a, (\sigma', q)) \in \text{trans } A;$
 $l' \in \text{labels } \Gamma q;$
 $(\sigma, pp) \in \text{oreachable } A S U;$
 $p \in \text{sterms } \Gamma pp;$
 $(\sigma', q) \in \text{oreachable } A S U;$
 $S \sigma \sigma' a$
 $\rrbracket \implies P (\sigma', l')$ "
 shows "A ⊨ (S, U →) onl Γ P"
 ⟨proof⟩

lemma oseq_invariant_ctermsI:

assumes wf: "wellformed Γ"
 and cw: "control_within Γ (init A)"
 and sl: "simple_labels Γ"
 and sp: "trans A = oseqp_sos Γ i"
 and init: " $\bigwedge \sigma p l. \llbracket$
 $(\sigma, p) \in \text{init } A;$
 $l \in \text{labels } \Gamma p$
 $\rrbracket \implies P (\sigma, l)$ "
 and other: " $\bigwedge \sigma \sigma' p l. \llbracket$
 wellformed Γ;
 $(\sigma, p) \in \text{oreachable } A S U;$
 $l \in \text{labels } \Gamma p;$
 $P (\sigma, l);$
 $U \sigma \sigma' \rrbracket \implies P (\sigma', l)$ "
 and local: " $\bigwedge p l \sigma a q l' \sigma' pp pn. \llbracket$
 wellformed Γ;
 $p \in \text{cterm } \Gamma pn;$
 not_call p;
 $l \in \text{labels } \Gamma p;$
 $P (\sigma, l);$
 $((\sigma, p), a, (\sigma', q)) \in \text{oseqp_sos } \Gamma i;$
 $((\sigma, p), a, (\sigma', q)) \in \text{trans } A;$
 $l' \in \text{labels } \Gamma q;$
 $(\sigma, pp) \in \text{oreachable } A S U;$
 $p \in \text{sterms } \Gamma pp;$
 $(\sigma', q) \in \text{oreachable } A S U;$
 $S \sigma \sigma' a$
 $\rrbracket \implies P (\sigma', l')$ "
 shows "A ⊨ (S, U →) onl Γ P"
 ⟨proof⟩

20.2 Open step invariants via labelled control terms

lemma onll_ostep_invariantI [intro]:

```

assumes *: "∧σ p l a σ' p' l'. [ (σ, p) ∈ oreachable A S U;
                                   ((σ, p), a, (σ', p')) ∈ trans A;
                                   S σ σ' a;
                                   l ∈ labels Γ p;
                                   l' ∈ labels Γ p' ]
        ⇒ P ((σ, l), a, (σ', l'))"
shows "A ⊨A (S, U →) onll Γ P"
⟨proof⟩

```

lemma onll_ostep_invariantE [elim]:

```

assumes "A ⊨A (S, U →) onll Γ P"
and "(σ, p) ∈ oreachable A S U"
and "((σ, p), a, (σ', p')) ∈ trans A"
and "S σ σ' a"
and lp: "l ∈ labels Γ p"
and lp': "l' ∈ labels Γ p'"
shows "P ((σ, l), a, (σ', l'))"
⟨proof⟩

```

lemma onll_ostep_invariantD [dest]:

```

assumes "A ⊨A (S, U →) onll Γ P"
and "(σ, p) ∈ oreachable A S U"
and "((σ, p), a, (σ', p')) ∈ trans A"
and "S σ σ' a"
shows "∀l ∈ labels Γ p. ∀l' ∈ labels Γ p'. P ((σ, l), a, (σ', l'))"
⟨proof⟩

```

lemma onll_ostep_invariant_weakenD [dest]:

```

assumes "A ⊨A (S', U' →) onll Γ P"
and "(σ, p) ∈ oreachable A S U"
and "((σ, p), a, (σ', p')) ∈ trans A"
and "S' σ σ' a"
and weakenS: "∧s s' a. S s s' a ⇒ S' s s' a"
and weakenU: "∧s s'. U s s' ⇒ U' s s'"
shows "∀l ∈ labels Γ p. ∀l' ∈ labels Γ p'. P ((σ, l), a, (σ', l'))"
⟨proof⟩

```

lemma onll_ostep_to_invariantI [intro]:

```

assumes sinv: "A ⊨A (S, U →) onll Γ Q"
and wf: "wellformed Γ"
and init: "∧σ l p. [ (σ, p) ∈ init A; l ∈ labels Γ p ] ⇒ P (σ, l)"
and other: "∧σ σ' p l.
            [ (σ, p) ∈ oreachable A S U;
              l ∈ labels Γ p;
              P (σ, l);
              U σ σ' ] ⇒ P (σ', l)"
and local: "∧σ p l σ' l' a.
            [ (σ, p) ∈ oreachable A S U;
              l ∈ labels Γ p;
              P (σ, l);
              Q ((σ, l), a, (σ', l'));
              S σ σ' a ] ⇒ P (σ', l)"
shows "A ⊨ (S, U →) onl Γ P"
⟨proof⟩

```

lemma onll_ostep_invariant_terms:

```

assumes wf: "wellformed Γ"
and si: "A ⊨A (S, U →) onll Γ P"
and sr: "(σ, p) ∈ oreachable A S U"
and sos: "((σ, p), a, (σ', q)) ∈ trans A"
and "S σ σ' a"
and "l' ∈ labels Γ q"

```

and "p' ∈ sterm s Γ p"
 and "l ∈ labels Γ p'"
 shows "P ((σ, l), a, (σ', l'))"
 ⟨proof⟩

lemma oseq_step_invariant_sterm s:

assumes inv: "A ⊢_A (S, U →) onll Γ P"
 and wf: "wellformed Γ"
 and sp: "trans A = oseqp_sos Γ i"
 and "l' ∈ labels Γ q"
 and sr: "(σ, p) ∈ oreachable A S U"
 and tr: "((σ, p'), a, (σ', q)) ∈ trans A"
 and "S σ σ' a"
 and "p' ∈ sterm s Γ p"
 shows "∀ l ∈ labels Γ p'. P ((σ, l), a, (σ', l'))"
 ⟨proof⟩

lemma oseq_step_invariant_sterm s_weaken:

assumes inv: "A ⊢_A (S, U →) onll Γ P"
 and wf: "wellformed Γ"
 and sp: "trans A = oseqp_sos Γ i"
 and "l' ∈ labels Γ q"
 and sr: "(σ, p) ∈ oreachable A S' U"
 and tr: "((σ, p'), a, (σ', q)) ∈ trans A"
 and "S' σ σ' a"
 and "p' ∈ sterm s Γ p"
 and weakenS: "∧ σ σ' a. S' σ σ' a ⇒ S σ σ' a"
 and weakenU: "∧ σ σ'. U' σ σ' ⇒ U σ σ'"
 shows "∀ l ∈ labels Γ p'. P ((σ, l), a, (σ', l'))"
 ⟨proof⟩

lemma onll_ostep_invariant_any_sterm s:

assumes wf: "wellformed Γ"
 and si: "A ⊢_A (S, U →) onll Γ P"
 and sr: "(σ, p) ∈ oreachable A S U"
 and sos: "((σ, p), a, (σ', q)) ∈ trans A"
 and "S σ σ' a"
 and "l' ∈ labels Γ q"
 shows "∀ p' ∈ sterm s Γ p. ∀ l ∈ labels Γ p'. P ((σ, l), a, (σ', l'))"
 ⟨proof⟩

lemma oseq_step_invariant_cterm I [intro]:

assumes wf: "wellformed Γ"
 and cw: "control_within Γ (init A)"
 and sl: "simple_labels Γ"
 and sp: "trans A = oseqp_sos Γ i"
 and local: "∧ p l σ a q l' σ' pp. [

 p ∈ cterm s Γ;

 l ∈ labels Γ p;

 ((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i;

 ((σ, p), a, (σ', q)) ∈ trans A;

 l' ∈ labels Γ q;

 (σ, pp) ∈ oreachable A S U;

 p ∈ sterm s Γ pp;

 (σ', q) ∈ oreachable A S U;

 S σ σ' a

] ⇒ P ((σ, l), a, (σ', l'))"

shows "A ⊢_A (S, U →) onll Γ P"
 ⟨proof⟩

lemma oseq_step_invariant_cterm sI [intro]:

assumes wf: "wellformed Γ"
 and "control_within Γ (init A)"
 and "simple_labels Γ"

```

and "trans A = oseqp_sos  $\Gamma$  i"
and local: " $\bigwedge p$  l  $\sigma$  a q l'  $\sigma'$  pp pn. [
  wellformed  $\Gamma$ ;
  p $\in$ ctermstl ( $\Gamma$  pn);
  not_call p;
  l $\in$ labels  $\Gamma$  p;
  (( $\sigma$ , p), a, ( $\sigma'$ , q))  $\in$  oseqp_sos  $\Gamma$  i;
  (( $\sigma$ , p), a, ( $\sigma'$ , q))  $\in$  trans A;
  l' $\in$ labels  $\Gamma$  q;
  ( $\sigma$ , pp)  $\in$  oreachable A S U;
  p $\in$ sterms  $\Gamma$  pp;
  ( $\sigma'$ , q)  $\in$  oreachable A S U;
  S  $\sigma$   $\sigma'$  a
]  $\implies$  P (( $\sigma$ , l), a, ( $\sigma'$ , l'))"
shows "A  $\models_A$  (S, U  $\rightarrow$ ) onll  $\Gamma$  P"
<proof>

```

```

lemma open_seqp_action [elim]:
  assumes "wellformed  $\Gamma$ "
  and "(( $\sigma$  i, p), a, ( $\sigma'$  i, p'))  $\in$  seqp_sos  $\Gamma$ "
  shows "(( $\sigma$ , p), a, ( $\sigma'$ , p'))  $\in$  oseqp_sos  $\Gamma$  i"
<proof>

```

end

21 Transfer standard invariants into open invariants

```

theory OAWN_Convert
imports AWN_SOS_Labels AWN_Invariants
  OAWN_SOS OAWN_Invariants
begin

```

```

definition initiali :: "'i  $\Rightarrow$  (('i  $\Rightarrow$  'g)  $\times$  'l) set  $\Rightarrow$  ('g  $\times$  'l) set  $\Rightarrow$  bool"
where "initiali i OI CI  $\equiv$  ({( $\sigma$  i, p) |  $\sigma$  p. ( $\sigma$ , p)  $\in$  OI} = CI)"

```

```

lemma initialiI [intro]:
  assumes OICI: " $\bigwedge \sigma$  p. ( $\sigma$ , p)  $\in$  OI  $\implies$  ( $\sigma$  i, p)  $\in$  CI"
  and CIOI: " $\bigwedge \xi$  p. ( $\xi$ , p)  $\in$  CI  $\implies$   $\exists \sigma$ .  $\xi = \sigma$  i  $\wedge$  ( $\sigma$ , p)  $\in$  OI"
  shows "initiali i OI CI"
<proof>

```

```

lemma open_from_initialiD [dest]:
  assumes "initiali i OI CI"
  and "( $\sigma$ , p)  $\in$  OI"
  shows " $\exists \xi$ .  $\sigma$  i =  $\xi$   $\wedge$  ( $\xi$ , p)  $\in$  CI"
<proof>

```

```

lemma closed_from_initialiD [dest]:
  assumes "initiali i OI CI"
  and "( $\xi$ , p)  $\in$  CI"
  shows " $\exists \sigma$ .  $\sigma$  i =  $\xi$   $\wedge$  ( $\sigma$ , p)  $\in$  OI"
<proof>

```

```

definition
  seq1 :: "'i  $\Rightarrow$  (('s  $\times$  'l)  $\Rightarrow$  bool)  $\Rightarrow$  (('i  $\Rightarrow$  's)  $\times$  'l)  $\Rightarrow$  bool"
where
  "seq1 i P  $\equiv$  ( $\lambda(\sigma$ , p). P ( $\sigma$  i, p))"

```

```

lemma seq1I [intro]:
  "P (fst s i, snd s)  $\implies$  seq1 i P s"
<proof>

```

```

lemma same_seq1 [elim]:
  assumes " $\forall j \in \{i\}$ .  $\sigma' j = \sigma j$ "

```

```

    and "seq1 i P ( $\sigma'$ , s)"
    shows "seq1 i P ( $\sigma$ , s)"
  <proof>

```

```

lemma seq1simp:
  "seq1 i P ( $\sigma$ , p) = P ( $\sigma$  i, p)"
  <proof>

```

```

lemma other_steps_resp_local [intro!, simp]: "other_steps (other A I) I"
  <proof>

```

```

lemma seq1_onl_swap:
  "seq1 i (onl  $\Gamma$  P) = onl  $\Gamma$  (seq1 i P)"
  <proof>

```

```

lemma oseqp_sos_resp_local_steps [intro!, simp]:
  fixes  $\Gamma$  :: "'p  $\Rightarrow$  ('s, 'm, 'p, 'l) seqp"
  shows "local_steps (oseqp_sos  $\Gamma$  i) {i}"
  <proof>

```

```

lemma oseqp_sos_subreachable [intro!, simp]:
  assumes "trans OA = oseqp_sos  $\Gamma$  i"
  shows "subreachable OA (other ANY {i}) {i}"
  <proof>

```

```

lemma oseq_step_is_seq_step:
  fixes  $\sigma$  :: "ip  $\Rightarrow$  's"
  assumes "( $(\sigma, p)$ , a :: 'm seq_action, ( $\sigma'$ , p'))  $\in$  oseqp_sos  $\Gamma$  i"
  and " $\sigma$  i =  $\xi$ "
  shows " $\exists \xi'. \sigma' i = \xi' \wedge ((\xi, p), a, (\xi', p')) \in$  seqp_sos  $\Gamma$ "
  <proof>

```

```

lemma reachable_oseq_seqp_sos:
  assumes " $(\sigma, p) \in$  reachable OA I"
  and "initiali i (init OA) (init A)"
  and spo: "trans OA = oseqp_sos  $\Gamma$  i"
  and sp: "trans A = seqp_sos  $\Gamma$ "
  shows " $\exists \xi. \sigma i = \xi \wedge (\xi, p) \in$  reachable A I"
  <proof>

```

```

lemma reachable_oseq_seqp_sos':
  assumes "s  $\in$  reachable OA I"
  and "initiali i (init OA) (init A)"
  and "trans OA = oseqp_sos  $\Gamma$  i"
  and "trans A = seqp_sos  $\Gamma$ "
  shows " $\exists \xi. (fst s) i = \xi \wedge (\xi, snd s) \in$  reachable A I"
  <proof>

```

Any invariant shown in the (simpler) closed semantics can be transferred to an invariant in the open semantics.

```

theorem open_seq_invariant [intro]:
  assumes "A  $\models (I \rightarrow) P$ "
  and "initiali i (init OA) (init A)"
  and spo: "trans OA = oseqp_sos  $\Gamma$  i"
  and sp: "trans A = seqp_sos  $\Gamma$ "
  shows "OA  $\models$  (act I, other ANY {i}  $\rightarrow$ ) (seq1 i P)"
  <proof>

```

```

definition
  seq11 :: "'i  $\Rightarrow$  ((('s  $\times$  'l)  $\times$  'a  $\times$  ('s  $\times$  'l))  $\Rightarrow$  bool)
            $\Rightarrow$  (((('i  $\Rightarrow$  's)  $\times$  'l)  $\times$  'a  $\times$  (('i  $\Rightarrow$  's)  $\times$  'l))  $\Rightarrow$  bool"

```

```

where
  "seq11 i P  $\equiv$  ( $\lambda((\sigma, p), a, (\sigma', p')). P ((\sigma i, p), a, (\sigma' i, p')))$ "

```

```

lemma same_seq11 [elim]:

```

```

assumes "∀j∈{i}. σ1' j = σ1 j"
  and "∀j∈{i}. σ2' j = σ2 j"
  and "seqll i P ((σ1', s), a, (σ2', s'))"
shows "seqll i P ((σ1, s), a, (σ2, s'))"
⟨proof⟩

lemma seqllI [intro!]:
  assumes "P ((σ i, p), a, (σ' i, p'))"
  shows "seqll i P ((σ, p), a, (σ', p'))"
⟨proof⟩

lemma seqllD [dest]:
  assumes "seqll i P ((σ, p), a, (σ', p'))"
  shows "P ((σ i, p), a, (σ' i, p'))"
⟨proof⟩

lemma seqllsimp:
  "seqll i P ((σ, p), a, (σ', p')) = P ((σ i, p), a, (σ' i, p'))"
⟨proof⟩

lemma seqll_onll_swap:
  "seqll i (onll Γ P) = onll Γ (seqll i P)"
⟨proof⟩

theorem open_seq_step_invariant [intro]:
  assumes "A ⊧A (I →) P"
  and "initiali i (init OA) (init A)"
  and spo: "trans OA = oseqp_sos Γ i"
  and sp: "trans A = seqp_sos Γ"
  shows "OA ⊧A (act I, other ANY {i} →) (seqll i P)"
⟨proof⟩

end



## 22 Model the standard queuing model



theory Qmsg
imports AWN_SOS_Labels AWN_Invariants
begin

Define the queue process

fun ΓQMSG :: "'m::msg) list, 'm, unit, unit label) seqp_env"
where
  "ΓQMSG () = labelled () (receive(λmsg msgs. msgs @ [msg]). call())
  ⊕ ⟨msgs. msgs ≠ []⟩
  (send(λmsgs. hd msgs).
  ([msgs. tl msgs] call())
  ⊕ receive(λmsg msgs. tl msgs @ [msg]). call())
  ⊕ receive(λmsg msgs. msgs @ [msg]). call())"

definition σQMSG :: "'m::msg) list × ('m list, 'm, unit, unit label) seqp) set"
where "σQMSG ≡ {([], ΓQMSG ())}"

abbreviation qmsg
  :: "'m::msg) list × ('m list, 'm, unit, unit label) seqp, 'm seq_action) automaton"
where
  "qmsg ≡ (| init = σQMSG, trans = seqp_sos ΓQMSG |)"

declare ΓQMSG.simps [simp del, code del]
lemmas ΓQMSG.simps [simp, code] = ΓQMSG.simps [simplified]

lemma σQMSG_not_empty [simp, intro]: "σQMSG ≠ {}"
⟨proof⟩

```

lemma σ_{QMSG_exists} [simp]: " $\exists qmsg\ q. (qmsg, q) \in \sigma_{QMSG}$ "
 <proof>

lemma $qmsg_wf$ [simp]: "wellformed Γ_{QMSG} "
 <proof>

lemmas $qmsg_labels_not_empty$ [simp] = labels_not_empty [OF $qmsg_wf$]

lemma $qmsg_control_within$ [simp]: "control_within Γ_{QMSG} (init $qmsg$)"
 <proof>

lemma $qmsg_simple_labels$ [simp]: "simple_labels Γ_{QMSG} "
 <proof>

lemma $qmsg_trans$: "trans $qmsg = seqp_sos\ \Gamma_{QMSG}$ "
 <proof>

lemma σ_{QMSG_labels} [simp]: " $(\xi, q) \in \sigma_{QMSG} \implies labels\ \Gamma_{QMSG}\ q = \{()\!-\!:\!0\}$ "
 <proof>

lemma $qmsg_proc_cases$ [dest]:
 fixes $p\ pn$
 shows " $p \in ctermsl\ (\Gamma_{QMSG}\ pn) \implies p \in ctermsl\ (\Gamma_{QMSG}\ ())$ "
 <proof>

declare

Γ_{QMSG_simps} [ctermenv]
 $qmsg_proc_cases$ [ctermcases]
 seq_invariant_ctermI [OF $qmsg_wf\ qmsg_control_within\ qmsg_simple_labels\ qmsg_trans, cterms_intros$]
 seq_step_invariant_ctermI [OF $qmsg_wf\ qmsg_control_within\ qmsg_simple_labels\ qmsg_trans, cterms_intros$]

end

23 Lifting rules for parallel compositions with QMSG

theory $Qmsg_Lifting$
 imports $Qmsg\ OAWN_SOS\ Inv_Cterms\ OAWN_Invariants$
 begin

lemma $oseq_no_change_on_send$:
 fixes $\sigma\ s\ a\ \sigma'\ s'$
 assumes " $((\sigma, s), a, (\sigma', s')) \in oseqp_sos\ \Gamma\ i$ "
 shows "case a of
 broadcast $m \implies \sigma'\ i = \sigma\ i$
 | groupcast $ips\ m \implies \sigma'\ i = \sigma\ i$
 | unicast $ips\ m \implies \sigma'\ i = \sigma\ i$
 | \neg unicast $ips \implies \sigma'\ i = \sigma\ i$
 | send $m \implies \sigma'\ i = \sigma\ i$
 | deliver $m \implies \sigma'\ i = \sigma\ i$
 | $_ \implies True$ "
 <proof>

lemma $qmsg_no_change_on_send_or_receive$:
 fixes $\sigma\ s\ a\ \sigma'\ s'$
 assumes " $((\sigma, s), a, (\sigma', s')) \in oparp_sos\ i\ (oseqp_sos\ \Gamma\ i)\ (seqp_sos\ \Gamma_{QMSG})$ "
 and " $a \neq \tau$ "
 shows " $\sigma'\ i = \sigma\ i$ "
 <proof>

lemma $qmsg_msgs_not_empty$:
 " $qmsg \Vdash onl\ \Gamma_{QMSG}\ (\lambda(msgs, l). l = ()\!-\!:\!1 \implies msgs \neq [])$ "
 <proof>

lemma $qmsg_send_from_queue$:

"qmsg $\models_A (\lambda((msgs, q), a, _). \text{sendmsg } (\lambda m. m \in \text{set } msgs) a)"$
 <proof>

lemma qmsg_queue_contents:

"qmsg $\models_A (\lambda((msgs, q), a, (msgs', q')). \text{case a of}$
 receive m $\Rightarrow \text{set } msgs' \subseteq \text{set } (msgs @ [m])$
 | _ $\Rightarrow \text{set } msgs' \subseteq \text{set } msgs)"$
 <proof>

lemma qmsg_send_receive_or_tau:

"qmsg $\models_A (\lambda(_, a, _). \exists m. a = \text{send } m \vee a = \text{receive } m \vee a = \tau)"$
 <proof>

lemma par_qmsg_oreachable:

assumes "(σ, ζ) \in oreachable (A $\langle\langle_i$ qmsg) (otherwith S {i} (orecvmsg R)) (other U {i}))"
 (is " $_ \in$ oreachable $_ ?owS _$ ")
 and pinv: "A \models_A (otherwith S {i} (orecvmsg R), other U {i} \rightarrow)
 globala ($\lambda(\sigma, _, \sigma'). U (\sigma i) (\sigma' i)$)"
 and ustutter: " $\bigwedge \xi. U \xi \xi$ "
 and sgivesu: " $\bigwedge \xi \xi'. S \xi \xi' \implies U \xi \xi'$ "
 and upreservesq: " $\bigwedge \sigma \sigma' m. [\forall j. U (\sigma j) (\sigma' j); R \sigma m] \implies R \sigma' m$ "
 shows "($\sigma, \text{fst } \zeta$) \in oreachable A $_ ?owS$ (other U {i})"
 $\wedge \text{snd } \zeta \in \text{reachable qmsg (recvmsg (R } \sigma))$
 $\wedge (\forall m \in \text{set } (\text{fst } (\text{snd } \zeta))). R \sigma m"$
 <proof>

lemma par_qmsg_oreachable_statelessasm:

assumes "(σ, ζ) \in oreachable (A $\langle\langle_i$ qmsg)
 ($\lambda \sigma _.$ orecvmsg ($\lambda _.$ R) σ) (other ($\lambda _ _.$ True) {i}))"
 and ustutter: " $\bigwedge \xi. U \xi \xi$ "
 shows "($\sigma, \text{fst } \zeta$) \in oreachable A ($\lambda \sigma _.$ orecvmsg ($\lambda _.$ R) σ) (other ($\lambda _ _.$ True) {i})"
 $\wedge \text{snd } \zeta \in \text{reachable qmsg (recvmsg R)}$
 $\wedge (\forall m \in \text{set } (\text{fst } (\text{snd } \zeta))). R m"$
 <proof>

lemma lift_into_qmsg:

assumes "A \models (otherwith S {i} (orecvmsg R), other U {i} \rightarrow) global P"
 and " $\bigwedge \xi. U \xi \xi$ "
 and " $\bigwedge \xi \xi'. S \xi \xi' \implies U \xi \xi'$ "
 and " $\bigwedge \sigma \sigma' m. [\forall j. U (\sigma j) (\sigma' j); R \sigma m] \implies R \sigma' m$ "
 and "A \models_A (otherwith S {i} (orecvmsg R), other U {i} \rightarrow)
 globala ($\lambda(\sigma, _, \sigma'). U (\sigma i) (\sigma' i)$)"
 shows "A $\langle\langle_i$ qmsg \models (otherwith S {i} (orecvmsg R), other U {i} \rightarrow) global P"
 <proof>

lemma lift_step_into_qmsg:

assumes inv: "A \models_A (otherwith S {i} (orecvmsg R), other U {i} \rightarrow) globala P"
 and ustutter: " $\bigwedge \xi. U \xi \xi$ "
 and sgivesu: " $\bigwedge \xi \xi'. S \xi \xi' \implies U \xi \xi'$ "
 and upreservesq: " $\bigwedge \sigma \sigma' m. [\forall j. U (\sigma j) (\sigma' j); R \sigma m] \implies R \sigma' m$ "
 and self_sync: "A \models_A (otherwith S {i} (orecvmsg R), other U {i} \rightarrow)
 globala ($\lambda(\sigma, _, \sigma'). U (\sigma i) (\sigma' i)$)"
 and recv_stutter: " $\bigwedge \sigma \sigma' m. [\forall j. U (\sigma j) (\sigma' j); \sigma' i = \sigma i] \implies P (\sigma, \text{receive } m, \sigma')$ "
 and receive_right: " $\bigwedge \sigma \sigma' m. P (\sigma, \text{receive } m, \sigma') \implies P (\sigma, \tau, \sigma')$ "
 shows "A $\langle\langle_i$ qmsg \models_A (otherwith S {i} (orecvmsg R), other U {i} \rightarrow) globala P"
 (is " $_ \models_A$ ($_ ?owS, ?U \rightarrow$) $_$ ")
 <proof>

lemma lift_step_into_qmsg_statelessasm:

assumes "A \models_A ($\lambda \sigma _.$ orecvmsg ($\lambda _.$ R) σ , other ($\lambda _ _.$ True) {i} \rightarrow) globala P"
 and " $\bigwedge \sigma \sigma' m. \sigma' i = \sigma i \implies P (\sigma, \text{receive } m, \sigma')$ "
 and " $\bigwedge \sigma \sigma' m. P (\sigma, \text{receive } m, \sigma') \implies P (\sigma, \tau, \sigma')$ "
 shows "A $\langle\langle_i$ qmsg \models_A ($\lambda \sigma _.$ orecvmsg ($\lambda _.$ R) σ , other ($\lambda _ _.$ True) {i} \rightarrow) globala P"

<proof>

end

24 Transfer open results onto closed models

theory *OClosed_Transfer*

imports *Closed OClosed_Lifting*

begin

locale *openproc* =

fixes *np* :: "*ip* \Rightarrow (*'s*, (*'m*::*msg*) *seq_action*) automaton"

and *onp* :: "*ip* \Rightarrow ((*ip* \Rightarrow *'g*) \times *'l*, *'m seq_action*) automaton"

and *sr* :: "*'s* \Rightarrow (*'g* \times *'l*)"

assumes *init*: "{ (σ , ζ) | $\sigma \zeta s$. *s* \in *init* (*np i*)

\wedge (σi , ζ) = *sr s*

\wedge ($\forall j. j \neq i \rightarrow \sigma j \in$ (*fst* \circ *sr*) ' *init* (*np j*)) } \subseteq *init* (*onp i*)"

and *init_notempty*: " $\forall j. \text{init } (np j) \neq \{\}$ "

and *trans*: " $\wedge s a s' \sigma \sigma'. \llbracket \sigma i = \text{fst } (sr s);$

$\sigma' i = \text{fst } (sr s');$

$(s, a, s') \in \text{trans } (np i) \rrbracket$

$\implies ((\sigma, \text{snd } (sr s)), a, (\sigma', \text{snd } (sr s')))) \in \text{trans } (onp i)"$

begin

lemma *init_pnet_p_NodeS*:

assumes "*NodeS i s R* \in *init* (*pnet np p*)"

shows "*p* = $\langle i; R \rangle$ "

<proof>

lemma *init_pnet_p_SubnetS*:

assumes "*SubnetS s1 s2* \in *init* (*pnet np p*)"

obtains *p1 p2* where "*p* = (*p1* \parallel *p2*)"

and "*s1* \in *init* (*pnet np p1*)"

and "*s2* \in *init* (*pnet np p2*)"

<proof>

lemma *init_pnet_fst_sr_netgmap*:

assumes "*s* \in *init* (*pnet np p*)"

and "*i* \in *net_ips s*"

and "*wf_net_tree p*"

shows "*the* (*fst* (*netgmap sr s*) *i*) \in (*fst* \circ *sr*) ' *init* (*np i*)"

<proof>

lemma *init_lifted*:

assumes "*wf_net_tree p*"

shows "{ (σ , *snd* (*netgmap sr s*)) | σs . *s* \in *init* (*pnet np p*)

\wedge ($\forall i. \text{if } i \in \text{net_tree_ips } p \text{ then } \sigma i = \text{the } (\text{fst } (\text{netgmap } sr s) i)$

$\text{else } \sigma i \in (\text{fst } \circ sr) ' \text{init } (np i) \}$ \subseteq *init* (*opnet onp p*)"

<proof>

lemma *init_pnet_opnet [elim]*:

assumes "*wf_net_tree p*"

and "*s* \in *init* (*pnet np p*)"

shows "*netgmap sr s* \in *netmask* (*net_tree_ips p*) ' *init* (*opnet onp p*)"

<proof>

lemma *transfer_connect*:

assumes " $(s, \text{connect}(i, i'), s') \in \text{trans } (pnet np n)"$

and "*s* \in *reachable* (*pnet np n*) *TT*"

and "*netgmap sr s* = *netmask* (*net_tree_ips n*) (σ , ζ)"

and "*wf_net_tree n*"

obtains $\sigma' \zeta'$ where " $((\sigma, \zeta), \text{connect}(i, i'), (\sigma', \zeta')) \in \text{trans } (opnet onp n)"$

and " $\forall j. j \notin \text{net_ips } \zeta \rightarrow \sigma' j = \sigma j"$

and "*netgmap sr s'* = *netmask* (*net_tree_ips n*) (σ' , ζ')"

<proof>

lemma transfer_disconnect:

assumes "(s, disconnect(i, i'), s') ∈ trans (pnet np n)"
and "s ∈ reachable (pnet np n) TT"
and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
and "wf_net_tree n"

obtains σ' ζ' where "((σ, ζ), disconnect(i, i'), (σ', ζ')) ∈ trans (opnet onp n)"
and "∀j. j ∉ net_ips ζ → σ' j = σ j"
and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ)'"

<proof>

lemma transfer_tau:

assumes "(s, τ, s') ∈ trans (pnet np n)"
and "s ∈ reachable (pnet np n) TT"
and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
and "wf_net_tree n"

obtains σ' ζ' where "((σ, ζ), τ, (σ', ζ')) ∈ trans (opnet onp n)"
and "∀j. j ∉ net_ips ζ → σ' j = σ j"
and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ)'"

<proof>

lemma transfer_deliver:

assumes "(s, i:deliver(d), s') ∈ trans (pnet np n)"
and "s ∈ reachable (pnet np n) TT"
and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
and "wf_net_tree n"

obtains σ' ζ' where "((σ, ζ), i:deliver(d), (σ', ζ')) ∈ trans (opnet onp n)"
and "∀j. j ∉ net_ips ζ → σ' j = σ j"
and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ)'"

<proof>

lemma transfer_arrive':

assumes "(s, H¬K:arrive(m), s') ∈ trans (pnet np n)"
and "s ∈ reachable (pnet np n) TT"
and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ)'"
and "wf_net_tree n"

shows "((σ, ζ), H¬K:arrive(m), (σ', ζ')) ∈ trans (opnet onp n)"

<proof>

lemma transfer_arrive:

assumes "(s, H¬K:arrive(m), s') ∈ trans (pnet np n)"
and "s ∈ reachable (pnet np n) TT"
and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
and "wf_net_tree n"

obtains σ' ζ' where "((σ, ζ), H¬K:arrive(m), (σ', ζ')) ∈ trans (opnet onp n)"
and "∀j. j ∉ net_ips ζ → σ' j = σ j"
and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ)'"

<proof>

lemma transfer_cast:

assumes "(s, mR:*cast(m), s') ∈ trans (pnet np n)"
and "s ∈ reachable (pnet np n) TT"
and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
and "wf_net_tree n"

obtains σ' ζ' where "((σ, ζ), mR:*cast(m), (σ', ζ')) ∈ trans (opnet onp n)"
and "∀j. j ∉ net_ips ζ → σ' j = σ j"
and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ)'"

<proof>

lemma transfer_pnet_action:

assumes "s ∈ reachable (pnet np n) TT"
and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"

```

    and "wf_net_tree n"
    and "(s, a, s') ∈ trans (pnet np n)"
obtains σ' ζ' where "((σ, ζ), a, (σ', ζ')) ∈ trans (opnet onp n)"
    and "∀j. j ∉ net_ips ζ → σ' j = σ j"
    and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
⟨proof⟩

lemma transfer_action_pnet_closed:
assumes "(s, a, s') ∈ trans (closed (pnet np n))"
obtains a' where "(s, a', s') ∈ trans (pnet np n)"
    and "∧σ ζ σ' ζ'. [ ((σ, ζ), a', (σ', ζ')) ∈ trans (opnet onp n);
    (∀j. j ∉ net_ips ζ → σ' j = σ j) ]
    ⇒ ((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))"
⟨proof⟩

lemma transfer_action:
assumes "s ∈ reachable (closed (pnet np n)) TT"
    and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
    and "wf_net_tree n"
    and "(s, a, s') ∈ trans (closed (pnet np n))"
obtains σ' ζ' where "((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))"
    and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
⟨proof⟩

lemma pnet_reachable_transfer':
assumes "wf_net_tree n"
    and "s ∈ reachable (closed (pnet np n)) TT"
shows "netgmap sr s ∈ netmask (net_tree_ips n) 'oreachable (oclosed (opnet onp n)) (λ_ _ . True)
U"
    (is " _ ∈ ?f ' ?oreachable n")
⟨proof⟩

definition
someinit :: "nat ⇒ 'g"
where
"someinit i ≡ SOME x. x ∈ (fst o sr) ' init (np i)"

definition
initmissing :: "(nat ⇒ 'g option) × 'a ⇒ (nat ⇒ 'g) × 'a"
where
"initmissing σ = (λi. case (fst σ) i of None ⇒ someinit i | Some s ⇒ s, snd σ)"

lemma initmissing_def':
"initmissing = apfst (default someinit)"
⟨proof⟩

lemma netmask_initmissing_netgmap:
"netmask (net_ips s) (initmissing (netgmap sr s)) = netgmap sr s"
⟨proof⟩

lemma snd_initmissing [simp]:
"snd (initmissing x) = snd x"
⟨proof⟩

lemma initmissing_snd_netgmap [simp]:
assumes "initmissing (netgmap sr s) = (σ, ζ)"
shows "snd (netgmap sr s) = ζ"
⟨proof⟩

lemma in_net_ips_fst_init_missing [simp]:
assumes "i ∈ net_ips s"
shows "fst (initmissing (netgmap sr s)) i = the (fst (netgmap sr s) i)"
⟨proof⟩

```

```

lemma not_in_net_ips_fst_init_missing [simp]:
  assumes "i ∉ net_ips s"
  shows "fst (initmissing (netgmap sr s)) i = someinit i"
  ⟨proof⟩

lemma initmissing_oreachable_netmask [elim]:
  assumes "initmissing (netgmap sr s) ∈ oreachable (oclosed (opnet onp n)) (λ_ _ . True) U"
    (is "_ ∈ ?oreachable n")
  and "net_ips s = net_tree_ips n"
  shows "netgmap sr s ∈ netmask (net_tree_ips n) ' ?oreachable n"
  ⟨proof⟩

lemma pnet_reachable_transfer:
  assumes "wf_net_tree n"
  and "s ∈ reachable (closed (pnet np n)) TT"
  shows "initmissing (netgmap sr s) ∈ oreachable (oclosed (opnet onp n)) (λ_ _ . True) U"
    (is "_ ∈ ?oreachable n")
  ⟨proof⟩

definition
  netglobal :: "(nat ⇒ 'g) ⇒ bool ⇒ 's net_state ⇒ bool"
where
  "netglobal P ≡ (λs. P (fst (initmissing (netgmap sr s))))"

lemma netglobalsimp [simp]:
  "netglobal P s = P (fst (initmissing (netgmap sr s)))"
  ⟨proof⟩

lemma netglobale [elim]:
  assumes "netglobal P s"
  and "∧σ. [ P σ; fst (initmissing (netgmap sr s)) = σ ] ⇒ Q σ"
  shows "netglobal Q s"
  ⟨proof⟩

lemma netglobal_weakenE [elim]:
  assumes "p ⊨ netglobal P"
  and "∧σ. P σ ⇒ Q σ"
  shows "p ⊨ netglobal Q"
  ⟨proof⟩

lemma close_opnet:
  assumes "wf_net_tree n"
  and "oclosed (opnet onp n) ⊨ (λ_ _ . True, U →) global P"
  shows "closed (pnet np n) ⊨ netglobal P"
  ⟨proof⟩

end

locale openproc_parq =
  op?: openproc np onp sr for np :: "'ip ⇒ ('s, ('m::msg) seq_action) automaton" and onp sr
  + fixes qp :: "('t, 'm seq_action) automaton"
  assumes init_qp_notempty: "init qp ≠ {}"

sublocale openproc_parq ⊆ openproc "λi. np i ⟨⟨ qp"
  "λi. onp i ⟨⟨i qp"
  "λ(p, q). (fst (sr p), (snd (sr p), q))"
  ⟨proof⟩

end

```

25 Import all AWN-related theories

```
theory AWN_Main
```

```

imports AWN_SOS AWN_SOS_Labels OAWN_SOS_Labels AWN_Invariants
         OAWN_Convert OClosed_Transfer
begin

end

```

26 Simple toy example

```

theory Toy
imports Main AWN_Main Qmsg_Lifting
begin

```

26.1 Messages used in the protocol

```

datatype msg =
  Pkt data ip
  | Newpkt data ip

instantiation msg :: msg
begin
definition newpkt_def [simp]: "newpkt  $\equiv$   $\lambda(d, did). \text{Newpkt } d \text{ did}"$ 
definition eq_newpkt_def: "eq_newpkt m  $\equiv$  case m of Newpkt d did  $\Rightarrow$  True | _  $\Rightarrow$  False"

instance <proof>
end

definition pkt :: "nat  $\times$  nat  $\Rightarrow$  msg"
where "pkt  $\equiv$   $\lambda(no, sid). \text{Pkt } no \text{ sid}"$ 

lemma pkt_simp [simp]:
  "pkt(no, sid) = Pkt no sid"
  <proof>

lemma not_eq_newpkt_pkt [simp]: " $\neg$ eq_newpkt (Pkt no sid)"
  <proof>

```

26.2 Protocol model

```

record state =
  id      :: "nat"
  no      :: "nat"
  nhid    :: "nat"

  msg     :: "msg"
  num     :: "nat"
  sid     :: "nat"

abbreviation toy_init :: "ip  $\Rightarrow$  state"
where "toy_init i  $\equiv$  (
  id = i,
  no = 0,
  nhid = i,

  msg = (SOME x. True),
  num = (SOME x. True),
  sid = (SOME x. True)
)"

lemma some_neq_not_eq [simp]: " $\neg((\text{SOME } x :: \text{nat}. x \neq i) = i)"$ 
  <proof>

definition clear_locals :: "state  $\Rightarrow$  state"
where "clear_locals  $\xi = \xi$  (
  msg := (SOME x. True),

```

```

    num    := (SOME x. True),
    sid    := (SOME x. True)
  )"

lemma clear_locals_but_not_globals [simp]:
  "id (clear_locals  $\xi$ ) = id  $\xi$ "
  "no (clear_locals  $\xi$ ) = no  $\xi$ "
  "nhid (clear_locals  $\xi$ ) = nhid  $\xi$ "
  <proof>

definition is_newpkt
where "is_newpkt  $\xi \equiv$  case msg  $\xi$  of
      Newpkt data did  $\Rightarrow$  {  $\xi$ (num := data) }
      | _  $\Rightarrow$  {}"

definition is_pkt
where "is_pkt  $\xi \equiv$  case msg  $\xi$  of
      Pkt num' sid'  $\Rightarrow$  {  $\xi$ ( num := num', sid := sid' ) }
      | _  $\Rightarrow$  {}"

lemmas is_msg_defs =
  is_pkt_def is_newpkt_def

lemma is_msg_inv_id [simp]:
  " $\xi' \in$  is_pkt  $\xi \implies$  id  $\xi' =$  id  $\xi$ "
  " $\xi' \in$  is_newpkt  $\xi \implies$  id  $\xi' =$  id  $\xi$ "
  <proof>

lemma is_msg_inv_sid [simp]:
  " $\xi' \in$  is_newpkt  $\xi \implies$  sid  $\xi' =$  sid  $\xi$ "
  <proof>

lemma is_msg_inv_no [simp]:
  " $\xi' \in$  is_pkt  $\xi \implies$  no  $\xi' =$  no  $\xi$ "
  " $\xi' \in$  is_newpkt  $\xi \implies$  no  $\xi' =$  no  $\xi$ "
  <proof>

lemma is_msg_inv_nhid [simp]:
  " $\xi' \in$  is_pkt  $\xi \implies$  nhid  $\xi' =$  nhid  $\xi$ "
  " $\xi' \in$  is_newpkt  $\xi \implies$  nhid  $\xi' =$  nhid  $\xi$ "
  <proof>

lemma is_msg_inv_msg [simp]:
  " $\xi' \in$  is_pkt  $\xi \implies$  msg  $\xi' =$  msg  $\xi$ "
  " $\xi' \in$  is_newpkt  $\xi \implies$  msg  $\xi' =$  msg  $\xi$ "
  <proof>

datatype pseqp =
  PToy

fun nat_of_seqp :: "pseqp  $\Rightarrow$  nat"
where
  "nat_of_seqp PToy = 1"

instantiation "pseqp" :: ord
begin
definition less_eq_seqp [iff]: "l1  $\leq$  l2 = (nat_of_seqp l1  $\leq$  nat_of_seqp l2)"
definition less_seqp [iff]: "l1 < l2 = (nat_of_seqp l1 < nat_of_seqp l2)"
instance <proof>
end

abbreviation Toy
where
  "Toy  $\equiv$   $\lambda$ _. [[clear_locals]] call(PToy)"

```

```

fun  $\Gamma_{TOY} :: \text{"(state, msg, pseq, pseq label) seqp\_env"}$ 
where
  " $\Gamma_{TOY}$  PToy = labelled PToy (
    receive( $\lambda$ msg'  $\xi$ .  $\xi$  ( $\mid$  msg := msg'  $\mid$ )).
    [ $\xi$ .  $\xi$  ( $\mid$ nhid := id  $\xi$ )])
    (  $\langle$ is_newpkt $\rangle$ 
      (
        [ $\xi$ .  $\xi$  ( $\mid$ no := max (no  $\xi$ ) (num  $\xi$ ) $\mid$ )]
        broadcast( $\lambda$  $\xi$ . pkt(no  $\xi$ , id  $\xi$ )). Toy()
      )
     $\oplus$   $\langle$ is_pkt $\rangle$ 
    (
      ( $\xi$ . num  $\xi$  > no  $\xi$ )
      [ $\xi$ .  $\xi$  ( $\mid$ no := num  $\xi$ )])
      [ $\xi$ .  $\xi$  ( $\mid$ nhid := sid  $\xi$ )])
      broadcast( $\lambda$  $\xi$ . pkt(no  $\xi$ , id  $\xi$ )). Toy()
     $\oplus$  ( $\xi$ . num  $\xi$   $\leq$  no  $\xi$ )
      Toy()
    )
  )"

```

```

declare  $\Gamma_{TOY}$ .simps [simp del, code del]
lemmas  $\Gamma_{TOY}$ _simps [simp, code] =  $\Gamma_{TOY}$ .simps [simplified]

```

```

fun  $\Gamma_{TOY\_skeleton}$ 
where " $\Gamma_{TOY\_skeleton}$  PToy = seqp_skeleton ( $\Gamma_{TOY}$  PToy)"

```

```

lemma  $\Gamma_{TOY\_skeleton\_wf}$  [simp]:
  "wellformed  $\Gamma_{TOY\_skeleton}$ "
   $\langle$ proof $\rangle$ 

```

```

declare  $\Gamma_{TOY\_skeleton}$ .simps [simp del, code del]
lemmas  $\Gamma_{TOY\_skeleton}$ _simps [simp, code] =  $\Gamma_{TOY\_skeleton}$ .simps [simplified  $\Gamma_{TOY}$ _simps seqp_skeleton.simps]

```

```

lemma toy_proc_cases [dest]:
  fixes p pn
  assumes "p  $\in$  cterms1 ( $\Gamma_{TOY}$  pn)"
  shows "p  $\in$  cterms1 ( $\Gamma_{TOY}$  PToy)"
   $\langle$ proof $\rangle$ 

```

```

definition  $\sigma_{TOY} :: \text{"ip} \Rightarrow \text{(state} \times \text{(state, msg, pseq, pseq label) seqp) set"}$ 
where " $\sigma_{TOY}$  i  $\equiv$  {(toy_init i,  $\Gamma_{TOY}$  PToy)}"

```

```

abbreviation ptoy
  :: "ip  $\Rightarrow$  (state  $\times$  (state, msg, pseq, pseq label) seqp, msg seq_action) automaton"
where
  "ptoy i  $\equiv$  ( $\mid$  init =  $\sigma_{TOY}$  i, trans = seqp_sos  $\Gamma_{TOY}$   $\mid$ )"

```

```

lemma toy_trans: "trans (ptoy i) = seqp_sos  $\Gamma_{TOY}$ "
   $\langle$ proof $\rangle$ 

```

```

lemma toy_control_within [simp]: "control_within  $\Gamma_{TOY}$  (init (ptoy i))"
   $\langle$ proof $\rangle$ 

```

```

lemma toy_wf [simp]:
  "wellformed  $\Gamma_{TOY}$ "
   $\langle$ proof $\rangle$ 

```

```

lemmas toy_labels_not_empty [simp] = labels_not_empty [OF toy_wf]

```

```

lemma toy_ex_label [intro]: " $\exists$ l. l $\in$ labels  $\Gamma_{TOY}$  p"
   $\langle$ proof $\rangle$ 

```

```

lemma toy_ex_labelE [elim]:
  assumes " $\forall l \in \text{labels } \Gamma_{TOY} p. P l p$ "
  and " $\exists p l. P l p \implies Q$ "
  shows "Q"
<proof>

```

```

lemma toy_simple_labels [simp]: "simple_labels  $\Gamma_{TOY}$ "
<proof>

```

```

lemma  $\sigma_{TOY}$ _labels [simp]: " $(\xi, p) \in \sigma_{TOY} i \implies \text{labels } \Gamma_{TOY} p = \{PToy-:0\}$ "
<proof>

```

By default, we no longer let the simplifier descend into process terms.

```

declare seqp_congs [cong]

```

```

declare

```

```

   $\Gamma_{TOY}$ _simps [cterms_env]
  toy_proc_cases [ctermssl_cases]
  seq_invariant_ctermsI [OF toy_wf toy_control_within toy_simple_labels toy_trans, cterms_intros]
  seq_step_invariant_ctermsI [OF toy_wf toy_control_within toy_simple_labels toy_trans, cterms_intros]

```

26.3 Define an open version of the protocol

```

definition  $\sigma_{OTOY} :: ((ip \Rightarrow state) \times ((state, msg, pseq, pseq label) seq)) \text{ set}$ "
where " $\sigma_{OTOY} \equiv \{(toy\_init, \Gamma_{TOY} PToy)\}$ "

```

```

abbreviation optoy

```

```

  :: " $ip \Rightarrow ((ip \Rightarrow state) \times (state, msg, pseq, pseq label) seq, msg seq\_action) \text{ automaton}$ "

```

```

where

```

```

  "optoy i  $\equiv$  (| init =  $\sigma_{OTOY}$ , trans = oseqp_sos  $\Gamma_{TOY} i$  |)"

```

```

lemma initiali_toy [intro!, simp]: "initiali i (init (optoy i)) (init (ptoy i))"
<proof>

```

```

lemma oaadv_control_within [simp]: "control_within  $\Gamma_{TOY}$  (init (optoy i))"
<proof>

```

```

lemma  $\sigma_{OTOY}$ _labels [simp]: " $(\sigma, p) \in \sigma_{OTOY} \implies \text{labels } \Gamma_{TOY} p = \{PToy-:0\}$ "
<proof>

```

```

lemma otoy_trans: "trans (optoy i) = oseqp_sos  $\Gamma_{TOY} i$ "
<proof>

```

```

declare

```

```

  oseq_invariant_ctermsI [OF toy_wf oaadv_control_within toy_simple_labels otoy_trans, cterms_intros]
  oseq_step_invariant_ctermsI [OF toy_wf oaadv_control_within toy_simple_labels otoy_trans, cterms_intros]

```

26.4 Predicates

```

definition msg_sender :: " $msg \Rightarrow ip$ "

```

```

where "msg_sender m  $\equiv$  case m of Pkt _ ipc  $\Rightarrow$  ipc"

```

```

lemma msg_sender_simps [simp]:

```

```

  " $\bigwedge d \text{ did } sid. \text{msg\_sender } (Pkt d sid) = sid$ "

```

```

<proof>

```

```

abbreviation not_Pkt :: " $msg \Rightarrow \text{bool}$ "

```

```

where "not_Pkt m  $\equiv$  case m of Pkt _ _  $\Rightarrow$  False | _  $\Rightarrow$  True"

```

```

definition nos_inc :: " $state \Rightarrow state \Rightarrow \text{bool}$ "

```

```

where "nos_inc  $\xi \xi' \equiv$  (no  $\xi \leq$  no  $\xi'$ )"

```

definition `msg_ok` :: "(ip \Rightarrow state) \Rightarrow msg \Rightarrow bool"
where "msg_ok σ m \equiv case m of Pkt num' sid' \Rightarrow num' \leq no (σ sid') | _ \Rightarrow True"

lemma `msg_okI` [intro]:
assumes " \bigwedge num' sid'. m = Pkt num' sid' \implies num' \leq no (σ sid')"
shows "msg_ok σ m"
 \langle proof \rangle

lemma `msg_ok_Pkt` [simp]:
"msg_ok σ (Pkt data src) = (data \leq no (σ src))"
 \langle proof \rangle

lemma `msg_ok_pkt` [simp]:
"msg_ok σ (pkt(data, src)) = (data \leq no (σ src))"
 \langle proof \rangle

lemma `msg_ok_Newpkt` [simp]:
"msg_ok σ (Newpkt data dst)"
 \langle proof \rangle

lemma `msg_ok_newpkt` [simp]:
"msg_ok σ (newpkt(data, dst))"
 \langle proof \rangle

26.5 Sequential Invariants

lemma `seq_no_leq_num`:
"ptoy i \models_{onl} Γ_{TOY} ($\lambda(\xi, l). l \in \{PToy-:7..PToy-:8\} \longrightarrow no \xi \leq num \xi$)"
 \langle proof \rangle

lemma `seq_nos_incs`:
"ptoy i \models_A $onll$ Γ_{TOY} ($\lambda((\xi, _), _, (\xi', _)). nos_inc \xi \xi'$)"
 \langle proof \rangle

lemma `seq_nos_incs'`:
"ptoy i \models_A ($\lambda((\xi, _), _, (\xi', _)). nos_inc \xi \xi'$)"
 \langle proof \rangle

lemma `sender_ip_valid`:
"ptoy i \models_A $onll$ Γ_{TOY} ($\lambda((\xi, _), a, _). anycast (\lambda m. msg_sender m = id \xi) a$)"
 \langle proof \rangle

lemma `id_constant`:
"ptoy i \models_{onl} Γ_{TOY} ($\lambda(\xi, _). id \xi = i$)"
 \langle proof \rangle

lemma `nhid_eq_id`:
"ptoy i \models_{onl} Γ_{TOY} ($\lambda(\xi, l). l \in \{PToy-:2..PToy-:8\} \longrightarrow nhid \xi = id \xi$)"
 \langle proof \rangle

lemma `seq_msg_ok`:
"ptoy i \models_A $onll$ Γ_{TOY} ($\lambda((\xi, _), a, _). anycast (\lambda m. case m of Pkt num' sid' \Rightarrow num' = no $\xi \wedge sid' = i$ | _ \Rightarrow True) a)$ "
 \langle proof \rangle

lemma `nhid_eq_i`:
"ptoy i \models_{onl} Γ_{TOY} ($\lambda(\xi, l). l \in \{PToy-:2..PToy-:8\} \longrightarrow nhid \xi = i$)"
 \langle proof \rangle

26.6 Global Invariants

lemma `nos_incD` [dest]:
assumes "nos_inc $\xi \xi'$ "
shows "no $\xi \leq no \xi'$ "

<proof>

lemma nos_inc_simp [simp]:

"nos_inc $\xi \xi' = (\text{no } \xi \leq \text{no } \xi')$ "

<proof>

lemmas oseq_nos_incs =

open_seq_step_invariant [OF seq_nos_incs initiali_toy otoy_trans toy_trans,
simplified seqll_onll_swap]

lemmas oseq_no_leq_num =

open_seq_invariant [OF seq_no_leq_num initiali_toy otoy_trans toy_trans,
simplified seql_onl_swap]

lemma all_nos_inc:

shows "optoy $i \models_A$ (otherwith nos_inc $\{i\}$ S,
other nos_inc $\{i\} \rightarrow$)

onll $\Gamma_{TOY} (\lambda((\sigma, _), a, (\sigma', _)). (\forall j. \text{nos_inc } (\sigma j) (\sigma' j)))$ "

<proof>

lemma oreceived_msg_inv:

assumes other: " $\bigwedge \sigma \sigma' m. \llbracket P \sigma m; \text{other } Q \{i\} \sigma \sigma' \rrbracket \implies P \sigma' m$ "

and local: " $\bigwedge \sigma m. P \sigma m \implies P (\sigma(i := \sigma i(\text{msg} := m))) m$ "

shows "optoy $i \models$ (otherwith $Q \{i\}$ (orecvmsg P), other $Q \{i\} \rightarrow$)

onl $\Gamma_{TOY} (\lambda(\sigma, l). l \in \{PToy-:1\} \longrightarrow P \sigma (\text{msg } (\sigma i)))$ "

<proof>

lemma msg_ok_other_nos_inc [elim]:

assumes "msg_ok σm "

and "other nos_inc $\{i\} \sigma \sigma'$ "

shows "msg_ok $\sigma' m$ "

<proof>

lemma msg_ok_no_leq_no [simp, elim]:

assumes "msg_ok σm "

and " $\forall j. \text{no } (\sigma j) \leq \text{no } (\sigma' j)$ "

shows "msg_ok $\sigma' m$ "

<proof>

lemma oreceived_msg_ok:

"optoy $i \models$ (otherwith nos_inc $\{i\}$ (orecvmsg msg_ok),
other nos_inc $\{i\} \rightarrow$)

onl $\Gamma_{TOY} (\lambda(\sigma, l). l \in \{PToy-:1..\} \longrightarrow \text{msg_ok } \sigma (\text{msg } (\sigma i)))$ "

(is " $_ \models (?S, ?U \rightarrow) _$ ")

<proof>

lemma is_pkt_handler_num_leq_no:

shows "optoy $i \models$ (otherwith nos_inc $\{i\}$ (orecvmsg msg_ok),
other nos_inc $\{i\} \rightarrow$)

onl $\Gamma_{TOY} (\lambda(\sigma, l). l \in \{PToy-:6..PToy-:10\} \longrightarrow \text{num } (\sigma i) \leq \text{no } (\sigma (\text{sid } (\sigma i))))$ "

<proof>

lemmas oseq_id_constant =

open_seq_invariant [OF id_constant initiali_toy otoy_trans toy_trans,
simplified seql_onl_swap]

lemmas oseq_nhid_eq_i =

open_seq_invariant [OF nhid_eq_i initiali_toy otoy_trans toy_trans,
simplified seql_onl_swap]

lemmas oseq_nhid_eq_id =

open_seq_invariant [OF nhid_eq_id initiali_toy otoy_trans toy_trans,
simplified seql_onl_swap]

```

lemma oseq_bigger_than_next:
  shows "optoy i  $\models$  (otherwith nos_inc {i} (orecvmsg msg_ok),
    other nos_inc {i}  $\rightarrow$ ) global ( $\lambda\sigma. \text{no } (\sigma \text{ i}) \leq \text{no } (\sigma \text{ (nhid } (\sigma \text{ i})))$ )"
    (is "_  $\models$  (?S, ?U  $\rightarrow$ ) ?P")
  <proof>

```

```

lemma anycast_weakenE [elim]:
  assumes "anycast P a"
    and " $\bigwedge m. P m \implies Q m$ "
  shows "anycast Q a"
  <proof>

```

```

lemma oseq_msg_ok:
  "optoy i  $\models_A$  (act TT, other U {i}  $\rightarrow$ ) globala ( $\lambda(\sigma, a, _). \text{anycast } (\text{msg\_ok } \sigma) a$ )"
  <proof>

```

26.7 Lifting

```

lemma opar_bigger_than_next:
  shows "optoy i  $\langle\langle_i \text{qmsg} \models$  (otherwith nos_inc {i} (orecvmsg msg_ok),
    other nos_inc {i}  $\rightarrow$ ) global ( $\lambda\sigma. \text{no } (\sigma \text{ i}) \leq \text{no } (\sigma \text{ (nhid } (\sigma \text{ i})))$ )"
  <proof>

```

```

lemma onode_bigger_than_next:
  " $\langle i : \text{optoy } i \langle\langle_i \text{qmsg} : R_i \rangle_o$ 
   $\models$  (otherwith nos_inc {i} (oarrivmsg msg_ok), other nos_inc {i}  $\rightarrow$ )
  global ( $\lambda\sigma. \text{no } (\sigma \text{ i}) \leq \text{no } (\sigma \text{ (nhid } (\sigma \text{ i})))$ )"
  <proof>

```

```

lemma node_local_nos_inc:
  " $\langle i : \text{optoy } i \langle\langle_i \text{qmsg} : R_i \rangle_o \models_A$  ( $\lambda\sigma \_ . \text{oarrivmsg } (\lambda\_ \_ . \text{True}) \sigma, \text{other } (\lambda\_ \_ . \text{True}) \{i\} \rightarrow$ )
  globala ( $\lambda(\sigma, \_ , \sigma'). \text{nos\_inc } (\sigma \text{ i}) (\sigma' \text{ i})$ )"
  <proof>

```

```

lemma opnet_bigger_than_next:
  "opnet ( $\lambda i. \text{optoy } i \langle\langle_i \text{qmsg} \rangle n$ 
   $\models$  (otherwith nos_inc (net_tree_ips n) (oarrivmsg msg_ok),
    other nos_inc (net_tree_ips n)  $\rightarrow$ )
  global ( $\lambda\sigma. \forall i \in \text{net\_tree\_ips } n. \text{no } (\sigma \text{ i}) \leq \text{no } (\sigma \text{ (nhid } (\sigma \text{ i})))$ )"
  <proof>

```

```

lemma ocnet_bigger_than_next:
  "oclosed (opnet ( $\lambda i. \text{optoy } i \langle\langle_i \text{qmsg} \rangle n$ 
   $\models$  ( $\lambda\_ \_ \_ . \text{True}, \text{other nos\_inc } (\text{net\_tree\_ips } n) \rightarrow$ )
  global ( $\lambda\sigma. \forall i \in \text{net\_tree\_ips } n. \text{no } (\sigma \text{ i}) \leq \text{no } (\sigma \text{ (nhid } (\sigma \text{ i})))$ )"
  <proof>

```

26.8 Transfer

definition

```

initmissing :: "(nat  $\Rightarrow$  state option)  $\times$  'a  $\Rightarrow$  (nat  $\Rightarrow$  state)  $\times$  'a"

```

where

```

"initmissing  $\sigma = (\lambda i. \text{case } (\text{fst } \sigma) \text{ i of None } \Rightarrow \text{toy\_init } i \mid \text{Some } s \Rightarrow s, \text{snd } \sigma)"$ 

```

```

lemma not_in_net_ips_fst_init_missing [simp]:
  assumes "i  $\notin$  net_ips  $\sigma$ "
  shows "fst (initmissing (netgmap fst  $\sigma$ )) i = toy_init i"
  <proof>

```

```

lemma fst_initmissing_netgmap_pair_fst [simp]:
  "fst (initmissing (netgmap ( $\lambda(p, q). (\text{fst } (\text{Fun.id } p), \text{snd } (\text{Fun.id } p), q)) s))
  = \text{fst } (\text{initmissing } (\text{netgmap } \text{fst } s))"$ 
  <proof>

```

interpretation `toy_openproc`: `openproc ptoy optoy Fun.id`

`rewrites "toy_openproc.initmissing = initmissing"`

`<proof>`

lemma `fst_initmissing_netgmap_default_toy_init_netlift`:

`"fst (initmissing (netgmap sr s)) = default toy_init (netlift sr s)"`

`<proof>`

definition

`netglobal :: "(nat \Rightarrow state) \Rightarrow bool \Rightarrow ((state \times 'b) \times 'c) net_state \Rightarrow bool"`

where

`"netglobal P \equiv (λ s. P (default toy_init (netlift fst s)))"`

interpretation `toy_openproc_par_qmsg`: `openproc_parq ptoy optoy Fun.id qmsg`

`rewrites "toy_openproc_par_qmsg.netglobal = netglobal"`

`and "toy_openproc_par_qmsg.initmissing = initmissing"`

`<proof>`

26.9 Final result

lemma `bigger_than_next`:

`assumes "wf_net_tree n"`

`shows "closed (pnet (λ i. ptoy i $\langle\langle$ qmsg) n) \models netglobal (λ σ . \forall i. no (σ i) \leq no (σ (nhid (σ i))))"`
`(is "_ \models netglobal (λ σ . \forall i. ?inv σ i))"`

`<proof>`

end

27 Acknowledgements

We thank Peter Höfner for agreeing to the inclusion of the simple ‘Toy’ example model.

References

- [1] T. Bourke, R. J. van Glabbeek, and P. Höfner. Showing invariance compositionally for a process algebra for network protocols, July 2014.
- [2] A. Fehnker, R. J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W. L. Tan. A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Technical Report 5513, NICTA, 2013.