Mechanization of the Algebra for Wireless Networks (AWN)

Timothy Bourke*

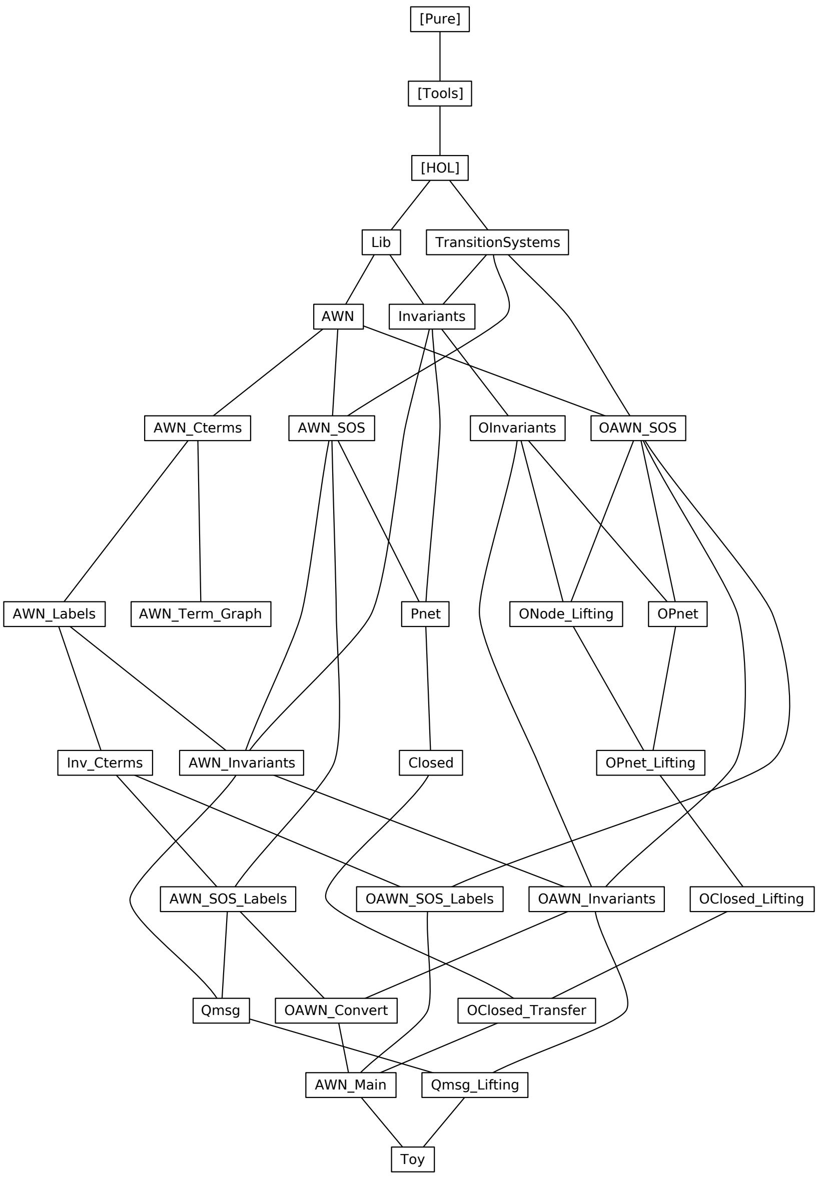
March 17, 2025

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 . . . 6

4 Open reachability and invariance **9**
4.1 Open reachability . . . 9
4.2 Open Invariance . . . 11
4.3 Standard assumption predicates . . . 18

5 Terms of the Algebra for Wireless Networks **20**
5.1 Sequential Processes . . . 20
5.2 Actions . . . 23
5.2.1 Sequential Actions (and related predicates) . . . 23
5.2.2 Node Actions (and related predicates) . . . 25
5.3 Networks . . . 26

6 Semantics of the Algebra of Wireless Networks **32**
6.1 Table 1: Structural operational semantics for sequential process expressions . . . 32
6.2 Table 2: Structural operational semantics for parallel process expressions . . . 33
6.3 Table 3: Structural operational semantics for node expressions . . . 35
6.4 Table 4: Structural operational semantics for partial network expressions . . . 37
6.5 Table 5: Structural operational semantics for complete network expressions . . . 38

7 Control terms and well-definedness of sequential processes **39**
7.1 Microsteps . . . 39
7.2 Wellformed process specifications . . . 41
7.3 Start terms (sterms) . . . 42
7.4 Start terms . . . 43
7.5 Derivative terms . . . 48
7.6 Control terms . . . 50
7.7 Local control terms . . . 51
7.8 Local deriviative terms . . . 54
7.9 More properties of control terms . . . 55

8 Labelling sequential processes **57**
8.1 Labels . . . 57

9 A custom tactic for showing invariants via control terms **60**

10 Configure the inv-cterms tactic for sequential processes **63**

11 Lemmas for partial networks **64**

12 Lemmas for closed networks **73**

13 Open semantics of the Algebra of Wireless Networks **76**
13.1 Open structural operational semantics for sequential process expressions . . . 76
13.2 Open structural operational semantics for parallel process expressions . . . 78
13.3 Open structural operational semantics for node expressions . . . 79
13.4 Open structural operational semantics for partial network expressions . . . 82
13.5 Open structural operational semantics for complete network expressions . . . 83

14 Configure the inv-cterms tactic for open sequential processes **85**

15 Lemmas for open partial networks **86**

16 Lifting rules for (open) nodes **87**

17 Lifting rules for (open) partial networks **95**

18 Lifting rules for (open) closed networks **108**

19 Generic invariants on sequential AWN processes **109**
19.1 Invariants via labelled control terms . . . 110
19.2 Step invariants via labelled control terms . . . 115

20 Generic open invariants on sequential AWN processes **119**
20.1 Open invariants via labelled control terms . . . 119
20.2 Open step invariants via labelled control terms . . . 126

21 Transfer standard invariants into open invariants **131**

22 Model the standard queuing model **135**

23 Lifting rules for parallel compositions with QMSG **136**

24 Transfer open results onto closed models **143**

25 Import all AWN-related theories **172**

26 Simple toy example **172**
26.1 Messages used in the protocol . . . 172
26.2 Protocol model . . . 172
26.3 Define an open version of the protocol . . . 175
26.4 Predicates . . . 176
26.5 Sequential Invariants . . . 176
26.6 Global Invariants . . . 177
26.7 Lifting . . . 182
26.8 Transfer . . . 183
26.9 Final result . . . 185

27 Acknowledgements **185**

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"
  unfolding TT_def by simp

lemma in_set_tl: "x ∈ set (tl xs) ⟹ x ∈ set xs"
  by (metis Nil_tl insert_iff list.collapse set_simps(2))

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"
  using assms unfolding nat_less_le by auto

lemma disjoint_commute:
  "(A ∩ B = {}) ⟹ (B ∩ A = {})"
  by auto

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_icases: "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"
  using assms(1) proof (induction "(ξ, p)" arbitrary: ξ p)
    fix ξ p
    assume "(ξ, p) ∈ init A"
    with assms(2) show "P ξ p" .
  next
    fix s a ξ' p'
    assume "s ∈ reachable A I"
       and tr: "(s, a, (ξ', p')) ∈ trans A"
       and "I a"
       and IH: "⋀ξ p. s = (ξ, p) ⟹ P ξ p"
    from this(1) obtain ξ p where "s = (ξ, p)"
                              and "(ξ, p) ∈ reachable A I"
      by (metis prod.collapse)
    note this(2)
    moreover from IH and ‹s = (ξ, p)› have "P ξ p" .
    moreover from tr and ‹s = (ξ, p)› have "((ξ, p), a, (ξ', p')) ∈ trans A" by simp
    ultimately show "P ξ' p'"
      using ‹I a› by (rule assms(3))
  qed

lemma reachable_weakenE [elim]:
  assumes "s ∈ reachable A P"
      and PQ: "⋀a. P a ⟹ Q a"
    shows "s ∈ reachable A Q"
  using assms(1)
  proof (induction)
    fix s assume "s ∈ init A"
    thus "s ∈ reachable A Q" ..
  next
    fix s a s'
    assume "s ∈ reachable A P"
       and "s ∈ reachable A Q"
       and "(s, a, s') ∈ trans A"
       and "P a"
    from ‹P a› have "Q a" by (rule PQ)
    with ‹s ∈ reachable A Q› and ‹(s, a, s') ∈ trans A› show "s' ∈ reachable A Q" ..
  qed

lemma reachable_weaken_TT [elim]:
  assumes "s ∈ reachable A I"
    shows "s ∈ reachable A TT"
  using assms by rule simp

lemma init_empty_reachable_empty:
  assumes "init A = {}"
    shows "reachable A I = {}"
  proof (rule ccontr)
    assume "reachable A I ≠ {}"
    then obtain s where "s ∈ reachable A I" by auto
    thus False
    proof (induction rule: reachable.induct)
```

```
    fix s
    assume "s ∈ init A"
    with ‹init A = {}› show False by simp
  qed
qed
```

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"
  unfolding invariant_def
  proof
      fix s
    assume "s ∈ reachable A I"
      thus "P s"
    proof induction
      fix s assume "s ∈ init A"
      thus "P s" by (rule init)
    next
      fix s a s'
      assume "s ∈ reachable A I"
         and "P s"
         and "(s, a, s') ∈ trans A"
         and "I a"
        thus "P s'" by (rule step)
    qed
  qed

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"
  using assms by auto

lemma invariant_arbitraryI:
  assumes "⋀s. s ∈ reachable A I ⟹ P s"
    shows "A ⊫ (I →) P"
  using assms unfolding invariant_def by simp

lemma invariantD [dest]:
  assumes "A ⊫ (I →) P"
      and "s ∈ reachable A I"
    shows "P s"
  using assms unfolding invariant_def by blast

lemma invariant_initE [elim]:
  assumes invP: "A ⊫ (I →) P"
```

```
    and init: "s ∈ init A"
  shows "P s"
proof -
  from init have "s ∈ reachable A I" ..
  with invP show ?thesis ..
qed

lemma invariant_weakenE [elim]:
  fixes T σ P Q
  assumes invP: "A ⊫ (PI →) P"
      and PQ:   "⋀s. P s ⟹ Q s"
      and QIPI: "⋀a. QI a ⟹ PI a"
    shows       "A ⊫ (QI →) Q"
  proof
    fix s
    assume "s ∈ init A"
    with invP have "P s" ..
    thus "Q s" by (rule PQ)
  next
    fix s a s'
    assume "s ∈ reachable A QI"
       and "(s, a, s') ∈ trans A"
       and "QI a"
    from ‹QI a› have "PI a" by (rule QIPI)
    from ‹s ∈ reachable A QI› and QIPI have "s ∈ reachable A PI" ..
    hence "s' ∈ reachable A PI" using ‹(s, a, s') ∈ trans A› and ‹PI a› ..
    with invP have "P s'" ..
    thus "Q s'" by (rule PQ)
  qed

definition
  step_invariant
  :: "('s, 'a) automaton ⇒ ('a ⇒ bool) ⇒ (('s, 'a) transition ⇒ bool) ⇒ bool"
  (‹_ ⊫_A (1'(_ →')/ _)› [100, 0, 0] 8)
where
  "(A ⊫_A (I →) P) = (∀a. I a ⟶ (∀s∈reachable A I. (∀s'.(s, a, s') ∈ trans A ⟶ P (s, a, s'))))"

lemma invariant_restrict_inD [dest]:
  assumes "A ⊫ (TT →) P"
    shows "A ⊫ (QI →) P"
  using assms by auto

abbreviation
  any_step_invariant
  :: "('s, 'a) automaton ⇒ (('s, 'a) transition ⇒ bool) ⇒ bool"
  (‹_ ⊫_A _› [100, 9] 8)
where
  "(A ⊫_A P) ≡ (A ⊫_A (TT →) P)"

lemma step_invariant_true:
  "p ⊫_A (λ(s, a, s'). True)"
  unfolding step_invariant_def by simp

lemma step_invariantI [intro]:
  assumes *: "⋀s a s'. ⟦ s∈reachable A I; (s, a, s')∈trans A; I a ⟧ ⟹ P (s, a, s')"
    shows "A ⊫_A (I →) P"
  unfolding step_invariant_def
  using assms by auto

lemma step_invariantD [dest]:
  assumes "A ⊫_A (I →) P"
      and "s∈reachable A I"
      and "(s, a, s') ∈ trans A"
      and "I a"
```

```
    shows "P (s, a, s')"
  using assms unfolding step_invariant_def by blast

lemma step_invariantE [elim]:
    fixes T σ P I s a s'
  assumes "A ⊫A (I →) P"
      and "s∈reachable A I"
      and "(s, a, s') ∈ trans A"
      and "I a"
      and "P (s, a, s') ⟹ Q"
    shows "Q"
  using assms by auto

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"
  using assms by auto

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"
  using assms by auto

lemma step_invariant_weakenE [elim!]:
  fixes T σ P Q
  assumes invP: "A ⊫A (PI →) P"
      and PQ:   "⋀t. P t ⟹ Q t"
      and QIPI: "⋀a. QI a ⟹ PI a"
    shows       "A ⊫A (QI →) Q"
  proof
    fix s a s'
    assume "s ∈ reachable A QI"
       and "(s, a, s') ∈ trans A"
       and "QI a"
    from ‹QI a› have "PI a" by (rule QIPI)
    from ‹s ∈ reachable A QI› have "s ∈ reachable A PI" using QIPI ..
    with invP have "P (s, a, s')" using ‹(s, a, s') ∈ trans A› ‹PI a› ..
    thus "Q (s, a, s')" by (rule PQ)
  qed

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; P s'; Q (s, a, s'); I a ⟧ ⟹ R (s, a, s')"
    shows "A ⊫A (I →) R"
  proof
    fix s a s'
    assume sr: "s ∈ reachable A I"
       and tr: "(s, a, s') ∈ trans A"
       and "I a"
    hence "s' ∈ reachable A I" ..
    with pinv have "P s'" ..
    from pinv and sr have "P s" ..
    from qinv sr tr ‹I a› have "Q (s, a, s')" ..
    with ‹P s› and ‹P s'› show "R (s, a, s')" using ‹I a› by (rule wr)
  qed

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
    fix s assume "s ∈ init A" thus "P s" by (rule init)
  next
    fix s s' a
    assume "s ∈ reachable A I"
       and "P s"
       and "(s, a, s') ∈ trans A"
       and "I a"
      show "P s'"
    proof -
      from sinv and ‹s∈reachable A I› and ‹(s, a, s')∈trans A› and ‹I a› have "Q (s, a, s')" ..
      with ‹s∈reachable A I› and ‹P s› show "P s'" using ‹I a› by (rule step)
    qed
  qed

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"
  using assms by (metis fst_conv oreachable.oreachable_local)

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

```
    from ‹U σ σ'› have "U (fst (σ, p)) σ'" by simp
    with ‹(σ, p) ∈ oreachable A S U› have "(σ', snd (σ, p)) ∈ oreachable A S U"
      by (rule oreachable_other)
    thus "(σ', p) ∈ oreachable A S U" by simp
  qed

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"
  using assms (1) proof (induction "(σ, p)" arbitrary: σ p)
    fix σ p
    assume "(σ, p) ∈ init A"
    with assms(2) show "P σ p" .
  next
    fix s σ'
    assume "s ∈ oreachable A S U"
       and "U (fst s) σ'"
       and IH: "⋀σ p. s = (σ, p) ⟹ P σ p"
    from this(1) obtain σ p where "s = (σ, p)"
                              and "(σ, p) ∈ oreachable A S U"
      by (metis surjective_pairing)
    note this(2)
    moreover from IH and ‹s = (σ, p)› have "P σ p" .
    moreover from ‹U (fst s) σ'› and ‹s = (σ, p)› have "U σ σ'" by simp
    ultimately have "P σ' p" by (rule assms(3))
    with ‹s = (σ, p)› show "P σ' (snd s)" by simp
  next
    fix s a σ' p'
    assume "s ∈ oreachable A S U"
       and tr: "(s, a, (σ', p')) ∈ trans A"
       and "S (fst s) (fst (σ', p')) a"
       and IH: "⋀σ p. s = (σ, p) ⟹ P σ p"
    from this(1) obtain σ p where "s = (σ, p)"
                              and "(σ, p) ∈ oreachable A S U"
      by (metis surjective_pairing)
    note this(2)
    moreover from IH ‹s = (σ, p)› have "P σ p" .
    moreover from tr and ‹s = (σ, p)› have "((σ, p), a, (σ', p')) ∈ trans A" by simp
    moreover from ‹S (fst s) (fst (σ', p')) a› and ‹s = (σ, p)› have "S σ σ' a" by simp
    ultimately show "P σ' p'" by (rule assms(4))
  qed

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"
  using assms(1)
  proof (induction)
    fix s assume "s ∈ init A"
    thus "s ∈ oreachable A QS QU" ..
  next
    fix s a s'
    assume "s ∈ oreachable A QS QU"
       and "(s, a, s') ∈ trans A"
       and "PS (fst s) (fst s') a"
    from ‹PS (fst s) (fst s') a› have "QS (fst s) (fst s') a" by (rule PSQS)
    with ‹s ∈ oreachable A QS QU› and ‹(s, a, s') ∈ trans A› show "s' ∈ oreachable A QS QU" ..
  next
    fix s g'
```

```
    assume "s ∈ oreachable A QS QU"
       and "PU (fst s) g'"
    from ‹PU (fst s) g'› have "QU (fst s) g'" by (rule PUQU)
    with ‹s ∈ oreachable A QS QU› show "(g', snd s) ∈ oreachable A QS QU" ..
  qed

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

lemma act_simp [iff]: "act I s s' a = I a"
  unfolding act_def ..

lemma reachable_in_oreachable [elim]:
    fixes s
  assumes "s ∈ reachable A I"
    shows "s ∈ oreachable A (act I) U"
  unfolding act_def using assms proof induction
    fix s
    assume "s ∈ init A"
    thus "s ∈ oreachable A (λ_ _. I) U" ..
  next
    fix s a s'
    assume "s ∈ oreachable A (λ_ _. I) U"
       and "(s, a, s') ∈ trans A"
       and "I a"
    thus "s' ∈ oreachable A (λ_ _. I) U"
      by (rule oreachable_local)
  qed
```

4.2 Open Invariance

```
definition oinvariant
  :: "('g × 'l, 'a) automaton
      ⇒ ('g ⇒ 'g ⇒ 'a ⇒ bool) ⇒ ('g ⇒ 'g ⇒ bool)
      ⇒ (('g × 'l) ⇒ bool) ⇒ bool"
  (‹_ ⊨ (1'((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: "⋀s. s ∈ init A ⟹ P s"
      and other: "⋀g g' l.
                  ⟦ (g, l) ∈ oreachable A S U; P (g, l); U g g' ⟧ ⟹ P (g', l)"
      and local: "⋀s a s'.
                  ⟦ s ∈ oreachable A S U; P s; (s, a, s') ∈ trans A; S (fst s) (fst s') a ⟧ ⟹ P s'"
    shows "A ⊨ (S, U →) P"
  unfolding oinvariant_def
  proof
      fix s
    assume "s ∈ oreachable A S U"
      thus "P s"
    proof induction
      fix s assume "s ∈ init A"
      thus "P s" by (rule init)
    next
      fix s a s'
      assume "s ∈ oreachable A S U"
         and "P s"
         and "(s, a, s') ∈ trans A"
         and "S (fst s) (fst s') a"
        thus "P s'" by (rule local)
```

```
    next
      fix s g'
      assume "s ∈ oreachable A S U"
         and "P s"
         and "U (fst s) g'"
        thus "P (g', snd s)"
          by - (rule other [where g="fst s"], simp_all)
    qed
  qed

lemma oinvariant_oreachableI:
  assumes "⋀σ s. (σ, s)∈oreachable A S U ⟹ P (σ, s)"
  shows "A ⊨ (S, U →) P"
  using assms unfolding oinvariant_def by auto

lemma oinvariant_pairI [intro]:
  assumes init: "⋀σ p. (σ, p) ∈ init A ⟹ P (σ, p)"
      and local: "⋀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')"
      and other: "⋀σ σ' p.
                  ⟦ (σ, p) ∈ oreachable A S U; P (σ, p); U σ σ' ⟧ ⟹ P (σ', p)"
    shows "A ⊨ (S, U →) P"
  by (rule oinvariantI)
     (clarsimp | erule init | erule(3) local | erule(2) other)+

lemma oinvariantD [dest]:
  assumes "A ⊨ (S, U →) P"
      and "s ∈ oreachable A S U"
    shows "P s"
  using assms unfolding oinvariant_def
  by clarsimp

lemma oinvariant_initD [dest, elim]:
  assumes invP: "A ⊨ (S, U →) P"
      and init: "s ∈ init A"
    shows "P s"
  proof -
    from init have "s ∈ oreachable A S U" ..
    with invP show ?thesis ..
  qed

lemma oinvariant_weakenE [elim!]:
  assumes invP: "A ⊨ (PS, PU →) P"
      and PQ:   "⋀s. P s ⟹ Q s"
      and QSPS: "⋀s s' a. QS s s' a ⟹ PS s s' a"
      and QUPU: "⋀s s'.   QU s s'   ⟹ PU s s'"
    shows       "A ⊨ (QS, QU →) Q"
  proof
    fix s
    assume "s ∈ init A"
    with invP have "P s" ..
    thus "Q s" by (rule PQ)
  next
    fix σ p σ' p' a
    assume "(σ, p) ∈ oreachable A QS QU"
       and "((σ, p), a, (σ', p')) ∈ trans A"
       and "QS σ σ' a"
    from this(3) have "PS σ σ' a" by (rule QSPS)
    from ‹(σ, p) ∈ oreachable A QS QU› and QSPS QUPU have "(σ, p) ∈ oreachable A PS PU" ..
    hence "(σ', p') ∈ oreachable A PS PU" using ‹((σ, p), a, (σ', p')) ∈ trans A› and ‹PS σ σ' a› ..
    with invP have "P (σ', p')" ..
    thus "Q (σ', p')" by (rule PQ)
  next
```

```
    fix σ σ' p
    assume "(σ, p) ∈ oreachable A QS QU"
       and "Q (σ, p)"
       and "QU σ σ'"
    from ‹QU σ σ'› have "PU σ σ'" by (rule QUPU)
    from ‹(σ, p) ∈ oreachable A QS QU› and QSPS QUPU have "(σ, p) ∈ oreachable A PS PU" ..
    hence "(σ', p) ∈ oreachable A PS PU" using ‹PU σ σ'› ..
    with invP have "P (σ', p)" ..
    thus "Q (σ', p)" by (rule PQ)
  qed

lemma oinvariant_weakenD [dest]:
  assumes "A ⊨ (S', U' →) P"
      and "(σ, p) ∈ oreachable A S U"
      and weakenS: "⋀s s' a. S s s' a ⟹ S' s s' a"
      and weakenU: "⋀s s'. U s s' ⟹ U' s s'"
    shows "P (σ, p)"
  proof -
    from ‹(σ, p) ∈ oreachable A S U› have "(σ, p) ∈ oreachable A S' U'"
      by (rule oreachable_weakenE)
         (erule weakenS, erule weakenU)
    with ‹A ⊨ (S', U' →) P› show "P (σ, p)" ..
  qed

lemma close_open_invariant:
  assumes oinv: "A ⊨ (act I, U →) P"
    shows "A ⊫ (I →) P"
  proof
    fix s
    assume "s ∈ init A"
    with oinv show "P s" ..
  next
    fix ξ p ξ' p' a
    assume sr: "(ξ, p) ∈ reachable A I"
       and step: "((ξ, p), a, (ξ', p')) ∈ trans A"
       and "I a"
    hence "(ξ', p') ∈ reachable A I" ..
    hence "(ξ', p') ∈ oreachable A (act I) U" ..
    with oinv show "P (ξ', p')" ..
  qed

definition local_steps :: "((('i ⇒ 's1) × 'l1) × 'a × ('i ⇒ 's2) × 'l2) set ⇒ 'i set ⇒ bool"
where "local_steps T J ≡
   (∀σ ζ s a σ' s'. ((σ, s), a, (σ', s')) ∈ T ∧ (∀j∈J. ζ j = σ j)
   ⟶ (∃ζ'. (∀j∈J. ζ' j = σ' j) ∧ ((ζ, s), a, (ζ', s')) ∈ T))"

lemma local_stepsI [intro!]:
  assumes "⋀σ ζ s a σ' ζ' s'. ⟦ ((σ, s), a, (σ', s')) ∈ T; ∀j∈J. ζ j = σ j ⟧
                                ⟹ (∃ζ'. (∀j∈J. ζ' j = σ' j) ∧ ((ζ, s), a, (ζ', s')) ∈ T)"
    shows "local_steps T J"
  unfolding local_steps_def using assms by clarsimp

lemma local_stepsE [elim, dest]:
  assumes "local_steps T J"
      and "((σ, s), a, (σ', s')) ∈ T"
      and "∀j∈J. ζ j = σ j"
    shows "∃ζ'. (∀j∈J. ζ' j = σ' j) ∧ ((ζ, s), a, (ζ', s')) ∈ T"
  using assms unfolding local_steps_def by blast

definition other_steps :: "(('i ⇒ 's) ⇒ ('i ⇒ 's) ⇒ bool) ⇒ 'i set ⇒ bool"
where "other_steps U J ≡ ∀σ σ'. U σ σ' ⟶ (∀j∈J. σ' j = σ j)"

lemma other_stepsI [intro!]:
  assumes "⋀σ σ' j. ⟦ U σ σ'; j ∈ J ⟧ ⟹ σ' j = σ j"
```

```
    shows "other_steps U J"
  using assms unfolding other_steps_def by simp

lemma other_stepsE [elim]:
  assumes "other_steps U J"
      and "U σ σ'"
    shows "∀j∈J. σ' j = σ j"
  using assms unfolding other_steps_def by simp

definition subreachable
where "subreachable A U J ≡ ∀I. ∀s ∈ oreachable A (λs s'. I) U.
                                  (∃σ. (∀j∈J. σ j = (fst s) j) ∧ (σ, snd s) ∈ reachable A I)"

lemma subreachableI [intro]:
  assumes "local_steps (trans A) J"
      and "other_steps U J"
    shows "subreachable A U J"
  unfolding subreachable_def
  proof (rule, rule)
    fix I s
    assume "s ∈ oreachable A (λs s'. I) U"
    thus "(∃σ. (∀j∈J. σ j = (fst s) j) ∧ (σ, snd s) ∈ reachable A I)"
    proof induction
      fix s
      assume "s ∈ init A"
      hence "(fst s, snd s) ∈ reachable A I"
        by simp (rule reachable_init)
      moreover have "∀j∈J. (fst s) j = (fst s) j"
        by simp
      ultimately show "∃σ. (∀j∈J. σ j = (fst s) j) ∧ (σ, snd s) ∈ reachable A I"
        by auto
    next
      fix s a s'
      assume "∃σ. (∀j∈J. σ j = (fst s) j) ∧ (σ, snd s) ∈ reachable A I"
         and "(s, a, s') ∈ trans A"
         and "I a"
      then obtain ζ where "∀j∈J. ζ j = (fst s) j"
                      and "(ζ, snd s) ∈ reachable A I" by auto

      from ‹(s, a, s') ∈ trans A› have "((fst s, snd s), a, (fst s', snd s')) ∈ trans A"
        by simp
      with ‹local_steps (trans A) J› obtain ζ' where "∀j∈J. ζ' j = (fst s') j"
                                                 and "((ζ, snd s), a, (ζ', snd s')) ∈ trans A"
        using ‹∀j∈J. ζ j = (fst s) j› by - (drule(2) local_stepsE, clarsimp)
      from ‹(ζ, snd s) ∈ reachable A I›
       and ‹((ζ, snd s), a, (ζ', snd s')) ∈ trans A›
       and ‹I a›
       have "(ζ', snd s') ∈ reachable A I" ..

      with ‹∀j∈J. ζ' j = (fst s') j›
        show "∃σ. (∀j∈J. σ j = (fst s') j) ∧ (σ, snd s') ∈ reachable A I" by auto
    next
      fix s σ'
      assume "∃σ. (∀j∈J. σ j = (fst s) j) ∧ (σ, snd s) ∈ reachable A I"
         and "U (fst s) σ'"
      then obtain σ where "∀j∈J. σ j = (fst s) j"
                      and "(σ, snd s) ∈ reachable A I" by auto
      from ‹other_steps U J› and ‹U (fst s) σ'› have "∀j∈J. σ' j = (fst s) j"
        by - (erule(1) other_stepsE)
      with ‹∀j∈J. σ j = (fst s) j› have "∀j∈J. σ j = σ' j"
        by clarsimp
      with ‹(σ, snd s) ∈ reachable A I›
       show "∃σ. (∀j∈J. σ j = fst (σ', snd s) j) ∧ (σ, snd (σ', snd s)) ∈ reachable A I"
         by auto
```

```
    qed
  qed

lemma subreachableE [elim]:
  assumes "subreachable A U J"
      and "s ∈ oreachable A (λs s'. I) U"
    shows "∃σ. (∀j∈J. σ j = (fst s) j) ∧ (σ, snd s) ∈ reachable A I"
  using assms unfolding subreachable_def by simp

lemma subreachableE_pair [elim]:
  assumes "subreachable A U J"
      and "(σ, s) ∈ oreachable A (λs s'. I) U"
    shows "∃ζ. (∀j∈J. ζ j = σ j) ∧ (ζ, s) ∈ reachable A I"
  using assms unfolding subreachable_def by (metis fst_conv snd_conv)

lemma subreachable_otherE [elim]:
  assumes "subreachable A U J"
      and "(σ, l) ∈ oreachable A (λs s'. I) U"
      and "U σ σ'"
    shows "∃ζ'. (∀j∈J. ζ' j = σ' j) ∧ (ζ', l) ∈ reachable A I"
  proof -
    from ‹(σ, l) ∈ oreachable A (λs s'. I) U› and ‹U σ σ'›
      have "(σ', l) ∈ oreachable A (λs s'. I) U"
      by - (rule oreachable_other')
    with ‹subreachable A U J› show ?thesis
      by auto
  qed

lemma open_closed_invariant:
    fixes J
  assumes "A ⊨ (I →) P"
      and "subreachable A U J"
      and localp: "⋀σ σ' s. ⟦ ∀j∈J. σ' j = σ j; P (σ', s) ⟧ ⟹ P (σ, s)"
    shows "A ⊨ (act I, U →) P"
  proof (rule, simp_all only: act_def)
    fix s
    assume "s ∈ init A"
    with ‹A ⊨ (I →) P› show "P s" ..
  next
    fix s a s'
    assume "s ∈ oreachable A (λ_ _. I) U"
       and "P s"
       and "(s, a, s') ∈ trans A"
       and "I a"
    hence "s' ∈ oreachable A (λ_ _. I) U"
      by (metis oreachable_local)
    with ‹subreachable A U J› obtain σ'
      where "∀j∈J. σ' j = (fst s') j"
        and "(σ', snd s') ∈ reachable A I"
        by (metis subreachableE)
    from ‹A ⊨ (I →) P› and ‹(σ', snd s') ∈ reachable A I› have "P (σ', snd s')" ..
    with ‹∀j∈J. σ' j = (fst s') j› show "P s'"
      by (metis localp prod.collapse)
  next
    fix g g' l
    assume or: "(g, l) ∈ oreachable A (λs s'. I) U"
       and "U g g'"
       and "P (g, l)"
    from ‹subreachable A U J› and or and ‹U g g'›
      obtain gg' where "∀j∈J. gg' j = g' j"
                   and "(gg', l) ∈ reachable A I"
        by (auto dest!: subreachable_otherE)
    from ‹A ⊨ (I →) P› and ‹(gg', l) ∈ reachable A I›
      have "P (gg', l)" ..
```

```
    with ‹∀j∈J. gg' j = g' j› show "P (g', l)"
      by (rule localp)
  qed

lemma oinvariant_anyact:
  assumes "A ⊨ (act TT, U →) P"
    shows "A ⊨ (S, U →) P"
  using assms by rule auto

definition
  ostep_invariant
  :: "('g × 'l, 'a) automaton
      ⇒ ('g ⇒ 'g ⇒ 'a ⇒ bool) ⇒ ('g ⇒ 'g ⇒ bool)
      ⇒ (('g × 'l, 'a) transition ⇒ bool) ⇒ bool"
  (‹_ ⊨A (1'((1_),/ (1_) →')/ _)› [100, 0, 0, 9] 8)
where
  "(A ⊨A (S, U →) P) =
     (∀s∈oreachable A S U. (∀a s'. (s, a, s') ∈ trans A ∧ S (fst s) (fst s') a ⟶ P (s, a, s')))"

lemma ostep_invariant_def':
  "(A ⊨A (S, U →) P) = (∀s∈oreachable A S U.
                          (∀a s'. (s, a, s') ∈ trans A ∧ S (fst s) (fst s') a ⟶ P (s, a, s')))"
  unfolding ostep_invariant_def by auto

lemma ostep_invariantI [intro]:
  assumes *: "⋀σ s a σ' s'. ⟦ (σ, s)∈oreachable A S U; ((σ, s), a, (σ', s')) ∈ trans A; S σ σ' a ⟧
                              ⟹ P ((σ, s), a, (σ', s'))"
    shows "A ⊨A (S, U →) P"
  unfolding ostep_invariant_def
  using assms by auto

lemma ostep_invariantD [dest]:
  assumes "A ⊨A (S, U →) P"
      and "(σ, s)∈oreachable A S U"
      and "((σ, s), a, (σ', s')) ∈ trans A"
      and "S σ σ' a"
    shows "P ((σ, s), a, (σ', s'))"
  using assms unfolding ostep_invariant_def' by clarsimp

lemma ostep_invariantE [elim]:
  assumes "A ⊨A (S, U →) P"
      and "(σ, s)∈oreachable A S U"
      and "((σ, s), a, (σ', s')) ∈ trans A"
      and "S σ σ' a"
      and "P ((σ, s), a, (σ', s')) ⟹ Q"
    shows "Q"
  using assms by auto

lemma ostep_invariant_weakenE [elim!]:
  assumes invP: "A ⊨A (PS, PU →) P"
      and PQ:   "⋀t. P t ⟹ Q t"
      and QSPS: "⋀σ σ' a. QS σ σ' a ⟹ PS σ σ' a"
      and QUPU: "⋀σ σ'.   QU σ σ'   ⟹ PU σ σ'"
    shows       "A ⊨A (QS, QU →) Q"
  proof
    fix σ s σ' s' a
    assume "(σ, s) ∈ oreachable A QS QU"
       and "((σ, s), a, (σ', s')) ∈ trans A"
       and "QS σ σ' a"
    from ‹QS σ σ' a› have "PS σ σ' a" by (rule QSPS)
    from ‹(σ, s) ∈ oreachable A QS QU› have "(σ, s) ∈ oreachable A PS PU" using QSPS QUPU ..
    with invP have "P ((σ, s), a, (σ', s'))" using ‹((σ, s), a, (σ', s')) ∈ trans A› ‹PS σ σ' a› ..
    thus "Q ((σ, s), a, (σ', s'))" by (rule PQ)
  qed
```

```
lemma ostep_invariant_weaken_with_invariantE [elim]:
  assumes pinv: "A ⊨ (S, U →) P"
      and qinv: "A ⊫A (S, U →) Q"
      and wr: "⋀σ s a σ' s'. ⟦ P (σ, s); P (σ', s'); Q ((σ, s), a, (σ', s')); S σ σ' a ⟧
                              ⟹ R ((σ, s), a, (σ', s'))"
    shows "A ⊫A (S, U →) R"
  proof
    fix σ s a σ' s'
    assume sr: "(σ, s) ∈ oreachable A S U"
       and tr: "((σ, s), a, (σ', s')) ∈ trans A"
       and "S σ σ' a"
    hence "(σ', s') ∈ oreachable A S U" ..
    with pinv have "P (σ', s')" ..
    from pinv and sr have "P (σ, s)" ..
    from qinv sr tr ‹S σ σ' a› have "Q ((σ, s), a, (σ', s'))" ..
    with ‹P (σ, s)› and ‹P (σ', s')› show "R ((σ, s), a, (σ', s'))" using ‹S σ σ' a› by (rule wr)
  qed

lemma ostep_to_invariantI:
  assumes sinv: "A ⊫A (S, U →) Q"
      and init: "⋀σ s. (σ, s) ∈ init A ⟹ P (σ, s)"
      and local: "⋀σ s σ' s' a.
                    ⟦ (σ, s) ∈ oreachable A S U;
                      P (σ, s);
                      Q ((σ, s), a, (σ', s'));
                      S σ σ' a ⟧ ⟹ P (σ', s')"
      and other: "⋀σ σ' s. ⟦ (σ, s) ∈ oreachable A S U; U σ σ'; P (σ, s) ⟧ ⟹ P (σ', s)"
    shows "A ⊨ (S, U →) P"
  proof
    fix σ s assume "(σ, s) ∈ init A" thus "P (σ, s)" by (rule init)
  next
    fix σ s σ' s' a
    assume "(σ, s) ∈ oreachable A S U"
       and "P (σ, s)"
       and "((σ, s), a, (σ', s')) ∈ trans A"
       and "S σ σ' a"
      show "P (σ', s')"
    proof -
      from sinv and ‹(σ, s)∈oreachable A S U› and ‹((σ, s), a, (σ', s')) ∈ trans A› and ‹S σ σ' a›
        have "Q ((σ, s), a, (σ', s'))" ..
      with ‹(σ, s)∈oreachable A S U› and ‹P (σ, s)› show "P (σ', s')"
        using ‹S σ σ' a› by (rule local)
    qed
  next
    fix σ σ' l
    assume "(σ, l) ∈ oreachable A S U"
       and "U σ σ'"
       and "P (σ, l)"
      thus "P (σ', l)" by (rule other)
  qed

lemma open_closed_step_invariant:
  assumes "A ⊫A (I →) P"
      and "local_steps (trans A) J"
      and "other_steps U J"
      and localp: "⋀σ ζ a σ' ζ' s s'.
                   ⟦ ∀j∈J. σ j = ζ j; ∀j∈J. σ' j = ζ' j; P ((σ, s), a, (σ', s')) ⟧
                   ⟹ P ((ζ, s), a, (ζ', s'))"
    shows "A ⊫A (act I, U →) P"
  proof
    fix σ s a σ' s'
    assume or: "(σ, s) ∈ oreachable A (act I) U"
       and tr: "((σ, s), a, (σ', s')) ∈ trans A"
```

```
      and "act I σ σ' a"
    from ‹act I σ σ' a› have "I a" ..
    from ‹local_steps (trans A) J› and ‹other_steps U J› have "subreachable A U J" ..
    then obtain ζ where "∀j∈J. ζ j = σ j"
                    and "(ζ, s) ∈ reachable A I"
      using or unfolding act_def
        by (auto dest!: subreachableE_pair)

     from ‹local_steps (trans A) J› and tr and ‹∀j∈J. ζ j = σ j›
      obtain ζ' where "∀j∈J. ζ' j = σ' j"
                  and "((ζ, s), a, (ζ', s')) ∈ trans A"
      by auto

    from ‹A ⊫A (I →) P› and ‹(ζ, s) ∈ reachable A I›
                        and ‹((ζ, s), a, (ζ', s')) ∈ trans A›
                        and ‹I a›
      have "P ((ζ, s), a, (ζ', s'))" ..
    with ‹∀j∈J. ζ j = σ j› and ‹∀j∈J. ζ' j = σ' j› show "P ((σ, s), a, (σ', s'))"
      by (rule localp)
  qed

lemma oinvariant_step_anyact:
  assumes "p ⊨A (act TT, U →) P"
    shows "p ⊨A (S, U →) P"
  using assms by rule auto
```

4.3 Standard assumption predicates

otherwith

```
definition otherwith :: "('s ⇒ 's ⇒ bool)
                         ⇒ 'i set
                         ⇒ (('i ⇒ 's) ⇒ 'a ⇒ bool)
                         ⇒ ('i ⇒ 's) ⇒ ('i ⇒ 's) ⇒ 'a ⇒ bool"
where "otherwith Q I P σ σ' a ≡ (∀i. i∉I ⟶ Q (σ i) (σ' i)) ∧ P σ a"

lemma otherwithI [intro]:
  assumes other: "⋀j. j∉I ⟹ Q (σ j) (σ' j)"
      and sync:  "P σ a"
    shows "otherwith Q I P σ σ' a"
  unfolding otherwith_def using assms by simp

lemma otherwithE [elim]:
  assumes "otherwith Q I P σ σ' a"
      and "⟦ P σ a; ∀j. j∉I ⟶ Q (σ j) (σ' j) ⟧ ⟹ R σ σ' a"
    shows "R σ σ' a"
  using assms unfolding otherwith_def by simp

lemma otherwith_actionD [dest]:
  assumes "otherwith Q I P σ σ' a"
    shows "P σ a"
  using assms by auto

lemma otherwith_syncD [dest]:
  assumes "otherwith Q I P σ σ' a"
    shows "∀j. j∉I ⟶ Q (σ j) (σ' j)"
  using assms by auto

lemma otherwithEI [elim]:
  assumes "otherwith P I PO σ σ' a"
      and "⋀σ a. PO σ a ⟹ QO σ a"
    shows "otherwith P I QO σ σ' a"
  using assms(1) unfolding otherwith_def
  by (clarsimp elim!: assms(2))
```

```
lemma all_but:
  assumes "⋀ξ. S ξ ξ"
      and "σ' i = σ i"
      and "∀j. j ≠ i ⟶ S (σ j) (σ' j)"
    shows "∀j. S (σ j) (σ' j)"
  using assms by metis

lemma all_but_eq [dest]:
  assumes "σ' i = σ i"
      and "∀j. j ≠ i ⟶ σ j = σ' j"
    shows "σ = σ'"
  using assms by - (rule ext, metis)
```

other

```
definition other :: "('s ⇒ 's ⇒ bool) ⇒ 'i set ⇒ ('i ⇒ 's) ⇒ ('i ⇒ 's) ⇒ bool"
where "other P I σ σ' ≡ ∀i. if i∈I then σ' i = σ i else P (σ i) (σ' i)"

lemma otherI [intro]:
  assumes local: "⋀i. i∈I ⟹ σ' i = σ i"
      and other: "⋀j. j∉I ⟹ P (σ j) (σ' j)"
    shows "other P I σ σ'"
  using assms unfolding other_def by clarsimp

lemma otherE [elim]:
  assumes "other P I σ σ'"
      and "⟦ ∀i∈I. σ' i = σ i; ∀j. j∉I ⟶ P (σ j) (σ' j) ⟧ ⟹ R σ σ'"
    shows "R σ σ'"
  using assms unfolding other_def by simp

lemma other_localD [dest]:
  "other P {i} σ σ' ⟹ σ' i = σ i"
  by auto

lemma other_otherD [dest]:
  "other P {i} σ σ' ⟹ ∀j. j≠i ⟶ P (σ j) (σ' j)"
  by auto

lemma other_bothE [elim]:
  assumes "other P {i} σ σ'"
  obtains "σ' i = σ i" and "∀j. j≠i ⟶ P (σ j) (σ' j)"
  using assms by auto

lemma weaken_local [elim]:
  assumes "other P I σ σ'"
      and PQ: "⋀ξ ξ'. P ξ ξ' ⟹ Q ξ ξ'"
    shows "other Q I σ σ'"
  using assms unfolding other_def by auto

definition global :: "((nat ⇒ 's) ⇒ bool) ⇒ (nat ⇒ 's) × 'local ⇒ bool"
where "global P ≡ (λ(σ, _). P σ)"

lemma globalsimp [simp]: "global P s = P (fst s)"
  unfolding global_def by (simp split: prod.split)

definition globala :: "((nat ⇒ 's, 'action) transition ⇒ bool)
                        ⇒ ((nat ⇒ 's) × 'local, 'action) transition ⇒ bool"
where "globala P ≡ (λ((σ, _), a, (σ', _)). P (σ, a, σ'))"

lemma globalasimp [simp]: "globala P s = P (fst (fst s), fst (snd s), fst (snd (snd s)))"
  unfolding globala_def by (simp split: prod.split)

end
```

5 Terms of the Algebra for Wireless Networks

```
theory AWN
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"  :: "[pttrn, bool,  ('s, 'm, 'p, unit) seqp] ⇒  ('s, 'm, 'p, unit) seqp"
                 (‹(‹unbreakable›⟨_. _⟩)//_› [0, 0, 60] 60)

  "_bassign"  :: "[pttrn, 'a,  ('s, 'm, 'p, unit) seqp] ⇒  ('s, 'm, 'p, unit) seqp"
                 (‹(‹unbreakable›⟦_. _⟧)//_› [0, 0, 60] 60)
  "_lbassign" :: "['a, pttrn, '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)

syntax_consts
  "_guard" "_lguard" "_ifguard" ⇌ GUARD and
  "_assign" "_lassign" "_bassign" "_lbassign" ⇌ ASSIGN and
  "_unicast" "_lunicast" ⇌ UCAST and
  "_bcast" "_lbcast" ⇌ BCAST and
  "_gcast" "_lgcast" ⇌ GCAST and
  "_send" "_lsend" ⇌ SEND and
  "_deliver" "_ldeliver" ⇌ DELIVER and
  "_receive" "_lreceive" ⇌ RECEIVE

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 ({l}⟨fg⟩ p)"
  "⋀l fa p.         not_call ({l}⟦fa⟧ p)"
  "⋀p1 p2.          not_call (p1 ⊕ p2)"
  "⋀l fip fmsg p q. not_call ({l}unicast(fip, fmsg).p ▷ q)"
  "⋀l fmsg p.       not_call ({l}broadcast(fmsg).p)"
  "⋀l fips fmsg p.  not_call ({l}groupcast(fips, fmsg).p)"
  "⋀l fmsg p.       not_call ({l}send(fmsg).p)"
  "⋀l fdata p.      not_call ({l}deliver(fdata).p)"
  "⋀l fmsg p.       not_call ({l}receive(fmsg).p)"
  "⋀l pn.         ¬(not_call (call(pn)))"
  unfolding not_call_def by auto

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

lemma not_choice_simps [simp]:
  "⋀l fg p.         not_choice ({l}⟨fg⟩ p)"
  "⋀l fa p.         not_choice ({l}⟦fa⟧ p)"
  "⋀p1 p2.        ¬(not_choice (p1 ⊕ p2))"
  "⋀l fip fmsg p q. not_choice ({l}unicast(fip, fmsg).p ▷ q)"
  "⋀l fmsg p.       not_choice ({l}broadcast(fmsg).p)"
  "⋀l fips fmsg p.  not_choice ({l}groupcast(fips, fmsg).p)"
  "⋀l fmsg p.       not_choice ({l}send(fmsg).p)"
  "⋀l fdata p.      not_choice ({l}deliver(fdata).p)"
  "⋀l fmsg p.       not_choice ({l}receive(fmsg).p)"
  "⋀l pn.           not_choice (call(pn))"
  unfolding not_choice_def by auto

lemma seqp_congs:
  "⋀l fg p. {l}⟨fg⟩ p = {l}⟨fg⟩ p"
  "⋀l fa p. {l}⟦fa⟧ p = {l}⟦fa⟧ p"
  "⋀p1 p2. p1 ⊕ p2 = p1 ⊕ p2"
  "⋀l fip fmsg p q. {l}unicast(fip, fmsg).p ▷ q = {l}unicast(fip, fmsg).p ▷ q"
  "⋀l fmsg p. {l}broadcast(fmsg).p = {l}broadcast(fmsg).p"
  "⋀l fips fmsg p. {l}groupcast(fips, fmsg).p = {l}groupcast(fips, fmsg).p"
  "⋀l fmsg p. {l}send(fmsg).p = {l}send(fmsg).p"
  "⋀l fdata p. {l}deliver(fdata).p = {l}deliver(fdata).p"
  "⋀l fmsg p. {l}receive(fmsg).p = {l}receive(fmsg).p"
  "⋀l pn. call(pn) = call(pn)"
  by auto
```

Remove data expressions from process terms.

```
fun seqp_skeleton :: "('s, 'm, 'p, 'l) seqp ⇒ (unit, unit, 'p, 'l) seqp"
where
    "seqp_skeleton ({l}⟨_⟩ p)                 = {l}⟨λ_. {()}⟩ (seqp_skeleton p)"
  | "seqp_skeleton ({l}⟦_⟧ p)                 = {l}⟦λ_. ()⟧ (seqp_skeleton p)"
  | "seqp_skeleton (p ⊕ q)                    = (seqp_skeleton p) ⊕ (seqp_skeleton q)"
  | "seqp_skeleton ({l}unicast(_, _). p ▷ q) = {l}unicast(λ_. 0, λ_. ()). (seqp_skeleton p) ▷ (seqp_skeleton
q)"
  | "seqp_skeleton ({l}broadcast(_). p)      = {l}broadcast(λ_. ()). (seqp_skeleton p)"
  | "seqp_skeleton ({l}groupcast(_, _). p)   = {l}groupcast(λ_. {}, λ_. ()). (seqp_skeleton p)"
  | "seqp_skeleton ({l}send(_). p)           = {l}send(λ_. ()). (seqp_skeleton p)"
  | "seqp_skeleton ({l}deliver(_). p)        = {l}deliver(λ_. 0). (seqp_skeleton p)"
  | "seqp_skeleton ({l}receive(_). p)        = {l}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 ({l}⟨fg⟩ p) = {{l}⟨fg⟩ p} ∪ subterms p"
  | "subterms ({l}⟦fa⟧ p) = {{l}⟦fa⟧ p} ∪ subterms p"
  | "subterms (p1 ⊕ p2) = {p1 ⊕ p2} ∪ subterms p1 ∪ subterms p2"
  | "subterms ({l}unicast(fip, fmsg). p ▷ q) =
       {{l}unicast(fip, fmsg). p ▷ q} ∪ subterms p ∪ subterms q"
```

```
  | "subterms ({l}broadcast(fmsg). p) = {{l}broadcast(fmsg). p} ∪ subterms p"
  | "subterms ({l}groupcast(fips, fmsg). p) = {{l}groupcast(fips, fmsg). p} ∪ subterms p"
  | "subterms ({l}send(fmsg). p) = {{l}send(fmsg).p} ∪ subterms p"
  | "subterms ({l}deliver(fdata). p) = {{l}deliver(fdata).p} ∪ subterms p"
  | "subterms ({l}receive(fmsg). p) = {{l}receive(fmsg).p} ∪ subterms p"
  | "subterms (call(pn)) = {call(pn)}"

lemma subterms_refl [simp]: "p ∈ subterms p"
  by (cases p) simp_all

lemma subterms_trans [elim]:
  assumes "q ∈ subterms p"
      and "r ∈ subterms q"
    shows "r ∈ subterms p"
  using assms by (induction p) auto

lemma root_in_subterms [simp]:
   "⋀Γ pn. ∃pn'. Γ pn ∈ subterms (Γ pn')"
  by (rule_tac x=pn in exI) simp

lemma deriv_in_subterms [elim, dest]:
  "⋀l f p q. {l}⟨f⟩ q ∈ subterms p ⟹ q ∈ subterms p"
  "⋀l fa p q. {l}⟦fa⟧ q ∈ subterms p ⟹ q ∈ subterms p"
  "⋀p1 p2 p. p1 ⊕ p2 ∈ subterms p ⟹ p1 ∈ subterms p"
  "⋀p1 p2 p. p1 ⊕ p2 ∈ subterms p ⟹ p2 ∈ subterms p"
  "⋀l fip fmsg p q r. {l}unicast(fip, fmsg). q ▷ r ∈ subterms p ⟹ q ∈ subterms p"
  "⋀l fip fmsg p q r. {l}unicast(fip, fmsg). q ▷ r ∈ subterms p ⟹ r ∈ subterms p"
  "⋀l fmsg p q. {l}broadcast(fmsg). q ∈ subterms p ⟹ q ∈ subterms p"
  "⋀l fips fmsg p q. {l}groupcast(fips, fmsg). q ∈ subterms p ⟹ q ∈ subterms p"
  "⋀l fmsg p q. {l}send(fmsg). q ∈ subterms p ⟹ q ∈ subterms p"
  "⋀l fdata p q. {l}deliver(fdata). q ∈ subterms p ⟹ q ∈ subterms p"
  "⋀l fmsg p q. {l}receive(fmsg). q ∈ subterms p ⟹ q ∈ subterms p"
  by auto
```

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 ..
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]:
```

```
  "⋀m.     recvmsg P (broadcast m)     = True"
  "⋀ips m. recvmsg P (groupcast ips m) = True"
  "⋀ip m.  recvmsg P (unicast ip m)    = True"
  "⋀ip.    recvmsg P (notunicast ip)   = True"
  "⋀m.     recvmsg P (send m)          = True"
  "⋀d.     recvmsg P (deliver d)       = True"
  "⋀m.     recvmsg P (receive m)       = P m"
  "        recvmsg P τs                = True"
  unfolding recvmsg_def by simp_all

lemma recvmsgTT [simp]: "recvmsg TT a"
  by (cases a) simp_all

lemma recvmsgE [elim]:
  assumes "recvmsg (R σ) a"
      and "⋀m. R σ m ⟹ R σ' m"
    shows "recvmsg (R σ') a"
  using assms(1) by (cases a) (auto elim!: assms(2))

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 τs                = True"
  unfolding anycast_def by simp_all

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 τs                = True"
  unfolding orecvmsg_def by simp_all

lemma orecvmsgEI [elim]:
  "⟦ orecvmsg P σ a; ⋀σ a. P σ a ⟹ Q σ a ⟧ ⟹ orecvmsg Q σ a"
  by (cases a) simp_all

lemma orecvmsg_stateless_recvmsg [elim]:
  "orecvmsg (λ_. P) σ a ⟹ recvmsg P a"
  by (cases a) simp_all

lemma orecvmsg_recv_weaken [elim]:
  "⟦ orecvmsg P σ a; ⋀σ a. P σ a ⟹ Q a ⟧ ⟹ recvmsg Q a"
  by (cases a) simp_all

lemma orecvmsg_recvmsg [elim]:
```

```
  "orecvmsg P σ a ⟹ recvmsg (P σ) a"
  by (cases a) simp_all

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 τs                = True"
  unfolding sendmsg_def by simp_all

type_synonym ('s, 'm, 'p, 'l) seqp_env = "'p ⇒ ('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 ≡ τn"
instance ..
end

definition arrivemsg :: "ip ⇒ ('m ⇒ bool) ⇒ 'm node_action ⇒ bool"
where "arrivemsg i P a ≡ case a of node_arrive ii ni m ⇒ ((ii = {i} ⟶ P m))
                                 | _ ⇒ True"

lemma arrivemsg_simps[simp]:
  "⋀R m.         arrivemsg i P (R:*cast(m))         = True"
  "⋀d m.         arrivemsg i P (d:deliver(m))       = True"
  "⋀i ii ni m.   arrivemsg i P (ii¬ni:arrive(m))    = (ii = {i} ⟶ P m)"
  "⋀i1 i2.       arrivemsg i P (connect(i1, i2))    = True"
  "⋀i1 i2.       arrivemsg i P (disconnect(i1, i2)) = True"
  "⋀i i' d di.   arrivemsg i P (i':newpkt(d, di))   = True"
  "              arrivemsg i P τn                   = True"
  unfolding arrivemsg_def by simp_all

lemma arrivemsgTT [simp]: "arrivemsg i TT = TT"
  by (rule ext) (clarsimp simp: arrivemsg_def split: node_action.split)

definition oarrivemsg :: "((ip ⇒ 's) ⇒ 'm ⇒ bool) ⇒ (ip ⇒ 's) ⇒ 'm node_action ⇒ bool"
where "oarrivemsg P σ a ≡ case a of node_arrive ii ni m ⇒ P σ m | _ ⇒ True"

lemma oarrivemsg_simps[simp]:
  "⋀R m.         oarrivemsg P σ (R:*cast(m))         = True"
  "⋀d m.         oarrivemsg P σ (d:deliver(m))       = True"
  "⋀i ii ni m.   oarrivemsg P σ (ii¬ni:arrive(m))    = P σ m"
  "⋀i1 i2.       oarrivemsg P σ (connect(i1, i2))    = True"
  "⋀i1 i2.       oarrivemsg P σ (disconnect(i1, i2)) = True"
  "⋀i i' d di.   oarrivemsg P σ (i':newpkt(d, di))   = True"
  "              oarrivemsg P σ τn                   = True"
```

```
  unfolding oarrivemsg_def by simp_all

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

definition castmsg :: "('m ⇒ bool) ⇒ 'm node_action ⇒ bool"
where "castmsg P a ≡ case a of _:*cast(m) ⇒ P m
                              | _ ⇒ True"

lemma castmsg_simps[simp]:
  "⋀R m.       castmsg P (R:*cast(m))         = P m"
  "⋀d m.       castmsg P (d:deliver(m))       = True"
  "⋀i ii ni m. castmsg P (ii¬ni:arrive(m))    = True"
  "⋀i1 i2.     castmsg P (connect(i1, i2))    = True"
  "⋀i1 i2.     castmsg P (disconnect(i1, i2)) = True"
  "⋀i i' d di. castmsg P (i':newpkt(d, di))   = True"
  "            castmsg P τn                   = True"
  unfolding castmsg_def by simp_all
```

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)"
  by simp (rule Un_commute)

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"
  using assms by simp

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"
  using assms by (induction ns) simp_all

lemma netmap_none_not_in_net_ips:
  assumes "netmap ns i = None"
    shows "i∉net_ips ns"
  using assms by (induction ns) auto

lemma net_ips_is_dom_netmap: "net_ips s = dom(netmap s)"
  proof (induction s)
    fix i R_i and p :: 's
    show "net_ips (NodeS i p R_i) = dom (netmap (NodeS i p R_i))"
      by auto
  next
    fix s1 s2 :: "'s net_state"
    assume "net_ips s1 = dom (netmap s1)"
       and "net_ips s2 = dom (netmap s2)"
    thus "net_ips (SubnetS s1 s2) = dom (netmap (SubnetS s1 s2))"
      by auto
  qed

lemma in_netmap [simp]:
  assumes "i ∈ net_ips ns"
    shows "netmap ns i ≠ None"
  using assms by (auto simp add: net_ips_is_dom_netmap)

lemma netmap_subnets_same:
  assumes "netmap s1 i = x"
      and "netmap s2 i = x"
    shows "netmap (SubnetS s1 s2) i = x"
  using assms by simp (metis map_add_dom_app_simps(1) map_add_dom_app_simps(3))

lemma netmap_subnets_samef:
  assumes "netmap s1 = f"
      and "netmap s2 = f"
    shows "netmap (SubnetS s1 s2) = f"
  using assms by simp (metis map_add_le_mapI map_le_antisym map_le_map_add map_le_refl)

lemma netmap_add_disjoint [elim]:
  assumes "∀i∈net_ips s1 ∪ net_ips s2. the ((netmap s1 ++ netmap s2) i) = σ i"
      and "net_ips s1 ∩ net_ips s2 = {}"
    shows "∀i∈net_ips s1. the (netmap s1 i) = σ i"
  proof
    fix i
    assume "i ∈ net_ips s1"
    hence "i ∈ dom(netmap s1)" by (simp add: net_ips_is_dom_netmap)
    moreover with assms(2) have "i ∉ dom(netmap s2)" by (auto simp add: net_ips_is_dom_netmap)
    ultimately have "the (netmap s1 i) = the ((netmap s1 ++ netmap s2) i)"
      by (simp add: map_add_dom_app_simps)
    with assms(1) and ‹i∈net_ips s1› show "the (netmap s1 i) = σ i" by simp
  qed

lemma netmap_add_disjoint2 [elim]:
  assumes "∀i∈net_ips s1 ∪ net_ips s2. the ((netmap s1 ++ netmap s2) i) = σ i"
    shows "∀i∈net_ips s2. the (netmap s2 i) = σ i"
  using assms by (simp add: net_ips_is_dom_netmap)
                 (metis Un_iff map_add_dom_app_simps(1))

lemma net_ips_netmap_subnet [elim]:
  assumes "net_ips s1 ∩ net_ips s2 = {}"
      and "∀i∈net_ips (SubnetS s1 s2). the (netmap (SubnetS s1 s2) i) = σ i"
    shows "∀i∈net_ips s1. the (netmap s1 i) = σ i"
      and "∀i∈net_ips s2. the (netmap s2 i) = σ i"
```

```
  proof -
    from assms(2) have "∀i∈net_ips s1 ∪ net_ips s2. the ((netmap s1 ++ netmap s2) i) = σ i" by auto
    with assms(1) show "∀i∈net_ips s1. the (netmap s1 i) = σ i"
      by - (erule(1) netmap_add_disjoint)
  next
    from assms(2) have "∀i∈net_ips s1 ∪ net_ips s2. the ((netmap s1 ++ netmap s2) i) = σ i" by auto
    thus "∀i∈net_ips s2. the (netmap s2 i) = σ i"
      by - (erule netmap_add_disjoint2)
  qed

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

lemma inclosed_simps [simp]:
  "⋀σ ii ni. inoclosed σ (ii¬ni:arrive(m))   = eq_newpkt m"
  "⋀σ d di.  inoclosed σ (i:newpkt(d, di))   = False"
  "⋀σ R m.   inoclosed σ (R:*cast(m))        = True"
  "⋀σ i d.   inoclosed σ (i:deliver(d))      = True"
  "⋀σ i i'.  inoclosed σ (connect(i, i'))    = True"
  "⋀σ i i'.  inoclosed σ (disconnect(i, i')) = True"
  "⋀σ.       inoclosed σ (τ)                 = True"
  by auto

definition
  netmask :: "ip set ⇒ ((ip ⇒ 's) × 'l) ⇒ ((ip ⇒ 's option) × 'l)"
where
  "netmask I s ≡ (λi. if i∈I then Some (fst s i) else None, snd s)"

lemma netmask_def' [simp]:
  "netmask I (σ, ζ) = (λi. if i∈I then Some (σ i) else None, ζ)"
  unfolding netmask_def by auto

fun netgmap :: "('s ⇒ 'g × 'l) ⇒ 's net_state ⇒ (nat ⇒ 'g option) × 'l net_state"
  where
    "netgmap sr (NodeS i s R) = ([i ↦ fst (sr s)], NodeS i (snd (sr s)) R)"
  | "netgmap sr (SubnetS s₁ s₂) = (let (σ₁, ss) = netgmap sr s₁ in
                                   let (σ₂, tt) = netgmap sr s₂ in
                                   (σ₁ ++ σ₂, SubnetS ss tt))"

lemma dom_fst_netgmap [simp, intro]: "dom (fst (netgmap sr n)) = net_ips n"
  proof (induction n)
    fix i s R
    show "dom (fst (netgmap sr (NodeS i s R))) = net_ips (NodeS i s R)"
      by simp
  next
    fix n1 n2
    assume a1: "dom (fst (netgmap sr n1)) = net_ips n1"
       and a2: "dom (fst (netgmap sr n2)) = net_ips n2"
    obtain σ₁ ζ₁ σ₂ ζ₂ where nm1: "netgmap sr n1 = (σ₁, ζ₁)"
                         and nm2: "netgmap sr n2 = (σ₂, ζ₂)"
      by (metis surj_pair)
    hence "netgmap sr (SubnetS n1 n2) = (σ₁ ++ σ₂, SubnetS ζ₁ ζ₂)" by simp
    hence "dom (fst (netgmap sr (SubnetS n1 n2))) = dom (σ₁ ++ σ₂)" by simp
    also from a1 a2 nm1 nm2 have "dom (σ₁ ++ σ₂) = net_ips (SubnetS n1 n2)" by auto
    finally show "dom (fst (netgmap sr (SubnetS n1 n2))) = net_ips (SubnetS n1 n2)" .
  qed

lemma netgmap_pair_dom [elim]:
  obtains σ ζ where "netgmap sr n = (σ, ζ)"
                and "dom σ = net_ips n"
    by (metis dom_fst_netgmap surjective_pairing)
```

```
lemma net_ips_netgmap [simp]:
  "net_ips (snd (netgmap sr s)) = net_ips s"
  proof (induction s)
    fix s1 s2
    assume "net_ips (snd (netgmap sr s1)) = net_ips s1"
       and "net_ips (snd (netgmap sr s2)) = net_ips s2"
    thus "net_ips (snd (netgmap sr (SubnetS s1 s2))) = net_ips (SubnetS s1 s2)"
      by (cases "netgmap sr s1", cases "netgmap sr s2") auto
  qed simp

lemma some_the_fst_netgmap:
  assumes "i ∈ net_ips s"
    shows "Some (the (fst (netgmap sr s) i)) = fst (netgmap sr s) i"
  using assms by (metis domIff dom_fst_netgmap option.collapse)


lemma fst_netgmap_none [simp]:
  assumes "i ∉ net_ips s"
    shows "fst (netgmap sr s) i = None"
  using assms by (metis domIff dom_fst_netgmap)

lemma fst_netgmap_subnet [simp]:
  "fst (case netgmap sr s1 of (σ₁, ss) ⇒
        case netgmap sr s2 of (σ₂, tt) ⇒
        (σ₁ ++ σ₂, SubnetS ss tt)) = (fst (netgmap sr s1) ++ fst (netgmap sr s2))"
  by (metis (mono_tags) fst_conv netgmap_pair_dom split_conv)

lemma snd_netgmap_subnet [simp]:
  "snd (case netgmap sr s1 of (σ₁, ss) ⇒
        case netgmap sr s2 of (σ₂, tt) ⇒
        (σ₁ ++ σ₂, SubnetS ss tt)) = (SubnetS (snd (netgmap sr s1)) (snd (netgmap sr s2)))"
  by (metis (lifting, no_types) Pair_inject split_beta' surjective_pairing)

lemma fst_netgmap_not_none [simp]:
  assumes "i ∈ net_ips s"
    shows "fst (netgmap sr s) i ≠ None"
  using assms by (induction s) auto

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 -
    from assms(1) have "i ∉ dom (fst (netgmap sr s2))" by simp
    thus ?thesis by (simp add: map_add_dom_app_simps)
  qed

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"
  using assms by (simp add: map_add_dom_app_simps)

lemma netgmap_netmask_subnets [elim]:
  assumes "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
      and "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
    shows "fst (netgmap sr (SubnetS s1 s2))
             = fst (netmask (net_tree_ips (n1 ∥ n2)) (σ, snd (netgmap sr (SubnetS s1 s2))))"
  proof (rule ext)
    fix i
    have "i ∈ net_tree_ips n1 ∨ i ∈ net_tree_ips n2 ∨ (i∉net_tree_ips n1 ∪ net_tree_ips n2)"
      by auto
    thus "fst (netgmap sr (SubnetS s1 s2)) i
             = fst (netmask (net_tree_ips (n1 ∥ n2)) (σ, snd (netgmap sr (SubnetS s1 s2)))) i"
    proof (elim disjE)
```

```
      assume "i ∈ net_tree_ips n1"
      with ‹netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))›
           ‹netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))›
        show ?thesis
          by (cases "netgmap sr s1", cases "netgmap sr s2", clarsimp)
             (metis (lifting, mono_tags) map_add_Some_iff)
    next
      assume "i ∈ net_tree_ips n2"
      with ‹netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))›
        show ?thesis
          by simp (metis (lifting, mono_tags) fst_conv map_add_find_right)
    next
      assume "i∉net_tree_ips n1 ∪ net_tree_ips n2"
      with ‹netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))›
           ‹netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))›
        show ?thesis
          by simp (metis (lifting, mono_tags) fst_conv)
    qed
  qed

lemma netgmap_netmask_subnets' [elim]:
  assumes "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
      and "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
      and "s = SubnetS s1 s2"
    shows "netgmap sr s = netmask (net_tree_ips (n1 ∥ n2)) (σ, snd (netgmap sr s))"
  by (simp only: assms(3))
     (rule prod_eqI [OF netgmap_netmask_subnets [OF assms(1-2)]], simp)

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 (rule prod_eqI)
    show "fst (netgmap sr s1) = fst (netmask (net_tree_ips n1) (σ, snd (netgmap sr s1)))"
    proof (rule ext, simp, intro conjI impI)
      fix i
      assume "i∈net_tree_ips n1"
      with ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}› have "i∉net_tree_ips n2"
        by auto
      from assms(1) [simplified prod_eq_iff]
        have "(fst (netgmap sr s1) ++ fst (netgmap sr s2)) i =
                 (if i ∈ net_tree_ips n1 ∨ i ∈ net_tree_ips n2 then Some (σ i) else None)"
          by simp
      also from ‹i∉net_tree_ips n2› and ‹net_ips s2 = net_tree_ips n2›
        have "(fst (netgmap sr s1) ++ fst (netgmap sr s2)) i = fst (netgmap sr s1) i"
          by (metis dom_fst_netgmap map_add_dom_app_simps(3))
      finally show "fst (netgmap sr s1) i = Some (σ i)"
        using ‹i∈net_tree_ips n1› by simp
    next
      fix i
      assume "i ∉ net_tree_ips n1"
      with ‹net_ips s1 = net_tree_ips n1› have "i ∉ net_ips s1" by simp
      thus "fst (netgmap sr s1) i = None" by simp
    qed
  qed simp

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 (rule prod_eqI)
```

```
    show "fst (netgmap sr s2) = fst (netmask (net_tree_ips n2) (σ, snd (netgmap sr s2)))"
    proof (rule ext, simp, intro conjI impI)
      fix i
      assume "i∈net_tree_ips n2"
      from assms(1) [simplified prod_eq_iff]
        have "(fst (netgmap sr s1) ++ fst (netgmap sr s2)) i =
                 (if i ∈ net_tree_ips n1 ∨ i ∈ net_tree_ips n2 then Some (σ i) else None)"
          by simp
      also from ‹i∈net_tree_ips n2› and ‹net_ips s2 = net_tree_ips n2›
        have "(fst (netgmap sr s1) ++ fst (netgmap sr s2)) i = fst (netgmap sr s2) i"
          by (metis dom_fst_netgmap map_add_dom_app_simps(1))
      finally show "fst (netgmap sr s2) i = Some (σ i)"
        using ‹i∈net_tree_ips n2› by simp
    next
      fix i
      assume "i ∉ net_tree_ips n2"
      with ‹net_ips s2 = net_tree_ips n2› have "i ∉ net_ips s2" by simp
      thus "fst (netgmap sr s2) i = None" by simp
    qed
  qed simp

lemma netmap_fst_netgmap_rel:
  shows "(λi. map_option (fst o sr) (netmap s i)) = fst (netgmap sr s)"
  proof (induction s)
    fix ii s R
    show "(λi. map_option (fst ∘ sr) (netmap (NodeS ii s R) i)) = fst (netgmap sr (NodeS ii s R))"
      by auto
  next
    fix s1 s2
    assume a1: "(λi. map_option (fst ∘ sr) (netmap s1 i)) = fst (netgmap sr s1)"
       and a2: "(λi. map_option (fst ∘ sr) (netmap s2 i)) = fst (netgmap sr s2)"
    show "(λi. map_option (fst ∘ sr) (netmap (SubnetS s1 s2) i)) = fst (netgmap sr (SubnetS s1 s2))"
    proof (rule ext)
      fix i
      from a1 a2 have "map_option (fst ∘ sr) ((netmap s1 ++ netmap s2) i)
                               = (fst (netgmap sr s1) ++ fst (netgmap sr s2)) i"
        by (metis fst_conv map_add_dom_app_simps(1) map_add_dom_app_simps(3)
                  net_ips_is_dom_netmap netgmap_pair_dom)
      thus "map_option (fst ∘ sr) (netmap (SubnetS s1 s2) i) = fst (netgmap sr (SubnetS s1 s2)) i"
        by simp
    qed
  qed

lemma netmap_is_fst_netgmap':
  assumes "netmap s' i = netmap s i"
    shows "fst (netgmap sr s') i = fst (netgmap sr s) i"
  using assms by (metis netmap_fst_netgmap_rel)

lemma netmap_is_fst_netgmap:
  assumes "netmap s' = netmap s"
    shows "fst (netgmap sr s') = fst (netgmap sr s)"
  by (rule ext) (metis assms netmap_fst_netgmap_rel)

lemma fst_netgmap_pair_fst [simp]:
  "fst (netgmap (λ(p, q). (fst p, snd p, q)) s) = fst (netgmap fst s)"
  by (induction s) auto
```

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"
  by (induction s) simp_all

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"
  by (induction s) simp_all

lemma netgmap_netlift_netliftl: "netgmap sr s = (netlift sr s, netliftl sr s)"
  by rule (simp_all add: fst_netgmap_netlift snd_netgmap_netliftl)

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 ⇒ ('s × ('s, 'm, 'p, 'l) seqp, 'm seq_action) transition set"
  for Γ :: "('s, 'm, 'p, 'l) seqp_env"
where
    broadcastT: "((ξ, {l}broadcast(s_msg).p),         broadcast (s_msg ξ),          (ξ, p)) ∈ seqp_sos Γ"
  | groupcastT: "((ξ, {l}groupcast(s_ips, s_msg).p),  groupcast (s_ips ξ) (s_msg ξ), (ξ, p)) ∈ seqp_sos Γ"
  | unicastT:   "((ξ, {l}unicast(s_ip, s_msg).p ▷ q), unicast (s_ip ξ) (s_msg ξ),    (ξ, p)) ∈ seqp_sos Γ"
  | notunicastT:"((ξ, {l}unicast(s_ip, s_msg).p ▷ q),  ¬unicast (s_ip ξ),           (ξ, q)) ∈ seqp_sos Γ"
  | sendT:      "((ξ, {l}send(s_msg).p),               send (s_msg ξ),              (ξ, p)) ∈ seqp_sos Γ"
  | deliverT:   "((ξ, {l}deliver(s_data).p),          deliver (s_data ξ),          (ξ, p)) ∈ seqp_sos Γ"
  | receiveT:   "((ξ, {l}receive(u_msg).p),            receive msg,      (u_msg msg ξ, p)) ∈ seqp_sos Γ"
  | assignT:    "((ξ, {l}⟦u⟧ p),                       τ,                       (u ξ, p)) ∈ seqp_sos Γ"

  | callT:      "⟦ ((ξ, Γ pn), a, (ξ', p')) ∈ seqp_sos Γ ⟧ ⟹
                 ((ξ, call(pn)), a, (ξ', p')) ∈ seqp_sos Γ"

  | choiceT1:   "((ξ, p), a, (ξ', p')) ∈ seqp_sos Γ  ⟹ ((ξ, p ⊕ q), a, (ξ', p')) ∈ seqp_sos Γ"
  | choiceT2:   "((ξ, q), a, (ξ', q')) ∈ seqp_sos Γ  ⟹ ((ξ, p ⊕ q), a, (ξ', q')) ∈ seqp_sos Γ"

  | guardT:     "ξ' ∈ g ξ ⟹ ((ξ, {l}⟨g⟩ p), τ, (ξ', p)) ∈ seqp_sos Γ"

inductive_cases
      seqp_callTE [elim]:       "((ξ, call(pn)), a, (ξ', q)) ∈ seqp_sos Γ"
  and seqp_choiceTE [elim]:     "((ξ, p1 ⊕ p2), a, (ξ', q)) ∈ seqp_sos Γ"

lemma seqp_broadcastTE [elim]:
  "⟦((ξ, {l}broadcast(s_msg). p), a, (ξ', q)) ∈ seqp_sos Γ;
    ⟦a = broadcast (s_msg ξ); ξ' = ξ; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((ξ, {l}broadcast(s_msg). p), a, (ξ', q)) ∈ seqp_sos Γ") simp

lemma seqp_groupcastTE [elim]:
  "⟦((ξ, {l}groupcast(s_ips, s_msg). p), a, (ξ', q)) ∈ seqp_sos Γ;
    ⟦a = groupcast (s_ips ξ) (s_msg ξ); ξ' = ξ; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((ξ, {l}groupcast(s_ips, s_msg). p), a, (ξ', q)) ∈ seqp_sos Γ") simp

lemma seqp_unicastTE [elim]:
  "⟦((ξ, {l}unicast(s_ip, s_msg). p ▷ q), a, (ξ', r)) ∈ seqp_sos Γ;
```

```
    ⟦a = unicast (s_ip ξ) (s_msg ξ); ξ' = ξ; r = p⟧ ⟹ P;
    ⟦a = ¬unicast (s_ip ξ); ξ' = ξ; r = q⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((ξ, {l}unicast(s_ip, s_msg). p ▷ q), a, (ξ', r)) ∈ seqp_sos Γ") simp_all
```

lemma *seqp_sendTE [elim]:*

```
  "⟦((ξ, {l}send(s_msg). p), a, (ξ', q)) ∈ seqp_sos Γ;
    ⟦a = send (s_msg ξ); ξ' = ξ; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((ξ, {l}send(s_msg). p), a, (ξ', q)) ∈ seqp_sos Γ") simp
```

lemma *seqp_deliverTE [elim]:*

```
  "⟦((ξ, {l}deliver(s_data). p), a, (ξ', q)) ∈ seqp_sos Γ;
    ⟦a = deliver (s_data ξ); ξ' = ξ; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((ξ, {l}deliver(s_data). p), a, (ξ', q)) ∈ seqp_sos Γ") simp
```

lemma *seqp_receiveTE [elim]:*

```
  "⟦((ξ, {l}receive(u_msg). p), a, (ξ', q)) ∈ seqp_sos Γ;
    ⋀msg. ⟦a = receive msg; ξ' = u_msg msg ξ; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((ξ, {l}receive(u_msg). p), a, (ξ', q)) ∈ seqp_sos Γ") simp
```

lemma *seqp_assignTE [elim]:*

```
  "⟦((ξ, {l}⟦u⟧ p), a, (ξ', q)) ∈ seqp_sos Γ; ⟦a = τ; ξ' = u ξ; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((ξ, {l}⟦u⟧ p), a, (ξ', q)) ∈ seqp_sos Γ") simp
```

lemma *seqp_guardTE [elim]:*

```
  "⟦((ξ, {l}⟨g⟩ p), a, (ξ', q)) ∈ seqp_sos Γ; ⟦a = τ; ξ' ∈ g ξ; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((ξ, {l}⟨g⟩ p), a, (ξ', q)) ∈ seqp_sos Γ") simp
```

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
                  ⇒ ('s2, 'm seq_action) transition set
                  ⇒ ('s1 × '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') ∈ S; ⋀m. a ≠ receive m ⟧ ⟹ ((s, t), a, (s', t)) ∈ parp_sos S T"
  | parright: "⟦ (t, a, t') ∈ T; ⋀m. a ≠ send m ⟧ ⟹ ((s, t), a, (s, t')) ∈ parp_sos S T"
  | parboth:  "⟦ (s, receive m, s') ∈ S; (t, send m, t') ∈ T ⟧
               ⟹((s, t), τ, (s', t')) ∈ parp_sos S T"
```

lemma *par_broadcastTE [elim]:*

```
  "⟦((s, t), broadcast m, (s', t')) ∈ parp_sos S T;
    ⟦(s, broadcast m, s') ∈ S; t' = t⟧ ⟹ P;
    ⟦(t, broadcast m, t') ∈ T; s' = s⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((s, t), broadcast m, (s', t')) ∈ parp_sos S T") simp_all
```

lemma *par_groupcastTE [elim]:*

```
  "⟦((s, t), groupcast ips m, (s', t')) ∈ parp_sos S T;
    ⟦(s, groupcast ips m, s') ∈ S; t' = t⟧ ⟹ P;
```

```
    ⟦(t, groupcast ips m, t') ∈ T; s' = s⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((s, t), groupcast ips m, (s', t')) ∈ parp_sos S T") simp_all

lemma par_unicastTE [elim]:
  "⟦((s, t), unicast i m, (s', t')) ∈ parp_sos S T;
    ⟦(s, unicast i m, s') ∈ S; t' = t⟧ ⟹ P;
    ⟦(t, unicast i m, t') ∈ T; s' = s⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((s, t), unicast i m, (s', t')) ∈ parp_sos S T") simp_all

lemma par_notunicastTE [elim]:
  "⟦((s, t), notunicast i, (s', t')) ∈ parp_sos S T;
    ⟦(s, notunicast i, s') ∈ S; t' = t⟧ ⟹ P;
    ⟦(t, notunicast i, t') ∈ T; s' = s⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((s, t), notunicast i, (s', t')) ∈ parp_sos S T") simp_all

lemma par_sendTE [elim]:
  "⟦((s, t), send m, (s', t')) ∈ parp_sos S T;
    ⟦(s, send m, s') ∈ S; t' = t⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((s, t), send m, (s', t')) ∈ parp_sos S T") auto

lemma par_deliverTE [elim]:
  "⟦((s, t), deliver d, (s', t')) ∈ parp_sos S T;
    ⟦(s, deliver d, s') ∈ S; t' = t⟧ ⟹ P;
    ⟦(t, deliver d, t') ∈ T; s' = s⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((s, t), deliver d, (s', t')) ∈ parp_sos S T") simp_all

lemma par_receiveTE [elim]:
  "⟦((s, t), receive m, (s', t')) ∈ parp_sos S T;
    ⟦(t, receive m, t') ∈ T; s' = s⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((s, t), receive m, (s', t')) ∈ parp_sos S T") auto

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"
  using assms by cases auto

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)"
  unfolding par_comp_def by simp

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

```
unfolding par_comp_def by simp
```

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}¬{}: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}: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)"
  using assms by cases auto

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})"
  using assms by rule simp_all

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)"
  unfolding node_comp_def by simp

lemma init_node_comp:
  "init (⟨i : np : Rᵢ⟩) = {NodeS i s Rᵢ|s. s ∈ init np}"
  unfolding node_comp_def by simp

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))"
  unfolding node_comp_def by simp

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

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

lemma node_sos_dest:
  assumes "(NodeS i p R, a, s') ∈ node_sos S"
    shows "∃P' R'. s' = NodeS i P' R'"
  using assms assms [THEN node_sos_dest_is_net_state]
  by - (erule node_sos.cases, auto)

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'"
  proof -
    assume [intro!]: "⋀i s R s' R'. ns = NodeS i s R ⟹ ns' = NodeS i s' R' ⟹ thesis"
    from assms(1) obtain i s R where "ns = NodeS i s R"
      by (cases ns) auto
    moreover with assms(1) obtain s' R' where "ns' = NodeS i s' R'"
      by (metis node_sos_dest)
    ultimately show thesis ..
  qed

lemma node_sos_cases [elim]:
  "(NodeS i p R, a, NodeS i p' R') ∈ node_sos S ⟹
  (⋀m .       ⟦ a = R:*cast(m);          R' = R; (p, broadcast m, p') ∈ S ⟧ ⟹ P) ⟹
  (⋀m D.      ⟦ a = (R ∩ D):*cast(m);    R' = R; (p, groupcast D m, p') ∈ S ⟧ ⟹ P) ⟹
  (⋀d m.      ⟦ a = {d}:*cast(m);        R' = R; (p, unicast d m, p') ∈ S; d ∈ R ⟧ ⟹ P) ⟹
  (⋀d.        ⟦ a = τ;                   R' = R; (p, ¬unicast d, p') ∈ S; d ∉ R ⟧ ⟹ P) ⟹
  (⋀d.        ⟦ a = i:deliver(d);        R' = R; (p, deliver d, p') ∈ S ⟧ ⟹ P) ⟹
  (⋀m.        ⟦ a = {i}¬{}:arrive(m);    R' = R; (p, receive m, p') ∈ S ⟧ ⟹ P) ⟹
  (           ⟦ a = τ;                   R' = R; (p, τ, p') ∈ S ⟧ ⟹ P) ⟹
  (⋀m.        ⟦ a = {}¬{i}:arrive(m);    R' = R; p = p' ⟧ ⟹ P) ⟹
  (⋀i i'.     ⟦ a = connect(i, i');      R' = R ∪ {i'}; p = p' ⟧ ⟹ P) ⟹
  (⋀i i'.     ⟦ a = connect(i', i);      R' = R ∪ {i'}; p = p' ⟧ ⟹ P) ⟹
  (⋀i i'.     ⟦ a = disconnect(i, i');   R' = R - {i'}; p = p' ⟧ ⟹ P) ⟹
  (⋀i i'.     ⟦ a = disconnect(i', i);   R' = R - {i'}; p = p' ⟧ ⟹ P) ⟹
  (⋀i i' i''. ⟦ a = connect(i', i'');    R' = R; p = p'; i ≠ i'; i ≠ i'' ⟧ ⟹ P) ⟹
  (⋀i i' i''. ⟦ a = disconnect(i', i''); R' = R; p = p'; i ≠ i'; i ≠ i'' ⟧ ⟹ P) ⟹
  P"
  by (erule node_sos.cases) simp_all
```

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: "⟦ (s, R:*cast(m), s') ∈ S; (t, H¬K:arrive(m), t') ∈ T; H ⊆ R; K ∩ R = {} ⟧
      ⟹ (SubnetS s t, R:*cast(m), SubnetS s' t') ∈ pnet_sos S T"

  | pnet_cast2: "⟦ (s, H¬K:arrive(m), s') ∈ S; (t, R:*cast(m), t') ∈ T;  H ⊆ R; K ∩ R = {} ⟧
      ⟹ (SubnetS s t, R:*cast(m), SubnetS s' t') ∈ pnet_sos S T"

  | pnet_arrive: "⟦ (s, H¬K:arrive(m), s') ∈ S; (t, H'¬K':arrive(m), t') ∈ T ⟧
      ⟹ (SubnetS s t,  (H ∪ H')¬(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: "⟦ (t, i:deliver(d), t') ∈ T ⟧
      ⟹ (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: "⟦ (s, connect(i, i'), s') ∈ S; (t, connect(i, i'), t') ∈ T ⟧
      ⟹ (SubnetS s t, connect(i, i'), SubnetS s' t') ∈ pnet_sos S T"

  | pnet_disconnect: "⟦ (s, disconnect(i, i'), s') ∈ S; (t, disconnect(i, i'), t') ∈ T ⟧
      ⟹ (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)"
  using assms(1) by cases (auto dest!: assms(2-3))

fun pnet :: "(ip ⇒ ('s, 'm seq_action) automaton)
              ⇒ net_tree ⇒ ('s net_state, 'm node_action) automaton"
where
    "pnet np (⟨i; Rᵢ⟩)  =  ⟨i : np i : Rᵢ⟩"
  | "pnet np (p₁ ∥ p₂) = ⦇ init = {SubnetS s₁ s₂ |s₁ s₂. s₁ ∈ init (pnet np p₁)
                                                  ∧ s₂ ∈ init (pnet np p₂)},
                          trans = pnet_sos (trans (pnet np p₁)) (trans (pnet np p₂)) ⦈"

lemma pnet_node_init [elim, simp]:
  assumes "s ∈ init (pnet np ⟨i; R⟩)"
    shows "s ∈ { NodeS i s R |s. s ∈ init (np i)}"
  using assms by (simp add: node_comp_def)

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)"
   using assms by (auto simp add: node_comp_def)
```

```
lemma pnet_node_trans [elim, simp]:
  assumes "(s, a, s') ∈ trans (pnet np ⟨i; R⟩)"
    shows "(s, a, s') ∈ node_sos (trans (np i))"
  using assms by (simp add: trans_node_comp)

lemma pnet_never_newpkt':
  assumes "(s, a, s') ∈ trans (pnet np n)"
    shows "∀ i d di. a ≠ i:newpkt(d, di)"
  using assms proof (induction n arbitrary: s a s')
    fix n1 n2 s a s'
    assume IH1: "⋀s a s'. (s, a, s') ∈ trans (pnet np n1) ⟹ ∀ i d di. a ≠ i:newpkt(d, di)"
       and IH2: "⋀s a s'. (s, a, s') ∈ trans (pnet np n2) ⟹ ∀ i d di. a ≠ i:newpkt(d, di)"
       and "(s, a, s') ∈ trans (pnet np (n1 ∥ n2))"
    show "∀ i d di. a ≠ i:newpkt(d, di)"
    proof (intro allI)
      fix i d di
      from ‹(s, a, s') ∈ trans (pnet np (n1 ∥ n2))›
        have "(s, a, s') ∈ pnet_sos (trans (pnet np n1)) (trans (pnet np n2))"
          by simp
      thus "a ≠ i:newpkt(d, di)"
        by (rule pnet_sos_never_newpkt) (auto dest!: IH1 IH2)
    qed
  qed (simp add: node_comps)

lemma pnet_never_newpkt:
  assumes "(s, a, s') ∈ trans (pnet np n)"
    shows "a ≠ i:newpkt(d, di)"
  proof -
    from assms have "∀ i d di. a ≠ i:newpkt(d, di)"
      by (rule pnet_never_newpkt')
    thus ?thesis by clarsimp
  qed
```

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, τ, s') ∈ cnet_sos S"
          and deliver_completeTE: "(s, i:deliver(d), s') ∈ cnet_sos S"
          and newpkt_completeTE: "(s, i:newpkt(d, di), s') ∈ 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') ∉ cnet_sos T"
  proof
    assume "(s, R:*cast(m), s') ∈ cnet_sos T"
    hence "R:*cast(m) ≠ R:*cast(m)"
```

```
    by (rule cnet_sos.cases) auto
   thus False by simp
  qed

lemma complete_no_arrive [simp]:
  "(s, ii¬ni:arrive(m), s') ∉ cnet_sos T"
  proof
   assume "(s, ii¬ni:arrive(m), s') ∈ cnet_sos T"
   hence "ii¬ni:arrive(m) ≠ ii¬ni:arrive(m)"
    by (rule cnet_sos.cases) auto
   thus False by simp
  qed

abbreviation
  closed :: "('s net_state, ('m::msg) node_action) automaton ⇒ ('s net_state, 'm node_action) automaton"
where
  "closed ≡ (λA. A ⦇ trans := cnet_sos (trans A) ⦈)"

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
              ⇒ ('s, 'm, 'p, 'l) seqp
              ⇒ ('s, 'm, 'p, 'l) seqp
              ⇒ bool"
for Γ :: "('s, 'm, 'p, 'l) seqp_env"
where
    microstep_choiceI1 [intro, simp]: "microstep Γ (p1 ⊕ p2) p1"
  | microstep_choiceI2 [intro, simp]: "microstep Γ (p1 ⊕ p2) p2"
  | microstep_callI [intro, simp]: "microstep Γ (call(pn)) (Γ pn)"

abbreviation microstep_rtcl
where "microstep_rtcl Γ p q ≡ (microstep Γ)** p q"

abbreviation microstep_tcl
where "microstep_tcl Γ p q ≡ (microstep Γ)++ p q"

syntax
  "_microstep"
    :: "[('s, 'm, 'p, 'l) seqp, ('s, 'm, 'p, 'l) seqp_env, ('s, 'm, 'p, 'l) seqp] ⇒ bool"
    ((‹(_) ↝_ (_)› [61, 0, 61] 50)
  "_microstep_rtcl"
    :: "[('s, 'm, 'p, 'l) seqp, ('s, 'm, 'p, 'l) seqp_env, ('s, 'm, 'p, 'l) seqp] ⇒ bool"
    ((‹(_) ↝_* (_)› [61, 0, 61] 50)
  "_microstep_tcl"
    :: "[('s, 'm, 'p, 'l) seqp, ('s, 'm, 'p, 'l) seqp_env, ('s, 'm, 'p, 'l) seqp] ⇒ bool"
    ((‹(_) ↝_+ (_)› [61, 0, 61] 50)

syntax_consts
  "_microstep" ⇌ microstep and
  "_microstep_rtcl" ⇌ microstep_rtcl and
  "_microstep_tcl" ⇌ microstep_tcl
```

```
translations
  "p1 ↝Γ p2"  ⇌ "CONST microstep Γ p1 p2"
  "p1 ↝Γ* p2" ⇌ "CONST microstep_rtcl Γ p1 p2"
  "p1 ↝Γ+ p2" ⇌ "CONST microstep_tcl Γ p1 p2"

lemma microstep_choiceD [dest]:
  "(p1 ⊕ p2) ↝Γ p ⟹ p = p1 ∨ p = p2"
  by (ind_cases "(p1 ⊕ p2) ↝Γ p") auto

lemma microstep_choiceE [elim]:
  "⟦ (p1 ⊕ p2) ↝Γ p;
     (p1 ⊕ p2) ↝Γ p1 ⟹ P;
     (p1 ⊕ p2) ↝Γ p2 ⟹ P ⟧ ⟹ P"
  by (blast)

lemma microstep_callD [dest]:
  "(call(pn)) ↝Γ p ⟹ p = Γ pn"
  by (ind_cases "(call(pn)) ↝Γ p")

lemma microstep_callE [elim]:
  "⟦ (call(pn)) ↝Γ p;  p = Γ(pn) ⟹ P ⟧ ⟹ P"
  by auto

lemma no_microstep_guard: "¬ (({l}⟨g⟩ p) ↝Γ q)"
  by (rule notI) (ind_cases "({l}⟨g⟩ p) ↝Γ q")

lemma no_microstep_assign: "¬ ({l}⟦f⟧ p) ↝Γ q"
  by (rule notI) (ind_cases "({l}⟦f⟧ p) ↝Γ q")

lemma no_microstep_unicast: "¬ (({l}unicast(s_ip, s_msg).p ▷ q) ↝Γ r)"
  by (rule notI) (ind_cases "({l}unicast(s_ip, s_msg).p ▷ q) ↝Γ r")

lemma no_microstep_broadcast: "¬ (({l}broadcast(s_msg).p) ↝Γ q)"
  by (rule notI) (ind_cases "({l}broadcast(s_msg).p) ↝Γ q")

lemma no_microstep_groupcast: "¬ (({l}groupcast(s_ips, s_msg).p) ↝Γ q)"
  by (rule notI) (ind_cases "({l}groupcast(s_ips, s_msg).p) ↝Γ q")

lemma no_microstep_send: "¬ (({l}send(s_msg).p) ↝Γ q)"
  by (rule notI) (ind_cases "({l}send(s_msg).p) ↝Γ q")

lemma no_microstep_deliver: "¬ (({l}deliver(s_data).p) ↝Γ q)"
  by (rule notI) (ind_cases "({l}deliver(s_data).p) ↝Γ q")

lemma no_microstep_receive: "¬ (({l}receive(u_msg).p) ↝Γ q)"
  by (rule notI) (ind_cases "({l}receive(u_msg).p) ↝Γ q")

lemma microstep_call_or_choice [dest]:
  assumes "p ↝Γ q"
    shows "(∃pn. p = call(pn)) ∨ (∃p1 p2. p = p1 ⊕ p2)"
  using assms by clarsimp (metis microstep.simps)

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: Γ_1 *p1 = call(p1)*

Γ_2 *p1 = send(msg) .*
call(p1)
⊕
call(p1)

Γ_3 *p1 = send(msg) .*

call(p2) Γ_3 *p2 = call(p3)* Γ_3 *p3 = call(p4)* Γ_3 *p4 = call(p5)* ...

```
definition
  wellformed :: "('s, 'm, 'p, 'l) seqp_env ⇒ bool"
where
  "wellformed Γ = wf {(q, p). p ↝Γ q}"

lemma wellformed_defP: "wellformed Γ = wfP (λq p. p ↝Γ q)"
  unfolding wellformed_def wfp_def by simp
```

The induction rule for *wellformed* Γ is stronger than ⟦⋀x1 x2 x3. ?P x3 ⟹ ?P ({x1}⟨x2⟩ x3); ⋀x1 x2 x3. ?P x3 ⟹ ?P ({x1}⟦x2⟧ x3); ⋀x1 x2. ⟦?P x1; ?P x2⟧ ⟹ ?P (x1 ⊕ x2); ⋀x1 x2 x3 x4 x5. ⟦?P x4; ?P x5⟧ ⟹ ?P ({x1}unicast(x2, x3) . x4 ▷ x5); ⋀x1 x2 x3. ?P x3 ⟹ ?P ({x1}broadcast(x2) . x3); ⋀x1 x2 x3 x4. ?P x4 ⟹ ?P ({x1}groupcast(x2, x3) . x4); ⋀x1 x2 x3. ?P x3 ⟹ ?P ({x1}send(x2) . x3); ⋀x1 x2 x3. ?P x3 ⟹ ?P ({x1}deliver(x2) . x3); ⋀x1 x2 x3. ?P x3 ⟹ ?P ({x1}receive(x2) . x3); ⋀x. ?P (call(x))⟧ ⟹ ?P ?seqp because the case for *call(pn)* can be shown with the assumption on Γ *pn*.

```
lemma wellformed_induct
  [consumes 1, case_names ASSIGN CHOICE CALL GUARD UCAST BCAST GCAST SEND DELIVER RECEIVE,
   induct set: wellformed]:
  assumes "wellformed Γ"
      and ASSIGN:  "⋀l f p.          wellformed Γ ⟹ P ({l}⟦f⟧ p)"
      and GUARD:   "⋀l f p.          wellformed Γ ⟹ P ({l}⟨f⟩ p)"
      and UCAST:   "⋀l fip fmsg p q. wellformed Γ ⟹ P ({l}unicast(fip, fmsg). p ▷ q)"
      and BCAST:   "⋀l fmsg p.       wellformed Γ ⟹ P ({l}broadcast(fmsg). p)"
      and GCAST:   "⋀l fips fmsg p.  wellformed Γ ⟹ P ({l}groupcast(fips, fmsg). p)"
      and SEND:    "⋀l fmsg p.       wellformed Γ ⟹ P ({l}send(fmsg). p)"
      and DELIVER: "⋀l fdata p.      wellformed Γ ⟹ P ({l}deliver(fdata). p)"
      and RECEIVE: "⋀l fmsg p.       wellformed Γ ⟹ P ({l}receive(fmsg). p)"
      and CHOICE:  "⋀p1 p2.         ⟦ wellformed Γ; P p1; P p2 ⟧ ⟹ P (p1 ⊕ p2)"
      and CALL:    "⋀pn.            ⟦ wellformed Γ; P (Γ pn) ⟧ ⟹ P (call(pn))"
    shows "P a"
  using assms(1) unfolding wellformed_defP
  proof (rule wfp_induct_rule, case_tac x, simp_all)
    fix p1 p2
    assume "⋀q. (p1 ⊕ p2) ↝Γ q ⟹ P q"
    then obtain "P p1" and "P p2" by (auto intro!: microstep.intros)
    thus "P (p1 ⊕ p2)" by (rule CHOICE [OF ‹wellformed Γ›])
  next
    fix pn
    assume "⋀q. (call(pn)) ↝Γ q ⟹ P q"
    hence "P (Γ pn)" by (auto intro!: microstep.intros)
    thus "P (call(pn))" by (rule CALL [OF ‹wellformed Γ›])
  qed (auto intro: assms)
```

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*

⊕

p2 are traversed, and not *call(p)* terms.

```
fun stermsl :: "('s, 'm, 'p, 'l) seqp ⇒ ('s, 'm, 'p , 'l) seqp set"
where
    "stermsl (p1 ⊕ p2) = stermsl p1 ∪ stermsl p2"
  | "stermsl p         = {p}"

lemma stermsl_nobigger: "q ∈ stermsl p ⟹ size q ≤ size p"
  by (induct p) auto

lemma stermsl_no_choice[simp]: "p1 ⊕ p2 ∉ stermsl p"
  by (induct p) simp_all

lemma stermsl_choice_disj[simp]:
  "p ∈ stermsl (p1 ⊕ p2) = (p ∈ stermsl p1 ∨ p ∈ stermsl p2)"
  by simp

lemma stermsl_in_branch[elim]:
  "⟦p ∈ stermsl (p1 ⊕ p2); p ∈ stermsl p1 ⟹ P; p ∈ stermsl p2 ⟹ P⟧ ⟹ P"
  by auto

lemma stermsl_commute:
  "stermsl (p1 ⊕ p2) = stermsl (p2 ⊕ p1)"
  by simp (rule Un_commute)

lemma stermsl_not_empty:
  "stermsl p ≠ {}"
  by (induct p) auto

lemma stermsl_idem [simp]:
  "(⋃q∈stermsl p. stermsl q) = stermsl p"
  by (induct p) simp_all

lemma stermsl_in_wfpf:
  assumes AA: "A ⊆ {(q, p). p ↝Γ q} `` A"
      and *: "p ∈ A"
    shows "∃r∈stermsl p. r ∈ A"
  using *
  proof (induction p)
    fix p1 p2
    assume IH1: "p1 ∈ A ⟹ ∃r∈stermsl p1. r ∈ A"
       and IH2: "p2 ∈ A ⟹ ∃r∈stermsl p2. r ∈ A"
       and *: "p1 ⊕ p2 ∈ A"
    from * and AA have "p1 ⊕ p2 ∈ {(q, p). p ↝Γ q} `` A" by auto
    hence "p1 ∈ A ∨ p2 ∈ A" by auto
    hence "(∃r∈stermsl p1. r ∈ A) ∨ (∃r∈stermsl p2. r ∈ A)"
      proof
        assume "p1 ∈ A" hence "∃r∈stermsl p1. r ∈ A" by (rule IH1) thus ?thesis ..
      next
        assume "p2 ∈ A" hence "∃r∈stermsl p2. r ∈ A" by (rule IH2) thus ?thesis ..
      qed
      hence "∃r∈stermsl p1 ∪ stermsl p2. r ∈ A" by blast
      thus "∃r∈stermsl (p1 ⊕ p2). r ∈ A" by simp
    next case UCAST from UCAST.prems show ?case by auto
  qed auto

lemma nocall_stermsl_max:
  assumes "r ∈ stermsl p"
      and "not_call r"
```

```
    shows "¬ (r ↝Γ q)"
  using assms
  by (induction p) auto

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"
  unfolding wellformed_def wfp_def
  proof (rule wfI_pf)
    fix A
    assume ARA: "A ⊆ {(q, p). p ↝Γ q} `` A"
    hence hasnext: "⋀p. p ∈ A ⟹ ∃q. p ↝Γ q ∧ q ∈ A" by auto
    show "A = {}"
    proof (rule Set.equals0I)
      fix p assume "p ∈ A" thus "False"
      proof (induction p)
        fix l f p'
        assume *: "{l}⟨f⟩ p' ∈ A"
        from hasnext [OF *] have "∃q. ({l}⟨f⟩ p') ↝Γ q" by simp
        thus "False" by simp
      next
        fix p1 p2
        assume *: "p1 ⊕ p2 ∈ A"
         and IH1: "p1 ∈ A ⟹ False"
         and IH2: "p2 ∈ A ⟹ False"
        have "∃q. (p1 ⊕ p2) ↝Γ q ∧ q ∈ A" by (rule hasnext [OF *])
        hence "p1 ∈ A ∨ p2 ∈ A" by auto
        thus "False" by (auto dest: IH1 IH2)
      next
        fix pn
        assume "call(pn) ∈ A"
        hence "∃q. (call(pn)) ↝Γ q ∧ q ∈ A" by (rule hasnext)
        hence "Γ(pn) ∈ A" by auto

        with ARA [THEN stermsl_in_wfpf] obtain q where "q∈stermsl (Γ pn)" and "q ∈ A" by metis
        hence "not_call q" using no_calls [of pn]
          unfolding not_call_def by auto

        from hasnext [OF ‹q ∈ A›] obtain q' where "q ↝Γ q'" by auto
        moreover from  ‹q ∈ stermsl (Γ pn)› ‹not_call q› have "¬ (q ↝Γ q')"
          by (rule nocall_stermsl_max)
        ultimately show "False" by simp
      qed (auto dest: hasnext)
    qed
  qed
```

7.4 Start terms

The start terms are those terms, relative to a wellformed process specification Γ, 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}"
  by pat_completeness auto

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

```
  proof (rule accpI)
    fix y
    assume "sterms_rel y (Γ, p)"
    with assms show "sterms_dom y"
      by (cases p) (auto simp: sterms_rel.simps)
  qed

lemma sterms_termination:
  assumes "wellformed Γ"
    shows "sterms_dom (Γ, p)"
  proof -
    have sterms_rel':
      "sterms_rel = (λgq gp. (gq, gp) ∈ {((Γ, q), (Γ', p)). Γ = Γ' ∧ p ⇝Γ q})"
      by (rule ext)+ (auto simp: sterms_rel.simps elim: microstep.cases)

    from assms have "∀x. x ∈ Wellfounded.acc {(q, p). p ⇝Γ q}"
      unfolding wellformed_def by (simp add: wf_iff_acc)
    hence "p ∈ Wellfounded.acc {(q, p). p ⇝Γ q}" ..

    hence "(Γ, p) ∈ Wellfounded.acc {((Γ, q), (Γ', p)). Γ = Γ' ∧ p ⇝Γ q}"
      by (rule acc_induct) (auto intro: accI)

    thus "sterms_dom (Γ, p)" unfolding sterms_rel' accp_acc_eq .
  qed

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"
  using assms by (cases p) auto

lemma sterms_choice_disj [simp]:
  assumes "wellformed Γ"
    shows "p ∈ sterms Γ (p1 ⊕ p2) = (p ∈ sterms Γ p1 ∨ p ∈ sterms Γ p2)"
  using assms by (simp)

lemma sterms_no_choice [simp]:
  assumes "wellformed Γ"
    shows "p1 ⊕ p2 ∉ sterms Γ p"
  using assms by induction auto

lemma sterms_not_choice [simp]:
  assumes "wellformed Γ"
      and "q ∈ sterms Γ p"
    shows "not_choice q"
  using assms unfolding not_choice_def
  by (auto dest: sterms_no_choice)

lemma sterms_no_call [simp]:
  assumes "wellformed Γ"
    shows "call(pn) ∉ sterms Γ p"
  using assms by induction auto

lemma sterms_not_call [simp]:
  assumes "wellformed Γ"
      and "q ∈ sterms Γ p"
    shows "not_call q"
  using assms unfolding not_call_def
  by (auto dest: sterms_no_call)
```

```
lemma sterms_in_branch:
  assumes "wellformed Γ"
      and "p ∈ sterms Γ (p1 ⊕ p2)"
      and "p ∈ sterms Γ p1 ⟹ P"
      and "p ∈ sterms Γ p2 ⟹ P"
  shows "P"
  using assms by auto

lemma sterms_commute:
  assumes "wellformed Γ"
    shows "sterms Γ (p1 ⊕ p2) = sterms Γ (p2 ⊕ p1)"
  using assms by simp (rule Un_commute)

lemma sterms_not_empty:
  assumes "wellformed Γ"
    shows "sterms Γ p ≠ {}"
  using assms
  by (induct p rule: sterms_pinduct [OF ‹wellformed Γ›]) simp_all

lemma sterms_sterms [simp]:
  assumes "wellformed Γ"
    shows "(⋃x∈sterms Γ p. sterms Γ x) = sterms Γ p"
  using assms by induction simp_all

lemma sterms_stermsl:
  assumes "ps ∈ sterms Γ p"
      and "wellformed Γ"
    shows "ps ∈ stermsl p ∨ (∃pn. ps ∈ stermsl (Γ pn))"
  using assms by (induction p rule: sterms_pinduct [OF ‹wellformed Γ›]) auto

lemma stermsl_sterms [elim]:
  assumes "q ∈ stermsl p"
      and "not_call q"
      and "wellformed Γ"
    shows "q ∈ sterms Γ p"
  using assms by (induct p) auto

lemma sterms_stermsl_heads:
  assumes "ps ∈ sterms Γ (Γ pn)"
      and "wellformed Γ"
    shows "∃pn. ps ∈ stermsl (Γ pn)"
  proof -
    from assms have "ps ∈ stermsl (Γ pn) ∨ (∃pn'. ps ∈ stermsl (Γ pn'))"
      by (rule sterms_stermsl)
    thus ?thesis by auto
  qed

lemma sterms_subterms [dest]:
  assumes "wellformed Γ"
      and "∃pn. p ∈ subterms (Γ pn)"
      and "q ∈ sterms Γ p"
    shows "∃pn. q ∈ subterms (Γ pn)"
  using assms by (induct p) auto

lemma no_microsteps_sterms_refl:
  assumes "wellformed Γ"
  shows "(¬(∃q. p ↝Γ q)) = (sterms Γ p = {p})"
  proof (cases p)
    fix p1 p2
    assume "p = p1 ⊕ p2"
    from ‹wellformed Γ› have "p1 ⊕ p2 ∉ sterms Γ (p1 ⊕ p2)" by simp
    hence "sterms Γ (p1 ⊕ p2) ≠ {p1 ⊕ p2}" by auto
    moreover have "∃q. (p1 ⊕ p2) ↝Γ q" by auto
```

```
    ultimately show ?thesis
      using ‹p = p1 ⊕ p2› by simp
  next
    fix pn
    assume "p = call(pn)"
    from ‹wellformed Γ› have "call(pn) ∉ sterms Γ (call(pn))" by simp
    hence "sterms Γ (call(pn)) ≠ {call(pn)}" by auto
    moreover have "∃q. (call(pn)) ↝Γ q" by auto
    ultimately show ?thesis
      using ‹p = call(pn)› by simp
  qed simp_all

lemma sterms_maximal [elim]:
  assumes "wellformed Γ"
      and "q ∈ sterms Γ p"
    shows "sterms Γ q = {q}"
  using assms by (cases q) auto

lemma microstep_rtranscl_equal:
  assumes "not_call p"
      and "not_choice p"
      and "p ↝Γ* q"
    shows "q = p"
  using assms(3) proof (rule converse_rtranclpE)
    fix p'
    assume "p ↝Γ p'"
    with assms(1-2) show "q = p"
      by (cases p) simp_all
  qed simp

lemma microstep_rtranscl_singleton [simp]:
  assumes "not_call p"
      and "not_choice p"
    shows "{q. p ↝Γ* q ∧ sterms Γ q = {q}} = {p}"
  proof (rule set_eqI)
    fix p'
    show "(p' ∈ {q. p ↝Γ* q ∧ sterms Γ q = {q}}) = (p' ∈ {p})"
    proof
      assume "p' ∈ {q. p ↝Γ* q ∧ sterms Γ q = {q}}"
      hence "(microstep Γ)** p p'" and "sterms Γ p' = {p'}" by auto
      from this(1) have "p' = p"
      proof (rule converse_rtranclpE)
        fix q assume "p ↝Γ q"
        with ‹not_call p› and ‹not_choice p› have False
          by (cases p) auto
        thus "p' = p" ..
      qed simp
      thus "p' ∈ {p}" by simp
    next
      assume "p' ∈ {p}"
      hence "p' = p" ..
      with ‹not_call p› and ‹not_choice p› show "p' ∈ {q. p ↝Γ* q ∧ sterms Γ q = {q}}"
        by (cases p) simp_all
    qed
  qed

theorem sterms_maximal_microstep:
  assumes "wellformed Γ"
    shows "sterms Γ p = {q. p ↝Γ* q ∧ ¬(∃q'. q ↝Γ q')}"
  proof
    from ‹wellformed Γ› have "sterms Γ p ⊆ {q. p ↝Γ* q ∧ sterms Γ q = {q}}"
    proof induction
     fix p1 p2
     assume IH1: "sterms Γ p1 ⊆ {q. p1 ↝Γ* q ∧ sterms Γ q = {q}}"
```

```
      and IH2: "sterms Γ p2 ⊆ {q. p2 ⇝Γ* q ∧ sterms Γ q = {q}}"
   have "sterms Γ p1 ⊆ {q. (p1 ⊕ p2) ⇝Γ* q ∧ sterms Γ q = {q}}"
   proof
     fix p'
     assume "p' ∈ sterms Γ p1"
     with IH1 have "p1 ⇝Γ* p'" by auto
     moreover have "(p1 ⊕ p2) ⇝Γ p1" ..
     ultimately have "(p1 ⊕ p2) ⇝Γ* p'"
       by - (rule converse_rtranclp_into_rtranclp)
     moreover from ‹wellformed Γ› and ‹p' ∈ sterms Γ p1› have "sterms Γ p' = {p'}" ..
     ultimately show "p' ∈ {q. (p1 ⊕ p2) ⇝Γ* q ∧ sterms Γ q = {q}}"
       by simp
   qed
   moreover have "sterms Γ p2 ⊆ {q. (p1 ⊕ p2) ⇝Γ* q ∧ sterms Γ q = {q}}"
   proof
     fix p'
     assume "p' ∈ sterms Γ p2"
     with IH2 have "p2 ⇝Γ* p'" and "sterms Γ p' = {p'}" by auto
     moreover have "(p1 ⊕ p2) ⇝Γ p2" ..
     ultimately have "(p1 ⊕ p2) ⇝Γ* p'"
       by - (rule converse_rtranclp_into_rtranclp)
     with ‹sterms Γ p' = {p'}› show "p' ∈ {q. (p1 ⊕ p2) ⇝Γ* q ∧ sterms Γ q = {q}}"
       by simp
   qed
   ultimately show "sterms Γ (p1 ⊕ p2) ⊆ {q. (p1 ⊕ p2) ⇝Γ* q ∧ sterms Γ q = {q}}"
     using ‹wellformed Γ› by simp
  next
    fix pn
    assume IH: "sterms Γ (Γ pn) ⊆ {q. Γ pn ⇝Γ* q ∧ sterms Γ q = {q}}"
    show "sterms Γ (call(pn)) ⊆ {q. (call(pn)) ⇝Γ* q ∧ sterms Γ q = {q}}"
    proof
      fix p'
      assume "p' ∈ sterms Γ (call(pn))"
      with ‹wellformed Γ› have "p' ∈ sterms Γ (Γ pn)" by simp
      with IH have "Γ pn ⇝Γ* p'" and "sterms Γ p' = {p'}" by auto
      note this(1)
      moreover have "(call(pn)) ⇝Γ Γ pn" by simp
      ultimately have "(call(pn)) ⇝Γ* p'"
        by - (rule converse_rtranclp_into_rtranclp)
      with ‹sterms Γ p' = {p'}› show "p' ∈ {q. (call(pn)) ⇝Γ* q ∧ sterms Γ q = {q}}"
        by simp
    qed
  qed simp_all
  with ‹wellformed Γ› show "sterms Γ p ⊆ {q. p ⇝Γ* q ∧ ¬(∃q'. q ⇝Γ q')}"
    by (simp only: no_microsteps_sterms_refl)
next
  from ‹wellformed Γ› have "{q. p ⇝Γ* q ∧ sterms Γ q = {q}} ⊆ sterms Γ p"
  proof (induction)
    fix p1 p2
    assume IH1: "{q. p1 ⇝Γ* q ∧ sterms Γ q = {q}} ⊆ sterms Γ p1"
       and IH2: "{q. p2 ⇝Γ* q ∧ sterms Γ q = {q}} ⊆ sterms Γ p2"
    show "{q. (p1 ⊕ p2) ⇝Γ* q ∧ sterms Γ q = {q}} ⊆ sterms Γ (p1 ⊕ p2)"
    proof (rule, drule CollectD, erule conjE)
      fix q'
      assume "(p1 ⊕ p2) ⇝Γ* q'"
         and "sterms Γ q' = {q'}"
      with ‹wellformed Γ› have "(p1 ⊕ p2) ⇝Γ+ q'"
        by (auto dest!: rtranclpD sterms_no_choice)
      hence "p1 ⇝Γ* q' ∨ p2 ⇝Γ* q'"
        by (auto dest: tranclpD)
      thus "q' ∈ sterms Γ (p1 ⊕ p2)"
      proof
        assume "p1 ⇝Γ* q'"
        with IH1 and ‹sterms Γ q' = {q'}› have "q' ∈ sterms Γ p1" by auto
```

```
        with ‹wellformed Γ› show ?thesis by auto
      next
        assume "p2 ↝Γ* q'"
        with IH2 and ‹sterms Γ q' = {q'}› have "q' ∈ sterms Γ p2" by auto
        with ‹wellformed Γ› show ?thesis by auto
      qed
    qed
  next
    fix pn
    assume IH: "{q. Γ pn ↝Γ* q ∧ sterms Γ q = {q}} ⊆ sterms Γ (Γ pn)"
    show   "{q. (call(pn)) ↝Γ* q ∧ sterms Γ q = {q}} ⊆ sterms Γ (call(pn))"
    proof (rule, drule CollectD, erule conjE)
      fix q'
      assume "(call(pn)) ↝Γ* q'"
         and "sterms Γ q' = {q'}"
      with ‹wellformed Γ› have "(call(pn)) ↝Γ+ q'"
        by (auto dest!: rtranclpD sterms_no_call)
      moreover have "(call(pn)) ↝Γ Γ pn" ..
      ultimately have "Γ pn ↝Γ* q'"
        by (auto dest!: tranclpD)
      with ‹sterms Γ q' = {q'}› and IH have "q' ∈ sterms Γ (Γ pn)" by auto
      with ‹wellformed Γ› show "q' ∈ sterms Γ (call(pn))" by simp
    qed
  qed simp_all
  with ‹wellformed Γ› show "{q. p ↝Γ* q ∧ ¬(∃q'. q ↝Γ q')} ⊆ sterms Γ p"
  by (simp only: no_microsteps_sterms_refl)
qed
```

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 sterms, 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(s_ip, s_msg).p ▷ q)     = sterms Γ p ∪ sterms Γ q"
  | "dterms Γ ({l}broadcast(s_msg). p)            = sterms Γ p"
  | "dterms Γ ({l}groupcast(s_ips, s_msg). p)     = sterms Γ p"
  | "dterms Γ ({l}send(s_msg).p)                  = sterms Γ p"
  | "dterms Γ ({l}deliver(s_data).p)              = sterms Γ p"
  | "dterms Γ ({l}receive(u_msg).p)               = sterms Γ p"
  | "dterms Γ (call(pn))                          = dterms Γ (Γ pn)"
  by pat_completeness auto

lemma dterms_dom_basic [simp]:
  assumes "not_call p"
      and "not_choice p"
    shows "dterms_dom (Γ, p)"
  proof (rule accpI)
    fix y
    assume "dterms_rel y (Γ, p)"
    with assms show "dterms_dom y"
      by (cases p) (auto simp: dterms_rel.simps)
  qed

lemma dterms_termination:
  assumes "wellformed Γ"
    shows "dterms_dom (Γ, p)"
  proof -
```

```
    have dterms_rel': "dterms_rel = (λgq gp. (gq, gp) ∈ {((Γ, q), (Γ', p)). Γ = Γ' ∧ p ⇝Γ q})"
      by (rule ext)+ (auto simp: dterms_rel.simps elim: microstep.cases)
    from ‹wellformed(Γ)› have "∀x. x ∈ Wellfounded.acc {(q, p). p ⇝Γ q}"
      unfolding wellformed_def by (simp add: wf_iff_acc)
    hence "p ∈ Wellfounded.acc {(q, p). p ⇝Γ q}" ..
    hence "(Γ, p) ∈ Wellfounded.acc {((Γ, q), Γ', p). Γ = Γ' ∧ p ⇝Γ q}"
      by (rule acc_induct) (auto intro: accI)
    thus "dterms_dom (Γ, p)"
      unfolding dterms_rel' by (subst accp_acc_eq)
  qed

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

lemma sterms_after_dterms [simp]:
  assumes "wellformed Γ"
  shows "(⋃x∈dterms Γ p. sterms Γ x) = dterms Γ p"
  using assms by (induction p) simp_all

lemma sterms_before_dterms [simp]:
  assumes "wellformed Γ"
  shows "(⋃x∈sterms Γ p. dterms Γ x) = dterms Γ p"
  using assms by (induction p) simp_all

lemma dterms_choice_disj [simp]:
  assumes "wellformed Γ"
    shows "p ∈ dterms Γ (p1 ⊕ p2) = (p ∈ dterms Γ p1 ∨ p ∈ dterms Γ p2)"
  using assms by (simp)

lemma dterms_in_branch:
  assumes "wellformed Γ"
      and "p ∈ dterms Γ (p1 ⊕ p2)"
      and "p ∈ dterms Γ p1 ⟹ P"
      and "p ∈ dterms Γ p2 ⟹ P"
  shows "P"
  using assms by auto

lemma dterms_no_choice:
  assumes "wellformed Γ"
    shows "p1 ⊕ p2 ∉ dterms Γ p"
  using assms by induction simp_all

lemma dterms_not_choice [simp]:
  assumes "wellformed Γ"
      and "q ∈ dterms Γ p"
    shows "not_choice q"
  using assms unfolding not_choice_def
  by (auto dest: dterms_no_choice)

lemma dterms_no_call:
  assumes "wellformed Γ"
    shows "call(pn) ∉ dterms Γ p"
  using assms by induction simp_all

lemma dterms_not_call [simp]:
  assumes "wellformed Γ"
      and "q ∈ dterms Γ p"
    shows "not_call q"
  using assms unfolding not_call_def
  by (auto dest: dterms_no_call)

lemma dterms_subterms:
  assumes wf: "wellformed Γ"
      and "∃pn. p ∈ subterms (Γ pn)"
```

```
    and "q ∈ dterms Γ p"
  shows "∃pn. q ∈ subterms (Γ pn)"
using assms
proof (induct p)
    fix p1 p2
  assume IH1: "∃pn. p1 ∈ subterms (Γ pn) ⟹ q ∈ dterms Γ p1 ⟹ ∃pn. q ∈ subterms (Γ pn)"
     and IH2: "∃pn. p2 ∈ subterms (Γ pn) ⟹ q ∈ dterms Γ p2 ⟹ ∃pn. q ∈ subterms (Γ pn)"
     and *: "∃pn. p1 ⊕ p2 ∈ subterms (Γ pn)"
     and "q ∈ dterms Γ (p1 ⊕ p2)"
  from * obtain pn where "p1 ⊕ p2 ∈ subterms (Γ pn)"
    by auto
  hence "p1 ∈ subterms (Γ pn)" and "p2 ∈ subterms (Γ pn)"
    by auto
  from ‹q ∈ dterms Γ (p1 ⊕ p2)› wf have "q ∈ dterms Γ p1 ∨ q ∈ dterms Γ p2"
    by auto
  thus "∃pn. q ∈ subterms (Γ pn)"
    proof
      assume "q ∈ dterms Γ p1"
      with ‹p1 ∈ subterms (Γ pn)› show ?thesis
        by (auto intro: IH1)
    next
      assume "q ∈ dterms Γ p2"
      with ‹p2 ∈ subterms (Γ pn)› show ?thesis
        by (auto intro: IH2)
    qed
qed auto
```

Note that the converse of ⟦*wellformed ?Γ; ∃pn. ?p ∈ subterms (?Γ pn); ?q ∈ dterms ?Γ ?p*⟧ ⟹ *∃pn. ?q ∈ subterms (?Γ 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 *call(pn)* terms, since the root terms of all processes are considered, and also *p1* ⊕ *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
  cterms :: "('s, 'm, 'p, 'l) seqp_env ⇒ ('s, 'm, 'p, 'l) seqp set"
  for Γ :: "('s, 'm, 'p, 'l) seqp_env"
where
    ctermsSI[intro]: "p ∈ sterms Γ (Γ pn) ⟹ p ∈ cterms Γ"
  | ctermsDI[intro]: "⟦ pp ∈ cterms Γ; p ∈ dterms Γ pp ⟧ ⟹ p ∈ cterms Γ"

lemma cterms_not_choice [simp]:
  assumes "wellformed Γ"
      and "p ∈ cterms Γ"
    shows "not_choice p"
  using assms
  proof (cases p)
    case CHOICE from ‹p ∈ cterms Γ› show ?thesis
      using ‹wellformed Γ› by cases simp_all
  qed simp_all

lemma cterms_no_choice [simp]:
  assumes "wellformed Γ"
    shows "p1 ⊕ p2 ∉ cterms Γ"
  using assms by (auto dest: cterms_not_choice)

lemma cterms_not_call [simp]:
  assumes "wellformed Γ"
```

```
    and "p ∈ cterms Γ"
  shows "not_call p"
  using assms
  proof (cases p)
    case CALL from ‹p ∈ cterms Γ› show ?thesis
      using ‹wellformed Γ› by cases simp_all
  qed simp_all

lemma cterms_no_call [simp]:
  assumes "wellformed Γ"
    shows "call(pn) ∉ cterms Γ"
  using assms by (auto dest: cterms_not_call)

lemma sterms_cterms [elim]:
  assumes "p ∈ cterms Γ"
      and "q ∈ sterms Γ p"
      and "wellformed Γ"
    shows "q ∈ cterms Γ"
  using assms by - (cases p, auto)

lemma dterms_cterms [elim]:
  assumes "p ∈ cterms Γ"
      and "q ∈ dterms Γ p"
      and "wellformed Γ"
    shows "q ∈ cterms Γ"
  using assms by (cases p) auto

lemma derivs_in_cterms [simp]:
  "⋀l f p. {l}⟨f⟩ p ∈ cterms Γ                              ⟹ sterms Γ p ⊆ cterms Γ"
  "⋀l f p. {l}⟦f⟧ p ∈ cterms Γ                              ⟹ sterms Γ p ⊆ cterms Γ"
  "⋀l fip fmsg q p. {l}unicast(fip, fmsg). p ▷ q ∈ cterms Γ
                              ⟹ sterms Γ p ⊆ cterms Γ ∧ sterms Γ q ⊆ cterms Γ"
  "⋀l fmsg p.      {l}broadcast(fmsg).p ∈ cterms Γ        ⟹ sterms Γ p ⊆ cterms Γ"
  "⋀l fips fmsg p. {l}groupcast(fips, fmsg).p ∈ cterms Γ ⟹ sterms Γ p ⊆ cterms Γ"
  "⋀l fmsg p.      {l}send(fmsg).p ∈ cterms Γ             ⟹ sterms Γ p ⊆ cterms Γ"
  "⋀l fdata p.     {l}deliver(fdata).p ∈ cterms Γ         ⟹ sterms Γ p ⊆ cterms Γ"
  "⋀l fmsg p.      {l}receive(fmsg).p ∈ cterms Γ          ⟹ sterms Γ p ⊆ cterms Γ"
  by (auto simp: dterms.psimps)
```

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 ⇒ ('s, 'm, 'p , 'l) seqp set"
where
    "ctermsl ({l}⟨g⟩ p)                        = insert ({l}⟨g⟩ p)  (ctermsl p)"
  | "ctermsl ({l}⟦u⟧ p)                        = insert ({l}⟦u⟧ p)  (ctermsl p)"
  | "ctermsl ({l}unicast(s_ip, s_msg). p ▷ q) = insert ({l}unicast(s_ip, s_msg). p ▷ q)
                                                              (ctermsl p ∪ 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 (p1 ⊕ p2)                     = ctermsl p1 ∪ ctermsl p2"
  | "ctermsl (call(pn))                    = {call(pn)}"
  by pat_completeness auto
  termination by (relation "measure(size)") (auto dest: stermsl_nobigger)

lemmas ctermsl_induct =
```

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

lemma ctermsl_refl [intro]: "not_choice p ⟹ p ∈ ctermsl p"
  by (cases p) auto

lemma ctermsl_subterms:
  "ctermsl p = {q. q ∈ subterms p ∧ not_choice q }" (is "?lhs = ?rhs")
  proof
    show "?lhs ⊆ ?rhs" by (induct p, auto) next
    show "?rhs ⊆ ?lhs" by (induct p, auto)
  qed

lemma ctermsl_trans [elim]:
  assumes "q ∈ ctermsl p"
      and "r ∈ ctermsl q"
    shows "r ∈ ctermsl p"
  using assms
  proof (induction p rule: ctermsl_induct)
    case (CHOICE p1 p2)
      have "(q ∈ ctermsl p1) ∨ (q ∈ ctermsl p2)"
        using CHOICE.prems(1) by simp
      hence "r ∈ ctermsl p1 ∨ r ∈ ctermsl p2"
      proof (rule disj_forward)
        assume "q ∈ ctermsl p1"
        thus "r ∈ ctermsl p1" using ‹r ∈ ctermsl q› by (rule CHOICE.IH)
      next
        assume "q ∈ ctermsl p2"
        thus "r ∈ ctermsl p2" using ‹r ∈ ctermsl q› by (rule CHOICE.IH)
      qed
      thus "r ∈ ctermsl (p1 ⊕ p2)" by simp
    qed auto

lemma ctermsl_ex_trans [elim]:
  assumes "∃q ∈ ctermsl p. r ∈ ctermsl q"
    shows "r ∈ ctermsl p"
  using assms by auto

lemma call_ctermsl_empty [elim]:
  "⟦ p ∈ ctermsl p'; not_call p ⟧ ⟹ not_call p'"
  unfolding not_call_def by (cases p) auto

lemma stermsl_ctermsl_choice1 [simp]:
  assumes "q ∈ stermsl p1"
    shows "q ∈ ctermsl (p1 ⊕ p2)"
  using assms by (induction p1) auto

lemma stermsl_ctermsl_choice2 [simp]:
  assumes "q ∈ stermsl p2"
    shows "q ∈ ctermsl (p1 ⊕ p2)"
  using assms by (induction p2) auto

lemma stermsl_ctermsl [elim]:
  assumes "q ∈ stermsl p"
    shows "q ∈ ctermsl p"
  using assms
  proof (cases p)
    case (CHOICE p1 p2)
    hence "q ∈ stermsl (p1 ⊕ p2)" using assms by simp
    hence "q ∈ stermsl p1 ∨ q ∈ stermsl p2" by simp
    hence "q ∈ ctermsl (p1 ⊕ p2)" by (rule) (simp_all del: ctermsl.simps)
    thus  "q ∈ ctermsl p" using CHOICE by simp
  qed simp_all
```

```
lemma stermsl_after_ctermsl [simp]:
  "(⋃x∈ctermsl p. stermsl x) = ctermsl p"
  by (induct p) auto

lemma stermsl_before_ctermsl [simp]:
  "(⋃x∈stermsl p. ctermsl x) = ctermsl p"
  by (induct p) simp_all

lemma ctermsl_no_choice: "p1 ⊕ p2 ∉ ctermsl p"
  by (induct p) simp_all

lemma ctermsl_ex_stermsl: "q ∈ ctermsl p ⟹ ∃ps∈stermsl p. q ∈ ctermsl ps"
  by (induct p) auto

lemma dterms_ctermsl [intro]:
  assumes "q ∈ dterms Γ p"
      and "wellformed Γ"
    shows "q ∈ ctermsl p ∨ (∃pn. q ∈ ctermsl (Γ pn))"
  using assms(1-2)
  proof (induction p rule: dterms_pinduct [OF ‹wellformed Γ›])
    fix Γ l fg p
    assume "q ∈ dterms Γ ({l}⟨fg⟩ p)"
       and "wellformed Γ"
    hence "q ∈ sterms Γ p" by simp
    hence "q ∈ stermsl p ∨ (∃pn. q ∈ stermsl (Γ pn))"
      using ‹wellformed Γ› by (rule sterms_stermsl)
    thus "q ∈ ctermsl ({l}⟨fg⟩ p) ∨ (∃pn. q ∈ ctermsl (Γ pn))"
    proof
      assume "q ∈ stermsl p"
      hence "q ∈ ctermsl p" by (rule stermsl_ctermsl)
      hence "q ∈ ctermsl ({l}⟨fg⟩ p)" by simp
      thus ?thesis ..
    next
      assume "∃pn. q ∈ stermsl (Γ pn)"
      then obtain pn where "q ∈ stermsl (Γ pn)" by auto
      hence "q ∈ ctermsl (Γ pn)" by (rule stermsl_ctermsl)
      hence "∃pn. q ∈ ctermsl (Γ pn)" ..
      thus ?thesis ..
    qed
  next
    fix Γ p1 p2
    assume "q ∈ dterms Γ (p1 ⊕ p2)"
       and IH1: "⟦ q ∈ dterms Γ p1; wellformed Γ ⟧ ⟹ q ∈ ctermsl p1 ∨ (∃pn. q ∈ ctermsl (Γ pn))"
       and IH2: "⟦ q ∈ dterms Γ p2; wellformed Γ ⟧ ⟹ q ∈ ctermsl p2 ∨ (∃pn. q ∈ ctermsl (Γ pn))"
       and "wellformed Γ"
    thus "q ∈ ctermsl (p1 ⊕ p2) ∨ (∃pn. q ∈ ctermsl (Γ pn))"
      by auto
  next
    fix Γ pn
    assume "q ∈ dterms Γ (call(pn))"
       and "wellformed Γ"
       and "⟦ q ∈ dterms Γ (Γ pn); wellformed Γ ⟧ ⟹ q ∈ ctermsl (Γ pn) ∨ (∃pn. q ∈ ctermsl (Γ pn))"
    thus "q ∈ ctermsl (call(pn)) ∨ (∃pn. q ∈ ctermsl (Γ pn))"
      by auto
  qed (simp_all, (metis sterms_stermsl stermsl_ctermsl)+)

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"
  using assms by (induct p rule: ctermsl.induct) auto
```

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 ({l}⟨fg⟩ p)                          = stermsl p"
  | "dtermsl ({l}⟦fa⟧ p)                          = stermsl p"
  | "dtermsl (p1 ⊕ p2)                            = dtermsl p1 ∪ dtermsl p2"
  | "dtermsl ({l}unicast(fip, fmsg).p ▷ q)  = stermsl p ∪ stermsl q"
  | "dtermsl ({l}broadcast(fmsg). p)          = stermsl p"
  | "dtermsl ({l}groupcast(fips, fmsg). p)   = stermsl p"
  | "dtermsl ({l}send(fmsg).p)                = stermsl p"
  | "dtermsl ({l}deliver(fdata).p)            = stermsl p"
  | "dtermsl ({l}receive(fmsg).p)             = stermsl p"
  | "dtermsl (call(pn))                        = {}"
  by pat_completeness auto
  termination by (relation "measure(size)") (auto dest: stermsl_nobigger)

lemma stermsl_after_dtermsl [simp]:
  shows "(⋃x∈dtermsl p. stermsl x) = dtermsl p"
  by (induct p) simp_all

lemma stermsl_before_dtermsl [simp]:
  "(⋃x∈stermsl p. dtermsl x) = dtermsl p"
  by (induct p) simp_all

lemma dtermsl_no_choice [simp]: "p1 ⊕ p2 ∉ dtermsl p"
  by (induct p) simp_all

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

lemma dtermsl_in_branch [elim]:
  "⟦p ∈ dtermsl (p1 ⊕ p2); p ∈ dtermsl p1 ⟹ P; p ∈ dtermsl p2 ⟹ P⟧ ⟹ P"
  by auto

lemma ctermsl_dtermsl [elim]:
  assumes "q ∈ dtermsl p"
    shows "q ∈ ctermsl p"
  using assms by (induct p) (simp_all, (metis stermsl_ctermsl)+)

lemma dtermsl_dterms [elim]:
  assumes "q ∈ dtermsl p"
      and "not_call q"
      and "wellformed Γ"
    shows "q ∈ dterms Γ p"
  using assms
  using assms by (induct p) (simp_all, (metis stermsl_sterms)+)

lemma ctermsl_stermsl_or_dtermsl:
  assumes "q ∈ ctermsl p"
    shows "q ∈ stermsl p ∨ (∃p'∈dtermsl p. q ∈ ctermsl p')"
  using assms by (induct p) (auto dest: ctermsl_ex_stermsl)

lemma dtermsl_add_stermsl_beforeD:
  assumes "q ∈ dtermsl p"
    shows "∃ps∈stermsl p. q ∈ dtermsl ps"
  proof -
    from assms have "q ∈ (⋃x∈stermsl p. dtermsl x)" by auto
    thus ?thesis
      by (rule UN_E) auto
  qed
```

```
lemma call_dtermsl_empty [elim]:
 "q ∈ dtermsl p ⟹ not_call p"
  by (cases p) simp_all
```

7.9 More properties of control terms

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

```
theorem cterms_def':
  assumes wfg: "wellformed Γ"
    shows "cterms Γ = { p |p pn. p ∈ ctermsl (Γ pn) ∧ not_call p }"
          (is "_ = ?ctermsl_set")
  proof (rule iffI [THEN set_eqI])
    fix p
    assume "p ∈ cterms Γ"
    thus "p ∈ ?ctermsl_set"
    proof (induction p)
      fix p pn
      assume "p ∈ sterms Γ (Γ pn)"
      then obtain pn' where "p ∈ stermsl (Γ pn')" using wfg
        by (blast dest: sterms_stermsl_heads)
      hence "p ∈ ctermsl (Γ pn')" ..
      moreover from ‹p ∈ sterms Γ (Γ pn)› wfg have "not_call p" by simp
      ultimately show "p ∈ ?ctermsl_set" by auto
    next
      fix pp p
      assume "pp ∈ cterms Γ"
         and IH: "pp ∈ ?ctermsl_set"
         and *: "p ∈ dterms Γ pp"
      from * have "p ∈ ctermsl pp ∨ (∃pn. p ∈ ctermsl (Γ pn))"
        using wfg by (rule dterms_ctermsl)
      hence "∃pn. p ∈ ctermsl (Γ pn)"
        proof
          assume "p ∈ ctermsl pp"
          from ‹pp ∈ cterms Γ› and IH obtain pn' where "pp ∈ ctermsl (Γ pn')"
            by auto
          with ‹p ∈ ctermsl pp› have "p ∈ ctermsl (Γ pn')" by auto
          thus "∃pn. p ∈ ctermsl (Γ pn)" ..
        qed -
      moreover from ‹p ∈ dterms Γ pp› wfg have "not_call p" by simp
      ultimately show "p ∈ ?ctermsl_set" by auto
    qed
  next
    fix p
    assume "p ∈ ?ctermsl_set"
    then obtain pn where *: "p ∈ ctermsl (Γ pn)" and "not_call p" by auto
    from * have "p ∈ stermsl (Γ pn) ∨ (∃p'∈dtermsl (Γ pn). p ∈ ctermsl p')"
      by (rule ctermsl_stermsl_or_dtermsl)
    thus "p ∈ cterms Γ"
    proof
      assume "p ∈ stermsl (Γ pn)"
      hence "p ∈ sterms Γ (Γ pn)" using ‹not_call p› wfg ..
      thus "p ∈ cterms Γ" ..
    next
      assume "∃p'∈dtermsl (Γ pn). p ∈ ctermsl p'"
      then obtain p' where p'1: "p' ∈ dtermsl (Γ pn)"
                       and p'2: "p ∈ ctermsl p'" ..
      from p'2 and ‹not_call p› have "not_call p'" ..
      from p'1 obtain ps where ps1: "ps ∈ stermsl (Γ pn)"
                           and ps2: "p' ∈ dtermsl ps"
        by (blast dest: dtermsl_add_stermsl_beforeD)
```

```
      from ps2 have "not_call ps" ..
      with ps1 have "ps ∈ cterms Γ" using wfg by auto
      with ‹p' ∈ dtermsl ps› and ‹not_call p'› have "p' ∈ cterms Γ" using wfg by auto
      hence "sterms Γ p' ⊆ cterms Γ" using wfg by auto
      with ‹p ∈ ctermsl p'› ‹not_call p› show "p ∈ cterms Γ" using wfg ..
    qed
  qed

lemma ctermsE [elim]:
  assumes "wellformed Γ"
      and "p ∈ cterms Γ"
  obtains pn where "p ∈ ctermsl (Γ pn)"
               and "not_call p"
  using assms(2) unfolding cterms_def' [OF assms(1)] by auto

corollary cterms_subterms:
  assumes "wellformed Γ"
    shows "cterms Γ = {p|p pn. p∈subterms (Γ pn) ∧ not_call p ∧ not_choice p}"
  by (subst cterms_def' [OF assms(1)], subst ctermsl_subterms) auto

lemma subterms_in_cterms [elim]:
  assumes "wellformed Γ"
      and "p∈subterms (Γ pn)"
      and "not_call p"
      and "not_choice p"
    shows "p ∈ cterms Γ"
  using assms unfolding cterms_subterms [OF ‹wellformed Γ›] by auto

lemma subterms_stermsl_ctermsl:
  assumes "q ∈ subterms p"
      and "r ∈ stermsl q"
    shows "r ∈ ctermsl p"
  using assms
  proof (induct p)
    fix p1 p2
    assume IH1: "q ∈ subterms p1 ⟹ r ∈ stermsl q ⟹ r ∈ ctermsl p1"
       and IH2: "q ∈ subterms p2 ⟹ r ∈ stermsl q ⟹ r ∈ ctermsl p2"
       and *: "q ∈ subterms (p1 ⊕ p2)"
       and "r ∈ stermsl q"
    from * have "q ∈ {p1 ⊕ p2} ∪ subterms p1 ∪ subterms p2" by simp
    thus "r ∈ ctermsl (p1 ⊕ p2)"
    proof (elim UnE)
      assume "q ∈ {p1 ⊕ p2}" with ‹r ∈ stermsl q› show ?thesis
      by simp (metis stermsl_ctermsl)
    next
      assume "q ∈ subterms p1" hence "r ∈ ctermsl p1" using ‹r ∈ stermsl q› by (rule IH1)
      thus ?thesis by simp
    next
      assume "q ∈ subterms p2" hence "r ∈ ctermsl p2" using ‹r ∈ stermsl q› by (rule IH2)
      thus ?thesis by simp
    qed
  qed auto

lemma subterms_sterms_cterms:
  assumes wf: "wellformed Γ"
      and "p ∈ subterms (Γ pn)"
    shows "sterms Γ p ⊆ cterms Γ"
  using assms(2)
  proof (induct p)
    fix p
    assume "call(p) ∈ subterms (Γ pn)"
    from wf have "sterms Γ (call(p)) = sterms Γ (Γ p)" by simp
      thus "sterms Γ (call(p)) ⊆ cterms Γ" by auto
  next
```

```
    fix p1 p2
    assume IH1: "p1 ∈ subterms (Γ pn) ⟹ sterms Γ p1 ⊆ cterms Γ"
       and IH2: "p2 ∈ subterms (Γ pn) ⟹ sterms Γ p2 ⊆ cterms Γ"
       and *: "p1 ⊕ p2 ∈ subterms (Γ pn)"
    from * have "p1 ∈ subterms (Γ pn)" by auto
    hence "sterms Γ p1 ⊆ cterms Γ" by (rule IH1)
    moreover from * have "p2 ∈ subterms (Γ pn)" by auto
      hence "sterms Γ p2 ⊆ cterms Γ" by (rule IH2)
    ultimately show "sterms Γ (p1 ⊕ p2 ) ⊆ cterms Γ" using wf by simp
  qed (auto elim!: subterms_in_cterms [OF ‹wellformed Γ›])

lemma subterms_sterms_in_cterms:
  assumes "wellformed Γ"
      and "p ∈ subterms (Γ pn)"
      and "q ∈ sterms Γ p"
    shows "q ∈ cterms Γ"
  using assms
  by (auto dest!: subterms_sterms_cterms [OF ‹wellformed Γ›])

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)"
  by pat_completeness auto

lemma labels_dom_basic [simp]:
  assumes "not_call p"
      and "not_choice p"
  shows "labels_dom (Γ, p)"
  proof (rule accpI)
    fix y
    assume "labels_rel y (Γ, p)"
    with assms show "labels_dom y"
      by (cases p) (auto simp: labels_rel.simps)
  qed

lemma labels_termination:
    fixes Γ p
  assumes "wellformed(Γ)"
    shows "labels_dom (Γ, p)"
  proof -
    have labels_rel': "labels_rel = (λgq gp. (gq, gp) ∈ {((Γ, q), (Γ', p)). Γ = Γ' ∧ p ↝Γ q})"
```

```
      by (rule ext)+ (auto simp: labels_rel.simps intro: microstep.intros elim: microstep.cases)
    from ‹wellformed(Γ)› have "∀x. x ∈ Wellfounded.acc {(q, p). p ↝Γ q}"
      unfolding wellformed_def by (simp add: wf_iff_acc)
    hence "p ∈ Wellfounded.acc {(q, p). p ↝Γ q}" ..
    hence "(Γ, p) ∈ Wellfounded.acc {((Γ, q), Γ', p). Γ = Γ' ∧ p ↝Γ q}"
      by (rule acc_induct) (auto intro: accI)
    thus "labels_dom (Γ, p)"
      unfolding labels_rel' by (subst accp_acc_eq)
  qed

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 Γ p
  assumes "wellformed Γ"
    shows "labels Γ p ≠ {}"
   by (induct p rule: labels_pinduct [OF ‹wellformed Γ›]) simp_all

lemma has_label [dest]:
    fixes Γ p
  assumes "wellformed Γ"
    shows "∃l. l ∈ labels Γ p"
  using labels_not_empty [OF assms] by auto

lemma singleton_labels [simp]:
  "⋀Γ l l' f p.          l ∈ labels Γ ({l'}⟨f⟩ p)                        = (l = l')"
  "⋀Γ l l' f p.          l ∈ labels Γ ({l'}⟦f⟧ p)                       = (l = l')"
  "⋀Γ l l' fip fmsg p q. l ∈ labels Γ ({l'}unicast(fip, fmsg).p ▷ q)  = (l = l')"
  "⋀Γ l l' fmsg p.       l ∈ labels Γ ({l'}broadcast(fmsg). p)         = (l = l')"
  "⋀Γ l l' fips fmsg p.  l ∈ labels Γ ({l'}groupcast(fips, fmsg). p)   = (l = l')"
  "⋀Γ l l' fmsg p.       l ∈ labels Γ ({l'}send(fmsg).p)               = (l = l')"
  "⋀Γ l l' fdata p.      l ∈ labels Γ ({l'}deliver(fdata).p)           = (l = l')"
  "⋀Γ l l' fmsg p.       l ∈ labels Γ ({l'}receive(fmsg).p)            = (l = l')"
  by auto

lemma in_labels_singletons [dest!]:
  "⋀Γ l l' f p.          l ∈ labels Γ ({l'}⟨f⟩ p)                        ⟹ l = l'"
  "⋀Γ l l' f p.          l ∈ labels Γ ({l'}⟦f⟧ p)                       ⟹ l = l'"
  "⋀Γ l l' fip fmsg p q. l ∈ labels Γ ({l'}unicast(fip, fmsg).p ▷ q)  ⟹ l = l'"
  "⋀Γ l l' fmsg p.       l ∈ labels Γ ({l'}broadcast(fmsg). p)         ⟹ l = l'"
  "⋀Γ l l' fips fmsg p.  l ∈ labels Γ ({l'}groupcast(fips, fmsg). p)   ⟹ l = l'"
  "⋀Γ l l' fmsg p.       l ∈ labels Γ ({l'}send(fmsg).p)               ⟹ l = l'"
  "⋀Γ l l' fdata p.      l ∈ labels Γ ({l'}deliver(fdata).p)           ⟹ l = l'"
  "⋀Γ l l' fmsg p.       l ∈ labels Γ ({l'}receive(fmsg).p)            ⟹ l = l'"
  by auto

definition
  simple_labels :: "('s, 'm, 'p, 'l) seqp_env ⇒ bool"
where
  "simple_labels Γ ≡ ∀pn. ∀p∈subterms (Γ pn). (∃!l. labels Γ p = {l})"

lemma simple_labelsI [intro]:
  assumes "⋀pn p. p∈subterms (Γ pn) ⟹ ∃!l. labels Γ p = {l}"
  shows "simple_labels Γ"
  using assms unfolding simple_labels_def by auto
```

The *simple_labels* Γ property is necessary to transfer results shown over the *cterms* of a process specification Γ to the reachable actions of that process.
Consider the process *{l₁}send(m1)* .
p1
⊕

{l2}send(m2) .

p2. The iteration over *cterms* Γ will cover the two transitions *(l1, send m1, p1)* and *(l2, send m2, p2)*, but reachability requires the four transitions *(l1, send m1, p1)*, *(l1, send m2, p2)*, *(l2, send m1, p1)*, and *(l2, 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, *{l1}⟦e⟧*

p

⊕

call(pn). Here l_1 must equal the label of Γ *pn*, 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*

⊕

p2 operator.

```
lemma simple_labels_in_sterms:
    fixes Γ l p
  assumes "simple_labels Γ"
      and "wellformed Γ"
      and "∃pn. p∈subterms (Γ pn)"
      and "l∈labels Γ p"
    shows "∀p'∈sterms Γ p. l∈labels Γ p'"
  using assms
  proof (induct p rule: labels_pinduct [OF ‹wellformed Γ›])
    fix Γ p1 p2
    assume sl: "simple_labels Γ"
       and wf: "wellformed Γ"
       and IH1: "⟦ simple_labels Γ; wellformed Γ;
                  ∃pn. p1 ∈ subterms (Γ pn); l ∈ labels Γ p1 ⟧
                 ⟹ ∀p'∈sterms Γ p1. l ∈ labels Γ p'"
       and IH2: "⟦ simple_labels Γ; wellformed Γ;
                  ∃pn. p2 ∈ subterms (Γ pn); l ∈ labels Γ p2 ⟧
                 ⟹ ∀p'∈sterms Γ p2. l ∈ labels Γ p'"
       and ein: "∃pn. p1 ⊕ p2 ∈ subterms (Γ pn)"
       and l12: "l ∈ labels Γ (p1 ⊕ p2)"
    from sl ein l12 have "labels Γ (p1 ⊕ p2) = {l}"
      unfolding simple_labels_def by (metis empty_iff insert_iff)
    with wf have "labels Γ p1 ∪ labels Γ p2 = {l}" by simp
    moreover have "labels Γ p1 ≠ {}" and "labels Γ p2 ≠ {}"
      using wf by (metis labels_not_empty)+
    ultimately have "l ∈ labels Γ p1" and "l ∈ labels Γ p2"
      by (metis Un_iff empty_iff insert_iff set_eqI)+
    moreover from ein have "∃pn. p1 ∈ subterms (Γ pn)"
                   and "∃pn. p2 ∈ subterms (Γ pn)"
       by auto
    ultimately show "∀p'∈sterms Γ (p1 ⊕ p2). l∈labels Γ p'"
      using wf IH1 [OF sl wf] IH2 [OF sl wf] by auto
  qed auto

lemma labels_in_sterms:
    fixes Γ l p
  assumes "wellformed Γ"
      and "l∈labels Γ p"
    shows "∃p'∈sterms Γ p. l∈labels Γ p'"
  using assms
  by (induct p rule: labels_pinduct [OF ‹wellformed Γ›]) (auto intro: Un_iff)

lemma labels_sterms_labels:
    fixes Γ p p' l
```

```
  assumes "wellformed Γ"
      and "p' ∈ sterms Γ p"
      and "l ∈ labels Γ p'"
    shows "l ∈ labels Γ p"
  using assms
  by (induct p rule: labels_pinduct [OF ‹wellformed Γ›]) auto

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 ..
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 aodv_proc_cases [ctermsl_cases]
3. add invariant intros declare seq_invariant_ctermsI [OF aodv_wf aodv_control_within aodv_simple_labels, cterms_intros] seq_step_invariant_ctermsI [OF aodv_wf aodv_control_within aodv_simple_labels, cterms_intros]

```
lemma has_ctermsl: "p ∈ ctermsl Γ ⟹ p ∈ ctermsl Γ" .

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 ‹
fun simp_only thms ctxt =
  asm_full_simp_tac
     (ctxt |> Raw_Simplifier.clear_simpset |> fold Simplifier.add_simp thms)

(* shallow_simp is useful for mopping up assumptions before really trying to simplify.
   Perhaps surprisingly, this saves minutes in some of the proofs that use a lot of
   invariants of the form (l = P-:n --> P). *)
fun shallow_simp ctxt =
  let val ctxt' = Config.put simp_depth_limit 2 ctxt in
    TRY o safe_asm_full_simp_tac ctxt'
  end

fun create_vcs ctxt i =
  let val main_simp_thms = rev (Named_Theorems.get ctxt @{named_theorems cterms_env})
      val ctermsl_cases = rev (Named_Theorems.get ctxt @{named_theorems ctermsl_cases})
  in
    dresolve_tac ctxt @{thms has_ctermsl} i
    THEN_ELSE (dmatch_tac ctxt ctermsl_cases i
               THEN
               TRY (REPEAT_ALL_NEW (ematch_tac ctxt [@{thm disjE}]) i)
               THEN
               PARALLEL_ALLGOALS
                 (fn i => simp_only main_simp_thms ctxt i
                  THEN TRY (REPEAT_ALL_NEW (ematch_tac ctxt [@{thm disjE}]) i)), all_tac)
  end

fun try_invs ctxt =
```

```
  let val inv_thms = rev (Named_Theorems.get ctxt @{named_theorems cterms_invs})
      fun fapp thm =
        TRY o (EVERY' (forward_tac ctxt [thm] :: replicate (Thm.nprems_of thm - 1) (assume_tac ctxt)))
  in
    EVERY' (map fapp inv_thms)
  end

fun try_final ctxt =
  let val final_thms = rev (Named_Theorems.get ctxt @{named_theorems cterms_final})
      fun eapp thm = EVERY' (eresolve_tac ctxt [thm] :: replicate (Thm.nprems_of thm - 1) (assume_tac ctxt))
  in
    TRY o (FIRST' (map eapp final_thms))
  end

fun each ctxt =
  (EVERY' ((ematch_tac ctxt (rev (Named_Theorems.get ctxt @{named_theorems cterms_elimders})) ::
    replicate 2 (assume_tac ctxt)))
   THEN' simp_only @{thms labels_psimps} ctxt
   THEN' (ematch_tac ctxt (rev (Named_Theorems.get ctxt @{named_theorems cterms_seqte}))
     THEN_ALL_NEW
       (fn j => simp_only [@{thm mem_Collect_eq}] ctxt j
                 THEN REPEAT (eresolve_tac ctxt @{thms exE} j)
                 THEN REPEAT (eresolve_tac ctxt @{thms conjE} j))))
  ORELSE' (SOLVED' (clarsimp_tac ctxt))

fun simp_all ctxt =
  let val ctxt' =
        ctxt |> fold Splitter.add_split [@{thm if_split_asm}]
  in
    PARALLEL_ALLGOALS (shallow_simp ctxt)
    THEN
    TRY (CHANGED_PROP (PARALLEL_ALLGOALS (asm_full_simp_tac ctxt' THEN' try_final ctxt)))
  end

fun intro_and_invs ctxt i =
  let val cterms_intros = rev (Named_Theorems.get ctxt @{named_theorems cterms_intros}) in
    match_tac ctxt cterms_intros i
    THEN PARALLEL_ALLGOALS (try_invs ctxt)
  end

fun process_vcs ctxt _ =
  ALLGOALS (create_vcs ctxt ORELSE' (SOLVED' (clarsimp_tac ctxt)))
  THEN PARALLEL_ALLGOALS (TRY o each ctxt)
›
```

method_setup *inv_cterms* = ‹

```
  let
    val intro_onlyN = "intro_only"
    val vcs_onlyN = "vcs_only"
    val invN = "inv"
    val solveN = "solve"

    val inv_cterms_options =
      (Args.parens (Args.$$$ intro_onlyN) >>  K intro_and_invs ||
       Args.parens (Args.$$$ vcs_onlyN) >>  K (fn ctxt => intro_and_invs ctxt
                                                     THEN' process_vcs ctxt) ||
       Scan.succeed (fn ctxt => intro_and_invs ctxt
                             THEN' process_vcs ctxt
                             THEN' K (simp_all ctxt)))
  in
    (Scan.lift inv_cterms_options --| Method.sections
      ((Args.$$$ invN -- Args.add -- Args.colon >>
        K (Method.modifier (Named_Theorems.add @{named_theorems cterms_invs}) here))
       :: (Args.$$$ solveN -- Args.colon >>
```

```
      K (Method.modifier (Named_Theorems.add @{named_theorems cterms_final}) here))
     :: Simplifier.simp_modifiers)
    >> (fn tac => SIMPLE_METHOD' o tac))
  end
› "solve invariants by considering all (interesting) control terms"

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"
  using assms by auto

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"
  using assms by auto

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"
  using assms by simp (erule seqpTEs, auto)

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"
  using assms by auto

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"
  using assms by auto

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"
  using assms by auto

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"
  using assms by auto

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"
  using assms by auto

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'"
  using assms proof (induction p arbitrary: s a s')
    fix i R σ s a s'
    assume "(s, a, s') ∈ trans (pnet np ⟨i; R⟩)"
    hence "(s, a, s') ∈ node_sos (trans (np i))" ..
    thus "net_ips s = net_ips s'"
      by (rule node_sos.cases) simp_all
  next
    fix p1 p2 s a s'
    assume "⋀s a s'. (s, a, s') ∈ trans (pnet np p1) ⟹ net_ips s = net_ips s'"
      and "⋀s a s'. (s, a, s') ∈ trans (pnet np p2) ⟹ net_ips s = net_ips s'"
      and "(s, a, s') ∈ trans (pnet np (p1 ∥ p2))"
    thus "net_ips s = net_ips s'"
      by simp (erule pnet_sos.cases, simp_all)
  qed

lemma pnet_net_ips_net_tree_ips [elim]:
```

```
  assumes "s ∈ reachable (pnet np p) I"
    shows "net_ips s = net_tree_ips p"
  using assms proof induction
    fix s
    assume "s ∈ init (pnet np p)"
    thus "net_ips s = net_tree_ips p"
    proof (induction p arbitrary: s)
      fix i R s
      assume "s ∈ init (pnet np ⟨i; R⟩)"
      then obtain ns where "s = NodeS i ns R" ..
      thus "net_ips s = net_tree_ips ⟨i; R⟩"
        by simp
    next
      fix p1 p2 s
      assume IH1: "⋀s. s ∈ init (pnet np p1) ⟹ net_ips s = net_tree_ips p1"
         and IH2: "⋀s. s ∈ init (pnet np p2) ⟹ net_ips s = net_tree_ips p2"
         and "s ∈ init (pnet np (p1 ∥ p2))"
      from this(3) obtain s1 s2 where "s1 ∈ init (pnet np p1)"
                              and "s2 ∈ init (pnet np p2)"
                              and "s = SubnetS s1 s2" by auto
      from this(1-2) have "net_ips s1 = net_tree_ips p1"
                   and "net_ips s2 = net_tree_ips p2"
        using IH1 IH2 by auto
      with ‹s = SubnetS s1 s2› show "net_ips s = net_tree_ips (p1 ∥ p2)" by auto
    qed
  next
    fix s a s'
    assume "(s, a, s') ∈ trans (pnet np p)"
       and "net_ips s = net_tree_ips p"
    from this(1) have "net_ips s = net_ips s'"
      by (rule pnet_maintains_dom)
    with ‹net_ips s = net_tree_ips p› show "net_ips s' = net_tree_ips p"
      by simp
  qed

lemma pnet_init_net_ips_net_tree_ips:
  assumes "s ∈ init (pnet np p)"
    shows "net_ips s = net_tree_ips p"
  using assms(1) by (rule reachable_init [THEN pnet_net_ips_net_tree_ips])

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"
  using assms by (clarsimp dest!: pnet_init_net_ips_net_tree_ips)

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"
  using assms by (clarsimp dest!: pnet_init_net_ips_net_tree_ips)

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
    assume "i ∈ net_ips s"
    with assms(1) have "i ∈ net_tree_ips p" ..
    with assms(2) show False ..
  qed

lemma net_node_reachable_is_node:
  assumes "st ∈ reachable (pnet np ⟨ii; R_i⟩) I"
```

```
    shows "∃ns R. st = NodeS ii ns R"
  using assms proof induct
    fix s
    assume "s ∈ init (pnet np ⟨ii; Rᵢ⟩)"
    thus "∃ns R. s = NodeS ii ns R"
      by (rule pnet_node_init') simp
  next
    fix s a s'
    assume "s ∈ reachable (pnet np ⟨ii; Rᵢ⟩) I"
       and "∃ns R. s = NodeS ii ns R"
       and "(s, a, s') ∈ trans (pnet np ⟨ii; Rᵢ⟩)"
       and "I a"
    thus "∃ns R. s' = NodeS ii ns R"
      by (auto simp add: trans_node_comp dest!: node_sos_dest)
  qed

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'"
  using assms by cases simp_all

lemma net_par_reachable_is_subnet:
  assumes "st ∈ reachable (pnet np (p1 ∥ p2)) I"
    shows "∃s t. st = SubnetS s t"
  using assms by induct (auto dest!: partial_net_preserves_subnets)

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"
  using assms(1) proof (induction "SubnetS s t" arbitrary: s t)
    fix s t
    assume "SubnetS s t ∈ init (pnet np (p1 ∥ p2))"
    with init show "P s t" .
  next
    fix st a s' t'
    assume "st ∈ reachable (pnet np (p1 ∥ p2)) I"
       and tr: "(st, a, SubnetS s' t') ∈ trans (pnet np (p1 ∥ p2))"
       and "I a"
       and IH: "⋀s t. st = SubnetS s t ⟹ P s t"
    from this(1) obtain s t where "st = SubnetS s t"
                              and str: "SubnetS s t ∈ reachable (pnet np (p1 ∥ p2)) I"
      by (metis net_par_reachable_is_subnet)
    note this(2)
    moreover from IH and ‹st = SubnetS s t› have "P s t" .
    moreover from ‹st = SubnetS s t› and tr
      have "(SubnetS s t, a, SubnetS s' t') ∈ trans (pnet np (p1 ∥ p2))" by simp
    ultimately show "P s' t'"
      using ‹I a› by (rule step)
  qed

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 -
    from assms have "s1 ∈ reachable (pnet np p1) TT
                     ∧ s2 ∈ reachable (pnet np p2) TT"
    proof (induction rule: reachable_par_subnet_induct)
      fix s1 s2
```

```
      assume "SubnetS s1 s2 ∈ init (pnet np (p1 ∥ p2))"
      thus "s1 ∈ reachable (pnet np p1) TT
          ∧ s2 ∈ reachable (pnet np p2) TT"
        by (auto dest: reachable_init)
    next
      case (step s1 s2 s1' s2' a)
      hence "SubnetS s1 s2 ∈ reachable (pnet np (p1 ∥ p2)) TT"
        and sr1: "s1 ∈ reachable (pnet np p1) TT"
        and sr2: "s2 ∈ reachable (pnet np p2) TT"
        and "(SubnetS s1 s2, a, SubnetS s1' s2') ∈ trans (pnet np (p1 ∥ p2))" by auto
      from this(4)
        have "(SubnetS s1 s2, a, SubnetS s1' s2') ∈ pnet_sos (trans (pnet np p1)) (trans (pnet np p2))"
          by simp
      thus "s1' ∈ reachable (pnet np p1) TT
         ∧ s2' ∈ reachable (pnet np p2) TT"
        by cases (insert sr1 sr2, auto elim: reachable_step)
    qed
    thus "s1 ∈ reachable (pnet np p1) TT"
         "s2 ∈ reachable (pnet np p2) TT" by auto
  qed

lemma delivered_to_node [elim]:
  assumes "s ∈ reachable (pnet np ⟨ii; R_i⟩) TT"
      and "(s, i:deliver(d), s') ∈ trans (pnet np ⟨ii; R_i⟩)"
    shows "i = ii"
  proof -
    from assms(1) obtain P R where "s = NodeS ii P R"
      by (metis net_node_reachable_is_node)
    with assms(2) show "i = ii"
       by (clarsimp simp add: trans_node_comp elim!: node_deliverTE')
  qed

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"
  using assms proof (induction p arbitrary: s s')
    fix ii R_i s s'
    assume sr: "s ∈ reachable (pnet np ⟨ii; R_i⟩) TT"
       and "(s, i:deliver(d), s') ∈ trans (pnet np ⟨ii; R_i⟩)"
    from this(2) have tr: "(s, i:deliver(d), s') ∈ node_sos (trans (np ii))" by simp
    from sr obtain P R where [simp]: "s = NodeS ii P R"
      by (metis net_node_reachable_is_node)
    moreover from tr obtain P' R' where [simp]: "s' = NodeS ii P' R'"
      by simp (metis node_sos_dest)
    ultimately have "i = ii" using tr by auto
    thus "i ∈ net_ips s" by simp
  next
    fix p1 p2 s s'
    assume IH1: "⋀s s'. ⟦ s ∈ reachable (pnet np p1) TT;
                          (s, i:deliver(d), s') ∈ trans (pnet np p1) ⟧ ⟹ i ∈ net_ips s"
       and IH2: "⋀s s'. ⟦ s ∈ reachable (pnet np p2) TT;
                          (s, i:deliver(d), s') ∈ trans (pnet np p2) ⟧ ⟹ i ∈ net_ips s"
       and sr: "s ∈ reachable (pnet np (p1 ∥ p2)) TT"
       and tr: "(s, i:deliver(d), s') ∈ trans (pnet np (p1 ∥ p2))"
    from tr have "(s, i:deliver(d), s') ∈ pnet_sos (trans (pnet np p1)) (trans (pnet np p2))"
      by simp
    thus "i ∈ net_ips s"
    proof (rule partial_deliverTE)
      fix s1 s1' s2
      assume "s = SubnetS s1 s2"
         and "s' = SubnetS s1' s2"
         and tr: "(s1, i:deliver(d), s1') ∈ trans (pnet np p1)"
      from sr have "s1 ∈ reachable (pnet np p1) TT"
```

```
      by (auto simp only: ‹s = SubnetS s1 s2› elim: subnet_reachable)
    hence "i ∈ net_ips s1" using tr by (rule IH1)
    thus "i ∈ net_ips s" by (simp add: ‹s = SubnetS s1 s2›)
  next
    fix s2 s2' s1
    assume "s = SubnetS s1 s2"
       and "s' = SubnetS s1 s2'"
       and tr: "(s2, i:deliver(d), s2') ∈ trans (pnet np p2)"
    from sr have "s2 ∈ reachable (pnet np p2) TT"
      by (auto simp only: ‹s = SubnetS s1 s2› elim: subnet_reachable)
    hence "i ∈ net_ips s2" using tr by (rule IH2)
    thus "i ∈ net_ips s" by (simp add: ‹s = SubnetS s1 s2›)
  qed
qed

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 -
    from ‹wf_net_tree (p1 ∥ p2)› have "net_tree_ips p1 ∩ net_tree_ips p2 = {}" by auto
    moreover from assms(2) have "net_ips s1 = net_tree_ips p1" ..
    moreover from assms(3) have "net_ips s2 = net_tree_ips p2" ..
    ultimately show ?thesis by simp
  qed

lemma init_mapstate_Some_aodv_init [elim]:
  assumes "s ∈ init (pnet np p)"
      and "netmap s i = Some v"
    shows "v ∈ init (np i)"
  using assms proof (induction p arbitrary: s)
    fix ii R s
    assume "s ∈ init (pnet np ⟨ii; R⟩)"
       and "netmap s i = Some v"
    from this(1) obtain ns where s: "s = NodeS ii ns R"
      and ns: "ns ∈ init (np ii)" ..
    from s and ‹netmap s i = Some v› have "i = ii"
      by simp (metis domI domIff)
    with s ns show "v ∈ init (np i)"
      using ‹netmap s i = Some v› by simp
  next
    fix p1 p2 s
    assume IH1: "⋀s. s ∈ init (pnet np p1) ⟹ netmap s i = Some v ⟹ v ∈ init (np i)"
       and IH2: "⋀s. s ∈ init (pnet np p2) ⟹ netmap s i = Some v ⟹ v ∈ init (np i)"
       and "s ∈ init (pnet np (p1 ∥ p2))"
       and "netmap s i = Some v"
    from this(3) obtain s1 s2 where "s = SubnetS s1 s2"
                                and "s1 ∈ init (pnet np p1)"
                                and "s2 ∈ init (pnet np p2)" by auto
    from this(1) and ‹netmap s i = Some v›
      have "netmap s1 i = Some v ∨ netmap s2 i = Some v" by auto
    thus "v ∈ init (np i)"
    proof
      assume "netmap s1 i = Some v"
      with ‹s1 ∈ init (pnet np p1)› show ?thesis by (rule IH1)
    next
      assume "netmap s2 i = Some v"
      with ‹s2 ∈ init (pnet np p2)› show ?thesis by (rule IH2)
    qed
  qed

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"
  using assms proof (induction n arbitrary: s s')
    fix ii Rᵢ s s'
    assume sr: "s ∈ reachable (pnet np ⟨ii; Rᵢ⟩) TT"
      and "(s, connect(i, i'), s') ∈ trans (pnet np ⟨ii; Rᵢ⟩)"
    from this(2) have tr: "(s, connect(i, i'), s') ∈ node_sos (trans (np ii))" ..
    from sr obtain p R where "s = NodeS ii p R"
      by (metis net_node_reachable_is_node)
    with tr show "netmap s' = netmap s"
      by (auto elim!: node_sos.cases)
  next
    fix p1 p2 s s'
    assume IH1: "⋀s s'. ⟦ s ∈ reachable (pnet np p1) TT;
                        (s, connect(i, i'), s') ∈ trans (pnet np p1) ⟧ ⟹ netmap s' = netmap s"
      and IH2: "⋀s s'. ⟦ s ∈ reachable (pnet np p2) TT;
                        (s, connect(i, i'), s') ∈ trans (pnet np p2) ⟧ ⟹ netmap s' = netmap s"
      and sr: "s ∈ reachable (pnet np (p1 ∥ p2)) TT"
      and tr: "(s, connect(i, i'), s') ∈ trans (pnet np (p1 ∥ p2))"
    from tr have "(s, connect(i, i'), s') ∈ pnet_sos (trans (pnet np p1)) (trans (pnet np p2))"
      by simp
    thus "netmap s' = netmap s"
    proof cases
      fix s1 s1' s2 s2'
      assume "s = SubnetS s1 s2"
         and "s' = SubnetS s1' s2'"
         and tr1: "(s1, connect(i, i'), s1') ∈ trans (pnet np p1)"
         and tr2: "(s2, connect(i, i'), s2') ∈ trans (pnet np p2)"
    from this(1) and sr
      have "SubnetS s1 s2 ∈ reachable (pnet np (p1 ∥ p2)) TT" by simp
    hence sr1: "s1 ∈ reachable (pnet np p1) TT"
      and sr2: "s2 ∈ reachable (pnet np p2) TT"
      by (auto intro: subnet_reachable)
    from sr1 tr1 have "netmap s1' = netmap s1" by (rule IH1)
    moreover from sr2 tr2 have "netmap s2' = netmap s2" by (rule IH2)
    ultimately show "netmap s' = netmap s"
      using ‹s = SubnetS s1 s2› and ‹s' = SubnetS s1' s2'› by simp
    qed simp_all
  qed

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"
  using assms proof (induction n arbitrary: s s')
    fix ii Rᵢ s s'
    assume sr: "s ∈ reachable (pnet np ⟨ii; Rᵢ⟩) TT"
      and "(s, disconnect(i, i'), s') ∈ trans (pnet np ⟨ii; Rᵢ⟩)"
    from this(2) have tr: "(s, disconnect(i, i'), s') ∈ node_sos (trans (np ii))" ..
    from sr obtain p R where "s = NodeS ii p R"
      by (metis net_node_reachable_is_node)
    with tr show "netmap s' = netmap s"
      by (auto elim!: node_sos.cases)
  next
    fix p1 p2 s s'
    assume IH1: "⋀s s'. ⟦ s ∈ reachable (pnet np p1) TT;
                        (s, disconnect(i, i'), s') ∈ trans (pnet np p1) ⟧ ⟹ netmap s' = netmap s"
      and IH2: "⋀s s'. ⟦ s ∈ reachable (pnet np p2) TT;
                        (s, disconnect(i, i'), s') ∈ trans (pnet np p2) ⟧ ⟹ netmap s' = netmap s"
      and sr: "s ∈ reachable (pnet np (p1 ∥ p2)) TT"
      and tr: "(s, disconnect(i, i'), s') ∈ trans (pnet np (p1 ∥ p2))"
    from tr have "(s, disconnect(i, i'), s') ∈ pnet_sos (trans (pnet np p1)) (trans (pnet np p2))"
      by simp
    thus "netmap s' = netmap s"
```

```
    proof cases
      fix s1 s1' s2 s2'
      assume "s = SubnetS s1 s2"
         and "s' = SubnetS s1' s2'"
         and tr1: "(s1, disconnect(i, i'), s1') ∈ trans (pnet np p1)"
         and tr2: "(s2, disconnect(i, i'), s2') ∈ trans (pnet np p2)"
    from this(1) and sr
      have "SubnetS s1 s2 ∈ reachable (pnet np (p1 ∥ p2)) TT" by simp
    hence sr1: "s1 ∈ reachable (pnet np p1) TT"
      and sr2: "s2 ∈ reachable (pnet np p2) TT"
      by (auto intro: subnet_reachable)
    from sr1 tr1 have "netmap s1' = netmap s1" by (rule IH1)
    moreover from sr2 tr2 have "netmap s2' = netmap s2" by (rule IH2)
    ultimately show "netmap s' = netmap s"
      using ‹s = SubnetS s1 s2› and ‹s' = SubnetS s1' s2'› by simp
    qed simp_all
  qed

fun net_ip_action :: "(ip ⇒ ('s, 'm seq_action) automaton)
                      ⇒ 'm node_action ⇒ ip ⇒ net_tree ⇒ 's net_state ⇒ 's net_state ⇒ 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')"
  using assms proof (induction p arbitrary: s s')
    fix ii R_i s s'
    assume "s ∈ reachable (pnet np ⟨ii; R_i⟩) TT"
       and "(s, τ, s') ∈ trans (pnet np ⟨ii; R_i⟩)"
    from this obtain p R p' R' where "s = NodeS ii p R" and "s' = NodeS ii p' R'"
      by (metis (opaque_lifting, no_types) TT_True net_node_reachable_is_node
                                   reachable_step)
    hence "net_ips s = {ii}"
      and "net_ips s' = {ii}" by simp_all
    hence "∃i∈dom (netmap s). ∀j. j ≠ i ⟶ netmap s' j = netmap s j"
      by (simp add: net_ips_is_dom_netmap)
    thus "∃i∈net_ips s. (∀j. j ≠ i ⟶ netmap s' j = netmap s j)
                        ∧ net_ip_action np τ i (⟨ii; R_i⟩) s s'"
      by (simp add: net_ips_is_dom_netmap)
  next
    fix p1 p2 s s'
    assume IH1: "⋀s s'. ⟦ wf_net_tree p1;
                          s ∈ reachable (pnet np p1) TT;
                          (s, τ, s') ∈ trans (pnet np p1) ⟧
                        ⟹ ∃i∈net_ips s. (∀j. j ≠ i ⟶ netmap s' j = netmap s j)
                                         ∧ net_ip_action np τ i p1 s s'"
       and IH2: "⋀s s'. ⟦ wf_net_tree p2;
                          s ∈ reachable (pnet np p2) TT;
                          (s, τ, s') ∈ trans (pnet np p2) ⟧
                        ⟹ ∃i∈net_ips s. (∀j. j ≠ i ⟶ netmap s' j = netmap s j)
                                         ∧ net_ip_action np τ i p2 s s'"
       and sr: "s ∈ reachable (pnet np (p1 ∥ p2)) TT"
       and "wf_net_tree (p1 ∥ p2)"
       and tr: "(s, τ, s') ∈ trans (pnet np (p1 ∥ p2))"
    from ‹wf_net_tree (p1 ∥ p2)› have "net_tree_ips p1 ∩ net_tree_ips p2 = {}"
```

```
                              and "wf_net_tree p1"
                              and "wf_net_tree p2" by auto
  from tr have "(s, τ, s') ∈ pnet_sos (trans (pnet np p1)) (trans (pnet np p2))" by simp
  thus "∃i∈net_ips s. (∀j. j ≠ i ⟶ netmap s' j = netmap s j)
                     ∧ net_ip_action np τ i (p1 ∥ p2) s s'"
  proof cases
    fix s1 s1' s2
    assume subs:  "s = SubnetS s1 s2"
       and subs': "s' = SubnetS s1' s2"
       and tr1: "(s1, τ, s1') ∈ trans (pnet np p1)"
    from sr have sr1: "s1 ∈ reachable (pnet np p1) TT"
             and "s2 ∈ reachable (pnet np p2) TT"
      by (simp_all only: subs) (erule subnet_reachable)+
    with ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}› have "dom(netmap s1) ∩ dom(netmap s2) = {}"
      by (metis net_ips_is_dom_netmap pnet_net_ips_net_tree_ips)
    from ‹wf_net_tree p1› sr1 tr1 obtain i where "i∈dom(netmap s1)"
                                           and *: "∀j. j ≠ i ⟶ netmap s1' j = netmap s1 j"
                                           and "net_ip_action np τ i p1 s1 s1'"
        by (auto simp add: net_ips_is_dom_netmap dest!: IH1)
    from this(1) and ‹dom(netmap s1) ∩ dom(netmap s2) = {}› have "i∉dom(netmap s2)"
      by auto
    with subs subs' tr1 ‹net_ip_action np τ i p1 s1 s1'› have "net_ip_action np τ i (p1 ∥ p2) s s'"
      by (simp add: net_ips_is_dom_netmap)
    moreover have "∀j. j ≠ i ⟶ (netmap s1' ++ netmap s2) j = (netmap s1 ++ netmap s2) j"
    proof (intro allI impI)
      fix j
      assume "j ≠ i"
      with * have "netmap s1' j = netmap s1 j" by simp
      thus "(netmap s1' ++ netmap s2) j = (netmap s1 ++ netmap s2) j"
        by (metis (opaque_lifting, mono_tags) map_add_dom_app_simps(1) map_add_dom_app_simps(3))
    qed
    ultimately show ?thesis using ‹i∈dom(netmap s1)› subs subs'
      by (auto simp add: net_ips_is_dom_netmap)
  next
    fix s2 s2' s1
    assume subs: "s = SubnetS s1 s2"
       and subs': "s' = SubnetS s1 s2'"
       and tr2: "(s2, τ, s2') ∈ trans (pnet np p2)"
    from sr have "s1 ∈ reachable (pnet np p1) TT"
             and sr2: "s2 ∈ reachable (pnet np p2) TT"
      by (simp_all only: subs) (erule subnet_reachable)+
    with ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}› have "dom(netmap s1) ∩ dom(netmap s2) = {}"
      by (metis net_ips_is_dom_netmap pnet_net_ips_net_tree_ips)
    from ‹wf_net_tree p2› sr2 tr2 obtain i where "i∈dom(netmap s2)"
                                           and *: "∀j. j ≠ i ⟶ netmap s2' j = netmap s2 j"
                                           and "net_ip_action np τ i p2 s2 s2'"
        by (auto simp add: net_ips_is_dom_netmap dest!: IH2)
    from this(1) and ‹dom(netmap s1) ∩ dom(netmap s2) = {}› have "i∉dom(netmap s1)"
      by auto
    with subs subs' tr2 ‹net_ip_action np τ i p2 s2 s2'› have "net_ip_action np τ i (p1 ∥ p2) s s'"
      by (simp add: net_ips_is_dom_netmap)
    moreover have "∀j. j ≠ i ⟶ (netmap s1 ++ netmap s2') j = (netmap s1 ++ netmap s2) j"
    proof (intro allI impI)
      fix j
      assume "j ≠ i"
      with * have "netmap s2' j = netmap s2 j" by simp
      thus "(netmap s1 ++ netmap s2') j = (netmap s1 ++ netmap s2) j"
        by (metis (opaque_lifting, mono_tags) domD map_add_Some_iff map_add_dom_app_simps(3))
    qed
    ultimately show ?thesis using ‹i∈dom(netmap s2)› subs subs'
      by (clarsimp simp add: net_ips_is_dom_netmap)
         (metis domI dom_map_add map_add_find_right)
  qed simp_all
qed
```

```
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'")
  using assms proof (induction p arbitrary: s s')
    fix ii Rᵢ s s'
    assume sr: "s ∈ reachable (pnet np ⟨ii; Rᵢ⟩) TT"
       and tr: "(s, i:deliver(d), s') ∈ trans (pnet np ⟨ii; Rᵢ⟩)"
    from this obtain p R p' R' where "s = NodeS ii p R" and "s' = NodeS ii p' R'"
      by (metis (opaque_lifting, no_types) TT_True net_node_reachable_is_node
                                          reachable_step)
    hence "net_ips s = {ii}"
      and "net_ips s' = {ii}" by simp_all
    hence "∀j. j ≠ ii ⟶ netmap s' j = netmap s j"
      by simp
    moreover from sr tr have "i = ii" by (rule delivered_to_node)
    ultimately show "(∀j. j ≠ i ⟶ netmap s' j = netmap s j)
                    ∧ net_ip_action np (i:deliver(d)) i (⟨ii; Rᵢ⟩) s s'"
      by simp
  next
    fix p1 p2 s s'
    assume IH1: "⋀s s'. ⟦ wf_net_tree p1;
                         s ∈ reachable (pnet np p1) TT;
                         (s, i:deliver(d), s') ∈ trans (pnet np p1) ⟧
                        ⟹ (∀j. j ≠ i ⟶ netmap s' j = netmap s j)
                           ∧ net_ip_action np (i:deliver(d)) i p1 s s'"
       and IH2: "⋀s s'. ⟦ wf_net_tree p2;
                         s ∈ reachable (pnet np p2) TT;
                         (s, i:deliver(d), s') ∈ trans (pnet np p2) ⟧
                        ⟹ (∀j. j ≠ i ⟶ netmap s' j = netmap s j)
                           ∧ net_ip_action np (i:deliver(d)) i p2 s s'"
       and sr: "s ∈ reachable (pnet np (p1 ∥ p2)) TT"
       and "wf_net_tree (p1 ∥ p2)"
       and tr: "(s, i:deliver(d), s') ∈ trans (pnet np (p1 ∥ p2))"
    from ‹wf_net_tree (p1 ∥ p2)› have "net_tree_ips p1 ∩ net_tree_ips p2 = {}"
                                and "wf_net_tree p1"
                                and "wf_net_tree p2" by auto
    from tr have "(s, i:deliver(d), s') ∈ pnet_sos (trans (pnet np p1)) (trans (pnet np p2))" by simp
    thus "(∀j. j ≠ i ⟶ netmap s' j = netmap s j)
          ∧ net_ip_action np (i:deliver(d)) i (p1 ∥ p2) s s'"
    proof cases
      fix s1 s1' s2
      assume subs:  "s = SubnetS s1 s2"
         and subs': "s' = SubnetS s1' s2"
         and tr1: "(s1, i:deliver(d), s1') ∈ trans (pnet np p1)"
      from sr have sr1: "s1 ∈ reachable (pnet np p1) TT"
               and "s2 ∈ reachable (pnet np p2) TT"
        by (simp_all only: subs) (erule subnet_reachable)+
      with ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}› have "dom(netmap s1) ∩ dom(netmap s2) = {}"
        by (metis net_ips_is_dom_netmap pnet_net_ips_net_tree_ips)
      moreover from sr1 tr1 have "i ∈ net_ips s1" by (rule delivered_to_net_ips)
      ultimately have "i∉dom(netmap s2)" by (auto simp add: net_ips_is_dom_netmap)

      from ‹wf_net_tree p1› sr1 tr1 have *: "∀j. j ≠ i ⟶ netmap s1' j = netmap s1 j"
                                  and "net_ip_action np (i:deliver(d)) i p1 s1 s1'"
          by (auto dest!: IH1)
      from subs subs' tr1 this(2) ‹i∉dom(netmap s2)›
        have "net_ip_action np (i:deliver(d)) i (p1 ∥ p2) s s'"
          by (simp add: net_ips_is_dom_netmap)
      moreover have "∀j. j ≠ i ⟶ (netmap s1' ++ netmap s2) j = (netmap s1 ++ netmap s2) j"
      proof (intro allI impI)
```

```
        fix j
        assume "j ≠ i"
        with * have "netmap s1' j = netmap s1 j" by simp
        thus "(netmap s1' ++ netmap s2) j = (netmap s1 ++ netmap s2) j"
          by (metis (opaque_lifting, mono_tags) map_add_dom_app_simps(1) map_add_dom_app_simps(3))
      qed
      ultimately show ?thesis using ‹i∈net_ips s1› subs subs' by auto
    next
      fix s2 s2' s1
      assume subs: "s = SubnetS s1 s2"
         and subs': "s' = SubnetS s1 s2'"
         and tr2: "(s2, i:deliver(d), s2') ∈ trans (pnet np p2)"
      from sr have "s1 ∈ reachable (pnet np p1) TT"
               and sr2: "s2 ∈ reachable (pnet np p2) TT"
        by (simp_all only: subs) (erule subnet_reachable)+
      with ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}› have "dom(netmap s1) ∩ dom(netmap s2) = {}"
        by (metis net_ips_is_dom_netmap pnet_net_ips_net_tree_ips)
      moreover from sr2 tr2 have "i ∈ net_ips s2" by (rule delivered_to_net_ips)
      ultimately have "i∉dom(netmap s1)" by (auto simp add: net_ips_is_dom_netmap)

      from ‹wf_net_tree p2› sr2 tr2 have *: "∀j. j ≠ i ⟶ netmap s2' j = netmap s2 j"
                                 and "net_ip_action np (i:deliver(d)) i p2 s2 s2'"
          by (auto dest!: IH2)
      from subs subs' tr2 this(2) ‹i∉dom(netmap s1)›
        have "net_ip_action np (i:deliver(d)) i (p1 ∥ p2) s s'"
          by (simp add: net_ips_is_dom_netmap)
      moreover have "∀j. j ≠ i ⟶ (netmap s1 ++ netmap s2') j = (netmap s1 ++ netmap s2) j"
      proof (intro allI impI)
        fix j
        assume "j ≠ i"
        with * have "netmap s2' j = netmap s2 j" by simp
        thus "(netmap s1 ++ netmap s2') j = (netmap s1 ++ netmap s2) j"
          by (metis (opaque_lifting, mono_tags) domD map_add_Some_iff map_add_dom_app_simps(3))
      qed
      ultimately show ?thesis using ‹i∈net_ips s2› subs subs' by auto
    qed simp_all
  qed

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'"
  using assms by cases (auto dest: partial_net_preserves_subnets)

lemma complete_net_reachable_is_subnet:
  assumes "st ∈ reachable (closed (pnet np (p1 ∥ p2))) I"
    shows "∃s t. st = SubnetS s t"
  using assms by induction (auto dest!: complete_net_preserves_subnets)

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"
```

```
  using assms(1) proof (induction "SubnetS s t" arbitrary: s t)
    fix s t
    assume "SubnetS s t ∈ init (closed (pnet np (p1 ∥ p2)))"
    with init show "P s t" .
  next
    fix st a s' t'
    assume "st ∈ reachable (closed (pnet np (p1 ∥ p2))) I"
       and tr: "(st, a, SubnetS s' t') ∈ trans (closed (pnet np (p1 ∥ p2)))"
       and "I a"
       and IH: "⋀s t. st = SubnetS s t ⟹ P s t"
    from this(1) obtain s t where "st = SubnetS s t"
                            and "SubnetS s t ∈ reachable (closed (pnet np (p1 ∥ p2))) I"
      by (metis complete_net_reachable_is_subnet)
    note this(2)
    moreover from IH and ‹st = SubnetS s t› have "P s t" .
    moreover from tr and ‹st = SubnetS s t›
      have "(SubnetS s t, a, SubnetS s' t') ∈ trans (closed (pnet np (p1 ∥ p2)))" by simp
    ultimately show "P s' t'"
      using ‹I a› by (rule assms(3))
  qed

lemma reachable_closed_reachable_pnet [elim]:
  assumes "s ∈ reachable (closed (pnet np n)) TT"
    shows "s ∈ reachable (pnet np n) TT"
  using assms proof (induction rule: reachable.induct)
    fix s s' a
    assume sr: "s ∈ reachable (pnet np n) TT"
       and "(s, a, s') ∈ trans (closed (pnet np n))"
    from this(2) have "(s, a, s') ∈ cnet_sos (trans (pnet np n))" by simp
    thus "s' ∈ reachable (pnet np n) TT"
      by cases (insert sr, auto elim!: reachable_step)
  qed (auto elim: reachable_init)

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"
  using assms by (metis net_node_reachable_is_node reachable_closed_reachable_pnet surj_pair)

lemma closed_subnet_net_state [elim]:
  assumes "st ∈ reachable (closed (pnet np (p1 ∥ p2))) TT"
  obtains s t where "st = SubnetS s t"
  using assms by (metis reachable_closed_reachable_pnet net_par_reachable_is_subnet)

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)"
  using assms by (auto elim!: cnet_sos.cases)

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"
  using assms proof induction
    fix s
    assume "s ∈ init (closed (pnet np n))"
       and "i ∉ net_tree_ips n"
    thus "netmap s i = None"
    proof (induction n arbitrary: s)
      fix ii R s
      assume "s ∈ init (closed (pnet np ⟨ii; R⟩))"
         and "i ∉ net_tree_ips ⟨ii; R⟩"
      from this(2) have "i ≠ ii" by simp
      moreover from ‹s ∈ init (closed (pnet np ⟨ii; R⟩))› obtain p where "s = NodeS ii p R"
        by simp (metis pnet.simps(1) pnet_node_init')
```

```
      ultimately show "netmap s i = None" by simp
    next
      fix p1 p2 s
      assume IH1: "⋀s. s ∈ init (closed (pnet np p1)) ⟹ i ∉ net_tree_ips p1
                       ⟹ netmap s i = None"
         and IH2: "⋀s. s ∈ init (closed (pnet np p2)) ⟹ i ∉ net_tree_ips p2
                       ⟹ netmap s i = None"
         and "s ∈ init (closed (pnet np (p1 ∥ p2)))"
         and "i ∉ net_tree_ips (p1 ∥ p2)"
      from this(3) obtain s1 s2 where "s = SubnetS s1 s2"
                                  and "s1 ∈ init (closed (pnet np p1))"
                                  and "s2 ∈ init (closed (pnet np p2))" by simp metis
      moreover from ‹i ∉ net_tree_ips (p1 ∥ p2)› have "i ∉ net_tree_ips p1"
                                                  and "i ∉ net_tree_ips p2" by auto
      ultimately have "netmap s1 i = None"
                  and "netmap s2 i = None"
        using IH1 IH2 by auto
      with ‹s = SubnetS s1 s2› show "netmap s i = None" by simp
    qed
  next
    fix s a s'
    assume sr: "s ∈ reachable (closed (pnet np n)) TT"
       and tr: "(s, a, s') ∈ trans (closed (pnet np n))"
       and IH: "i ∉ net_tree_ips n ⟹ netmap s i = None"
       and "i ∉ net_tree_ips n"
    from this(3-4) have "i∉net_ips s" by auto
    with tr have "i∉net_ips s'"
      by simp (erule cnet_sos.cases, (metis net_ips_is_dom_netmap pnet_maintains_dom)+)
    thus "netmap s' i = None" by simp
  qed

lemma closed_pnet_aodv_init [elim]:
  assumes "s ∈ init (closed (pnet np n))"
      and "i∈net_tree_ips n"
    shows "the (netmap s i) ∈ init (np i)"
  using assms proof (induction n arbitrary: s)
    fix ii R s
    assume "s ∈ init (closed (pnet np ⟨ii; R⟩))"
       and "i∈net_tree_ips ⟨ii; R⟩"
    hence "s ∈ init (pnet np ⟨i; R⟩)" by simp
    then obtain p where "s = NodeS i p R"
                    and "p ∈ init (np i)" ..
    with ‹s = NodeS i p R› have "netmap s = [i ↦ p]" by simp
    with ‹p ∈ init (np i)› show "the (netmap s i) ∈ init (np i)" by simp
  next
    fix p1 p2 s
    assume IH1: "⋀s. s ∈ init (closed (pnet np p1)) ⟹
                    i∈net_tree_ips p1 ⟹ the (netmap s i) ∈ init (np i)"
       and IH2: "⋀s. s ∈ init (closed (pnet np p2)) ⟹
                   i∈net_tree_ips p2 ⟹ the (netmap s i) ∈ init (np i)"
       and "s ∈ init (closed (pnet np (p1 ∥ p2)))"
       and "i∈net_tree_ips (p1 ∥ p2)"
    from this(3) obtain s1 s2 where "s = SubnetS s1 s2"
                                and "s1 ∈ init (closed (pnet np p1))"
                                and "s2 ∈ init (closed (pnet np p2))"
      by auto
    from this(2) have "net_tree_ips p1 = net_ips s1"
      by (clarsimp dest!: pnet_init_net_ips_net_tree_ips)
    from ‹s2 ∈ init (closed (pnet np p2))› have "net_tree_ips p2 = net_ips s2"
      by (clarsimp dest!: pnet_init_net_ips_net_tree_ips)
    show "the (netmap s i) ∈ init (np i)"
    proof (cases "i∈net_tree_ips p2")
      assume "i∈net_tree_ips p2"
      with ‹s2 ∈ init (closed (pnet np p2))› have "the (netmap s2 i) ∈ init (np i)"
```

```
      by (rule IH2)
    moreover from ‹i∈net_tree_ips p2› and ‹net_tree_ips p2 = net_ips s2›
      have "i∈net_ips s2" by simp
    ultimately show ?thesis
      using ‹s = SubnetS s1 s2› by (auto simp add: net_ips_is_dom_netmap)
  next
    assume "i∉net_tree_ips p2"
    with ‹i∈net_tree_ips (p1 ∥ p2)› have "i∈net_tree_ips p1" by simp
    with ‹s1 ∈ init (closed (pnet np p1))› have "the (netmap s1 i) ∈ init (np i)"
      by (rule IH1)
    moreover from ‹i∈net_tree_ips p1› and ‹net_tree_ips p1 = net_ips s1›
      have "i∈net_ips s1" by simp
    moreover from ‹i∉net_tree_ips p2› and ‹net_tree_ips p2 = net_ips s2›
      have "i∉net_ips s2" by simp
    ultimately show ?thesis
      using ‹s = SubnetS s1 s2›
      by (simp add: map_add_dom_app_simps net_ips_is_dom_netmap)
  qed
qed

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 Γ :: "('s, 'm, 'p, 'l) seqp_env"
  and i :: ip
where
    obroadcastT: "σ' i = σ i ⟹
                ((σ, {l}broadcast(s_msg).p),       broadcast (s_msg (σ i)),              (σ', p)) ∈ oseqp_
Γ i"
  | ogroupcastT: "σ' i = σ i ⟹
                ((σ, {l}groupcast(s_ips, s_msg).p),  groupcast (s_ips (σ i)) (s_msg (σ i)), (σ', p)) ∈ oseqp_s
Γ i"
  | ounicastT:   "σ' i = σ i ⟹
                ((σ, {l}unicast(s_ip, s_msg).p ▷ q), unicast (s_ip (σ i)) (s_msg (σ i)),    (σ', p)) ∈ oseqp_
Γ i"
  | onotunicastT:"σ' i = σ i ⟹
                ((σ, {l}unicast(s_ip, s_msg).p ▷ q), ¬unicast (s_ip (σ i)),               (σ', q)) ∈ oseqp_so
Γ i"
  | osendT:      "σ' i = σ i ⟹
                ((σ, {l}send(s_msg).p),            send (s_msg (σ i)),                   (σ', p)) ∈ oseqp_s
Γ i"
  | odeliverT:   "σ' i = σ i ⟹
                ((σ, {l}deliver(s_data).p),        deliver (s_data (σ i)),               (σ', p)) ∈ oseqp_sos
Γ i"
  | oreceiveT:   "σ' i = u_msg msg (σ i) ⟹
                ((σ, {l}receive(u_msg).p),         receive msg,                          (σ', p)) ∈ oseqp_sos
Γ i"
  | oassignT:    "σ' i = u (σ i) ⟹
                ((σ, {l}⟦u⟧ p),                    τ,                                    (σ', p)) ∈ oseqp_sos
Γ i"
```

```
  | ocallT:      "((σ, Γ pn), a, (σ', p')) ∈ oseqp_sos Γ i ⟹
                  ((σ, call(pn)), a, (σ', p')) ∈ oseqp_sos Γ i"

  | ochoiceT1:   "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i ⟹
                  ((σ, p ⊕ q), a, (σ', p')) ∈ oseqp_sos Γ i"
  | ochoiceT2:   "((σ, q), a, (σ', q')) ∈ oseqp_sos Γ i ⟹
                  ((σ, p ⊕ q), a, (σ', q')) ∈ oseqp_sos Γ i"

  | oguardT:     "σ' i ∈ g (σ i) ⟹ ((σ, {l}⟨g⟩ p), τ, (σ', p)) ∈ oseqp_sos Γ i"

inductive_cases
      oseq_callTE [elim]:      "((σ, call(pn)), a, (σ', q)) ∈ oseqp_sos Γ i"
  and oseq_choiceTE [elim]:    "((σ, p1 ⊕ p2), a, (σ', q)) ∈ oseqp_sos Γ i"

lemma oseq_broadcastTE [elim]:
  "⟦((σ, {l}broadcast(s_msg). p), a, (σ', q)) ∈ oseqp_sos Γ i;
    ⟦a = broadcast (s_msg (σ i)); σ' i = σ i; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((σ, {l}broadcast(s_msg). p), a, (σ', q)) ∈ oseqp_sos Γ i") simp

lemma oseq_groupcastTE [elim]:
  "⟦((σ, {l}groupcast(s_ips, s_msg). p), a, (σ', q)) ∈ oseqp_sos Γ i;
    ⟦a = groupcast (s_ips (σ i)) (s_msg (σ i)); σ' i = σ i; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((σ, {l}groupcast(s_ips, s_msg). p), a, (σ', q)) ∈ oseqp_sos Γ i") simp

lemma oseq_unicastTE [elim]:
  "⟦((σ, {l}unicast(s_ip, s_msg). p ▷ q), a, (σ', r)) ∈ oseqp_sos Γ i;
    ⟦a = unicast (s_ip (σ i)) (s_msg (σ i)); σ' i = σ i; r = p⟧ ⟹ P;
    ⟦a = ¬unicast (s_ip (σ i)); σ' i = σ i; r = q⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((σ, {l}unicast(s_ip, s_msg). p ▷ q), a, (σ', r)) ∈ oseqp_sos Γ i") simp_all

lemma oseq_sendTE [elim]:
  "⟦((σ, {l}send(s_msg). p), a, (σ', q)) ∈ oseqp_sos Γ i;
    ⟦a = send (s_msg (σ i)); σ' i = σ i; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((σ, {l}send(s_msg). p), a, (σ', q)) ∈ oseqp_sos Γ i") simp

lemma oseq_deliverTE [elim]:
  "⟦((σ, {l}deliver(s_data). p), a, (σ', q)) ∈ oseqp_sos Γ i;
    ⟦a = deliver (s_data (σ i)); σ' i = σ i; q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((σ, {l}deliver(s_data). p), a, (σ', q)) ∈ oseqp_sos Γ i") simp

lemma oseq_receiveTE [elim]:
  "⟦((σ, {l}receive(u_msg). p), a, (σ', q)) ∈ oseqp_sos Γ i;
    ⋀msg. ⟦a = receive msg; σ' i = u_msg msg (σ i); q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((σ, {l}receive(u_msg). p), a, (σ', q)) ∈ oseqp_sos Γ i") simp

lemma oseq_assignTE [elim]:
  "⟦((σ, {l}⟦u⟧ p), a, (σ', q)) ∈ oseqp_sos Γ i; ⟦a = τ; σ' i = u (σ i); q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((σ, {l}⟦u⟧ p), a, (σ', q)) ∈ oseqp_sos Γ i") simp

lemma oseq_guardTE [elim]:
  "⟦((σ, {l}⟨g⟩ p), a, (σ', q)) ∈ oseqp_sos Γ i; ⟦a = τ; σ' i ∈ g (σ i); q = p⟧ ⟹ P⟧ ⟹ P"
  by (ind_cases "((σ, {l}⟨g⟩ p), a, (σ', q)) ∈ oseqp_sos Γ i") simp

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"
  by (ind_cases "((σ, (s, t)), broadcast m, (σ', (s', t'))) ∈ oparp_sos i S T") simp_all

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"
  by (ind_cases "((σ, (s, t)), groupcast ips m, (σ', (s', t'))) ∈ oparp_sos i S T") simp_all

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"
  by (ind_cases "((σ, (s, t)), unicast i m, (σ', (s', t'))) ∈ oparp_sos i S T") simp_all

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"
  by (ind_cases "((σ, (s, t)), notunicast i, (σ', (s', t'))) ∈ oparp_sos i S T") simp_all

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"
  by (ind_cases "((σ, (s, t)), send m, (σ', (s', t'))) ∈ oparp_sos i S T") auto

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"
  by (ind_cases "((σ, (s, t)), deliver d, (σ', (s', t'))) ∈ oparp_sos i S T") simp_all

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"
  by (ind_cases "((σ, (s, t)), receive m, (σ', (s', t'))) ∈ oparp_sos i S T") auto

inductive_cases opar_tauTE: "((σ, (s, t)), τ, (σ', (s', t'))) ∈ 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 "((σ, (s, t)), a, (σ', (s', t'))) ∈ oparp_sos i S T"
      and "⟦ ((σ, s), a, (σ', s')) ∈ S; ⋀m. a ≠ receive m; t' = t ⟧ ⟹ P"
      and "⟦ (t, a, t') ∈ T; ⋀m. a ≠ send m; s' = s; σ' i = σ i ⟧ ⟹ P"
      and "⋀m. ⟦ a = τ; ((σ, s), receive m, (σ', s')) ∈ S; (t, send m, t') ∈ T ⟧ ⟹ P"
    shows "P"
  using assms by cases auto

definition extg :: "('a × 'b) × 'c ⇒ 'a × 'b × 'c"
where "extg ≡ λ((σ, l1), l2). (σ, (l1, l2))"

lemma extgsimp [simp]:
  "extg ((σ, l1), l2) = (σ, (l1, l2))"
  unfolding extg_def by simp

lemma extg_range_prod: "extg ‘ (i1 × i2) = {(σ, (s1, s2))|σ s1 s2. (σ, s1) ∈ i1 ∧ s2 ∈ i2}"
  unfolding image_def extg_def
  by (rule Collect_cong) (auto split: prod.split)

definition
  opar_comp :: "((ip ⇒ 's) × 's1, 'm seq_action) automaton
              ⇒ ip
              ⇒ ('s2, 'm seq_action) automaton
              ⇒ ((ip ⇒ 's) × 's1 × 's2, 'm seq_action) automaton"
  (‹(_ ⟨⟨_ _)› [102, 0, 103] 102)
where
  "s ⟨⟨i t ≡ ⦇ init = extg ‘ (init s × init t), trans = oparp_sos i (trans s) (trans t) ⦈"

lemma opar_comp_def':
  "s ⟨⟨i t = ⦇ init = {(σ, (sl, tl))|σ sl tl. (σ, sl) ∈ init s ∧ tl ∈ init t},
              trans = oparp_sos i (trans s) (trans t) ⦈"
  unfolding opar_comp_def extg_def image_def by (auto split: prod.split)

lemma trans_opar_comp [simp]:
  "trans (s ⟨⟨i t) = oparp_sos i (trans s) (trans t)"
  unfolding opar_comp_def by simp

lemma init_opar_comp [simp]:
  "init (s ⟨⟨i t) = extg ‘ (init s × init t)"
  unfolding opar_comp_def by simp
```

13.3 Open structural operational semantics for node expressions

```
inductive_set
  onode_sos :: "((ip ⇒ 's) × 'l, 'm seq_action) transition set
             ⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) transition set"
  for S :: "((ip ⇒ 's) × 'l, 'm seq_action) transition set"
where
    onode_bcast:
    "((σ, s), broadcast m, (σ', s')) ∈ S ⟹ ((σ, NodeS i s R), R:*cast(m), (σ', NodeS i s' R)) ∈ onode_sos
S"

  | onode_gcast:
    "((σ, s), groupcast D m, (σ', s')) ∈ S ⟹ ((σ, NodeS i s R), (R∩D):*cast(m), (σ', 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}¬{}:arrive(m), (σ', NodeS i s' R)) ∈ onode
S"

  | onode_arrive:
    "σ' i = σ i ⟹ ((σ, NodeS i s R), {}¬{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})"
  using assms by rule simp_all

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 : Ri⟩o ≡ ⦇ init = {(σ, NodeS i s Ri)|σ s. (σ, s) ∈ init onp},
                       trans = onode_sos (trans onp) ⦈"

lemma trans_onode_comp:
  "trans (⟨i : S : R⟩o) = onode_sos (trans S)"
  unfolding onode_comp_def by simp

lemma init_onode_comp:
  "init (⟨i : S : R⟩o) = {(σ, NodeS i s R)|σ s. (σ, s) ∈ init S}"
  unfolding onode_comp_def by simp

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))"
  unfolding onode_comp_def by simp

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}"
  unfolding onode_comp_def by (simp add: extg_range_prod)

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')"
  using assms proof -
    assume "((σ, p), a, s') ∈ onode_sos S"
    then obtain σ' i' ζ' R' where "s' = (σ', NodeS i' ζ' R')"
      by (cases s') (auto elim!: onode_sos.cases)
    thus ?thesis by simp
  qed

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')"
  using assms proof -
    assume "((σ, NodeS i p R), a, s') ∈ onode_sos S"
    then obtain σ' ζ' R' where "s' = (σ', NodeS i ζ' R')"
      by (cases s') (auto elim!: onode_sos.cases)
    thus ?thesis by simp
  qed

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 -
    define ns' where "ns' = (σ', s')"
    with assms have "((σ, NodeS i p R), a, ns') ∈ onode_sos S" by simp
    then obtain σ'' ζ' R' where "ns' = (σ'', NodeS i ζ' R')"
      by (metis onode_sos_dest_is_net_state')
    hence "s' = NodeS i ζ' R'" by (simp add: ns'_def)
    thus ?thesis by simp
  qed

lemma onode_sos_src_is_net_state:
  assumes "((σ, p), a, s') ∈ onode_sos S"
    shows "∃i ζ R. p = NodeS i ζ R"
  using assms proof -
    assume "((σ, p), a, s') ∈ onode_sos S"
```

```
    then obtain i ζ R where "p = NodeS i ζ R"
      by (cases s') (auto elim!: onode_sos.cases)
    thus ?thesis by simp
  qed

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 -
    from assms obtain i ζ R where "s = NodeS i ζ R"
      by (metis onode_sos_src_is_net_state)
    moreover with assms obtain ζ' R' where "s' = NodeS i ζ' R'"
      by (auto dest!: onode_sos_dest_is_net_state')
    ultimately show ?thesis by simp
  qed

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}¬{}: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"
  by (erule onode_sos.cases) (simp | metis)+
```

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)
             ⇒ net_tree ⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) automaton"
where
    "opnet onp (⟨i; R_i⟩)  =  ⟨i : onp i : R_i⟩_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)}"
  using assms by (simp add: onode_comp_def)

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)"
   using assms by (auto simp add: onode_comp_def)

lemma opnet_node_trans [elim, simp]:
  assumes "(s, a, s') ∈ trans (opnet onp ⟨i; R⟩)"
    shows "(s, a, s') ∈ onode_sos (trans (onp i))"
  using assms by (simp add: trans_onode_comp)
```

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]:
  "((σ, s), R:*cast(m), (σ', s')) ∉ ocnet_sos T"
  proof
    assume "((σ, s), R:*cast(m), (σ', s')) ∈ ocnet_sos T"
    hence "R:*cast(m) ≠ R:*cast(m)"
     by (rule ocnet_sos.cases) auto
    thus False by simp
  qed

lemma ocomplete_no_arrive [simp]:
  "((σ, s), ii¬ni:arrive(m), (σ', s')) ∉ ocnet_sos T"
  proof
    assume "((σ, s), ii¬ni:arrive(m), (σ', s')) ∈ ocnet_sos T"
    hence "ii¬ni:arrive(m) ≠ ii¬ni:arrive(m)"
     by (rule ocnet_sos.cases) auto
    thus False by simp
  qed

lemma ocomplete_no_change [elim]:
  assumes "((σ, s), a, (σ', s')) ∈ ocnet_sos T"
      and "j ∉ net_ips s"
    shows "σ' j = σ j"
  using assms by cases simp_all

lemma ocomplete_transE [elim]:
  assumes "((σ, ζ), a, (σ', ζ')) ∈ ocnet_sos (trans (opnet onp n))"
  obtains a' where "((σ, ζ), a', (σ', ζ')) ∈ trans (opnet onp n)"
  using assms by (cases a) (auto elim!: ocompleteTEs [simplified])

abbreviation
  oclosed :: "((ip ⇒ 's) × 'l net_state, ('m::msg) node_action) automaton
             ⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) automaton"
where
  "oclosed ≡ (λA. A ⦇ trans := ocnet_sos (trans A) ⦈)"
```

end

14 Configure the inv-cterms tactic for open sequential processes

```
theory OAWN_SOS_Labels
imports OAWN_SOS Inv_Cterms
begin

lemma oelimder_guard:
  assumes "p = {l}⟨fg⟩ qq"
      and "l' ∈ labels Γ q"
      and "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
  obtains p' where "p = {l}⟨fg⟩ p'"
               and "l' ∈ labels Γ qq"
  using assms by auto

lemma oelimder_assign:
  assumes "p = {l}⟦fa⟧ qq"
      and "l' ∈ labels Γ q"
      and "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
  obtains p' where "p = {l}⟦fa⟧ p'"
               and "l' ∈ labels Γ qq"
  using assms by auto

lemma oelimder_ucast:
  assumes "p = {l}unicast(fip, fmsg).q1 ▷ q2"
      and "l' ∈ labels Γ q"
      and "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
  obtains p' pp' where "p = {l}unicast(fip, fmsg).p' ▷ pp'"
                   and "case a of unicast _ _ ⇒ l' ∈ labels Γ q1
                                          | _ ⇒ l' ∈ labels Γ q2"
  using assms by simp (erule oseqpTEs, auto)

lemma oelimder_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"
  using assms by auto

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"
  using assms by auto

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"
  using assms by auto

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"
  using assms by auto
```

```
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"
  using assms by auto

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'"
  using assms proof (induction p arbitrary: σ ns a σ' ns')
    fix i R σ ns a σ' ns'
    assume "((σ, ns), a, (σ', ns')) ∈ trans (opnet np ⟨i; R⟩)"
    hence "((σ, ns), a, (σ', ns')) ∈ onode_sos (trans (np i))" ..
    thus "net_ips ns = net_ips ns'"
      by (simp add: net_ips_is_dom_netmap)
         (erule onode_sos.cases, simp_all)
  next
    fix p1 p2 σ ns a σ' ns'
    assume "⋀σ ns a σ' ns'. ((σ, ns), a, (σ', ns')) ∈ trans (opnet np p1) ⟹ net_ips ns = net_ips ns'"
       and "⋀σ ns a σ' ns'. ((σ, ns), a, (σ', ns')) ∈ trans (opnet np p2) ⟹ net_ips ns = net_ips ns'"
       and "((σ, ns), a, (σ', ns')) ∈ trans (opnet np (p1 ∥ p2))"
    thus "net_ips ns = net_ips ns'"
      by simp (erule opnet_sos.cases, simp_all)
  qed

lemma opnet_net_ips_net_tree_ips:
  assumes "(σ, ns) ∈ oreachable (opnet np p) S U"
    shows "net_ips ns = net_tree_ips p"
  using assms proof (induction rule: oreachable_pair_induct)
    fix σ s
    assume "(σ, s) ∈ init (opnet np p)"
    thus "net_ips s = net_tree_ips p"
    proof (induction p arbitrary: σ s)
      fix p1 p2 σ s
      assume IH1: "(⋀σ s. (σ, s) ∈ init (opnet np p1) ⟹ net_ips s = net_tree_ips p1)"
         and IH2: "(⋀σ s. (σ, s) ∈ init (opnet np p2) ⟹ net_ips s = net_tree_ips p2)"
         and "(σ, s) ∈ init (opnet np (p1 ∥ p2))"
      thus "net_ips s = net_tree_ips (p1 ∥ p2)"
```

```
      by (clarsimp simp add: net_ips_is_dom_netmap)
         (metis Un_commute)
    qed (clarsimp simp add: onode_comps)
  next
    fix σ s σ' s' a
    assume "(σ, s) ∈ oreachable (opnet np p) S U"
       and "net_ips s = net_tree_ips p"
       and "((σ, s), a, (σ', s')) ∈ trans (opnet np p)"
       and "S σ σ' a"
    thus "net_ips s' = net_tree_ips p"
      by (simp add: net_ips_is_dom_netmap)
         (metis net_ips_is_dom_netmap opnet_maintains_dom)
  qed simp

lemma opnet_net_ips_net_tree_ips_init:
  assumes "(σ, ns) ∈ init (opnet np p)"
    shows "net_ips ns = net_tree_ips p"
  using assms(1) by (rule oreachable_init [THEN opnet_net_ips_net_tree_ips])

lemma opartial_net_preserves_subnets:
  assumes "((σ, SubnetS s t), a, (σ', st')) ∈ opnet_sos (trans (opnet np p1)) (trans (opnet np p2))"
    shows "∃s' t'. st' = SubnetS s' t'"
  using assms by cases simp_all

lemma net_par_oreachable_is_subnet:
  assumes "(σ, st) ∈ oreachable (opnet np (p1 ∥ p2)) S U"
    shows "∃s t. st = SubnetS s t"
  proof -
    define p where "p = (σ, st)"
    with assms have "p ∈ oreachable (opnet np (p1 ∥ p2)) S U" by simp
    hence "∃σ s t. p = (σ, SubnetS s t)"
      by induct (auto dest!: opartial_net_preserves_subnets)
    with p_def show ?thesis by simp
  qed

end
```

16 Lifting rules for (open) nodes

```
theory ONode_Lifting
imports AWN OAWN_SOS OInvariants
begin

lemma node_net_state':
  assumes "s ∈ oreachable (⟨i : T : R_i⟩_o) S U"
    shows "∃σ ζ R. s = (σ, NodeS i ζ R)"
  using assms proof induction
    fix s
    assume "s ∈ init (⟨i : T : R_i⟩_o)"
    then obtain σ ζ where "s = (σ, NodeS i ζ R_i)"
      by (auto simp: onode_comps)
    thus "∃σ ζ R. s = (σ, NodeS i ζ R)" by auto
  next
    fix s a σ'
    assume rt: "s ∈ oreachable (⟨i : T : R_i⟩_o) S U"
       and ih: "∃σ ζ R. s = (σ, NodeS i ζ R)"
       and "U (fst s) σ'"
    then obtain σ ζ R
      where "(σ, NodeS i ζ R)  ∈ oreachable (⟨i : T : R_i⟩_o) S U"
        and "U σ σ'" and "snd s = NodeS i ζ R" by auto
    from this(1-2)
      have "(σ', NodeS i ζ R) ∈ oreachable (⟨i : T : R_i⟩_o) S U"
        by - (erule(1) oreachable_other')
    with ‹snd s = NodeS i ζ R› show "∃σ ζ R. (σ', snd s) = (σ, NodeS i ζ R)" by simp
```

```
  next
    fix s a s'
    assume rt: "s ∈ oreachable (⟨i : T : R_i⟩_o) S U"
       and ih: "∃σ ζ R. s = (σ, NodeS i ζ R)"
       and tr: "(s, a, s') ∈ trans (⟨i : T : R_i⟩_o)"
       and "S (fst s) (fst s') a"
     from ih obtain σ ζ R where "s = (σ, NodeS i ζ R)" by auto
     with tr have "((σ, NodeS i ζ R), a, s') ∈ onode_sos (trans T)"
       by (simp add: onode_comps)
     then obtain σ' ζ' R' where "s' = (σ', NodeS i ζ' R')"
       using onode_sos_dest_is_net_state' by metis
     with tr ‹s = (σ, NodeS i ζ R)› show "∃σ ζ R. s' = (σ, NodeS i ζ R)"
       by simp
  qed

lemma node_net_state:
  assumes "(σ, s) ∈ oreachable (⟨i : T : R_i⟩_o) S U"
    shows "∃ζ R. s = NodeS i ζ R"
  using assms
  by (metis Pair_inject node_net_state')

lemma node_net_state_trans [elim]:
  assumes sor: "(σ, s) ∈ oreachable (⟨i : ζ_i : R_i⟩_o) S U"
      and str: "((σ, s), a, (σ', s')) ∈ trans (⟨i : ζ_i : R_i⟩_o)"
  obtains ζ R ζ' R'
    where "s = NodeS i ζ R"
      and "s' = NodeS i ζ' R'"
  proof -
    assume *: "⋀ζ R ζ' R'. s = NodeS i ζ R ⟹ s' = NodeS i ζ' R' ⟹ thesis"
    from sor obtain ζ R where "s = NodeS i ζ R"
      by (metis node_net_state)
    moreover with str obtain ζ' R' where "s' = NodeS i ζ' R'"
      by (simp only: onode_comps)
         (metis onode_sos_dest_is_net_state'')
    ultimately show thesis by (rule *)
  qed

lemma nodemap_induct' [consumes, case_names init other local]:
  assumes "(σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
      and init: "⋀σ ζ. (σ, NodeS ii ζ R_i) ∈ init (⟨ii : T : R_i⟩_o) ⟹ P (σ, NodeS ii ζ R_i)"
      and other: "⋀σ ζ R σ' a.
                  ⟦ (σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U;
                    U σ σ'; P (σ, NodeS ii ζ R) ⟧ ⟹ P (σ', NodeS ii ζ R)"
      and local: "⋀σ ζ R σ' ζ' R' a.
                  ⟦ (σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U;
                    ((σ, NodeS ii ζ R), a, (σ', NodeS ii ζ' R')) ∈ trans (⟨ii : T : R_i⟩_o);
                    S σ σ' a; P (σ, NodeS ii ζ R) ⟧ ⟹ P (σ', NodeS ii ζ' R')"
    shows "P (σ, NodeS ii ζ R)"
  using assms(1) proof induction
    fix s
    assume "s ∈ init (⟨ii : T : R_i⟩_o)"
    hence "s ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
      by (rule oreachable_init)
    with ‹s ∈ init (⟨ii : T : R_i⟩_o)› obtain σ ζ where "s = (σ, NodeS ii ζ R_i)"
      by (simp add: onode_comps) metis
    with ‹s ∈ init (⟨ii : T : R_i⟩_o)› and init show "P s" by simp
  next
    fix s a σ'
    assume sr: "s ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
       and "U (fst s) σ'"
       and "P s"
    from sr obtain σ ζ R where "s = (σ, NodeS ii ζ R)"
      using node_net_state' by metis
    with sr ‹U (fst s) σ'› ‹P s› show "P (σ', snd s)"
```

```
    by simp (metis other)
  next
    fix s a s'
    assume sr: "s ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
       and tr: "(s, a, s') ∈ trans (⟨ii : T : R_i⟩_o)"
       and "S (fst s) (fst s') a"
       and "P s"
    from this(1-3) have "s' ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
      by - (erule(2) oreachable_local)
    then obtain σ' ζ' R' where [simp]: "s' = (σ', NodeS ii ζ' R')"
      using node_net_state' by metis
    from sr and ‹P s› obtain σ ζ R
      where [simp]: "s = (σ, NodeS ii ζ R)"
        and A1: "(σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
        and A4: "P (σ, NodeS ii ζ R)"
      using node_net_state' by metis
    with tr and ‹S (fst s) (fst s') a›
      have A2: "((σ, NodeS ii ζ R), a, (σ', NodeS ii ζ' R')) ∈ trans (⟨ii : T : R_i⟩_o)"
       and A3: "S σ σ' a" by simp_all
    from A1 A2 A3 A4 have "P (σ', NodeS ii ζ' R')" by (rule local)
    thus "P s'" by simp
  qed

lemma nodemap_induct [consumes, case_names init step]:
  assumes "(σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
      and init: "⋀σ ζ. (σ, NodeS ii ζ R_i) ∈ init (⟨ii : T : R_i⟩_o) ⟹ P σ ζ R_i"
      and other: "⋀σ ζ R σ' a.
                  ⟦ (σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U;
                    U σ σ'; P σ ζ R ⟧ ⟹ P σ' ζ R"
      and local: "⋀σ ζ R σ' ζ' R' a.
                  ⟦ (σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U;
                    ((σ, NodeS ii ζ R), a, (σ', NodeS ii ζ' R')) ∈ trans (⟨ii : T : R_i⟩_o);
                    S σ σ' a; P σ ζ R ⟧ ⟹ P σ' ζ' R'"
    shows "P σ ζ R"
  using assms(1) proof (induction "(σ, NodeS ii ζ R)" arbitrary: σ ζ R)
    fix σ ζ R
    assume a1: "(σ, NodeS ii ζ R) ∈ init (⟨ii : T : R_i⟩_o)"
    hence "R = R_i" by (simp add: init_onode_comp)
    with a1 have "(σ, NodeS ii ζ R_i) ∈ init (⟨ii : T : R_i⟩_o)" by simp
    with init and ‹R = R_i› show "P σ ζ R" by simp
  next
    fix st a σ' ζ' R'
    assume "st ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
       and tr: "(st, a, (σ', NodeS ii ζ' R')) ∈ trans (⟨ii : T : R_i⟩_o)"
       and "S (fst st) (fst (σ', NodeS ii ζ' R')) a"
       and IH: "⋀σ ζ R. st = (σ, NodeS ii ζ R) ⟹ P σ ζ R"
    from this(1) obtain σ ζ R where "st = (σ, NodeS ii ζ R)"
                              and "(σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
      by (metis node_net_state')
    note this(2)
    moreover from tr and ‹st = (σ, NodeS ii ζ R)›
      have "((σ, NodeS ii ζ R), a, (σ', NodeS ii ζ' R')) ∈ trans (⟨ii : T : R_i⟩_o)" by simp
    moreover from ‹S (fst st) (fst (σ', NodeS ii ζ' R')) a› and ‹st = (σ, NodeS ii ζ R)›
      have "S σ σ' a" by simp
    moreover from IH and ‹st = (σ, NodeS ii ζ R)› have "P σ ζ R" .
    ultimately show "P σ' ζ' R'" by (rule local)
  next
    fix st σ' ζ R
    assume "st ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
       and "U (fst st) σ'"
       and "snd st = NodeS ii ζ R"
       and IH: "⋀σ ζ R. st = (σ, NodeS ii ζ R) ⟹ P σ ζ R"
    from this(1,3) obtain σ where "st = (σ, NodeS ii ζ R)"
                            and "(σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
```

```
    by (metis surjective_pairing)
  note this(2)
  moreover from ‹U (fst st) σ'› and ‹st = (σ, NodeS ii ζ R)› have "U σ σ'" by simp
  moreover from IH and ‹st = (σ, NodeS ii ζ R)› have "P σ ζ R" .
  ultimately show "P σ' ζ R" by (rule other)
qed

lemma node_addressD [dest, simp]:
  assumes "(σ, NodeS i ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o) S U"
    shows "i = ii"
  using assms by (clarsimp dest!: node_net_state')

lemma node_proc_reachable [dest]:
  assumes "(σ, NodeS i ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o)
                                          (otherwith S {ii} (oarrivemsg I)) (other U {ii})"
      and sgivesu: "⋀ξ ξ'. S ξ ξ' ⟹ U ξ ξ'"
    shows "(σ, ζ) ∈ oreachable T (otherwith S {ii} (orecvmsg I)) (other U {ii})"
  proof -
    from assms(1) have "(σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o)
                                              (otherwith S {ii} (oarrivemsg I)) (other U {ii})"
      by - (frule node_addressD, simp)
    thus ?thesis
    proof (induction rule: nodemap_induct)
      fix σ ζ
      assume "(σ, NodeS ii ζ R_i) ∈ init (⟨ii : T : R_i⟩_o)"
      hence "(σ, ζ) ∈ init T" by (auto simp: onode_comps)
      thus "(σ, ζ) ∈ oreachable T (otherwith S {ii} (orecvmsg I)) (other U {ii})"
        by (rule oreachable_init)
    next
      fix σ ζ R σ' ζ' R' a
      assume "other U {ii} σ σ'"
         and "(σ, ζ) ∈ oreachable T (otherwith S {ii} (orecvmsg I)) (other U {ii})"
      thus "(σ', ζ) ∈ oreachable T (otherwith S {ii} (orecvmsg I)) (other U {ii})"
        by - (rule oreachable_other')
    next
      fix σ ζ R σ' ζ' R' a
      assume rs: "(σ, NodeS ii ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o)
                                          (otherwith S {ii} (oarrivemsg I)) (other U {ii})"
         and tr: "((σ, NodeS ii ζ R), a, (σ', NodeS ii ζ' R')) ∈ trans (⟨ii : T : R_i⟩_o)"
         and ow: "otherwith S {ii} (oarrivemsg I) σ σ' a"
         and ih: "(σ, ζ) ∈ oreachable T (otherwith S {ii} (orecvmsg I)) (other U {ii})"

      from ow have *: "σ' ii = σ ii ⟹ other U {ii} σ σ'"
        by (clarsimp elim!: otherwithE) (rule otherI, simp_all, metis sgivesu)
      from tr have "((σ, NodeS ii ζ R), a, (σ', NodeS ii ζ' R')) ∈ onode_sos (trans T)"
        by (simp add: onode_comps)
      thus "(σ', ζ') ∈ oreachable T (otherwith S {ii} (orecvmsg I)) (other U {ii})"
      proof cases
        case onode_bcast
        with ih and ow show ?thesis
          by (auto elim!: oreachable_local' otherwithE)
      next
        case onode_gcast
        with ih and ow show ?thesis
          by (auto elim!: oreachable_local' otherwithE)
      next
        case onode_ucast
        with ih and ow show ?thesis
          by (auto elim!: oreachable_local' otherwithE)
      next
        case onode_notucast
        with ih and ow show ?thesis
          by (auto elim!: oreachable_local' otherwithE)
      next
```

```
        case onode_deliver
        with ih and ow show ?thesis
          by (auto elim!: oreachable_local' otherwithE)
      next
        case onode_tau
        with ih and ow show ?thesis
          by (auto elim!: oreachable_local' otherwithE)
      next
        case onode_receive
        with ih and ow show ?thesis
          by (auto elim!: oreachable_local' otherwithE)
      next
        case (onode_arrive m)
        hence "ζ' = ζ" and "σ' ii = σ ii" by auto
        from this(2) have "other U {ii} σ σ'" by (rule *)
        with ih and ‹ζ' = ζ› show ?thesis by auto
      next
        case onode_connect1
        hence "ζ' = ζ" and "σ' ii = σ ii" by auto
        from this(2) have "other U {ii} σ σ'" by (rule *)
        with ih and ‹ζ' = ζ› show ?thesis by auto
      next
        case onode_connect2
        hence "ζ' = ζ" and "σ' ii = σ ii" by auto
        from this(2) have "other U {ii} σ σ'" by (rule *)
        with ih and ‹ζ' = ζ› show ?thesis by auto
      next
        case onode_connect_other
        hence "ζ' = ζ" and "σ' ii = σ ii" by auto
        from this(2) have "other U {ii} σ σ'" by (rule *)
        with ih and ‹ζ' = ζ› show ?thesis by auto
      next
        case onode_disconnect1
        hence "ζ' = ζ" and "σ' ii = σ ii" by auto
        from this(2) have "other U {ii} σ σ'" by (rule *)
        with ih and ‹ζ' = ζ› show ?thesis by auto
      next
        case onode_disconnect2
        hence "ζ' = ζ" and "σ' ii = σ ii" by auto
        from this(2) have "other U {ii} σ σ'" by (rule *)
        with ih and ‹ζ' = ζ› show ?thesis by auto
      next
        case onode_disconnect_other
        hence "ζ' = ζ" and "σ' ii = σ ii" by auto
        from this(2) have "other U {ii} σ σ'" by (rule *)
        with ih and ‹ζ' = ζ› show ?thesis by auto
      qed
    qed
  qed

lemma node_proc_reachable_statelessassm [dest]:
  assumes "(σ, NodeS i ζ R) ∈ oreachable (⟨ii : T : R_i⟩_o)
                                          (otherwith (λ_ _. True) {ii} (oarrivemsg I))
                                          (other (λ_ _. True) {ii})"
    shows "(σ, ζ) ∈ oreachable T
                             (otherwith (λ_ _. True) {ii} (orecvmsg I)) (other (λ_ _. True) {ii})"
  using assms
  by (rule node_proc_reachable) simp_all

lemma node_lift:
  assumes "T ⊨ (otherwith S {ii} (orecvmsg I), other U {ii} →) global P"
      and "⋀ξ ξ'. S ξ ξ' ⟹ U ξ ξ'"
    shows "⟨ii : T : R_i⟩_o ⊨ (otherwith S {ii} (oarrivemsg I), other U {ii} →) global P"
  proof (rule oinvariant_oreachableI)
```

```
    fix σ ζ
    assume "(σ, ζ) ∈ oreachable (⟨ii : T : R_i⟩_o) (otherwith S {ii} (oarrivemsg I)) (other U {ii})"
    moreover then obtain i s R where "ζ = NodeS i s R"
      by (metis node_net_state)
    ultimately have "(σ, NodeS i s R) ∈ oreachable (⟨ii : T : R_i⟩_o)
                                                   (otherwith S {ii} (oarrivemsg I)) (other U {ii})"
      by simp
    hence "(σ, s) ∈ oreachable T (otherwith S {ii} (orecvmsg I)) (other U {ii})"
      by - (erule node_proc_reachable, erule assms(2))
    with assms(1) show "global P (σ, ζ)"
      by (metis fst_conv globalsimp oinvariantD)
  qed

lemma node_lift_step [intro]:
  assumes pinv: "T ⊨_A (otherwith S {i} (orecvmsg I), other U {i} →) globala (λ(σ, _, σ'). Q σ σ')"
      and other: "⋀σ σ'. other U {i} σ σ' ⟹ Q σ σ'"
      and sgivesu: "⋀ξ ξ'. S ξ ξ' ⟹ U ξ ξ'"
    shows "⟨i : T : R_i⟩_o ⊨_A (otherwith S {i} (oarrivemsg I), other U {i} →)
                              globala (λ(σ, _, σ'). Q σ σ')"
    (is "_ ⊨_A (?S, ?U →) _")
  proof (rule ostep_invariantI, simp)
    fix σ s a σ' s'
    assume rs: "(σ, s) ∈ oreachable (⟨i : T : R_i⟩_o) ?S ?U"
       and tr: "((σ, s), a, (σ', s')) ∈ trans (⟨i : T : R_i⟩_o)"
       and ow: "?S σ σ' a"
    from ow have *: "σ' i = σ i ⟹ other U {i} σ σ'"
      by (clarsimp elim!: otherwithE) (rule otherI, simp_all, metis sgivesu)
    from rs tr obtain ζ R
      where [simp]: "s = NodeS i ζ R"
        and "(σ, NodeS i ζ R) ∈ oreachable (⟨i : T : R_i⟩_o) ?S ?U"
      by (metis node_net_state)
    from this(2) have or: "(σ, ζ) ∈ oreachable T (otherwith S {i} (orecvmsg I)) ?U"
      by (rule node_proc_reachable [OF _ assms(3)])
    from tr have "((σ, NodeS i ζ R), a, (σ', s')) ∈ onode_sos (trans T)"
      by (simp add: onode_comps)
    thus "Q σ σ'"
    proof cases
      fix m ζ'
      assume "a = R:*cast(m)"
         and tr': "((σ, ζ), broadcast m, (σ', ζ')) ∈ trans T"
      from this(1) and <?S σ σ' a> have "otherwith S {i} (orecvmsg I) σ σ' (broadcast m)"
        by (auto elim!: otherwithE)
      with or tr' show ?thesis by (rule ostep_invariantD [OF pinv, simplified])
    next
      fix D m ζ'
      assume "a = (R ∩ D):*cast(m)"
         and tr': "((σ, ζ), groupcast D m, (σ', ζ')) ∈ trans T"
      from this(1) and <?S σ σ' a> have "otherwith S {i} (orecvmsg I) σ σ' (groupcast D m)"
        by (auto elim!: otherwithE)
      with or tr' show ?thesis by (rule ostep_invariantD [OF pinv, simplified])
    next
      fix d m ζ'
      assume "a = {d}:*cast(m)"
         and tr': "((σ, ζ), unicast d m, (σ', ζ')) ∈ trans T"
      from this(1) and <?S σ σ' a> have "otherwith S {i} (orecvmsg I) σ σ' (unicast d m)"
        by (auto elim!: otherwithE)
      with or tr' show ?thesis by (rule ostep_invariantD [OF pinv, simplified])
    next
      fix d ζ'
      assume "a = τ"
         and tr': "((σ, ζ), ¬unicast d, (σ', ζ')) ∈ trans T"
      from this(1) and <?S σ σ' a> have "otherwith S {i} (orecvmsg I) σ σ' (¬unicast d)"
        by (auto elim!: otherwithE)
      with or tr' show ?thesis by (rule ostep_invariantD [OF pinv, simplified])
```

```
next
  fix d ζ'
  assume "a = i:deliver(d)"
     and tr': "((σ, ζ), deliver d, (σ', ζ')) ∈ trans T"
  from this(1) and ‹?S σ σ' a› have "otherwith S {i} (orecvmsg I) σ σ' (deliver d)"
    by (auto elim!: otherwithE)
  with or tr' show ?thesis by (rule ostep_invariantD [OF pinv, simplified])
next
  fix ζ'
  assume "a = τ"
     and tr': "((σ, ζ), τ, (σ', ζ')) ∈ trans T"
  from this(1) and ‹?S σ σ' a› have "otherwith S {i} (orecvmsg I) σ σ' τ"
    by (auto elim!: otherwithE)
  with or tr' show ?thesis by (rule ostep_invariantD [OF pinv, simplified])
next
  fix m ζ'
  assume "a = {i}¬{}:arrive(m)"
     and tr': "((σ, ζ), receive m, (σ', ζ')) ∈ trans T"
  from this(1) and ‹?S σ σ' a› have "otherwith S {i} (orecvmsg I) σ σ' (receive m)"
    by (auto elim!: otherwithE)
  with or tr' show ?thesis by (rule ostep_invariantD [OF pinv, simplified])
next
  fix m
  assume "a = {}¬{i}:arrive(m)"
     and "σ' i = σ i"
  from this(2) have "other U {i} σ σ'" by (rule *)
  thus ?thesis by (rule other)
next
  fix i'
  assume "a = connect(i, i')"
     and "σ' i = σ i"
  from this(2) have "other U {i} σ σ'" by (rule *)
  thus ?thesis by (rule other)
next
  fix i'
  assume "a = connect(i', i)"
     and "σ' i = σ i"
  from this(2) have "other U {i} σ σ'" by (rule *)
  thus ?thesis by (rule other)
next
  fix i' i''
  assume "a = connect(i', i'')"
     and "σ' i = σ i"
  from this(2) have "other U {i} σ σ'" by (rule *)
  thus ?thesis by (rule other)
next
  fix i'
  assume "a = disconnect(i, i')"
     and "σ' i = σ i"
  from this(2) have "other U {i} σ σ'" by (rule *)
  thus ?thesis by (rule other)
next
  fix i'
  assume "a = disconnect(i', i)"
     and "σ' i = σ i"
  from this(2) have "other U {i} σ σ'" by (rule *)
  thus ?thesis by (rule other)
next
  fix i' i''
  assume "a = disconnect(i', i'')"
     and "σ' i = σ i"
  from this(2) have "other U {i} σ σ'" by (rule *)
  thus ?thesis by (rule other)
qed
```

```
  qed

lemma node_lift_step_statelessassm [intro]:
  assumes "T ⊨A (λσ _. orecvmsg I σ, other (λ_ _. True) {i} →)
                    globala (λ(σ, _, σ'). Q (σ i) (σ' i))"
      and "⋀ξ. Q ξ ξ"
    shows "⟨i : T : Ri⟩o ⊨A (λσ _. oarrivemsg I σ, other (λ_ _. True) {i} →)
                         globala (λ(σ, _, σ'). Q (σ i) (σ' i))"
  proof -
    from assms(1)
      have "T ⊨A (otherwith (λ_ _. True) {i} (orecvmsg I), other (λ_ _. True) {i} →)
                globala (λ(σ, _, σ'). Q (σ i) (σ' i))"
        by rule auto
    with assms(2) have "⟨i : T : Ri⟩o ⊨A (otherwith (λ_ _. True) {i} (oarrivemsg I),
                                         other (λ_ _. True) {i} →)
                                        globala (λ(σ, _, σ'). Q (σ i) (σ' i))"
      by - (rule node_lift_step, auto)
    thus ?thesis by rule auto
  qed

lemma node_lift_anycast [intro]:
  assumes pinv: "T ⊨A (otherwith S {i} (orecvmsg I), other U {i} →)
                    globala (λ(σ, a, σ'). anycast (Q σ σ') a)"
      and "⋀ξ ξ'. S ξ ξ' ⟹ U ξ ξ'"
    shows "⟨i : T : Ri⟩o ⊨A (otherwith S {i} (oarrivemsg I), other U {i} →)
                         globala (λ(σ, a, σ'). castmsg (Q σ σ') a)"
    (is "_ ⊨A (?S, ?U →) _")
  proof (rule ostep_invariantI, simp)
    fix σ s a σ' s'
    assume rs: "(σ, s) ∈ oreachable (⟨i : T : Ri⟩o) ?S ?U"
       and tr: "((σ, s), a, (σ', s')) ∈ trans (⟨i : T : Ri⟩o)"
       and "?S σ σ' a"
    from this(1-2) obtain ζ R
      where [simp]: "s = NodeS i ζ R"
        and "(σ, NodeS i ζ R) ∈ oreachable (⟨i : T : Ri⟩o) ?S ?U"
      by (metis node_net_state)
    from this(2) have "(σ, ζ) ∈ oreachable T (otherwith S {i} (orecvmsg I)) ?U"
      by (rule node_proc_reachable [OF _ assms(2)])
    moreover from tr have "((σ, NodeS i ζ R), a, (σ', s')) ∈ onode_sos (trans T)"
      by (simp add: onode_comps)
    ultimately show "castmsg (Q σ σ') a" using ‹?S σ σ' a›
      by - (erule onode_sos.cases, auto elim!: otherwithE dest!: ostep_invariantD [OF pinv])
  qed

lemma node_lift_anycast_statelessassm [intro]:
  assumes pinv: "T ⊨A (λσ _. orecvmsg I σ, other (λ_ _. True) {i} →)
                    globala (λ(σ, a, σ'). anycast (Q σ σ') a)"
    shows "⟨i : T : Ri⟩o ⊨A (λσ _. oarrivemsg I σ, other (λ_ _. True) {i} →)
                         globala (λ(σ, a, σ'). castmsg (Q σ σ') a)"
    (is "_ ⊨A (?S, _ →) _")
  proof -
    from assms(1)
      have "T ⊨A (otherwith (λ_ _. True) {i} (orecvmsg I), other (λ_ _. True) {i} →)
                globala (λ(σ, a, σ'). anycast (Q σ σ') a)"
        by rule auto
    hence "⟨i : T : Ri⟩o ⊨A (otherwith (λ_ _. True) {i} (oarrivemsg I), other (λ_ _. True) {i} →)
                         globala (λ(σ, a, σ'). castmsg (Q σ σ') a)"
      by (rule node_lift_anycast) simp_all
    thus ?thesis
      by rule auto
  qed

lemma node_local_deliver:
  "⟨i : ζi : Ri⟩o ⊨A (S, U →) globala (λ(_, a, _). ∀j. j≠i ⟶ (∀d. a ≠ j:deliver(d)))"
```

```
  proof (rule ostep_invariantI, simp)
    fix σ s a σ' s'
    assume 1: "(σ, s) ∈ oreachable (⟨i : ζ_i : R_i⟩_o) S U"
       and 2: "((σ, s), a, (σ', s')) ∈ trans (⟨i : ζ_i : R_i⟩_o)"
       and "S σ σ' a"
    moreover from 1 2 obtain ζ R ζ' R' where "s = NodeS i ζ R" and "s' = NodeS i ζ' R'" ..
    ultimately show "∀j. j≠i ⟶ (∀d. a ≠ j:deliver(d))"
      by (cases a) (auto simp add: onode_comps)
  qed

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 (rule ostep_invariantI, clarsimp simp only: globalasimp snd_conv fst_conv)
    fix σ s a σ' s' j
    assume 1: "(σ, s) ∈ oreachable (⟨i : ζ_i : R_i⟩_o) S U"
       and 2: "((σ, s), a, (σ', s')) ∈ trans (⟨i : ζ_i : R_i⟩_o)"
       and "S σ σ' a"
       and "a = τ ∨ (∃i d. a = i:deliver(d))"
       and "j ≠ i"
    moreover from 1 2 obtain ζ R ζ' R' where "s = NodeS i ζ R" and "s' = NodeS i ζ' R'" ..
    ultimately show "σ' j = σ j"
      by (cases a) (auto simp del: step_node_tau simp add: onode_comps)
  qed

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 (p₁ ∥ p₂)) S U"
      and init: "⋀σ s t. (σ, SubnetS s t) ∈ init (opnet onp (p₁ ∥ p₂)) ⟹ P σ s t"
      and other: "⋀σ s t σ'. ⟦ (σ, SubnetS s t) ∈ oreachable (opnet onp (p₁ ∥ p₂)) 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 (p₁ ∥ p₂)) S U;
                  ((σ, SubnetS s t), a, (σ', SubnetS s' t')) ∈ trans (opnet onp (p₁ ∥ p₂));
                  S σ σ' a; P σ s t ⟧ ⟹ P σ' s' t'"
    shows "P σ s t"
  using assms(1) proof (induction "(σ, SubnetS s t)" arbitrary: s t σ)
    fix s t σ
    assume "(σ, SubnetS s t) ∈ init (opnet onp (p₁ ∥ p₂))"
    with init show "P σ s t" .
  next
    fix st a s' t' σ'
    assume "st ∈ oreachable (opnet onp (p₁ ∥ p₂)) S U"
       and tr: "(st, a, (σ', SubnetS s' t')) ∈ trans (opnet onp (p₁ ∥ p₂))"
       and "S (fst st) (fst (σ', SubnetS s' t')) a"
       and IH: "⋀s t σ. st = (σ, SubnetS s t) ⟹ P σ s t"
    from this(1) obtain s t σ where "st = (σ, SubnetS s t)"
                              and "(σ, SubnetS s t) ∈ oreachable (opnet onp (p₁ ∥ p₂)) S U"
      by (metis net_par_oreachable_is_subnet prod.collapse)
    note this(2)
    moreover from tr and ‹st = (σ, SubnetS s t)›
      have "((σ, SubnetS s t), a, (σ', SubnetS s' t')) ∈ trans (opnet onp (p₁ ∥ p₂))" by simp
    moreover from ‹S (fst st) (fst (σ', SubnetS s' t')) a› and ‹st = (σ, SubnetS s t)›
      have "S σ σ' a" by simp
    moreover from IH and ‹st = (σ, SubnetS s t)› have "P σ s t" .
    ultimately show "P σ' s' t'" by (rule local)
  next
    fix st σ' s t
```

```
    assume "st ∈ oreachable (opnet onp (p₁ ∥ p₂)) S U"
       and "U (fst st) σ'"
       and "snd st = SubnetS s t"
       and IH: "⋀s t σ. st = (σ, SubnetS s t) ⟹ P σ s t"
    from this(1,3) obtain σ where "st = (σ, SubnetS s t)"
                                 and "(σ, SubnetS s t) ∈ oreachable (opnet onp (p₁ ∥ p₂)) S U"
      by (metis prod.collapse)
    note this(2)
    moreover from ‹U (fst st) σ'› and ‹st = (σ, SubnetS s t)› have "U σ σ'" by simp
    moreover from IH and ‹st = (σ, SubnetS s t)› have "P σ s t" .
    ultimately show "P σ' s t" by (rule other)
  qed

lemma other_net_tree_ips_par_left:
  assumes "other U (net_tree_ips (p₁ ∥ p₂)) σ σ'"
      and "⋀ξ. U ξ ξ"
    shows "other U (net_tree_ips p₁) σ σ'"
  proof -
    from assms(1) obtain ineq: "∀i∈net_tree_ips (p₁ ∥ p₂). σ' i = σ i"
                     and outU: "∀j. j∉net_tree_ips (p₁ ∥ p₂) ⟶ U (σ j) (σ' j)" ..
    show ?thesis
    proof (rule otherI)
      fix i
      assume "i∈net_tree_ips p₁"
      hence "i∈net_tree_ips (p₁ ∥ p₂)" by simp
      with ineq show "σ' i = σ i" ..
    next
      fix j
      assume "j∉net_tree_ips p₁"
      show "U (σ j) (σ' j)"
      proof (cases "j∈net_tree_ips p₂")
        assume "j∈net_tree_ips p₂"
        hence "j∈net_tree_ips (p₁ ∥ p₂)" by simp
        with ineq have "σ' j = σ j" ..
        thus "U (σ j) (σ' j)"
          by simp (rule ‹⋀ξ. U ξ ξ›)
      next
        assume "j∉net_tree_ips p₂"
        with ‹j∉net_tree_ips p₁› have "j∉net_tree_ips (p₁ ∥ p₂)" by simp
        with outU show "U (σ j) (σ' j)" by simp
      qed
    qed
  qed

lemma other_net_tree_ips_par_right:
  assumes "other U (net_tree_ips (p₁ ∥ p₂)) σ σ'"
      and "⋀ξ. U ξ ξ"
    shows "other U (net_tree_ips p₂) σ σ'"
  proof -
    from assms(1) have "other U (net_tree_ips (p₂ ∥ p₁)) σ σ'"
      by (subst net_tree_ips_commute)
    thus ?thesis using ‹⋀ξ. U ξ ξ›
      by (rule other_net_tree_ips_par_left)
  qed

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 -
    from assms(2) have "(σ, s) ∈ oreachable p (λσ _ a. oarrivemsg I σ a) U"
      by (rule oreachable_weakenE) auto
```

```
    thus "P ((σ, s), a, (σ', s'))"
      using assms(3-4) by (rule ostep_invariantD [OF assms(1)])
  qed

lemma opnet_sync_action_subnet_oreachable:
  assumes "(σ, SubnetS s t) ∈ oreachable (opnet onp (p1 ∥ p2))
                                  (λσ _. oarrivemsg I σ) (other U (net_tree_ips (p1 ∥ p2)))"
          (is "_ ∈ oreachable _ (?S (p1 ∥ p2)) (?U (p1 ∥ p2))")

      and "⋀ξ. U ξ ξ"

      and act1: "opnet onp p1 ⊨A (λσ _. oarrivemsg I σ, other U (net_tree_ips p1) →)
                 globala (λ(σ, a, σ'). castmsg (I σ) a
                                  ∧ (a = τ ∨ (∃i d. a = i:deliver(d)) ⟶
                                        ((∀i∈net_tree_ips p1. U (σ i) (σ' i))
                                      ∧ (∀i. i∉net_tree_ips p1 ⟶ σ' i = σ i))))"

      and act2: "opnet onp p2 ⊨A (λσ _. oarrivemsg I σ, other U (net_tree_ips p2) →)
                 globala (λ(σ, a, σ'). castmsg (I σ) a
                                  ∧ (a = τ ∨ (∃i d. a = i:deliver(d)) ⟶
                                        ((∀i∈net_tree_ips p2. U (σ i) (σ' i))
                                      ∧ (∀i. i∉net_tree_ips p2 ⟶ σ' i = σ i))))"

    shows "(σ, s) ∈ oreachable (opnet onp p1) (λσ _. oarrivemsg I σ) (other U (net_tree_ips p1))
          ∧ (σ, t) ∈ oreachable (opnet onp p2) (λσ _. oarrivemsg I σ) (other U (net_tree_ips p2))
          ∧ net_tree_ips p1 ∩ net_tree_ips p2 = {}"
  using assms(1)
  proof (induction rule: oreachable_par_subnet_induct)
    case (init σ s t)
    hence sinit: "(σ, s) ∈ init (opnet onp p1)"
      and tinit: "(σ, t) ∈ init (opnet onp p2)"
      and "net_ips s ∩ net_ips t = {}" by auto
    moreover from sinit have "net_ips s = net_tree_ips p1"
      by (rule opnet_net_ips_net_tree_ips_init)
    moreover from tinit have "net_ips t = net_tree_ips p2"
      by (rule opnet_net_ips_net_tree_ips_init)
    ultimately show ?case by (auto elim: oreachable_init)
  next
    case (other σ s t σ')
    hence "other U (net_tree_ips (p1 ∥ p2)) σ σ'"
      and IHs: "(σ, s) ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
      and IHt: "(σ, t) ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
      and "net_tree_ips p1 ∩ net_tree_ips p2 = {}" by auto

    have "(σ', s) ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
    proof -
      from ‹?U (p1 ∥ p2) σ σ'› and ‹⋀ξ. U ξ ξ› have "?U p1 σ σ'"
        by (rule other_net_tree_ips_par_left)
      with IHs show ?thesis by - (erule(1) oreachable_other')
    qed

    moreover have "(σ', t) ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
    proof -
      from ‹?U (p1 ∥ p2) σ σ'› and ‹⋀ξ. U ξ ξ› have "?U p2 σ σ'"
        by (rule other_net_tree_ips_par_right)
      with IHt show ?thesis by - (erule(1) oreachable_other')
    qed

    ultimately show ?case using ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}› by simp
  next
    case (local σ s t σ' s' t' a)
    hence stor: "(σ, SubnetS s t) ∈ oreachable (opnet onp (p1 ∥ p2)) (?S (p1 ∥ p2)) (?U (p1 ∥ p2))"
      and tr: "((σ, SubnetS s t), a, (σ', SubnetS s' t')) ∈ trans (opnet onp (p1 ∥ p2))"
      and "oarrivemsg I σ a"
```

```
  and sor: "(σ, s) ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
  and tor: "(σ, t) ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
  and "net_tree_ips p₁ ∩ net_tree_ips p₂ = {}" by auto
from tr have "((σ, SubnetS s t), a, (σ', SubnetS s' t'))
               ∈ opnet_sos (trans (opnet onp p₁)) (trans (opnet onp p₂))" by simp
hence "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)
     ∧ (σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
proof (cases)
  fix H K m H' K'
  assume "a = (H ∪ H')¬(K ∪ K'):arrive(m)"
     and str: "((σ, s), H¬K:arrive(m), (σ', s')) ∈ trans (opnet onp p₁)"
     and ttr: "((σ, t), H'¬K':arrive(m), (σ', t')) ∈ trans (opnet onp p₂)"
  from this(1) and ‹oarrivemsg I σ a› have "I σ m" by simp

  with sor str
    have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
      by - (erule(1) oreachable_local, auto)
  moreover from ‹I σ m› tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix R m H K
  assume str: "((σ, s), R:*cast(m), (σ', s')) ∈ trans (opnet onp p₁)"
     and ttr: "((σ, t), H¬K:arrive(m), (σ', t')) ∈ trans (opnet onp p₂)"
  from sor str have "I σ m"
    by - (drule(1) ostep_invariantD [OF act1], simp_all)
  with sor str
    have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
      by - (erule(1) oreachable_local, auto)
  moreover from ‹I σ m› tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix R m H K
  assume str: "((σ, s), H¬K:arrive(m), (σ', s')) ∈ trans (opnet onp p₁)"
     and ttr: "((σ, t), R:*cast(m), (σ', t')) ∈ trans (opnet onp p₂)"
  from tor ttr have "I σ m"
    by - (drule(1) ostep_invariantD [OF act2], simp_all)
  with sor str
    have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
      by - (erule(1) oreachable_local, auto)
  moreover from ‹I σ m› tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix i i'
  assume str: "((σ, s), connect(i, i'), (σ', s')) ∈ trans (opnet onp p₁)"
     and ttr: "((σ, t), connect(i, i'), (σ', t')) ∈ trans (opnet onp p₂)"
  with sor str
    have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
      by - (erule(1) oreachable_local, auto)
  moreover from tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix i i'
  assume str: "((σ, s), disconnect(i, i'), (σ', s')) ∈ trans (opnet onp p₁)"
     and ttr: "((σ, t), disconnect(i, i'), (σ', t')) ∈ trans (opnet onp p₂)"
  with sor str
    have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
```

```
      by - (erule(1) oreachable_local, auto)
  moreover from tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix i d
  assume "t' = t"
     and str: "((σ, s), i:deliver(d), (σ', s')) ∈ trans (opnet onp p₁)"

  from sor str have "∀j. j∉net_tree_ips p₁ ⟶ σ' j = σ j"
    by - (drule(1) ostep_invariantD [OF act1], simp_all)
  moreover with ‹net_tree_ips p₁ ∩ net_tree_ips p₂ = {}›
    have "∀j. j∈net_tree_ips p₂ ⟶ σ' j = σ j" by auto
  moreover from sor str have "∀j∈net_tree_ips p₁. U (σ j) (σ' j)"
    by - (drule(1) ostep_invariantD [OF act1], simp_all)
  ultimately have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
    using tor ‹t' = t› by (clarsimp elim!: oreachable_other')
                          (metis otherI ‹⋀ξ. U ξ ξ›)+

  moreover from sor str
    have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis by (rule conjI [rotated])
next
  fix i d
  assume "s' = s"
     and ttr: "((σ, t), i:deliver(d), (σ', t')) ∈ trans (opnet onp p₂)"

  from tor ttr have "∀j. j∉net_tree_ips p₂ ⟶ σ' j = σ j"
    by - (drule(1) ostep_invariantD [OF act2], simp_all)
  moreover with ‹net_tree_ips p₁ ∩ net_tree_ips p₂ = {}›
    have "∀j. j∈net_tree_ips p₁ ⟶ σ' j = σ j" by auto
  moreover from tor ttr have "∀j∈net_tree_ips p₂. U (σ j) (σ' j)"
    by - (drule(1) ostep_invariantD [OF act2], simp_all)
  ultimately have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
    using sor ‹s' = s› by (clarsimp elim!: oreachable_other')
                          (metis otherI ‹⋀ξ. U ξ ξ›)+

  moreover from tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  assume "t' = t"
     and str: "((σ, s), τ, (σ', s')) ∈ trans (opnet onp p₁)"

  from sor str have "∀j. j∉net_tree_ips p₁ ⟶ σ' j = σ j"
    by - (drule(1) ostep_invariantD [OF act1], simp_all)
  moreover with ‹net_tree_ips p₁ ∩ net_tree_ips p₂ = {}›
    have "∀j. j∈net_tree_ips p₂ ⟶ σ' j = σ j" by auto
  moreover from sor str have "∀j∈net_tree_ips p₁. U (σ j) (σ' j)"
    by - (drule(1) ostep_invariantD [OF act1], simp_all)
  ultimately have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
    using tor ‹t' = t› by (clarsimp elim!: oreachable_other')
                          (metis otherI ‹⋀ξ. U ξ ξ›)+

  moreover from sor str
    have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis by (rule conjI [rotated])
next
  assume "s' = s"
     and ttr: "((σ, t), τ, (σ', t')) ∈ trans (opnet onp p₂)"
```

```
    from tor ttr have "∀j. j∉net_tree_ips p₂ ⟶ σ' j = σ j"
      by - (drule(1) ostep_invariantD [OF act2], simp_all)
    moreover with ‹net_tree_ips p₁ ∩ net_tree_ips p₂ = {}›
      have "∀j. j∈net_tree_ips p₁ ⟶ σ' j = σ j" by auto
    moreover from tor ttr have "∀j∈net_tree_ips p₂. U (σ j) (σ' j)"
      by - (drule(1) ostep_invariantD [OF act2], simp_all)
    ultimately have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
      using sor ‹s' = s› by (clarsimp elim!: oreachable_other')
                          (metis otherI ‹⋀ξ. U ξ ξ›)+

    moreover from tor ttr
      have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
        by - (erule(1) oreachable_local, auto)
    ultimately show ?thesis ..
  qed
  with ‹net_tree_ips p₁ ∩ net_tree_ips p₂ = {}› show ?case by simp
qed
```

'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 "(σ, SubnetS s₁ s₂) ∈ oreachable (opnet onp (p₁ ∥ p₂)) (λ_ _ _. True) (λ_ _. True)"
    shows "(σ, s₁) ∈ oreachable (opnet onp p₁) (λ_ _ _. True) (λ_ _. True)"
          "(σ, s₂) ∈ oreachable (opnet onp p₂) (λ_ _ _. True) (λ_ _. True)"
          (is "_ ∈ ?oreachable p₂")
  using assms proof -
    from assms have "(σ, s₁) ∈ ?oreachable p₁ ∧ (σ, s₂) ∈ ?oreachable p₂"
    proof (induction rule: oreachable_par_subnet_induct)
      fix σ s₁ s₂
      assume "(σ, SubnetS s₁ s₂) ∈ init (opnet onp (p₁ ∥ p₂))"
      thus "(σ, s₁) ∈ ?oreachable p₁ ∧ (σ, s₂) ∈ ?oreachable p₂"
        by (auto dest: oreachable_init)
    next
      case (local σ s₁ s₂ σ' s₁' s₂' a)
      hence "(σ, SubnetS s₁ s₂) ∈ ?oreachable (p₁ ∥ p₂)"
        and sr1: "(σ, s₁) ∈ ?oreachable p₁"
        and sr2: "(σ, s₂) ∈ ?oreachable p₂"
        and "((σ, SubnetS s₁ s₂), a, (σ', SubnetS s₁' s₂')) ∈ trans (opnet onp (p₁ ∥ p₂))" by auto
      from this(4)
        have "((σ, SubnetS s₁ s₂), a, (σ', SubnetS s₁' s₂'))
                ∈ opnet_sos (trans (opnet onp p₁)) (trans (opnet onp p₂))" by simp
      thus "(σ', s₁') ∈ ?oreachable p₁ ∧ (σ', s₂') ∈ ?oreachable p₂"
      proof cases
        fix R m H K
        assume "a = R:*cast(m)"
           and tr1: "((σ, s₁), R:*cast(m), (σ', s₁')) ∈ trans (opnet onp p₁)"
           and tr2: "((σ, s₂), H¬K:arrive(m), (σ', s₂')) ∈ trans (opnet onp p₂)"
        from sr1 and tr1 and TrueI have "(σ', s₁') ∈ ?oreachable p₁"
          by (rule oreachable_local')
        moreover from sr2 and tr2 and TrueI have "(σ', s₂') ∈ ?oreachable p₂"
          by (rule oreachable_local')
        ultimately show ?thesis ..
      next
        assume "a = τ"
           and "s₂' = s₂"
           and tr1: "((σ, s₁), τ, (σ', s₁')) ∈ trans (opnet onp p₁)"
        from sr2 and this(2) have "(σ', s₂') ∈ ?oreachable p₂" by auto
        moreover have "(λ_ _. True) σ σ'" by (rule TrueI)
        ultimately have "(σ', s₂') ∈ ?oreachable p₂"
          by (rule oreachable_other')
        moreover from sr1 and tr1 and TrueI have "(σ', s₁') ∈ ?oreachable p₁"
          by (rule oreachable_local')
      qed (insert sr1 sr2, simp_all, (metis (no_types) oreachable_local'
```

```
                                                    oreachable_other')+)
    qed auto
    thus "(σ, s₁) ∈ ?oreachable p₁"
         "(σ, s₂) ∈ ?oreachable p₂" by auto
  qed
```

It may also be tempting to try splitting from the assumption *(σ, SubnetS s_1 s_2) ∈ oreachable (opnet onp (p_1 ∥ p_2)) (λ_ _ _. True) (λ_ _. 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 "⋀ξ. U ξ ξ"
      and act1: "⋀i R. ⟨i : onp i : R⟩ₒ ⊨A (λσ _. oarrivemsg I σ, other U {i} →)
                      globala (λ(σ, a, _). castmsg (I σ) a)"
      and act2: "⋀i R. ⟨i : onp i : R⟩ₒ ⊨A (λσ _. oarrivemsg I σ, other U {i} →)
                      globala (λ(σ, a, σ'). (a ≠ τ ∧ (∀d. a ≠ i:deliver(d)) ⟶ S (σ i) (σ' i)))"
      and act3: "⋀i R. ⟨i : onp i : R⟩ₒ ⊨A (λσ _. oarrivemsg 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 ⊨A (λσ _. oarrivemsg I σ, other U (net_tree_ips p) →)
                       globala (λ(σ, a, σ'). castmsg (I σ) a
                                         ∧ (a ≠ τ ∧ (∀i d. a ≠ i:deliver(d)) ⟶
                                                (∀i∈net_tree_ips p. S (σ i) (σ' i)))
                                         ∧ (a = τ ∨ (∃i d. a = i:deliver(d)) ⟶
                                                ((∀i∈net_tree_ips p. U (σ i) (σ' i))
                                              ∧ (∀i. i∉net_tree_ips p ⟶ σ' i = σ i))))"
    (is "opnet onp p ⊨A (?I, ?U p →) ?inv (net_tree_ips p)")
  proof (induction p)
    fix i R
    show "opnet onp ⟨i; R⟩ ⊨A (?I, ?U ⟨i; R⟩ →) ?inv (net_tree_ips ⟨i; R⟩)"
    proof (rule ostep_invariantI, simp only: opnet.simps net_tree_ips.simps)
      fix σ s a σ' s'
      assume sor: "(σ, s) ∈ oreachable (⟨i : onp i : R⟩ₒ) (λσ _. oarrivemsg I σ) (other U {i})"
         and str: "((σ, s), a, (σ', s')) ∈ trans (⟨i : onp i : R⟩ₒ)"
         and oam: "oarrivemsg I σ a"
      hence "castmsg (I σ) a"
        by - (drule(2) ostep_invariantD [OF act1], simp)
      moreover from sor str oam have "a ≠ τ ∧ (∀i d. a ≠ i:deliver(d)) ⟶ S (σ i) (σ' i)"
        by - (drule(2) ostep_invariantD [OF act2], simp)
      moreover have "a = τ ∨ (∃i d. a = i:deliver(d)) ⟶ U (σ i) (σ' i)"
      proof -
        from sor str oam have "a = τ ∨ (∃d. a = i:deliver(d)) ⟶ U (σ i) (σ' i)"
          by - (drule(2) ostep_invariantD [OF act3], simp)
        moreover from sor str oam have "∀j. j≠i ⟶ (∀d. a ≠ j:deliver(d))"
          by - (drule(2) ostep_invariantD [OF node_local_deliver], simp)
        ultimately show ?thesis
          by clarsimp metis
      qed
      moreover from sor str oam have "∀j. j≠i ⟶ (∀d. a ≠ j:deliver(d))"
        by - (drule(2) ostep_invariantD [OF node_local_deliver], simp)
      moreover from sor str oam have "a = τ ∨ (∃i d. a = i:deliver(d)) ⟶ (∀j. j≠i ⟶ σ' j = σ j)"
        by - (drule(2) ostep_invariantD [OF node_tau_deliver_unchanged], simp)
      ultimately show "?inv {i} ((σ, s), a, (σ', s'))" by simp
    qed
  next
    fix p₁ p₂
    assume inv1: "opnet onp p₁ ⊨A (?I, ?U p₁ →) ?inv (net_tree_ips p₁)"
       and inv2: "opnet onp p₂ ⊨A (?I, ?U p₂ →) ?inv (net_tree_ips p₂)"
    show "opnet onp (p₁ ∥ p₂) ⊨A (?I, ?U (p₁ ∥ p₂) →) ?inv (net_tree_ips (p₁ ∥ p₂))"
    proof (rule ostep_invariantI)
      fix σ st a σ' st'
      assume "(σ, st) ∈ oreachable (opnet onp (p₁ ∥ p₂)) ?I (?U (p₁ ∥ p₂))"
         and "((σ, st), a, (σ', st')) ∈ trans (opnet onp (p₁ ∥ p₂))"
         and "oarrivemsg I σ a"
      from this(1) obtain s t
```

```
    where "st = SubnetS s t"
      and *: "(σ, SubnetS s t) ∈ oreachable (opnet onp (p₁ ∥ p₂)) ?I (?U (p₁ ∥ p₂))"
    by - (frule net_par_oreachable_is_subnet, metis)

  from this(2) and inv1 and inv2
    obtain sor: "(σ, s) ∈ oreachable (opnet onp p₁) ?I (?U p₁)"
       and tor: "(σ, t) ∈ oreachable (opnet onp p₂) ?I (?U p₂)"
       and "net_tree_ips p₁ ∩ net_tree_ips p₂ = {}"
      by - (drule opnet_sync_action_subnet_oreachable [OF _ ‹⋀ξ. U ξ ξ›], auto)

  from * and ‹((σ, st), a, (σ', st')) ∈ trans (opnet onp (p₁ ∥ p₂))› and ‹st = SubnetS s t›
    obtain s' t' where "st' = SubnetS s' t'"
                 and "((σ, SubnetS s t), a, (σ', SubnetS s' t'))
                        ∈ opnet_sos (trans (opnet onp p₁)) (trans (opnet onp p₂))"
      by clarsimp (frule opartial_net_preserves_subnets, metis)

  from this(2)
    have"castmsg (I σ) a
         ∧ (a ≠ τ ∧ (∀i d. a ≠ i:deliver(d)) ⟶ (∀i∈net_tree_ips (p₁ ∥ p₂). S (σ i) (σ' i)))
         ∧ (a = τ ∨ (∃i d. a = i:deliver(d)) ⟶ (∀i∈net_tree_ips (p₁ ∥ p₂). U (σ i) (σ' i))
                                                ∧ (∀i. i ∉ net_tree_ips (p₁ ∥ p₂) ⟶ σ' i = σ i))"
  proof cases
    fix R m H K
    assume "a = R:*cast(m)"
       and str: "((σ, s), R:*cast(m), (σ', s')) ∈ trans (opnet onp p₁)"
       and ttr: "((σ, t), H¬K:arrive(m), (σ', t')) ∈ trans (opnet onp p₂)"
    from sor and str have "I σ m ∧ (∀i∈net_tree_ips p₁. S (σ i) (σ' i))"
      by (auto dest: ostep_invariantD [OF inv1])
    moreover with tor and ttr have "∀i∈net_tree_ips p₂. S (σ i) (σ' i)"
      by (auto dest: ostep_invariantD [OF inv2])
    ultimately show ?thesis
      using ‹a = R:*cast(m)› by auto
  next
    fix R m H K
    assume "a = R:*cast(m)"
       and str: "((σ, s), H¬K:arrive(m), (σ', s')) ∈ trans (opnet onp p₁)"
       and ttr: "((σ, t), R:*cast(m), (σ', t')) ∈ trans (opnet onp p₂)"
    from tor and ttr have "I σ m ∧ (∀i∈net_tree_ips p₂. S (σ i) (σ' i))"
      by (auto dest: ostep_invariantD [OF inv2])
    moreover with sor and str have "∀i∈net_tree_ips p₁. S (σ i) (σ' i)"
      by (auto dest: ostep_invariantD [OF inv1])
    ultimately show ?thesis
      using ‹a = R:*cast(m)› by auto
  next
    fix H K m H' K'
    assume "a = (H ∪ H')¬(K ∪ K'):arrive(m)"
       and str: "((σ, s), H¬K:arrive(m), (σ', s')) ∈ trans (opnet onp p₁)"
       and ttr: "((σ, t), H'¬K':arrive(m), (σ', t')) ∈ trans (opnet onp p₂)"
    from this(1) and ‹oarrivemsg I σ a› have "I σ m" by simp
    with sor and str have "∀i∈net_tree_ips p₁. S (σ i) (σ' i)"
      by (auto dest: ostep_invariantD [OF inv1])
    moreover from tor and ttr and ‹I σ m› have "∀i∈net_tree_ips p₂. S (σ i) (σ' i)"
      by (auto dest: ostep_invariantD [OF inv2])
    ultimately show ?thesis
      using ‹a = (H ∪ H')¬(K ∪ K'):arrive(m)› by auto
  next
    fix i d
    assume "a = i:deliver(d)"
       and str: "((σ, s), i:deliver(d), (σ', s')) ∈ trans (opnet onp p₁)"
    with sor have "((∀i∈net_tree_ips p₁. U (σ i) (σ' i))
                 ∧ (∀i. i∉net_tree_ips p₁ ⟶ σ' i = σ i))"
      by (auto dest!: ostep_invariantD [OF inv1])
    with ‹a = i:deliver(d)› and ‹⋀ξ. U ξ ξ› show ?thesis
      by auto
```

```
    next
      fix i d
      assume "a = i:deliver(d)"
         and ttr: "((σ, t), i:deliver(d), (σ', t')) ∈ trans (opnet onp p₂)"
      with tor have "((∀i∈net_tree_ips p₂. U (σ i) (σ' i))
                    ∧ (∀i. i∉net_tree_ips p₂ ⟶ σ' i = σ i))"
        by (auto dest!: ostep_invariantD [OF inv2])
      with ‹a = i:deliver(d)› and ‹⋀ξ. U ξ ξ› show ?thesis
        by auto
    next
      assume "a = τ"
         and str: "((σ, s), τ, (σ', s')) ∈ trans (opnet onp p₁)"
      with sor have "((∀i∈net_tree_ips p₁. U (σ i) (σ' i))
                    ∧ (∀i. i∉net_tree_ips p₁ ⟶ σ' i = σ i))"
        by (auto dest!: ostep_invariantD [OF inv1])
      with ‹a = τ› and ‹⋀ξ. U ξ ξ› show ?thesis
        by auto
    next
      assume "a = τ"
         and ttr: "((σ, t), τ, (σ', t')) ∈ trans (opnet onp p₂)"
      with tor have "((∀i∈net_tree_ips p₂. U (σ i) (σ' i))
                    ∧ (∀i. i∉net_tree_ips p₂ ⟶ σ' i = σ i))"
        by (auto dest!: ostep_invariantD [OF inv2])
      with ‹a = τ› and ‹⋀ξ. U ξ ξ› show ?thesis
        by auto
    next
      fix i i'
      assume "a = connect(i, i')"
         and str: "((σ, s), connect(i, i'), (σ', s')) ∈ trans (opnet onp p₁)"
         and ttr: "((σ, t), connect(i, i'), (σ', t')) ∈ trans (opnet onp p₂)"
      from sor and str have "∀i∈net_tree_ips p₁. S (σ i) (σ' i)"
        by (auto dest: ostep_invariantD [OF inv1])
      moreover from tor and ttr have "∀i∈net_tree_ips p₂. S (σ i) (σ' i)"
        by (auto dest: ostep_invariantD [OF inv2])
      ultimately show ?thesis
        using ‹a = connect(i, i')› by auto
    next
      fix i i'
      assume "a = disconnect(i, i')"
         and str: "((σ, s), disconnect(i, i'), (σ', s')) ∈ trans (opnet onp p₁)"
         and ttr: "((σ, t), disconnect(i, i'), (σ', t')) ∈ trans (opnet onp p₂)"
      from sor and str have "∀i∈net_tree_ips p₁. S (σ i) (σ' i)"
        by (auto dest: ostep_invariantD [OF inv1])
      moreover from tor and ttr have "∀i∈net_tree_ips p₂. S (σ i) (σ' i)"
        by (auto dest: ostep_invariantD [OF inv2])
      ultimately show ?thesis
        using ‹a = disconnect(i, i')› by auto
    qed
    thus "?inv (net_tree_ips (p₁ ∥ p₂)) ((σ, st), a, (σ', st'))" by simp
  qed
qed

theorem subnet_oreachable:
  assumes "(σ, SubnetS s t) ∈ oreachable (opnet onp (p₁ ∥ p₂))
                                (otherwith S (net_tree_ips (p₁ ∥ p₂)) (oarrivemsg I))
                                (other U (net_tree_ips (p₁ ∥ p₂)))"
        (is "_ ∈ oreachable _ (?S (p₁ ∥ p₂)) (?U (p₁ ∥ p₂))")

    and "⋀ξ. S ξ ξ"
    and "⋀ξ. U ξ ξ"

    and node1: "⋀i R. ⟨i : onp i : R⟩ₒ ⊨_A (λσ _. oarrivemsg I σ, other U {i} →)
                        globala (λ(σ, a, _). castmsg (I σ) a)"
    and node2: "⋀i R. ⟨i : onp i : R⟩ₒ ⊨_A (λσ _. oarrivemsg 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 _. oarrivemsg I σ, other U {i} →)
                  globala (λ(σ, a, σ'). (a = τ ∨ (∃d. a = i:deliver(d)) ⟶ U (σ i) (σ' i)))"

  shows "(σ, s) ∈ oreachable (opnet onp p1)
                         (otherwith S (net_tree_ips p1) (oarrivemsg I))
                         (other U (net_tree_ips p1))
       ∧ (σ, t) ∈ oreachable (opnet onp p2)
                           (otherwith S (net_tree_ips p2) (oarrivemsg I))
                           (other U (net_tree_ips p2))
       ∧ net_tree_ips p1 ∩ net_tree_ips p2 = {}"
using assms(1) proof (induction rule: oreachable_par_subnet_induct)
  case (init σ s t)
  hence sinit: "(σ, s) ∈ init (opnet onp p1)"
    and tinit: "(σ, t) ∈ init (opnet onp p2)"
    and "net_ips s ∩ net_ips t = {}" by auto
  moreover from sinit have "net_ips s = net_tree_ips p1"
    by (rule opnet_net_ips_net_tree_ips_init)
  moreover from tinit have "net_ips t = net_tree_ips p2"
    by (rule opnet_net_ips_net_tree_ips_init)
  ultimately show ?case by (auto elim: oreachable_init)
next
  case (other σ s t σ')
  hence "other U (net_tree_ips (p1 ∥ p2)) σ σ'"
    and IHs: "(σ, s) ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
    and IHt: "(σ, t) ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
    and "net_tree_ips p1 ∩ net_tree_ips p2 = {}" by auto

  have "(σ', s) ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
  proof -
    from ‹?U (p1 ∥ p2) σ σ'› and ‹⋀ξ. U ξ ξ› have "?U p1 σ σ'"
      by (rule other_net_tree_ips_par_left)
    with IHs show ?thesis by - (erule(1) oreachable_other')
  qed

  moreover have "(σ', t) ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
  proof -
    from ‹?U (p1 ∥ p2) σ σ'› and ‹⋀ξ. U ξ ξ› have "?U p2 σ σ'"
      by (rule other_net_tree_ips_par_right)
    with IHt show ?thesis by - (erule(1) oreachable_other')
  qed

  ultimately show ?case using ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}› by simp
next
  case (local σ s t σ' s' t' a)
  hence stor: "(σ, SubnetS s t) ∈ oreachable (opnet onp (p1 ∥ p2)) (?S (p1 ∥ p2)) (?U (p1 ∥ p2))"
    and tr: "((σ, SubnetS s t), a, (σ', SubnetS s' t')) ∈ trans (opnet onp (p1 ∥ p2))"
    and "?S (p1 ∥ p2) σ σ' a"
    and sor: "(σ, s) ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
    and tor: "(σ, t) ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
    and "net_tree_ips p1 ∩ net_tree_ips p2 = {}" by auto

  have act: "⋀p. opnet onp p ⊨A (λσ _. oarrivemsg I σ, other U (net_tree_ips p) →)
               globala (λ(σ, a, σ'). castmsg (I σ) a
                                     ∧ (a ≠ τ ∧ (∀i d. a ≠ i:deliver(d)) ⟶
                                            (∀i∈net_tree_ips p. S (σ i) (σ' i)))
                                     ∧ (a = τ ∨ (∃i d. a = i:deliver(d)) ⟶
                                            ((∀i∈net_tree_ips p. U (σ i) (σ' i))
                                          ∧ (∀i. i∉net_tree_ips p ⟶ σ' i = σ i))))"
    by (rule lift_opnet_sync_action [OF assms(3-6)])

  from ‹?S (p1 ∥ p2) σ σ' a› have "∀j. j ∉ net_tree_ips (p1 ∥ p2) ⟶ S (σ j) (σ' j)"
                              and "oarrivemsg I σ a"
    by (auto elim!: otherwithE)
```

```
from tr have "((σ, SubnetS s t), a, (σ', SubnetS s' t'))
               ∈ opnet_sos (trans (opnet onp p1)) (trans (opnet onp p2))" by simp
hence "(σ', s') ∈ oreachable (opnet onp p1) (?S p1) (?U p1)
     ∧ (σ', t') ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
proof (cases)
  fix H K m H' K'
  assume "a = (H ∪ H')¬(K ∪ K'):arrive(m)"
     and str: "((σ, s), H¬K:arrive(m), (σ', s')) ∈ trans (opnet onp p1)"
     and ttr: "((σ, t), H'¬K':arrive(m), (σ', t')) ∈ trans (opnet onp p2)"
  from this(1) and ‹?S (p1 ∥ p2) σ σ' a› have "I σ m" by auto

  with sor str have "∀i∈net_tree_ips p1. S (σ i) (σ' i)"
    by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
  moreover from ‹I σ m› tor ttr have "∀i∈net_tree_ips p2. S (σ i) (σ' i)"
    by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
  ultimately have "∀i. S (σ i) (σ' i)"
    using ‹∀j. j ∉ net_tree_ips (p1 ∥ p2) ⟶ S (σ j) (σ' j)› by auto

  with ‹I σ m› sor str
    have "(σ', s') ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
      by - (erule(1) oreachable_local, auto)
  moreover from ‹∀i. S (σ i) (σ' i)› ‹I σ m› tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix R m H K
  assume str: "((σ, s), R:*cast(m), (σ', s')) ∈ trans (opnet onp p1)"
     and ttr: "((σ, t), H¬K:arrive(m), (σ', t')) ∈ trans (opnet onp p2)"
  from sor str have "I σ m"
    by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
  with sor str tor ttr have "∀i. S (σ i) (σ' i)"
    using ‹∀j. j ∉ net_tree_ips (p1 ∥ p2) ⟶ S (σ j) (σ' j)›
    by (fastforce dest!: ostep_arrive_invariantD [OF act] ostep_arrive_invariantD [OF act])
  with ‹I σ m› sor str
    have "(σ', s') ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
      by - (erule(1) oreachable_local, auto)
  moreover from ‹∀i. S (σ i) (σ' i)› ‹I σ m› tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix R m H K
  assume str: "((σ, s), H¬K:arrive(m), (σ', s')) ∈ trans (opnet onp p1)"
     and ttr: "((σ, t), R:*cast(m), (σ', t')) ∈ trans (opnet onp p2)"
  from tor ttr have "I σ m"
    by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
  with sor str tor ttr have "∀i. S (σ i) (σ' i)"
    using ‹∀j. j ∉ net_tree_ips (p1 ∥ p2) ⟶ S (σ j) (σ' j)›
    by (fastforce dest!: ostep_arrive_invariantD [OF act] ostep_arrive_invariantD [OF act])
  with ‹I σ m› sor str
    have "(σ', s') ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
      by - (erule(1) oreachable_local, auto)
  moreover from ‹∀i. S (σ i) (σ' i)› ‹I σ m› tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix i i'
  assume str: "((σ, s), connect(i, i'), (σ', s')) ∈ trans (opnet onp p1)"
     and ttr: "((σ, t), connect(i, i'), (σ', t')) ∈ trans (opnet onp p2)"
  with sor tor have "∀i. S (σ i) (σ' i)"
    using ‹∀j. j ∉ net_tree_ips (p1 ∥ p2) ⟶ S (σ j) (σ' j)›
    by (fastforce dest!: ostep_arrive_invariantD [OF act] ostep_arrive_invariantD [OF act])
```

```
  with sor str
    have "(σ', s') ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
      by - (erule(1) oreachable_local, auto)
  moreover from ‹∀i. S (σ i) (σ' i)› tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix i i'
  assume str: "((σ, s), disconnect(i, i'), (σ', s')) ∈ trans (opnet onp p1)"
     and ttr: "((σ, t), disconnect(i, i'), (σ', t')) ∈ trans (opnet onp p2)"
  with sor tor have "∀i. S (σ i) (σ' i)"
    using ‹∀j. j ∉ net_tree_ips (p1 ∥ p2) ⟶ S (σ j) (σ' j)›
    by (fastforce dest!: ostep_arrive_invariantD [OF act] ostep_arrive_invariantD [OF act])
  with sor str
    have "(σ', s') ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
      by - (erule(1) oreachable_local, auto)
  moreover from ‹∀i. S (σ i) (σ' i)› tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
      by - (erule(1) oreachable_local, auto)
  ultimately show ?thesis ..
next
  fix i d
  assume "t' = t"
     and str: "((σ, s), i:deliver(d), (σ', s')) ∈ trans (opnet onp p1)"
  from sor str have "∀j. j∉net_tree_ips p1 ⟶ σ' j = σ j"
    by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
  hence "∀j. j∉net_tree_ips p1 ⟶ S (σ j) (σ' j)"
     by (auto intro: ‹⋀ξ. S ξ ξ›)
  with sor str
    have "(σ', s') ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
      by - (erule(1) oreachable_local, auto)

  moreover have "(σ', t') ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
  proof -
    from ‹∀j. j∉net_tree_ips p1 ⟶ σ' j = σ j› and ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}›
      have "∀j. j∈net_tree_ips p2 ⟶ σ' j = σ j" by auto
    moreover from sor str have "∀j∈net_tree_ips p1. U (σ j) (σ' j)"
      by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
    ultimately show ?thesis
      using tor ‹t' = t› ‹∀j. j ∉ net_tree_ips p1 ⟶ σ' j = σ j›
      by (clarsimp elim!: oreachable_other')
         (metis otherI ‹⋀ξ. U ξ ξ›)+
  qed
  ultimately show ?thesis ..
next
  fix i d
  assume "s' = s"
     and ttr: "((σ, t), i:deliver(d), (σ', t')) ∈ trans (opnet onp p2)"
  from tor ttr have "∀j. j∉net_tree_ips p2 ⟶ σ' j = σ j"
    by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
  hence "∀j. j∉net_tree_ips p2 ⟶ S (σ j) (σ' j)"
     by (auto intro: ‹⋀ξ. S ξ ξ›)
  with tor ttr
    have "(σ', t') ∈ oreachable (opnet onp p2) (?S p2) (?U p2)"
      by - (erule(1) oreachable_local, auto)

  moreover have "(σ', s') ∈ oreachable (opnet onp p1) (?S p1) (?U p1)"
  proof -
    from ‹∀j. j∉net_tree_ips p2 ⟶ σ' j = σ j› and ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}›
      have "∀j. j∈net_tree_ips p1 ⟶ σ' j = σ j" by auto
    moreover from tor ttr have "∀j∈net_tree_ips p2. U (σ j) (σ' j)"
      by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
    ultimately show ?thesis
```

```
        using sor ‹s' = s› ‹∀j. j ∉ net_tree_ips p₂ ⟶ σ' j = σ j›
        by (clarsimp elim!: oreachable_other')
           (metis otherI ‹⋀ξ. U ξ ξ›)+
    qed
    ultimately show ?thesis by - (rule conjI)
  next
    assume "s' = s"
       and ttr: "((σ, t), τ, (σ', t')) ∈ trans (opnet onp p₂)"
    from tor ttr have "∀j. j∉net_tree_ips p₂ ⟶ σ' j = σ j"
      by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
    hence "∀j. j∉net_tree_ips p₂ ⟶ S (σ j) (σ' j)"
       by (auto intro: ‹⋀ξ. S ξ ξ›)
    with tor ttr
      have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
        by - (erule(1) oreachable_local, auto)

    moreover have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
    proof -
      from ‹∀j. j∉net_tree_ips p₂ ⟶ σ' j = σ j› and ‹net_tree_ips p₁ ∩ net_tree_ips p₂ = {}›
        have "∀j. j∈net_tree_ips p₁ ⟶ σ' j = σ j" by auto
      moreover from tor ttr have "∀j∈net_tree_ips p₂. U (σ j) (σ' j)"
        by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
      ultimately show ?thesis
        using sor ‹s' = s› ‹∀j. j ∉ net_tree_ips p₂ ⟶ σ' j = σ j›
        by (clarsimp elim!: oreachable_other')
           (metis otherI ‹⋀ξ. U ξ ξ›)+
    qed
    ultimately show ?thesis by - (rule conjI)
  next
    assume "t' = t"
       and str: "((σ, s), τ, (σ', s')) ∈ trans (opnet onp p₁)"
    from sor str have "∀j. j∉net_tree_ips p₁ ⟶ σ' j = σ j"
      by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
    hence "∀j. j∉net_tree_ips p₁ ⟶ S (σ j) (σ' j)"
       by (auto intro: ‹⋀ξ. S ξ ξ›)
    with sor str
      have "(σ', s') ∈ oreachable (opnet onp p₁) (?S p₁) (?U p₁)"
        by - (erule(1) oreachable_local, auto)

    moreover have "(σ', t') ∈ oreachable (opnet onp p₂) (?S p₂) (?U p₂)"
    proof -
      from ‹∀j. j∉net_tree_ips p₁ ⟶ σ' j = σ j› and ‹net_tree_ips p₁ ∩ net_tree_ips p₂ = {}›
        have "∀j. j∈net_tree_ips p₂ ⟶ σ' j = σ j" by auto
      moreover from sor str have "∀j∈net_tree_ips p₁. U (σ j) (σ' j)"
        by - (drule(1) ostep_arrive_invariantD [OF act], simp_all)
      ultimately show ?thesis
        using tor ‹t' = t› ‹∀j. j ∉ net_tree_ips p₁ ⟶ σ' j = σ j›
        by (clarsimp elim!: oreachable_other')
           (metis otherI ‹⋀ξ. U ξ ξ›)+
    qed
    ultimately show ?thesis ..
  qed
  with ‹net_tree_ips p₁ ∩ net_tree_ips p₂ = {}› show ?case by simp
qed

lemmas subnet_oreachable1 [dest] = subnet_oreachable [THEN conjunct1, rotated 1]
lemmas subnet_oreachable2 [dest] = subnet_oreachable [THEN conjunct2, THEN conjunct1, rotated 1]
lemmas subnet_oreachable_disjoint [dest] = subnet_oreachable
                                          [THEN conjunct2, THEN conjunct2, rotated 1]

corollary pnet_lift:
  assumes "⋀ii Rᵢ. ⟨ii : onp ii : Rᵢ⟩ₒ
            ⊨ (otherwith S {ii} (oarrivemsg 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 _. oarrivemsg I σ, other U {i} →)
                   globala (λ(σ, a, _). castmsg (I σ) a)"
    and node2: "⋀i R. ⟨i : onp i : R⟩o ⊨A (λσ _. oarrivemsg 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 _. oarrivemsg 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) (oarrivemsg I),
                 other U (net_tree_ips p) →) global (λσ. ∀i∈net_tree_ips p. P i σ)"
    (is "_ ⊨ (?owS p, ?U p →) _")
proof (induction p)
  fix ii Ri
  from assms(1) show "opnet onp ⟨ii; Ri⟩ ⊨ (?owS ⟨ii; Ri⟩, ?U ⟨ii; Ri⟩ →)
                              global (λσ. ∀i∈net_tree_ips ⟨ii; Ri⟩. P i σ)" by auto
next
  fix p1 p2
  assume ih1: "opnet onp p1 ⊨ (?owS p1, ?U p1 →) global (λσ. ∀i∈net_tree_ips p1. P i σ)"
     and ih2: "opnet onp p2 ⊨ (?owS p2, ?U p2 →) global (λσ. ∀i∈net_tree_ips p2. P i σ)"
  show "opnet onp (p1 ∥ p2) ⊨ (?owS (p1 ∥ p2), ?U (p1 ∥ p2) →)
                      global (λσ. ∀i∈net_tree_ips (p1 ∥ p2). P i σ)"
  unfolding oinvariant_def
  proof
    fix pq
    assume "pq ∈ oreachable (opnet onp (p1 ∥ p2)) (?owS (p1 ∥ p2)) (?U (p1 ∥ p2))"
    moreover then obtain σ s t where "pq = (σ, SubnetS s t)"
      by (metis net_par_oreachable_is_subnet surjective_pairing)
    ultimately have "(σ, SubnetS s t) ∈ oreachable (opnet onp (p1 ∥ p2))
                                   (?owS (p1 ∥ p2)) (?U (p1 ∥ p2))" by simp
    then obtain sor: "(σ, s) ∈ oreachable (opnet onp p1) (?owS p1) (?U p1)"
           and tor: "(σ, t) ∈ oreachable (opnet onp p2) (?owS p2) (?U p2)"
      by - (drule subnet_oreachable [OF _ _ _ node1 node2 node3], auto intro: assms(2-3))
    from sor have "∀i∈net_tree_ips p1. P i σ"
      by (auto dest: oinvariantD [OF ih1])
    moreover from tor have "∀i∈net_tree_ips p2. P i σ"
      by (auto dest: oinvariantD [OF ih2])
    ultimately have "∀i∈net_tree_ips (p1 ∥ p2). P i σ" by auto
    with ‹pq = (σ, SubnetS s t)› show "global (λσ. ∀i∈net_tree_ips (p1 ∥ p2). P i σ) pq" by simp
  qed
qed

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"
  by (metis prod.exhaust oconnect_completeTE)

lemma trans_fst_oclosed_fst2 [dest]:
  "(s, disconnect(i, i'), s') ∈ ocnet_sos (trans p) ⟹ (s, disconnect(i, i'), s') ∈ trans p"
  by (metis prod.exhaust odisconnect_completeTE)

lemma trans_fst_oclosed_fst3 [dest]:
  "(s, i:deliver(d), s') ∈ ocnet_sos (trans p) ⟹      (s, i:deliver(d), s') ∈ trans p"
  by (metis prod.exhaust odeliver_completeTE)

lemma oclosed_oreachable_inclosed:
```

```
  assumes "(σ, ζ) ∈ oreachable (oclosed (opnet np p)) (λ_ _ _. True) U"
    shows "(σ, ζ) ∈ oreachable (opnet np p) (otherwith ((=)) (net_tree_ips p) inoclosed) U"
    (is "_ ∈ oreachable _ ?owS _")
  using assms proof (induction rule: oreachable_pair_induct)
    fix σ ζ
    assume "(σ, ζ) ∈ init (oclosed (opnet np p))"
    hence "(σ, ζ) ∈ init (opnet np p)" by simp
    thus "(σ, ζ) ∈ oreachable (opnet np p) ?owS U" ..
  next
    fix σ ζ σ'
    assume "(σ, ζ) ∈ oreachable (opnet np p) ?owS U"
       and "U σ σ'"
    thus "(σ', ζ) ∈ oreachable (opnet np p) ?owS U"
      by - (rule oreachable_other')
  next
    fix σ ζ σ' ζ' a
    assume zor: "(σ, ζ) ∈ oreachable (opnet np p) ?owS U"
       and ztr: "((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet np p))"
    from this(1) have [simp]: "net_ips ζ = net_tree_ips p"
      by (rule opnet_net_ips_net_tree_ips)
    from ztr have "((σ, ζ), a, (σ', ζ')) ∈ ocnet_sos (trans (opnet np p))" by simp
    thus "(σ', ζ') ∈ oreachable (opnet np p) ?owS U"
    proof cases
      fix i K d di
      assume "a = i:newpkt(d, di)"
         and tr: "((σ, ζ), {i}¬K:arrive(msg_class.newpkt (d, di)), (σ', ζ')) ∈ trans (opnet np p)"
         and "∀j. j ∉ net_ips ζ ⟶ σ' j = σ j"
      from this(3) have "∀j. j ∉ net_tree_ips p ⟶ σ' j = σ j"
        using ‹net_ips ζ = net_tree_ips p› by auto
      hence "otherwith ((=)) (net_tree_ips p) inoclosed σ σ' ({i}¬K:arrive(msg_class.newpkt (d, di)))"
        by auto
      with zor tr show ?thesis
        by - (rule oreachable_local')
    next
      assume "a = τ"
         and tr: "((σ, ζ), τ, (σ', ζ')) ∈ trans (opnet np p)"
         and "∀j. j ∉ net_ips ζ ⟶ σ' j = σ j"
      from this(3) have "∀j. j ∉ net_tree_ips p ⟶ σ' j = σ j"
        using ‹net_ips ζ = net_tree_ips p› by auto
      hence "otherwith ((=)) (net_tree_ips p) inoclosed σ σ' τ"
        by auto
      with zor tr show ?thesis by - (rule oreachable_local')
    qed (insert ‹net_ips ζ = net_tree_ips p›,
         auto elim!: oreachable_local' [OF zor])
  qed

lemma oclosed_oreachable_oreachable [elim]:
  assumes "(σ, ζ) ∈ oreachable (oclosed (opnet onp p)) (λ_ _ _. True) U"
    shows "(σ, ζ) ∈ oreachable (opnet onp p) (λ_ _ _. True) U"
  using assms by (rule oclosed_oreachable_inclosed [THEN oreachable_weakenE]) simp

lemma inclosed_closed [intro]:
  assumes cinv: "opnet np p ⊨ (otherwith ((=)) (net_tree_ips p) inoclosed, U →) P"
    shows "oclosed (opnet np p) ⊨ (λ_ _ _. True, U →) P"
  using assms unfolding oinvariant_def
  by (clarsimp dest!: oclosed_oreachable_inclosed)

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 ⇒ ('z × ('s, 'm, 'p, 'l) seqp) set ⇒ bool"
where
  "control_within Γ σ ≡ ∀(ξ, p)∈σ. ∃pn. p ∈ subterms (Γ pn)"

lemma control_withinI [intro]:
  assumes "⋀p. p ∈ Range σ ⟹ ∃pn. p ∈ subterms (Γ pn)"
    shows "control_within Γ σ"
  using assms unfolding control_within_def by auto

lemma control_withinD [dest]:
  assumes "control_within Γ σ"
      and "(ξ, p) ∈ σ"
    shows "∃pn. p ∈ subterms (Γ pn)"
  using assms unfolding control_within_def by blast

lemma control_within_topI [intro]:
  assumes "⋀p. p ∈ Range σ ⟹ ∃pn. p = Γ pn"
    shows "control_within Γ σ"
  using assms unfolding control_within_def
  by clarsimp (metis Range.RangeI subterms_refl)

lemma seqp_sos_subterms:
  assumes "wellformed Γ"
      and "∃pn. p ∈ subterms (Γ pn)"
      and "((ξ, p), a, (ξ', p')) ∈ seqp_sos Γ"
    shows "∃pn. p' ∈ subterms (Γ pn)"
  using assms
  proof (induct p)
    fix p1 p2
    assume IH1: "∃pn. p1 ∈ subterms (Γ pn) ⟹
                   ((ξ, p1), a, (ξ', p')) ∈ seqp_sos Γ ⟹
                   ∃pn. p' ∈ subterms (Γ pn)"
      and IH2: "∃pn. p2 ∈ subterms (Γ pn) ⟹
                   ((ξ, p2), a, (ξ', p')) ∈ seqp_sos Γ ⟹
                   ∃pn. p' ∈ subterms (Γ pn)"
      and "∃pn. p1 ⊕ p2 ∈ subterms (Γ pn)"
      and "((ξ, p1 ⊕ p2), a, (ξ', p')) ∈ seqp_sos Γ"
    from ‹∃pn. p1 ⊕ p2 ∈ subterms (Γ pn)› obtain pn
                                          where "p1 ∈ subterms (Γ pn)"
                                            and "p2 ∈ subterms (Γ pn)" by auto
    from ‹((ξ, p1 ⊕ p2), a, (ξ', p')) ∈ seqp_sos Γ›
      have "((ξ, p1), a, (ξ', p')) ∈ seqp_sos Γ
            ∨ ((ξ, p2), a, (ξ', p')) ∈ seqp_sos Γ" by auto
    thus "∃pn. p' ∈ subterms (Γ pn)"
    proof
      assume "((ξ, p1), a, (ξ', p')) ∈ seqp_sos Γ"
      with ‹p1 ∈ subterms (Γ pn)› show ?thesis by (auto intro: IH1)
    next
      assume "((ξ, p2), a, (ξ', p')) ∈ seqp_sos Γ"
      with ‹p2 ∈ subterms (Γ pn)› show ?thesis by (auto intro: IH2)
    qed
  qed auto

lemma reachable_subterms:
  assumes "wellformed Γ"
      and "control_within Γ (init A)"
      and "trans A = seqp_sos Γ"
      and "(ξ, p) ∈ reachable A I"
    shows "∃pn. p ∈ subterms (Γ pn)"
```

```
  using assms(4)
  proof (induct rule: reachable_pair_induct)
    fix ξ p
    assume "(ξ, p) ∈ init A"
    with ‹control_within Γ (init A)› show "∃pn. p ∈ subterms (Γ pn)" ..
  next
    fix ξ p a ξ' p'
    assume "(ξ, p) ∈ reachable A I"
       and "∃pn. p ∈ subterms (Γ pn)"
       and *: "((ξ, p), a, (ξ', p')) ∈ trans A"
       and "I a"
    moreover from * and assms(3) have "((ξ, p), a, (ξ', p')) ∈ seqp_sos Γ" by simp
    ultimately show "∃pn. p' ∈ subterms (Γ pn)"
    using ‹wellformed Γ›
      by (auto elim: seqp_sos_subterms)
  qed

definition
  onl :: "('s, 'm, 'p, 'l) seqp_env
          ⇒ ('z × 'l ⇒ bool)
          ⇒ 'z × ('s, 'm, 'p, 'l) seqp
          ⇒ bool"
where
  "onl Γ P ≡ (λ(ξ, p). ∀l∈labels Γ p. P (ξ, l))"

lemma onlI [intro]:
  assumes "⋀l. l∈labels Γ p ⟹ P (ξ, l)"
    shows "onl Γ P (ξ, p)"
  using assms unfolding onl_def by simp

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

lemma onlD [dest]:
  assumes "onl Γ P (ξ, p)"
    shows "∀l∈labels Γ p. P (ξ, l)"
  using assms unfolding onl_def by simp

lemma onl_invariantI [intro]:
  assumes init: "⋀ξ p l. ⟦ (ξ, p) ∈ init A; l ∈ labels Γ p ⟧ ⟹ P (ξ, l)"
      and step: "⋀ξ p a ξ' p' l'.
                 ⟦ (ξ, p) ∈ reachable A I;
                   ∀l∈labels Γ p. P (ξ, l);
                   ((ξ, p), a, (ξ', p')) ∈ trans A;
                   l' ∈ labels Γ p';
                   I a ⟧ ⟹ P (ξ', l')"
    shows "A ⊫ (I →) onl Γ P"
  proof (rule invariant_pairI)
    fix ξ p
    assume "(ξ, p) ∈ init A"
    hence "∀l∈labels Γ p. P (ξ, l)" using init by simp
    thus "onl Γ P (ξ, p)" ..
  next
    fix ξ p a ξ' p'
    assume rp: "(ξ, p) ∈ reachable A I"
       and "onl Γ P (ξ, p)"
       and tr: "((ξ, p), a, (ξ', p')) ∈ trans A"
       and "I a"
    from ‹onl Γ P (ξ, p)› have "∀l∈labels Γ p. P (ξ, l)" ..
    with rp tr ‹I a› have "∀l'∈labels Γ p'. P (ξ', l')" by (auto elim: step)
    thus "onl Γ P (ξ', p')" ..
  qed

lemma onl_invariantD [dest]:
  assumes "A ⊫ (I →) onl Γ P"
```

```
    and "(ξ, p) ∈ reachable A I"
    and "l ∈ labels Γ p"
  shows "P (ξ, l)"
  using assms unfolding onl_def by auto

lemma onl_invariant_initD [dest]:
  assumes invP: "A ⊫ (I →) onl Γ P"
      and init: "(ξ, p) ∈ init A"
      and pnl:  "l ∈ labels Γ p"
    shows "P (ξ, l)"
  proof -
    from init have "(ξ, p) ∈ reachable A I" ..
    with invP show ?thesis using pnl ..
  qed

lemma onl_invariant_sterms:
  assumes wf: "wellformed Γ"
      and il: "A ⊫ (I →) onl Γ P"
      and rp: "(ξ, p) ∈ reachable A I"
      and "p'∈sterms Γ p"
      and "l∈labels Γ p'"
    shows "P (ξ, l)"
  proof -
    from wf ‹p'∈sterms Γ p› ‹l∈labels Γ p'› have "l∈labels Γ p"
      by (rule labels_sterms_labels)
    with il rp show "P (ξ, l)" ..
  qed

lemma onl_invariant_sterms_weaken:
  assumes wf: "wellformed Γ"
      and il: "A ⊫ (I →) onl Γ P"
      and rp: "(ξ, p) ∈ reachable A I'"
      and "p'∈sterms Γ p"
      and "l∈labels Γ p'"
      and weaken: "⋀a. I' a ⟹ I a"
    shows "P (ξ, l)"
  proof -
    from ‹(ξ, p) ∈ reachable A I'› have "(ξ, p) ∈ reachable A I"
      by (rule reachable_weakenE)
         (erule weaken)
    with assms(1-2) show ?thesis using assms(4-5) by (rule onl_invariant_sterms)
  qed

lemma onl_invariant_sterms_TT:
  assumes wf: "wellformed Γ"
      and il: "A ⊫ onl Γ P"
      and rp: "(ξ, p) ∈ reachable A I"
      and "p'∈sterms Γ p"
      and "l∈labels Γ p'"
    shows "P (ξ, l)"
  using assms by (rule onl_invariant_sterms_weaken) simp

lemma trans_from_sterms:
  assumes "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
      and "wellformed Γ"
    shows "∃p'∈sterms Γ p. ((ξ, p'), a, (ξ', q)) ∈ seqp_sos Γ"
  using assms by (induction p rule: sterms_pinduct [OF ‹wellformed Γ›]) auto

lemma trans_from_sterms':
  assumes "((ξ, p'), a, (ξ', q)) ∈ seqp_sos Γ"
      and "wellformed Γ"
      and "p' ∈ sterms Γ p"
    shows "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
  using assms by (induction p rule: sterms_pinduct [OF ‹wellformed Γ›]) auto
```

```
lemma trans_to_dterms:
  assumes "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
      and "wellformed Γ"
   shows "∀r∈sterms Γ q. r ∈ dterms Γ p"
  using assms by (induction q) auto

theorem cterms_includes_sterms_of_seq_reachable:
  assumes "wellformed Γ"
      and "control_within Γ (init A)"
      and "trans A = seqp_sos Γ"
    shows "⋃(sterms Γ ` snd ` reachable A I) ⊆ cterms Γ"
  proof
    fix qs
    assume "qs ∈ ⋃(sterms Γ ` snd ` reachable A I)"
    then obtain ξ and q where  *: "(ξ, q) ∈ reachable A I"
                          and **: "qs ∈ sterms Γ q" by auto
    from * have "⋀x. x ∈ sterms Γ q ⟹ x ∈ cterms Γ"
    proof (induction rule: reachable_pair_induct)
      fix ξ p q
      assume "(ξ, p) ∈ init A"
         and "q ∈ sterms Γ p"
      from ‹control_within Γ (init A)› and ‹(ξ, p) ∈ init A›
        obtain pn where "p ∈ subterms (Γ pn)" by auto
      with ‹wellformed Γ› show "q ∈ cterms Γ" using ‹q∈sterms Γ p›
        by (rule subterms_sterms_in_cterms)
    next
      fix p ξ a ξ' q x
      assume "(ξ, p) ∈ reachable A I"
         and IH: "⋀x. x ∈ sterms Γ p ⟹ x ∈ cterms Γ"
         and "((ξ, p), a, (ξ', q)) ∈ trans A"
         and "x ∈ sterms Γ q"
      from this(3) and ‹trans A = seqp_sos Γ› have "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ" by simp
      from this and ‹wellformed Γ› obtain ps
        where ps: "ps ∈ sterms Γ p"
          and step: "((ξ, ps), a, (ξ', q)) ∈ seqp_sos Γ"
        by (rule trans_from_sterms [THEN bexE])
      from ps have "ps ∈ cterms Γ" by (rule IH)
      moreover from step ‹wellformed Γ› ‹x ∈ sterms Γ q› have "x ∈ dterms Γ ps"
        by (rule trans_to_dterms [rule_format])
      ultimately show "x ∈ cterms Γ" by (rule ctermsDI)
    qed
    thus "qs ∈ cterms Γ" using ** .
  qed

corollary seq_reachable_in_cterms:
  assumes "wellformed Γ"
      and "control_within Γ (init A)"
      and "trans A = seqp_sos Γ"
      and "(ξ, p) ∈ reachable A I"
      and "p' ∈ sterms Γ p"
    shows "p' ∈ cterms Γ"
  using assms(1-3)
  proof (rule cterms_includes_sterms_of_seq_reachable [THEN subsetD])
    from assms(4-5) show "p' ∈ ⋃(sterms Γ ` snd ` reachable A I)"
      by (auto elim!: rev_bexI)
  qed

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: "⋀ξ p l. ⟦
```

```
                (ξ, p) ∈ init A;
                l∈labels Γ p
              ⟧ ⟹ P (ξ, l)"
    and step: "⋀p l ξ a q l' ξ' pp. ⟦
              p∈cterms Γ;
              l∈labels Γ p;
              P (ξ, l);
              ((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ;
              ((ξ, p), a, (ξ', q)) ∈ trans A;
              l'∈labels Γ q;
              (ξ, pp)∈reachable A I;
              p∈sterms Γ pp;
              (ξ', q)∈reachable A I;
              I a
            ⟧ ⟹ P (ξ', l')"
  shows "A ⊫ (I →) onl Γ P"
proof
    fix ξ p l
  assume "(ξ, p) ∈ init A"
    and *: "l ∈ labels Γ p"
    with init show "P (ξ, l)" by auto
next
    fix ξ p a ξ' q l'
  assume sr: "(ξ, p) ∈ reachable A I"
    and pl: "∀l∈labels Γ p. P (ξ, l)"
    and tr: "((ξ, p), a, (ξ', q)) ∈ trans A"
    and A6: "l' ∈ labels Γ q"
    and "I a"
   from this(3) and ‹trans A = seqp_sos Γ› have tr': "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ" by simp
   show "P (ξ', l')"
  proof -
    from sr and tr and ‹I a› have A7: "(ξ', q) ∈ reachable A I" ..
    from tr' obtain p' where "p' ∈ sterms Γ p"
                        and "((ξ, p'), a, (ξ', q)) ∈ seqp_sos Γ"
      by (blast dest: trans_from_sterms [OF _ wf])
    from wf cw sp sr this(1) have A1: "p'∈cterms Γ"
      by (rule seq_reachable_in_cterms)
    from labels_not_empty [OF wf] obtain ll where A2: "ll∈labels Γ p'"
        by blast
    with ‹p'∈sterms Γ p› have "ll∈labels Γ p"
      by (rule labels_sterms_labels [OF wf])
    with pl have A3: "P (ξ, ll)" by simp
    from ‹((ξ, p'), a, (ξ', q)) ∈ seqp_sos Γ› and sp
      have A5: "((ξ, p'), a, (ξ', q)) ∈ trans A" by simp
    with sp have A4: "((ξ, p'), a, (ξ', q)) ∈ seqp_sos Γ" by simp
    from sr ‹p'∈sterms Γ p›
      obtain pp where A7: "(ξ, pp)∈reachable A I"
                  and A8: "p'∈sterms Γ pp"
      by auto
    from sr tr ‹I a› have A9: "(ξ', q) ∈ reachable A I" ..
    from A1 A2 A3 A4 A5 A6 A7 A8 A9 ‹I a› show ?thesis by (rule step)
  qed
qed

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: "⋀ξ p l. ⟦
                (ξ, p) ∈ init A;
                l∈labels Γ p
              ⟧ ⟹ P (ξ, l)"
    and step: "⋀p l ξ a q l' ξ' pp pn. ⟦
```

```
                wellformed Γ;
                p∈ctermsl (Γ pn);
                not_call p;
                l∈labels Γ p;
                P (ξ, l);
                ((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ;
                ((ξ, p), a, (ξ', q)) ∈ trans A;
                l'∈labels Γ q;
                (ξ, pp)∈reachable A I;
                p∈sterms Γ pp;
                (ξ', q)∈reachable A I;
                I a
              ⟧ ⟹ P (ξ', l')"
    shows "A ⊫ (I →) onl Γ P"
  using assms(1-4) proof (rule seq_invariant_ctermI)
    fix ξ p l
    assume "(ξ, p) ∈ init A"
       and "l ∈ labels Γ p"
    thus "P (ξ, l)" by (rule init)
  next
    fix p l ξ a q l' ξ' pp
    assume "p ∈ cterms Γ"
       and otherassms: "l ∈ labels Γ p"
           "P (ξ, l)"
           "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
           "((ξ, p), a, (ξ', q)) ∈ trans A"
           "l' ∈ labels Γ q"
           "(ξ, pp) ∈ reachable A I"
           "p ∈ sterms Γ pp"
           "(ξ', q) ∈ reachable A I"
           "I a"
    from this(1) obtain pn where "p ∈ ctermsl(Γ pn)"
                             and "not_call p"
      unfolding cterms_def' [OF wf] by auto
    with wf show "P (ξ', l')"
      using otherassms by (rule step)
  qed
```

19.2 Step invariants via labelled control terms

```
definition
  onll :: "('s, 'm, 'p, 'l) seqp_env
          ⇒ (('z × 'l, 'a) transition ⇒ bool)
          ⇒ ('z × ('s, 'm, 'p, 'l) seqp, 'a) transition ⇒ bool"
where
  "onll Γ P ≡ (λ((ξ, p), a, (ξ', p')). ∀l∈labels Γ p. ∀l'∈labels Γ p'. P ((ξ, l), a, (ξ', l')))"

lemma onllI [intro]:
  assumes "⋀l l'. ⟦ l∈labels Γ p; l'∈labels Γ p' ⟧ ⟹ P ((ξ, l), a, (ξ', l'))"
    shows "onll Γ P ((ξ, p), a, (ξ', p'))"
  using assms unfolding onll_def by simp

lemma onllI1 [intro]:
  assumes "∀l∈labels Γ p. ∀l'∈labels Γ p'. P ((ξ, l), a, (ξ', l'))"
    shows "onll Γ P ((ξ, p), a, (ξ', p'))"
  using assms by auto

lemma onllD [dest]:
  assumes "onll Γ P ((ξ, p), a, (ξ', p'))"
    shows "∀l∈labels Γ p. ∀l'∈labels Γ p'. P ((ξ, l), a, (ξ', l'))"
  using assms unfolding onll_def by simp

lemma onl_weaken [elim!]: "⋀Γ P Q s. ⟦ onl Γ P s; ⋀s. P s ⟹ Q s ⟧ ⟹ onl Γ Q s"
  by (clarsimp dest!: onlD intro!: onlI)
```

```
lemma onll_weaken [elim!]: "⋀Γ P Q s. ⟦ onll Γ P s; ⋀s. P s ⟹ Q s ⟧ ⟹ onll Γ Q s"
  by (clarsimp dest!: onllD intro!: onllI)

lemma onll_weaken' [elim!]: "⋀Γ P Q s. ⟦ onll Γ P ((ξ, p), a, (ξ', p'));
                                          ⋀l l'. P ((ξ, l), a, (ξ', l')) ⟹ Q ((ξ, l), a, (ξ', l')) ⟧
                                         ⟹ onll Γ Q ((ξ, p), a, (ξ', p'))"
  by (clarsimp dest!: onllD intro!: onllI)

lemma onll_step_invariantI [intro]:
  assumes *: "⋀ξ p l a ξ' p' l'. ⟦ (ξ, p)∈reachable A I;
                                   ((ξ, p), a, (ξ', p')) ∈ trans A;
                                   I a;
                                   l ∈labels Γ p;
                                   l'∈labels Γ p' ⟧
                                 ⟹ P ((ξ, l), a, (ξ', l'))"
    shows "A ⊫A (I →) onll Γ P"
  proof
    fix ξ p ξ' p' a
    assume "(ξ, p) ∈ reachable A I"
       and "((ξ, p), a, (ξ', p')) ∈ trans A"
       and "I a"
    hence "∀l∈labels Γ p. ∀l'∈labels Γ p'. P ((ξ, l), a, (ξ', l'))" by (auto elim!: *)
    thus "onll Γ P ((ξ, p), a, (ξ', p'))" ..
  qed

lemma onll_step_invariantE [elim]:
  assumes "A ⊫A (I →) onll Γ P"
      and "(ξ, p) ∈ reachable A I"
      and "((ξ, p), a, (ξ', p')) ∈ trans A"
      and "I a"
      and lp:  "l ∈labels Γ p"
      and lp': "l'∈labels Γ p'"
    shows "P ((ξ, l), a, (ξ', l'))"
  proof -
    from assms(1-4) have "onll Γ P ((ξ, p), a, (ξ', p'))" ..
    with lp lp' show "P ((ξ, l), a, (ξ', l'))" by auto
  qed

lemma onll_step_invariantD [dest]:
  assumes "A ⊫A (I →) onll Γ P"
      and "(ξ, p) ∈ reachable A I"
      and "((ξ, p), a, (ξ', p')) ∈ trans A"
      and "I a"
    shows "∀l∈labels Γ p. ∀l'∈labels Γ p'. P ((ξ, l), a, (ξ', l'))"
  using assms by auto

lemma onll_step_to_invariantI [intro]:
  assumes sinv: "A ⊫A (I →) onll Γ Q"
      and wf: "wellformed Γ"
      and init: "⋀ξ l p. ⟦ (ξ, p) ∈ init A; l∈labels Γ p ⟧ ⟹ P (ξ, l)"
      and step: "⋀ξ p l ξ' l' a.
                 ⟦ (ξ, p) ∈ reachable A I;
                   l∈labels Γ p;
                   P (ξ, l);
                   Q ((ξ, l), a, (ξ', l'));
                   I a⟧ ⟹ P (ξ', l')"
    shows "A ⊨ (I →) onl Γ P"
  proof
    fix ξ p l
    assume "(ξ, p) ∈ init A" and "l∈labels Γ p"
      thus "P (ξ, l)" by (rule init)
  next
    fix ξ p a ξ' p' l'
```

```
    assume sr: "(ξ, p) ∈ reachable A I"
       and lp: "∀l∈labels Γ p. P (ξ, l)"
       and tr: "((ξ, p), a, (ξ', p')) ∈ trans A"
       and "I a"
       and lp': "l' ∈ labels Γ p'"
      show "P (ξ', l')"
    proof -
      from lp obtain l where "l∈labels Γ p" and "P (ξ, l)"
        using labels_not_empty [OF wf] by auto
      from sinv sr tr ‹I a› this(1) lp' have "Q ((ξ, l), a, (ξ', l'))" ..
      with sr ‹l∈labels Γ p› ‹P (ξ, l)› show "P (ξ', l')" using ‹I a› by (rule step)
    qed
  qed

lemma onll_step_invariant_sterms:
  assumes wf: "wellformed Γ"
      and si: "A ⊫_A (I →) onll Γ P"
      and sr: "(ξ, p) ∈ reachable A I"
      and sos: "((ξ, p), a, (ξ', q)) ∈ trans A"
      and "I a"
      and "l'∈labels Γ q"
      and "p'∈sterms Γ p"
      and "l∈labels Γ p'"
    shows "P ((ξ, l), a, (ξ', l'))"
  proof -
    from wf ‹p'∈sterms Γ p› ‹l∈labels Γ p'› have "l∈labels Γ p"
      by (rule labels_sterms_labels)
    with si sr sos ‹I a› show "P ((ξ, l), a, (ξ', l'))" using ‹l'∈labels Γ q› ..
  qed

lemma seq_step_invariant_sterms:
  assumes inv: "A ⊫_A (I →) onll Γ P"
      and wf: "wellformed Γ"
      and sp: "trans A = seqp_sos Γ"
      and "l'∈labels Γ q"
      and sr: "(ξ, p) ∈ reachable A I"
      and tr: "((ξ, p'), a, (ξ', q)) ∈ trans A"
      and "I a"
      and "p'∈sterms Γ p"
    shows "∀l∈labels Γ p'. P ((ξ, l), a, (ξ', l'))"
  proof
    from tr and sp have "((ξ, p'), a, (ξ', q)) ∈ seqp_sos Γ" by simp
    hence "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
      using wf ‹p'∈sterms Γ p› by  (rule trans_from_sterms')
    with sp have trp: "((ξ, p), a, (ξ', q)) ∈ trans A" by simp
    fix l assume "l ∈ labels Γ p'"
    with wf inv sr trp ‹I a› ‹l'∈labels Γ q› ‹p'∈sterms Γ p›
      show "P ((ξ, l), a, (ξ', l'))" by (rule onll_step_invariant_sterms)
  qed

lemma seq_step_invariant_sterms_weaken:
  assumes "A ⊫_A (I →) onll Γ P"
      and "wellformed Γ"
      and "trans A = seqp_sos Γ"
      and "l'∈labels Γ q"
      and "(ξ, p) ∈ reachable A I'"
      and "((ξ, p'), a, (ξ', q)) ∈ trans A"
      and "I' a"
      and "p'∈sterms Γ p"
      and weaken: "⋀a. I' a ⟹ I a"
    shows "∀l∈labels Γ p'. P ((ξ, l), a, (ξ', l'))"
  proof -
    from ‹I' a› have "I a" by (rule weaken)
    from ‹(ξ, p) ∈ reachable A I'› have Ir: "(ξ, p) ∈ reachable A I"
```

```
      by (rule reachable_weakenE) (erule weaken)
    with assms(1-4) show ?thesis
      using ‹((ξ, p'), a, (ξ', q)) ∈ trans A› ‹I a› and ‹p'∈sterms Γ p›
      by (rule seq_step_invariant_sterms)
  qed

lemma seq_step_invariant_sterms_TT:
  assumes "A ⊫A onll Γ P"
      and "wellformed Γ"
      and "trans A = seqp_sos Γ"
      and "l'∈labels Γ q"
      and "(ξ, p) ∈ reachable A I"
      and "((ξ, p'), a, (ξ', q)) ∈ trans A"
      and "I a"
      and "p'∈sterms Γ p"
    shows "∀l∈labels Γ p'. P ((ξ, l), a, (ξ', l'))"
  using assms by (rule seq_step_invariant_sterms_weaken) simp

lemma onll_step_invariant_any_sterms:
  assumes "wellformed Γ"
      and "A ⊫A (I →) onll Γ P"
      and "(ξ, p) ∈ reachable A I"
      and "((ξ, p), a, (ξ', q)) ∈ trans A"
      and "I a"
      and "l'∈labels Γ q"
    shows "∀p'∈sterms Γ p. ∀l∈labels Γ p'. P ((ξ, l), a, (ξ', l'))"
  by (intro ballI) (rule onll_step_invariant_sterms [OF assms])

lemma seq_step_invariant_ctermI [intro]:
  assumes wf: "wellformed Γ"
      and cw: "control_within Γ (init A)"
      and sl: "simple_labels Γ"
      and sp: "trans A = seqp_sos Γ"
      and step: "⋀p pp l ξ a q l' ξ'. ⟦
                p∈cterms Γ;
                l∈labels Γ p;
                ((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ;
                ((ξ, p), a, (ξ', q)) ∈ trans A;
                l'∈labels Γ q;
                (ξ, pp) ∈ reachable A I;
                p∈sterms Γ pp;
                (ξ', q) ∈ reachable A I;
                I a
              ⟧ ⟹ P ((ξ, l), a, (ξ', l'))"
    shows "A ⊫A (I →) onll Γ P"
  proof
      fix ξ p l a ξ' q l'
    assume sr: "(ξ, p) ∈ reachable A I"
       and tr: "((ξ, p), a, (ξ', q)) ∈ trans A"
       and "I a"
       and pl: "l ∈ labels Γ p"
       and A5: "l' ∈ labels Γ q"
    from this(2) and sp have tr': "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ" by simp
    then obtain p' where "p' ∈ sterms Γ p"
                     and A3: "((ξ, p'), a, (ξ', q)) ∈ seqp_sos Γ"
      by (blast dest: trans_from_sterms [OF _ wf])
    from wf cw sp sr this(1) have A1: "p'∈cterms Γ"
      by (rule seq_reachable_in_cterms)
    from ‹((ξ, p'), a, (ξ', q)) ∈ seqp_sos Γ› and sp
      have A4: "((ξ, p'), a, (ξ', q)) ∈ trans A" by simp
    from sr ‹p'∈sterms Γ p› obtain pp where A6: "(ξ, pp)∈reachable A I"
                                      and A7: "p'∈sterms Γ pp"
      by auto
    from sr tr ‹I a› have A8: "(ξ', q)∈reachable A I" ..
```

```
    from wf cw sp sr have "∃pn. p ∈ subterms (Γ pn)"
      by (rule reachable_subterms)
    with sl wf have "∀p'∈sterms Γ p. l ∈ labels Γ p'"
      using pl by (rule simple_labels_in_sterms)
    with ‹p' ∈ sterms Γ p› have "l ∈ labels Γ p'" by simp
    with A1 show "P ((ξ, l), a, (ξ', l'))" using A3 A4 A5 A6 A7 A8 ‹I a›
      by (rule step)
  qed

lemma seq_step_invariant_ctermsI [intro]:
  assumes wf: "wellformed Γ"
      and cw: "control_within Γ (init A)"
      and sl: "simple_labels Γ"
      and sp: "trans A = seqp_sos Γ"
      and step: "⋀p l ξ a q l' ξ' pp pn. ⟦
                  wellformed Γ;
                  p∈ctermsl (Γ pn);
                  not_call p;
                  l∈labels Γ p;
                  ((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ;
                  ((ξ, p), a, (ξ', q)) ∈ trans A;
                  l'∈labels Γ q;
                  (ξ, pp) ∈ reachable A I;
                  p∈sterms Γ pp;
                  (ξ', q) ∈ reachable A I;
                  I a
                ⟧ ⟹ P ((ξ, l), a, (ξ', l'))"
    shows "A ⊫_A (I →) onll Γ P"
  using assms(1-4) proof (rule seq_step_invariant_ctermI)
    fix p pp l ξ a q l' ξ'
    assume "p ∈ cterms Γ"
       and otherassms: "l ∈ labels Γ p"
           "((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"
           "((ξ, p), a, (ξ', q)) ∈ trans A"
           "l' ∈ labels Γ q"
           "(ξ, pp) ∈ reachable A I"
           "p ∈ sterms Γ pp"
           "(ξ', q) ∈ reachable A I"
           "I a"
    from this(1) obtain pn where "p ∈ ctermsl(Γ pn)"
                            and "not_call p"
      unfolding cterms_def' [OF wf] by auto
    with wf show "P ((ξ, l), a, (ξ', l'))"
      using otherassms by (rule step)
  qed

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 Γ"
      and "∃pn. p ∈ subterms (Γ pn)"
      and "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i"
    shows "∃pn. p' ∈ subterms (Γ pn)"
  using assms
```

```
  proof (induct p)
    fix p1 p2
    assume IH1: "∃pn. p1 ∈ subterms (Γ pn) ⟹
                   ((σ, p1), a, (σ', p')) ∈ oseqp_sos Γ i ⟹
                   ∃pn. p' ∈ subterms (Γ pn)"
       and IH2: "∃pn. p2 ∈ subterms (Γ pn) ⟹
                   ((σ, p2), a, (σ', p')) ∈ oseqp_sos Γ i ⟹
                   ∃pn. p' ∈ subterms (Γ pn)"
       and "∃pn. p1 ⊕ p2 ∈ subterms (Γ pn)"
       and "((σ, p1 ⊕ p2), a, (σ', p')) ∈ oseqp_sos Γ i"
    from ‹∃pn. p1 ⊕ p2 ∈ subterms (Γ pn)› obtain pn
                                              where "p1 ∈ subterms (Γ pn)"
                                                and "p2 ∈ subterms (Γ pn)" by auto
    from ‹((σ, p1 ⊕ p2), a, (σ', p')) ∈ oseqp_sos Γ i›
      have "((σ, p1), a, (σ', p')) ∈ oseqp_sos Γ i
            ∨ ((σ, p2), a, (σ', p')) ∈ oseqp_sos Γ i" by auto
    thus "∃pn. p' ∈ subterms (Γ pn)"
    proof
      assume "((σ, p1), a, (σ', p')) ∈ oseqp_sos Γ i"
      with ‹p1 ∈ subterms (Γ pn)› show ?thesis by (auto intro: IH1)
    next
      assume "((σ, p2), a, (σ', p')) ∈ oseqp_sos Γ i"
      with ‹p2 ∈ subterms (Γ pn)› show ?thesis by (auto intro: IH2)
    qed
  qed auto

lemma oreachable_subterms:
  assumes "wellformed Γ"
      and "control_within Γ (init A)"
      and "trans A = oseqp_sos Γ i"
      and "(σ, p) ∈ oreachable A S U"
    shows "∃pn. p ∈ subterms (Γ pn)"
  using assms(4)
  proof (induct rule: oreachable_pair_induct)
    fix σ p
    assume "(σ, p) ∈ init A"
    with ‹control_within Γ (init A)› show "∃pn. p ∈ subterms (Γ pn)" ..
  next
    fix σ p a σ' p'
    assume "(σ, p) ∈ oreachable A S U"
       and "∃pn. p ∈ subterms (Γ pn)"
       and 3: "((σ, p), a, (σ', p')) ∈ trans A"
       and "S σ σ' a"
    moreover from 3 and ‹trans A = oseqp_sos Γ i›
      have "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i" by simp
    ultimately show "∃pn. p' ∈ subterms (Γ pn)"
    using ‹wellformed Γ›
      by (auto elim: oseqp_sos_subterms)
  qed

lemma onl_oinvariantI [intro]:
  assumes init: "⋀σ p l. ⟦ (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' ⟧ ⟹ ∀l∈labels Γ p. P (σ', l)"
      and step: "⋀σ p a σ' p' l'.
                 ⟦ (σ, p) ∈ oreachable A S U;
                   ∀l∈labels Γ p. P (σ, l);
                   ((σ, p), a, (σ', p')) ∈ trans A;
                   l' ∈ labels Γ p';
                   S σ σ' a ⟧ ⟹ P (σ', l')"
    shows "A ⊨ (S, U →) onl Γ P"
  proof
    fix σ p
```

```
    assume "(σ, p) ∈ init A"
    hence "∀l∈labels Γ p. P (σ, l)" using init by simp
    thus "onl Γ P (σ, p)" ..
  next
    fix σ p a σ' p'
    assume rp: "(σ, p) ∈ oreachable A S U"
       and "onl Γ P (σ, p)"
       and tr: "((σ, p), a, (σ', p')) ∈ trans A"
       and "S σ σ' a"
    from ‹onl Γ P (σ, p)› have "∀l∈labels Γ p. P (σ, l)" ..
    with rp tr ‹S σ σ' a› have "∀l'∈labels Γ p'. P (σ', l')" by (auto elim: step)
    thus "onl Γ P (σ', p')" ..
  next
    fix σ σ' p
    assume "(σ, p) ∈ oreachable A S U"
       and "onl Γ P (σ, p)"
       and "U σ σ'"
    from ‹onl Γ P (σ, p)› have "∀l∈labels Γ p. P (σ, l)" by auto
    with ‹(σ, p) ∈ oreachable A S U› have "∀l∈labels Γ p. P (σ', l)"
        using ‹U σ σ'› by (rule other)
    thus "onl Γ P (σ', p)" by auto
  qed

lemma global_oinvariantI [intro]:
  assumes init: "⋀σ p. (σ, p) ∈ init A ⟹ P σ"
      and other: "⋀σ σ' p l. ⟦ (σ, p) ∈ oreachable A S U; P σ; U σ σ' ⟧ ⟹ P σ'"
      and step: "⋀σ p a σ' p'.
                  ⟦ (σ, p) ∈ oreachable A S U;
                    P σ;
                    ((σ, p), a, (σ', p')) ∈ trans A;
                    S σ σ' a ⟧ ⟹ P σ'"
    shows "A ⊨ (S, U →) (λ(σ, _). P σ)"
  proof
    fix σ p
    assume "(σ, p) ∈ init A"
    thus "(λ(σ, _). P σ) (σ, p)"
      by simp (erule init)
  next
    fix σ p a σ' p'
    assume rp: "(σ, p) ∈ oreachable A S U"
       and "(λ(σ, _). P σ) (σ, p)"
       and tr: "((σ, p), a, (σ', p')) ∈ trans A"
       and "S σ σ' a"
    from ‹(λ(σ, _). P σ) (σ, p)› have "P σ" by simp
    with rp have "P σ'"
      using tr ‹S σ σ' a› by (rule step)
    thus "(λ(σ, _). P σ) (σ', p')" by simp
  next
    fix σ σ' p
    assume "(σ, p) ∈ oreachable A S U"
       and "(λ(σ, _). P σ) (σ, p)"
       and "U σ σ'"
    hence "P σ'" by simp (erule other)
    thus "(λ(σ, _). P σ) (σ', p)" by simp
  qed

lemma onl_oinvariantD [dest]:
  assumes "A ⊨ (S, U →) onl Γ P"
      and "(σ, p) ∈ oreachable A S U"
      and "l ∈ labels Γ p"
    shows "P (σ, l)"
  using assms unfolding onl_def by auto

lemma onl_oinvariant_weakenD [dest]:
```

```
  assumes "A ⊨ (S', U' →) onl Γ P"
      and "(σ, p) ∈ oreachable A S U"
      and "l ∈ labels Γ p"
      and weakenS: "⋀s s' a. S s s' a ⟹ S' s s' a"
      and weakenU: "⋀s s'. U s s' ⟹ U' s s'"
    shows "P (σ, l)"
  proof -
    from ‹(σ, p) ∈ oreachable A S U› have "(σ, p) ∈ oreachable A S' U'"
      by (rule oreachable_weakenE)
         (erule weakenS, erule weakenU)
    with ‹A ⊨ (S', U' →) onl Γ P› show "P (σ, l)"
      using ‹l ∈ labels Γ p› ..
  qed

lemma onl_oinvariant_initD [dest]:
  assumes invP: "A ⊨ (S, U →) onl Γ P"
      and init: "(σ, p) ∈ init A"
      and pnl:  "l ∈ labels Γ p"
    shows "P (σ, l)"
  proof -
    from init have "(σ, p) ∈ oreachable A S U" ..
    with invP show ?thesis using pnl ..
  qed

lemma onl_oinvariant_sterms:
  assumes wf: "wellformed Γ"
      and il: "A ⊨ (S, U →) onl Γ P"
      and rp: "(σ, p) ∈ oreachable A S U"
      and "p'∈sterms Γ p"
      and "l∈labels Γ p'"
    shows "P (σ, l)"
  proof -
    from wf ‹p'∈sterms Γ p› ‹l∈labels Γ p'› have "l∈labels Γ p"
      by (rule labels_sterms_labels)
    with il rp show "P (σ, l)" ..
  qed

lemma onl_oinvariant_sterms_weaken:
  assumes wf: "wellformed Γ"
      and il: "A ⊨ (S', U' →) onl Γ P"
      and rp: "(σ, p) ∈ oreachable A S U"
      and "p'∈sterms Γ p"
      and "l∈labels 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 "P (σ, l)"
  proof -
    from ‹(σ, p) ∈ oreachable A S U› have "(σ, p) ∈ oreachable A S' U'"
      by (rule oreachable_weakenE)
         (erule weakenS, erule weakenU)
    with assms(1-2) show ?thesis using assms(4-5)
      by (rule onl_oinvariant_sterms)
  qed

lemma otrans_from_sterms:
  assumes "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
      and "wellformed Γ"
    shows "∃p'∈sterms Γ p. ((σ, p'), a, (σ', q)) ∈ oseqp_sos Γ i"
  using assms by (induction p rule: sterms_pinduct [OF ‹wellformed Γ›]) auto

lemma otrans_from_sterms':
  assumes "((σ, p'), a, (σ', q)) ∈ oseqp_sos Γ i"
      and "wellformed Γ"
      and "p' ∈ sterms Γ p"
```

```
    shows "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
  using assms by (induction p rule: sterms_pinduct [OF ‹wellformed Γ›]) auto

lemma otrans_to_dterms:
  assumes "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
      and "wellformed Γ"
   shows "∀r∈sterms Γ q. r ∈ dterms Γ p"
  using assms by (induction q) auto

theorem cterms_includes_sterms_of_oseq_reachable:
  assumes "wellformed Γ"
      and "control_within Γ (init A)"
      and "trans A = oseqp_sos Γ i"
    shows "⋃(sterms Γ ` snd ` oreachable A S U) ⊆ cterms Γ"
  proof
    fix qs
    assume "qs ∈ ⋃(sterms Γ ` snd ` oreachable A S U)"
    then obtain ξ and q where  *: "(ξ, q) ∈ oreachable A S U"
                          and **: "qs ∈ sterms Γ q" by auto
    from * have "⋀x. x ∈ sterms Γ q ⟹ x ∈ cterms Γ"
    proof (induction rule: oreachable_pair_induct)
      fix σ p q
      assume "(σ, p) ∈ init A"
         and "q ∈ sterms Γ p"
      from ‹control_within Γ (init A)› and ‹(σ, p) ∈ init A›
        obtain pn where "p ∈ subterms (Γ pn)" by auto
      with ‹wellformed Γ› show "q ∈ cterms Γ" using ‹q∈sterms Γ p›
        by (rule subterms_sterms_in_cterms)
    next
      fix p σ a σ' q x
      assume "(σ, p) ∈ oreachable A S U"
         and IH: "⋀x. x ∈ sterms Γ p ⟹ x ∈ cterms Γ"
         and "((σ, p), a, (σ', q)) ∈ trans A"
         and "x ∈ sterms Γ q"
      from this(3) and ‹trans A = oseqp_sos Γ i›
        have step: "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i" by simp
      from step ‹wellformed Γ› obtain ps
        where ps: "ps ∈ sterms Γ p"
          and step': "((σ, ps), a, (σ', q)) ∈ oseqp_sos Γ i"
        by (rule otrans_from_sterms [THEN bexE])
      from ps have "ps ∈ cterms Γ" by (rule IH)
      moreover from step' ‹wellformed Γ› ‹x ∈ sterms Γ q› have "x ∈ dterms Γ ps"
        by (rule otrans_to_dterms [rule_format])
      ultimately show "x ∈ cterms Γ" by (rule ctermsDI)
    qed
    thus "qs ∈ cterms Γ" using ** .
  qed

corollary oseq_reachable_in_cterms:
  assumes "wellformed Γ"
      and "control_within Γ (init A)"
      and "trans A = oseqp_sos Γ i"
      and "(σ, p) ∈ oreachable A S U"
      and "p' ∈ sterms Γ p"
    shows "p' ∈ cterms Γ"
  using assms(1-3)
  proof (rule cterms_includes_sterms_of_oseq_reachable [THEN subsetD])
    from assms(4-5) show "p' ∈ ⋃(sterms Γ ` snd ` oreachable A S U)"
      by (auto elim!: rev_bexI)
  qed

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: "⋀σ p l. ⟦
                 (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: "⋀p l σ a q l' σ' pp. ⟦
               p∈cterms Γ;
               l∈labels Γ p;
               P (σ, l);
               ((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∈sterms Γ pp;
               (σ', q)∈oreachable A S U;
               S σ σ' a
             ⟧ ⟹ P (σ', l')"
  shows "A ⊨ (S, U →) onl Γ P"
proof
    fix σ p l
  assume "(σ, p) ∈ init A"
    and *: "l ∈ labels Γ p"
  with init show "P (σ, l)" by auto
next
    fix σ p a σ' q l'
  assume sr: "(σ, p) ∈ oreachable A S U"
    and pl: "∀l∈labels Γ p. P (σ, l)"
    and tr: "((σ, p), a, (σ', q)) ∈ trans A"
    and A6: "l' ∈ labels Γ q"
    and "S σ σ' a"
  thus "P (σ', l')"
  proof -
    from sr and tr and ‹S σ σ' a› have A7: "(σ', q) ∈ oreachable A S U"
      by - (rule oreachable_local')
    from tr and sp have tr': "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i" by simp
    then obtain p' where "p' ∈ sterms Γ p"
                     and A4: "((σ, p'), a, (σ', q)) ∈ oseqp_sos Γ i"
      by (blast dest: otrans_from_sterms [OF _ wf])
    from wf cw sp sr this(1) have A1: "p'∈cterms Γ"
      by (rule oseq_reachable_in_cterms)
    from labels_not_empty [OF wf] obtain ll where A2: "ll∈labels Γ p'"
        by blast
    with ‹p'∈sterms Γ p› have "ll∈labels Γ p"
      by (rule labels_sterms_labels [OF wf])
    with pl have A3: "P (σ, ll)" by simp
    from sr ‹p'∈sterms Γ p›
      obtain pp where A7: "(σ, pp)∈oreachable A S U"
                  and A8: "p'∈sterms Γ pp"
      by auto
    from sr tr ‹S σ σ' a› have A9: "(σ', q)∈oreachable A S U"
      by - (rule oreachable_local')
    from sp and ‹((σ, p'), a, (σ', q)) ∈ oseqp_sos Γ i›
      have A5: "((σ, p'), a, (σ', q)) ∈ trans A" by simp
    from A1 A2 A3 A4 A5 A6 A7 A8 A9 ‹S σ σ' a› show ?thesis by (rule local)
  qed
next
  fix σ σ' p l
  assume sr: "(σ, p) ∈ oreachable A S U"
```

```
      and "∀l∈labels Γ p. P (σ, l)"
      and "U σ σ'"
    show "∀l∈labels Γ p. P (σ', l)"
    proof
      fix l
      assume "l∈labels Γ p"
      with ‹∀l∈labels Γ p. P (σ, l)› have "P (σ, l)" ..
      with sr and ‹l∈labels Γ p›
        show "P (σ', l)" using ‹U σ σ'› by (rule other)
    qed
  qed

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: "⋀σ p l. ⟦
                  (σ, p) ∈ init A;
                  l∈labels Γ p
                ⟧ ⟹ P (σ, l)"
      and other: "⋀σ σ' p l. ⟦
                  wellformed Γ;
                  (σ, p) ∈ oreachable A S U;
                  l∈labels Γ p;
                  P (σ, l);
                  U σ σ' ⟧ ⟹ P (σ', l)"
      and local: "⋀p l σ a q l' σ' pp pn. ⟦
                wellformed Γ;
                p∈ctermsl (Γ pn);
                not_call p;
                l∈labels Γ p;
                P (σ, l);
                ((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∈sterms Γ pp;
                (σ', q)∈oreachable A S U;
                S σ σ' a
              ⟧ ⟹ P (σ', l')"
    shows "A ⊨ (S, U →) onl Γ P"
  proof (rule oseq_invariant_ctermI [OF wf cw sl sp])
    fix σ p l
    assume "(σ, p) ∈ init A"
      and "l ∈ labels Γ p"
    thus "P (σ, l)" by (rule init)
  next
    fix σ σ' p l
    assume "(σ, p) ∈ oreachable A S U"
      and "l ∈ labels Γ p"
      and "P (σ, l)"
      and "U σ σ'"
    with wf show "P (σ', l)" by (rule other)
  next
    fix p l σ a q l' σ' pp
    assume "p ∈ cterms Γ"
      and otherassms: "l ∈ labels Γ p"
        "P (σ, l)"
        "((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 ∈ sterms Γ pp"
```

```
        "(σ', q) ∈ oreachable A S U"
        "S σ σ' a"
  from this(1) obtain pn where "p ∈ ctermsl(Γ pn)"
                         and "not_call p"
    unfolding cterms_def' [OF wf] by auto
  with wf show "P (σ', l')"
    using otherasms by (rule local)
qed
```

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
    fix σ p σ' p' a
    assume "(σ, p) ∈ oreachable A S U"
       and "((σ, p), a, (σ', p')) ∈ trans A"
       and "S σ σ' a"
    hence "∀l∈labels Γ p. ∀l'∈labels Γ p'. P ((σ, l), a, (σ', l'))" by (auto elim!: *)
    thus "onll Γ P ((σ, p), a, (σ', p'))" ..
  qed

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 -
    from assms(1-4) have "onll Γ P ((σ, p), a, (σ', p'))" ..
    with lp lp' show "P ((σ, l), a, (σ', l'))" by auto
  qed

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'))"
  using assms by auto

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 -
    from ‹(σ, p) ∈ oreachable A S U› have "(σ, p) ∈ oreachable A S' U'"
      by (rule oreachable_weakenE)
         (erule weakenS, erule weakenU)
    with ‹A ⊨_A (S', U' →) onll Γ P› show ?thesis
      using ‹((σ, p), a, (σ', p')) ∈ trans A› and ‹S' σ σ' a› ..
  qed
```

```
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
    fix σ p l
    assume "(σ, p) ∈ init A" and "l∈labels Γ p"
      thus "P (σ, l)" by (rule init)
  next
    fix σ p a σ' p' l'
    assume sr: "(σ, p) ∈ oreachable A S U"
       and lp: "∀l∈labels Γ p. P (σ, l)"
       and tr: "((σ, p), a, (σ', p')) ∈ trans A"
       and "S σ σ' a"
       and lp': "l' ∈ labels Γ p'"
      show "P (σ', l')"
    proof -
      from lp obtain l where "l∈labels Γ p" and "P (σ, l)"
        using labels_not_empty [OF wf] by auto
      from sinv sr tr ‹S σ σ' a› this(1) lp' have "Q ((σ, l), a, (σ', l'))" ..
      with sr ‹l∈labels Γ p› ‹P (σ, l)› show "P (σ', l')" using ‹S σ σ' a› by (rule local)
    qed
  next
    fix σ σ' p l
    assume "(σ, p) ∈ oreachable A S U"
       and "∀l∈labels Γ p. P (σ, l)"
       and "U σ σ'"
      show "∀l∈labels Γ p. P (σ', l)"
    proof
      fix l
      assume "l∈labels Γ p"
      with ‹∀l∈labels Γ p. P (σ, l)› have "P (σ, l)" ..
      with ‹(σ, p) ∈ oreachable A S U› and ‹l∈labels Γ p›
      show "P (σ', l)" using ‹U σ σ'› by (rule other)
    qed
  qed

lemma onll_ostep_invariant_sterms:
  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'∈sterms Γ p"
      and "l∈labels Γ p'"
    shows "P ((σ, l), a, (σ', l'))"
  proof -
    from wf ‹p'∈sterms Γ p› ‹l∈labels Γ p'› have "l∈labels Γ p"
      by (rule labels_sterms_labels)
    with si sr sos ‹S σ σ' a› show "P ((σ, l), a, (σ', l'))" using ‹l'∈labels Γ q› ..
```

```
  qed

lemma oseq_step_invariant_sterms:
  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'∈sterms Γ p"
    shows "∀l∈labels Γ p'. P ((σ, l), a, (σ', l'))"
  proof
    from assms(3, 6) have "((σ, p'), a, (σ', q)) ∈ oseqp_sos Γ i" by simp
    hence "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i"
      using wf ‹p'∈sterms Γ p›  by (rule otrans_from_sterms')
    with assms(3) have trp: "((σ, p), a, (σ', q)) ∈ trans A" by simp
    fix l assume "l ∈ labels Γ p'"
    with wf inv sr trp ‹S σ σ' a› ‹l'∈labels Γ q› ‹p'∈sterms Γ p›
      show "P ((σ, l), a, (σ', l'))"
        by - (erule(7) onll_ostep_invariant_sterms)
  qed

lemma oseq_step_invariant_sterms_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'∈sterms 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 -
    from ‹S' σ σ' a› have "S σ σ' a" by (rule weakenS)
    from ‹(σ, p) ∈ oreachable A S' U'›
      have Ir: "(σ, p) ∈ oreachable A S U"
        by (rule oreachable_weakenE)
           (erule weakenS, erule weakenU)
    with assms(1-4) show ?thesis
      using tr ‹S σ σ' a› ‹p'∈sterms Γ p›
      by (rule oseq_step_invariant_sterms)
  qed

lemma onll_ostep_invariant_any_sterms:
  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'∈sterms Γ p. ∀l∈labels Γ p'. P ((σ, l), a, (σ', l'))"
  by (intro ballI) (rule onll_ostep_invariant_sterms [OF assms])

lemma oseq_step_invariant_ctermI [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∈cterms 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∈sterms 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
      fix σ p l a σ' q l'
    assume sr: "(σ, p) ∈ oreachable A S U"
      and tr: "((σ, p), a, (σ', q)) ∈ trans A"
      and "S σ σ' a"
      and pl: "l ∈ labels Γ p"
      and A5: "l' ∈ labels Γ q"
    from this(2) and sp have "((σ, p), a, (σ', q)) ∈ oseqp_sos Γ i" by simp
    then obtain p' where "p' ∈ sterms Γ p"
                   and A3: "((σ, p'), a, (σ', q)) ∈ oseqp_sos Γ i"
      by (blast dest: otrans_from_sterms [OF _ wf])
    from this(2) and sp have A4: "((σ, p'), a, (σ', q)) ∈ trans A" by simp
    from wf cw sp sr ‹p'∈sterms Γ p› have A1: "p'∈cterms Γ"
      by (rule oseq_reachable_in_cterms)
    from sr ‹p'∈sterms Γ p›
      obtain pp where A6: "(σ, pp)∈oreachable A S U"
                and A7: "p'∈sterms Γ pp"
      by auto
    from sr tr ‹S σ σ' a› have A8: "(σ', q)∈oreachable A S U"
      by - (erule(2) oreachable_local')
    from wf cw sp sr have "∃pn. p ∈ subterms (Γ pn)"
      by (rule oreachable_subterms)
    with sl wf have "∀p'∈sterms Γ p. l ∈ labels Γ p'"
      using pl by (rule simple_labels_in_sterms)
    with ‹p' ∈ sterms Γ p› have "l ∈ labels Γ p'" by simp
    with A1 show "P ((σ, l), a, (σ', l'))" using A3 A4 A5 A6 A7 A8 ‹S σ σ' a›
      by (rule local)
  qed

lemma oseq_step_invariant_ctermsI [intro]:
  assumes wf: "wellformed Γ"
      and "control_within Γ (init A)"
      and "simple_labels Γ"
      and "trans A = oseqp_sos Γ i"
      and local: "⋀p l σ a q l' σ' pp pn. ⟦
                wellformed Γ;
                p∈ctermsl (Γ pn);
                not_call p;
                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∈sterms 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"
  using assms(1-4) proof (rule oseq_step_invariant_ctermI)
    fix p l σ a q l' σ' pp
    assume "p ∈ cterms Γ"
      and otherassms: "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 ∈ sterms Γ pp"
        "(σ', q) ∈ oreachable A S U"
        "S σ σ' a"
  from this(1) obtain pn where "p ∈ ctermsl(Γ pn)"
                         and "not_call p"
    unfolding cterms_def' [OF wf] by auto
  with wf show "P ((σ, l), a, (σ', l'))"
    using otherassms by (rule local)
qed

lemma open_seqp_action [elim]:
 assumes "wellformed Γ"
     and "((σ i, p), a, (σ' i, p')) ∈ seqp_sos Γ"
   shows "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i"
 proof -
   from assms obtain ps where "ps∈sterms Γ p"
                        and "((σ i, ps), a, (σ' i, p')) ∈ seqp_sos Γ"
     by - (drule trans_from_sterms, auto)
   thus ?thesis
   proof (induction p)
     fix p1 p2
     assume "⟦ ps ∈ sterms Γ p1; ((σ i, ps), a, σ' i, p') ∈ seqp_sos Γ ⟧
             ⟹ ((σ, p1), a, (σ', p')) ∈ oseqp_sos Γ i"
        and "⟦ ps ∈ sterms Γ p2; ((σ i, ps), a, σ' i, p') ∈ seqp_sos Γ ⟧
             ⟹ ((σ, p2), a, (σ', p')) ∈ oseqp_sos Γ i"
        and "ps ∈ sterms Γ (p1 ⊕ p2)"
        and "((σ i, ps), a, (σ' i, p')) ∈ seqp_sos Γ"
     with assms(1) show "((σ, p1 ⊕ p2), a, (σ', p')) ∈ oseqp_sos Γ i"
       by simp (metis oseqp_sos.ochoiceT1 oseqp_sos.ochoiceT2)
   next
     fix l fip fmsg p1 p2
     assume IH1: "⟦ ps ∈ sterms Γ p1; ((σ i, ps), a, σ' i, p') ∈ seqp_sos Γ ⟧
                 ⟹ ((σ, p1), a, (σ', p')) ∈ oseqp_sos Γ i"
        and IH2: "⟦ ps ∈ sterms Γ p2; ((σ i, ps), a, σ' i, p') ∈ seqp_sos Γ ⟧
                 ⟹ ((σ, p2), a, (σ', p')) ∈ oseqp_sos Γ i"
        and "ps ∈ sterms Γ ({l}unicast(fip, fmsg). p1 ▷ p2)"
        and "((σ i, ps), a, (σ' i, p')) ∈ seqp_sos Γ"
     from this(3-4) have "((σ i, {l}unicast(fip, fmsg). p1 ▷ p2), a, (σ' i, p')) ∈ seqp_sos Γ"
       by simp
     thus "((σ, {l}unicast(fip, fmsg). p1 ▷ p2), a, (σ', p')) ∈ oseqp_sos Γ i"
     proof (rule seqp_unicastTE)
       assume "a = unicast (fip (σ i)) (fmsg (σ i))"
          and "σ' i = σ i"
          and "p' = p1"
       thus ?thesis by auto
     next
       assume "a = ¬unicast (fip (σ i))"
          and "σ' i = σ i"
          and "p' = p2"
       thus ?thesis by auto
     qed
   next
     fix p
     assume "ps ∈ sterms Γ (call(p))"
        and "((σ i, ps), a, (σ' i, p')) ∈ seqp_sos Γ"
     with assms(1) have "((σ, ps), a, (σ', p')) ∈ oseqp_sos Γ i"
       by (cases ps) auto
     with assms(1) ‹ps ∈ sterms Γ (call(p))› have "((σ, Γ p), a, (σ', p')) ∈ oseqp_sos Γ i"
       by - (rule otrans_from_sterms', simp_all)
     thus "((σ, call(p)), a, (σ', p')) ∈ oseqp_sos Γ i" by auto
   qed auto
 qed
```

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 ⇒ (('i ⇒ 'g) × 'l) set ⇒ ('g × 'l) set ⇒ bool"
where "initiali i OI CI ≡ ({(σ i, p)|σ p. (σ, p) ∈ OI} = CI)"

lemma initialiI [intro]:
  assumes OICI: "⋀σ p. (σ, p) ∈ OI ⟹ (σ i, p) ∈ CI"
      and CIOI: "⋀ξ p. (ξ, p) ∈ CI ⟹ ∃σ. ξ = σ i ∧ (σ, p) ∈ OI"
    shows "initiali i OI CI"
  unfolding initiali_def
  by (intro set_eqI iffI) (auto elim!: OICI CIOI)

lemma open_from_initialiD [dest]:
  assumes "initiali i OI CI"
      and "(σ, p) ∈ OI"
    shows "∃ξ. σ i = ξ ∧ (ξ, p) ∈ CI"
  using assms unfolding initiali_def by auto

lemma closed_from_initialiD [dest]:
  assumes "initiali i OI CI"
      and "(ξ, p) ∈ CI"
    shows "∃σ. σ i = ξ ∧ (σ, p) ∈ OI"
  using assms unfolding initiali_def by auto

definition
  seql :: "'i ⇒ (('s × 'l) ⇒ bool) ⇒ (('i ⇒ 's) × 'l) ⇒ bool"
where
  "seql i P ≡ (λ(σ, p). P (σ i, p))"

lemma seqlI [intro]:
  "P (fst s i, snd s) ⟹ seql i P s"
  by (clarsimp simp: seql_def)

lemma same_seql [elim]:
  assumes "∀j∈{i}. σ' j = σ j"
      and "seql i P (σ', s)"
    shows "seql i P (σ, s)"
  using assms unfolding seql_def by (clarsimp)

lemma seqlsimp:
  "seql i P (σ, p) = P (σ i, p)"
  unfolding seql_def by simp

lemma other_steps_resp_local [intro!, simp]: "other_steps (other A I) I"
  by (clarsimp elim!: otherE)

lemma seql_onl_swap:
  "seql i (onl Γ P) = onl Γ (seql i P)"
  unfolding seql_def onl_def by simp

lemma oseqp_sos_resp_local_steps [intro!, simp]:
  fixes Γ :: "'p ⇒ ('s, 'm, 'p, 'l) seqp"
  shows "local_steps (oseqp_sos Γ i) {i}"
  proof
    fix σ σ' ζ ζ' :: "nat ⇒ 's" and s a s'
    assume tr: "((σ, s), a, σ', s') ∈ oseqp_sos Γ i"
       and "∀j∈{i}. ζ j = σ j"
```

```
    thus "∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, s), a, (ζ', s')) ∈ oseqp_sos Γ i"
    proof induction
      fix σ σ' l ms p
      assume "σ' i = σ i"
         and "∀j∈{i}. ζ j = σ j"
      hence "((ζ, {l}broadcast(ms).p), broadcast (ms (σ i)), (σ', p)) ∈ oseqp_sos Γ i"
        by (metis obroadcastT singleton_iff)
      with ‹∀j∈{i}. ζ j = σ j› show "∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧
           ((ζ, {l}broadcast(ms).p), broadcast (ms (σ i)), (ζ', p)) ∈ oseqp_sos Γ i"
        by blast
    next
      fix σ σ' :: "nat ⇒ 's" and fmsg :: "'m ⇒ 's ⇒ 's" and msg l p
      assume *:  "σ' i = fmsg msg (σ i)"
         and **: "∀j∈{i}. ζ j = σ j"
      hence "∀j∈{i}. (ζ(i := fmsg msg (ζ i))) j = σ' j" by clarsimp
      moreover from * **
        have "((ζ, {l}receive(fmsg).p), receive msg, (ζ(i := fmsg msg (ζ i)), p)) ∈ oseqp_sos Γ i"
        by (metis fun_upd_same oreceiveT)
      ultimately show "∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧
                              ((ζ, {l}receive(fmsg).p), receive msg, (ζ', p)) ∈ oseqp_sos Γ i"
        by blast
    next
      fix σ' σ l p and fas :: "'s ⇒ 's"
      assume *:  "σ' i = fas (σ i)"
         and **: "∀j∈{i}. ζ j = σ j"
      hence "∀j∈{i}. (ζ(i := fas (ζ i))) j = σ' j" by clarsimp
      moreover from * ** have "((ζ, {l}⟦fas⟧ p), τ, (ζ(i := fas (ζ i)), p)) ∈ oseqp_sos Γ i"
        by (metis fun_upd_same oassignT)
      ultimately show "∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, {l}⟦fas⟧ p), τ, (ζ', p)) ∈ oseqp_sos Γ i"
        by blast
    next
      fix g :: "'s ⇒ 's set" and σ σ' l p
      assume *:  "σ' i ∈ g (σ i)"
         and **: "∀j∈{i}. ζ j = σ j"
      hence "∀j∈{i}. (SOME ζ'. ζ' i = σ' i) j = σ' j" by simp (metis (lifting, full_types) some_eq_ex)
      moreover with * ** have "((ζ, {l}⟨g⟩ p), τ, (SOME ζ'. ζ' i = σ' i, p)) ∈ oseqp_sos Γ i"
        by simp (metis oguardT step_seq_tau)
      ultimately show "∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, {l}⟨g⟩ p), τ, (ζ', p)) ∈ oseqp_sos Γ i"
        by blast
    next
      fix σ pn a σ' p'
      assume "((σ, Γ pn), a, (σ', p')) ∈ oseqp_sos Γ i"
         and IH: "∀j∈{i}. ζ j = σ j ⟹ ∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, Γ pn), a, (ζ', p')) ∈ oseqp_sos
  Γ i"
         and "∀j∈{i}. ζ j = σ j"
      then obtain ζ' where "∀j∈{i}. ζ' j = σ' j"
                     and "((ζ, Γ pn), a, (ζ', p')) ∈ oseqp_sos Γ i"
        by blast
      thus "∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, call(pn)), a, (ζ', p')) ∈ oseqp_sos Γ i"
        by blast
    next
      fix σ p a σ' p' q
      assume "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i"
         and "∀j∈{i}. ζ j = σ j ⟹ ∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, p), a, (ζ', p')) ∈ oseqp_sos Γ
  i"
         and "∀j∈{i}. ζ j = σ j"
      then obtain ζ' where "∀j∈{i}. ζ' j = σ' j"
                     and "((ζ, p), a, (ζ', p')) ∈ oseqp_sos Γ i"
        by blast
      thus "∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, p ⊕ q), a, (ζ', p')) ∈ oseqp_sos Γ i"
        by blast
    next
      fix σ p a σ' q q'
      assume "((σ, q), a, (σ', q')) ∈ oseqp_sos Γ i"
```

```
        and "∀j∈{i}. ζ j = σ j ⟹ ∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, q), a, (ζ', q')) ∈ oseqp_sos Γ
i"
        and "∀j∈{i}. ζ j = σ j"
      then obtain ζ' where "∀j∈{i}. ζ' j = σ' j"
                       and "((ζ, q), a, (ζ', q')) ∈ oseqp_sos Γ i"
        by blast
      thus "∃ζ'. (∀j∈{i}. ζ' j = σ' j) ∧ ((ζ, p ⊕ q), a, (ζ', q')) ∈ oseqp_sos Γ i"
        by blast
    qed (simp_all, (metis ogroupcastT ounicastT onotunicastT osendT odeliverT)+)
  qed

lemma oseqp_sos_subreachable [intro!, simp]:
  assumes "trans OA = oseqp_sos Γ i"
    shows "subreachable OA (other ANY {i}) {i}"
  by rule (clarsimp simp add: assms(1))+

lemma oseq_step_is_seq_step:
    fixes σ :: "ip ⇒ 's"
  assumes "((σ, p), a :: 'm seq_action, (σ', p')) ∈ oseqp_sos Γ i"
      and "σ i = ξ"
    shows "∃ξ'. σ' i = ξ' ∧ ((ξ, p), a, (ξ', p')) ∈ seqp_sos Γ"
  using assms proof induction
    fix σ σ' l ms p
    assume "σ' i = σ i"
      and "σ i = ξ"
    hence "σ' i = ξ" by simp
    have "((ξ, {l}broadcast(ms).p), broadcast (ms ξ), (ξ, p)) ∈ seqp_sos Γ"
      by auto
    with ‹σ i = ξ› and ‹σ' i = ξ› show "∃ξ'. σ' i = ξ'
            ∧ ((ξ, {l}broadcast(ms).p), broadcast (ms (σ i)), (ξ', p)) ∈ seqp_sos Γ"
      by clarsimp
  next
    fix fmsg :: "'m ⇒ 's ⇒ 's" and msg :: 'm and σ' σ l p
    assume "σ' i = fmsg msg (σ i)"
      and "σ i = ξ"
    have "((ξ, {l}receive(fmsg).p), receive msg, (fmsg msg ξ, p)) ∈ seqp_sos Γ"
      by auto
    with ‹σ' i = fmsg msg (σ i)› and ‹σ i = ξ›
      show "∃ξ'. σ' i = ξ' ∧ ((ξ, {l}receive(fmsg).p), receive msg, (ξ', p)) ∈ seqp_sos Γ"
        by clarsimp
  qed (simp_all, (metis assignT choiceT1 choiceT2 groupcastT guardT
                      callT unicastT notunicastT sendT deliverT step_seq_tau)+)

lemma reachable_oseq_seqp_sos:
  assumes "(σ, p) ∈ reachable OA I"
      and "initiali i (init OA) (init A)"
      and spo: "trans OA = oseqp_sos Γ i"
      and sp: "trans A = seqp_sos Γ"
      shows "∃ξ. σ i = ξ ∧ (ξ, p) ∈ reachable A I"
  using assms(1) proof (induction rule: reachable_pair_induct)
    fix σ p
    assume "(σ, p) ∈ init OA"
    with ‹initiali i (init OA) (init A)› obtain ξ where "σ i = ξ"
                                                  and "(ξ, p) ∈ init A"
      by auto
    from ‹(ξ, p) ∈ init A› have "(ξ, p) ∈ reachable A I" ..
    with ‹σ i = ξ› show "∃ξ. σ i = ξ ∧ (ξ, p) ∈ reachable A I"
      by auto
  next
    fix σ p σ' p' a
    assume "(σ, p) ∈ reachable OA I"
      and IH: "∃ξ. σ i = ξ ∧ (ξ, p) ∈ reachable A I"
      and otr: "((σ, p), a, (σ', p')) ∈ trans OA"
      and "I a"
```

```
    from IH obtain ξ where "σ i = ξ"
                  and cr: "(ξ, p) ∈ reachable A I"
      by clarsimp
    from otr and spo have "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i" by simp
    with ‹σ i = ξ› obtain ξ' where "σ' i = ξ'"
                          and "((ξ, p), a, (ξ', p')) ∈ seqp_sos Γ"
        by (auto dest!: oseq_step_is_seq_step)
    from this(2) and sp have ctr: "((ξ, p), a, (ξ', p')) ∈ trans A" by simp
    from ‹(ξ, p) ∈ reachable A I› and ctr and ‹I a›
      have "(ξ', p') ∈ reachable A I" ..
    with ‹σ' i = ξ'› show "∃ξ. σ' i = ξ ∧ (ξ, p') ∈ reachable A I"
      by blast
  qed

lemma reachable_oseq_seqp_sos':
  assumes "s ∈ reachable OA I"
      and "initiali i (init OA) (init A)"
      and "trans OA = oseqp_sos Γ i"
      and "trans A = seqp_sos Γ"
    shows "∃ξ. (fst s) i = ξ ∧ (ξ, snd s) ∈ reachable A I"
  using assms
  by - (cases s, auto dest: reachable_oseq_seqp_sos)
```

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 ||= (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 |= (act I, other ANY {i} →) (seql i P)"
  proof -
    have "OA ||= (I →) (seql i P)"
      proof (rule invariant_arbitraryI)
        fix s
        assume "s ∈ reachable OA I"
        with ‹initiali i (init OA) (init A)› obtain ξ where "(fst s) i = ξ"
                                                     and "(ξ, snd s) ∈ reachable A I"
          by (auto dest: reachable_oseq_seqp_sos' [OF _ _ spo sp])
        with ‹A ||= (I →) P› have "P (ξ, snd s)" by auto
        with ‹(fst s) i = ξ› show "seql i P s" by auto
      qed
    moreover from spo have "subreachable OA (other ANY {i}) {i}" ..
    ultimately show ?thesis
    proof (rule open_closed_invariant)
      fix σ σ' s
      assume "∀j∈{i}. σ' j = σ j"
         and "seql i P (σ', s)"
      thus "seql i P (σ, s)" ..
    qed
  qed

definition
  seqll :: "'i ⇒ ((('s × 'l) × 'a × ('s × 'l)) ⇒ bool)
              ⇒ ((('i ⇒ 's) × 'l) × 'a × (('i ⇒ 's) × 'l)) ⇒ bool"
where
  "seqll i P ≡ (λ((σ, p), a, (σ', p')). P ((σ i, p), a, (σ' i, p')))"

lemma same_seqll [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'))"
  using assms unfolding seqll_def by (clarsimp)
```

```
lemma seqllI [intro!]:
  assumes "P ((σ i, p), a, (σ' i, p'))"
    shows "seqll i P ((σ, p), a, (σ', p'))"
  using assms unfolding seqll_def by simp

lemma seqllD [dest]:
  assumes "seqll i P ((σ, p), a, (σ', p'))"
    shows "P ((σ i, p), a, (σ' i, p'))"
  using assms unfolding seqll_def by simp

lemma seqllsimp:
  "seqll i P ((σ, p), a, (σ', p')) = P ((σ i, p), a, (σ' i, p'))"
  unfolding seqll_def by simp

lemma seqll_onll_swap:
  "seqll i (onll Γ P) = onll Γ (seqll i P)"
  unfolding seqll_def onll_def by simp

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 -
    have "OA ⊫A (I →) (seqll i P)"
    proof (rule step_invariant_arbitraryI)
      fix σ p a σ' p'
      assume or: "(σ, p) ∈ reachable OA I"
         and otr: "((σ, p), a, (σ', p')) ∈ trans OA"
         and "I a"
      from or ‹initiali i (init OA) (init A)› spo sp obtain ξ where "σ i = ξ"
                                                               and cr: "(ξ, p) ∈ reachable A I"
        by - (drule(3) reachable_oseq_seqp_sos', auto)
      from otr and spo have "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i" by simp
      with ‹σ i = ξ› obtain ξ' where "σ' i = ξ'"
                                 and ctr: "((ξ, p), a, (ξ', p')) ∈ seqp_sos Γ"
        by (auto dest!: oseq_step_is_seq_step)
      with sp have "((ξ, p), a, (ξ', p')) ∈ trans A" by simp
      with ‹A ⊫A (I →) P› cr have "P ((ξ, p), a, (ξ', p'))" using ‹I a› ..
      with ‹σ i = ξ› and ‹σ' i = ξ'› have "P ((σ i, p), a, (σ' i, p'))" by simp
      thus "seqll i P ((σ, p), a, (σ', p'))" ..
    qed
    moreover from spo have "local_steps (trans OA) {i}" by simp
    moreover have "other_steps (other ANY {i}) {i}" ..
    ultimately show ?thesis
    proof (rule open_closed_step_invariant)
      fix σ ζ a σ' ζ' s s'
      assume "∀j∈{i}. σ j = ζ j"
         and "∀j∈{i}. σ' j = ζ' j"
         and "seqll i P ((σ, s), a, (σ', s'))"
        thus "seqll i P ((ζ, s), a, (ζ', s'))" ..
    qed
  qed

end
```

22 Model the standard queuing model

```
theory Qmsg
imports AWN_SOS_Labels AWN_Invariants
begin
```

Define the queue process

```
fun ΓQMSG :: "('m 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 ≠ {}"
  unfolding σQMSG_def by simp

lemma σQMSG_exists [simp]: "∃qmsg q. (qmsg, q) ∈ σQMSG"
  unfolding σQMSG_def by simp

lemma qmsg_wf [simp]: "wellformed ΓQMSG"
  by (rule wf_no_direct_calls) auto

lemmas qmsg_labels_not_empty [simp] = labels_not_empty [OF qmsg_wf]

lemma qmsg_control_within [simp]: "control_within ΓQMSG (init qmsg)"
  unfolding σQMSG_def by (rule control_withinI) (auto simp del: ΓQMSG_simps)

lemma qmsg_simple_labels [simp]: "simple_labels ΓQMSG"
  unfolding simple_labels_def by auto

lemma qmsg_trans: "trans qmsg = seqp_sos ΓQMSG"
  by simp

lemma σQMSG_labels [simp]: "(ξ, q) ∈ σQMSG ⟹ labels ΓQMSG q = {()-:0}"
  unfolding σQMSG_def by simp

lemma qmsg_proc_cases [dest]:
  fixes p pn
  shows "p ∈ ctermsl (ΓQMSG pn) ⟹ p ∈ ctermsl (ΓQMSG ())"
  by simp

declare
  ΓQMSG_simps [cterms_env]
  qmsg_proc_cases [ctermsl_cases]
  seq_invariant_ctermsI [OF qmsg_wf qmsg_control_within qmsg_simple_labels qmsg_trans, cterms_intros]
  seq_step_invariant_ctermsI [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 σ s a σ' s'
```

```
  assumes "((σ, s), a, (σ', s')) ∈ oseqp_sos Γ i"
  shows "case a of
           broadcast m      ⇒ σ' i = σ i
         | groupcast ips m ⇒ σ' i = σ i
         | unicast ips m    ⇒ σ' i = σ i
         | ¬unicast ips     ⇒ σ' i = σ i
         | send m           ⇒ σ' i = σ i
         | deliver m        ⇒ σ' i = σ i
         | _ ⇒ True"
  using assms by induction simp_all

lemma qmsg_no_change_on_send_or_receive:
    fixes σ s a σ' s'
  assumes "((σ, s), a, (σ', s')) ∈ oparp_sos i (oseqp_sos Γ i) (seqp_sos Γ_QMSG)"
      and "a ≠ τ"
    shows "σ' i = σ i"
  proof -
    from assms(1) obtain p q p' q'
      where "((σ, (p, q)), a, (σ', (p', q'))) ∈ oparp_sos i (oseqp_sos Γ i) (seqp_sos Γ_QMSG)"
      by (cases s, cases s', simp)
    thus ?thesis
    proof
      assume "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i"
         and "⋀m. a ≠ receive m"
      with ‹a ≠ τ› show "σ' i = σ i"
        by - (drule oseq_no_change_on_send, cases a, auto)
    next
      assume "(q, a, q') ∈ seqp_sos Γ_QMSG"
         and "σ' i = σ i"
        thus "σ' i = σ i" by simp
    next
      assume "a = τ" with ‹a ≠ τ› show ?thesis by auto
    qed
  qed

lemma qmsg_msgs_not_empty:
  "qmsg ⊨ onl Γ_QMSG (λ(msgs, l). l = ()-:1 ⟶ msgs ≠ [])"
  by inv_cterms

lemma qmsg_send_from_queue:
  "qmsg ⊨_A (λ((msgs, q), a, _). sendmsg (λm. m∈set msgs) a)"
  proof -
    have "qmsg ⊨_A onll Γ_QMSG (λ((msgs, _), a, _). sendmsg (λm. m∈set msgs) a)"
      by (inv_cterms inv add: onl_invariant_sterms [OF qmsg_wf qmsg_msgs_not_empty])
    thus ?thesis
      by (rule step_invariant_weakenE) (auto dest!: onllD)
  qed

lemma qmsg_queue_contents:
  "qmsg ⊨_A (λ((msgs, q), a, (msgs', q')). case a of
                                             receive m ⇒ set msgs' ⊆ set (msgs @ [m])
                                           | _ ⇒ set msgs' ⊆ set msgs)"
  proof -
    have "qmsg ⊨_A onll Γ_QMSG (λ((msgs, q), a, (msgs', q')).
                                 case a of
                                   receive m ⇒ set msgs' ⊆ set (msgs @ [m])
                                 | _ ⇒ set msgs' ⊆ set msgs)"
      by (inv_cterms) (clarsimp simp add: in_set_tl)+
    thus ?thesis
      by (rule step_invariant_weakenE) (auto dest!: onllD)
  qed

lemma qmsg_send_receive_or_tau:
  "qmsg ⊨_A (λ(_, a, _). ∃m. a = send m ∨ a = receive m ∨ a = τ)"
```

```
  proof -
   have "qmsg ⊫A onll ΓQMSG (λ(_, a, _). ∃m. a = send m ∨ a = receive m ∨ a = τ)"
     by inv_cterms
   thus ?thesis
    by rule (auto dest!: onllD)
  qed

lemma par_qmsg_oreachable:
  assumes "(σ, ζ) ∈ oreachable (A ⟨⟨i qmsg) (otherwith S {i} (orecvmsg R)) (other U {i})"
          (is "_ ∈ oreachable _ ?owS _")
      and pinv: "A ⊫A (otherwith S {i} (orecvmsg R), other U {i} →)
                       globala (λ(σ, _, σ'). U (σ i) (σ' i))"
      and ustutter: "⋀ξ. U ξ ξ"
      and sgivesu: "⋀ξ ξ'. S ξ ξ' ⟹ U ξ ξ'"
      and upreservesq: "⋀σ σ' m. ⟦ ∀j. U (σ j) (σ' j); R σ m ⟧ ⟹ R σ' m"
  shows "(σ, fst ζ) ∈ oreachable A ?owS (other U {i})
         ∧ snd ζ ∈ reachable qmsg (recvmsg (R σ))
         ∧ (∀m∈set (fst (snd ζ)). R σ m)"
  using assms(1) proof (induction rule: oreachable_pair_induct)
    fix σ pq
    assume "(σ, pq) ∈ init (A ⟨⟨i qmsg)"
    then obtain p ms q where "pq = (p, (ms, q))"
                         and "(σ, p) ∈ init A"
                         and "(ms, q) ∈ init qmsg"
      by (clarsimp simp del: ΓQMSG_simps)
    from this(2) have "(σ, p) ∈ oreachable A ?owS (other U {i})" ..
    moreover from ‹(ms, q) ∈ init qmsg› have "(ms, q) ∈ reachable qmsg (recvmsg (R σ))" ..
    moreover from ‹(ms, q) ∈ init qmsg› have "ms = []"
        unfolding σQMSG_def by simp
    ultimately show "(σ, fst pq) ∈ oreachable A ?owS (other U {i})
                    ∧ snd pq ∈ reachable qmsg (recvmsg (R σ))
                    ∧ (∀m∈set (fst (snd pq)). R σ m)"
      using ‹pq = (p, (ms, q))› by simp
  next
    note ΓQMSG_simps [simp del]
    case (other σ pq σ')
    hence "(σ, fst pq) ∈ oreachable A ?owS (other U {i})"
      and "other U {i} σ σ'"
      and qr: "snd pq ∈ reachable qmsg (recvmsg (R σ))"
      and "∀m∈set (fst (snd pq)). R σ m"
      by simp_all
    from ‹other U {i} σ σ'› and ustutter have "∀j. U (σ j) (σ' j)"
        by (clarsimp elim!: otherE) metis
    from ‹other U {i} σ σ'›
     and ‹(σ, fst pq) ∈ oreachable A ?owS (other U {i})›
      have "(σ', fst pq) ∈ oreachable A ?owS (other U {i})"
        by - (rule oreachable_other')
    moreover have "∀m∈set (fst (snd pq)). R σ' m"
    proof
      fix m assume "m ∈ set (fst (snd pq))"
      with ‹∀m∈set (fst (snd pq)). R σ m› have "R σ m" ..
      with ‹∀j. U (σ j) (σ' j)› show "R σ' m" by (rule upreservesq)
    qed
    moreover from qr have "snd pq ∈ reachable qmsg (recvmsg (R σ'))"
    proof
      fix a
      assume "recvmsg (R σ) a"
      thus "recvmsg (R σ') a"
      proof (rule recvmsgE [where R=R])
        fix m assume "R σ m"
        with ‹∀j. U (σ j) (σ' j)› show "R σ' m" by (rule upreservesq)
      qed
    qed
    ultimately show ?case using qr by simp
```

```
next
  case (local σ pq σ' pq' a)
  obtain p ms q p' ms' q' where "pq = (p, (ms, q))"
                           and "pq' = (p', (ms', q'))"
    by (cases pq, cases pq') metis
  with local.hyps local.IH
    have pqtr: "((σ, (p, (ms, q))), a, (σ', (p', (ms', q'))))
                 ∈ oparp_sos i (trans A) (seqp_sos Γ_QMSG)"
      and por: "(σ, p) ∈ oreachable A ?owS (other U {i})"
      and qr: "(ms, q) ∈ reachable qmsg (recvmsg (R σ))"
      and "∀m∈set ms. R σ m"
      and "?owS σ σ' a"
    by (simp_all del: Γ_QMSG_simps)

  from ‹?owS σ σ' a› have "∀j. j≠i ⟶ S (σ j) (σ' j)"
    by (clarsimp dest!: otherwith_syncD)
  with sgivesu have "∀j. j≠i ⟶ U (σ j) (σ' j)" by simp

  from ‹?owS σ σ' a› have "orecvmsg R σ a" by (rule otherwithE)
  hence "recvmsg (R σ) a" ..

  from pqtr have "(σ', p') ∈ oreachable A ?owS (other U {i})
              ∧ (ms', q') ∈ reachable qmsg (recvmsg (R σ'))
              ∧ (∀m∈set ms'. R σ' m)"
  proof
    assume "((σ, p), a, (σ', p')) ∈ trans A"
      and "⋀m. a ≠ receive m"
      and "(ms', q') = (ms, q)"
    from this(1) have ptr: "((σ, p), a, (σ', p')) ∈ trans A" by simp
    with pinv por and ‹?owS σ σ' a› have "U (σ i) (σ' i)"
      by (auto dest!: ostep_invariantD)
    with ‹∀j. j≠i ⟶ U (σ j) (σ' j)› have "∀j. U (σ j) (σ' j)" by auto

    hence recvmsg': "⋀a. recvmsg (R σ) a ⟹ recvmsg (R σ') a"
      by (auto elim!: recvmsgE [where R=R] upreservesq)

    from por ptr ‹?owS σ σ' a› have "(σ', p') ∈ oreachable A ?owS (other U {i})"
      by - (rule oreachable_local')

    moreover have "(ms', q') ∈ reachable qmsg (recvmsg (R σ'))"
    proof -
      from qr and ‹(ms', q') = (ms, q)›
        have "(ms', q') ∈ reachable qmsg (recvmsg (R σ))" by simp
      thus ?thesis by (rule reachable_weakenE) (erule recvmsg')
    qed

    moreover have "∀m∈set ms'. R σ' m"
    proof
      fix m
      assume "m∈set ms'"
      with ‹(ms', q') = (ms, q)› have "m∈set ms" by simp
      with ‹∀m∈set ms. R σ m› have "R σ m" ..
      with ‹∀j. U (σ j) (σ' j)› show "R σ' m"
        by (rule upreservesq)
    qed

    ultimately show
      "(σ', p') ∈ oreachable A ?owS (other U {i})
        ∧ (ms', q') ∈ reachable qmsg (recvmsg (R σ'))
        ∧ (∀m∈set ms'. R σ' m)" by simp_all
  next
    assume qtr: "((ms, q), a, (ms', q')) ∈ seqp_sos Γ_QMSG"
      and "⋀m. a ≠ send m"
      and "p' = p"
```

```
      and "σ' i = σ i"

    from this(4) and ‹⋀ξ. U ξ ξ› have "U (σ i) (σ' i)" by simp
    with ‹∀j. j≠i ⟶ U (σ j) (σ' j)› have "∀j. U (σ j) (σ' j)" by auto

    hence recvmsg': "⋀a. recvmsg (R σ) a ⟹ recvmsg (R σ') a"
      by (auto elim!: recvmsgE [where R=R] upreservesq)

    from qtr have tqtr: "((ms, q), a, (ms', q')) ∈ trans qmsg" by simp

    from ‹∀j. U (σ j) (σ' j)› and  ‹σ' i = σ i› have "other U {i} σ σ'" by auto
    with por and ‹p' = p›
      have "(σ', p') ∈ oreachable A ?owS (other U {i})"
        by (auto dest: oreachable_other)

    moreover have "(ms', q') ∈ reachable qmsg (recvmsg (R σ'))"
    proof (rule reachable_weakenE [where P="recvmsg (R σ)"])
      from qr tqtr ‹recvmsg (R σ) a› show "(ms', q') ∈ reachable qmsg (recvmsg (R σ))" ..
    qed (rule recvmsg')

    moreover have "∀m∈set ms'. R σ' m"
    proof
      fix m
      assume "m ∈ set ms'"
      moreover have "case a of receive m ⇒ set ms' ⊆ set (ms @ [m]) | _ ⇒ set ms' ⊆ set ms"
        proof -
          from qr have "(ms, q) ∈ reachable qmsg TT" ..
          thus ?thesis using tqtr
            by (auto dest!: step_invariantD [OF qmsg_queue_contents])
        qed
      ultimately have "R σ m" using ‹∀m∈set ms. R σ m› and ‹orecvmsg R σ a›
        by (cases a) auto
      with ‹∀j. U (σ j) (σ' j)› show "R σ' m"
        by (rule upreservesq)
    qed

    ultimately show "(σ', p') ∈ oreachable A ?owS (other U {i})
                   ∧ (ms', q') ∈ reachable qmsg (recvmsg (R σ'))
                   ∧ (∀m∈set ms'. R σ' m)" by simp
  next
    fix m
    assume "a = τ"
      and "((σ, p), receive m, (σ', p')) ∈ trans A"
      and "((ms, q), send m, (ms', q')) ∈ seqp_sos Γ_QMSG"
    from this(2-3)
      have ptr: "((σ, p), receive m, (σ', p')) ∈ trans A"
       and qtr: "((ms, q), send m, (ms', q')) ∈ trans qmsg" by simp_all

    from qr have "(ms, q) ∈ reachable qmsg TT" ..
    with qtr have "m ∈ set ms"
      by (auto dest!: step_invariantD [OF qmsg_send_from_queue])
    with ‹∀m∈set ms. R σ m› have "R σ m" ..
    hence "orecvmsg R σ (receive m)" by simp

    with ‹∀j. j≠i ⟶ S (σ j) (σ' j)› have "?owS σ σ' (receive m)"
      by (auto intro!: otherwithI)
    with pinv por ptr have "U (σ i) (σ' i)"
      by (auto dest!: ostep_invariantD)
    with ‹∀j. j≠i ⟶ U (σ j) (σ' j)› have "∀j. U (σ j) (σ' j)" by auto
    hence recvmsg': "⋀a. recvmsg (R σ) a ⟹ recvmsg (R σ') a"
      by (auto elim!: recvmsgE [where R=R] upreservesq)

    from por ptr have "(σ', p') ∈ oreachable A ?owS (other U {i})"
      using ‹?owS σ σ' (receive m)› by - (erule(1) oreachable_local, simp)
```

```
    moreover have "(ms', q') ∈ reachable qmsg (recvmsg (R σ'))"
    proof (rule reachable_weakenE [where P="recvmsg (R σ)"])
      have "recvmsg (R σ) (send m)" by simp
      with qr qtr show "(ms', q') ∈ reachable qmsg (recvmsg (R σ))" ..
    qed (rule recvmsg')

    moreover have "∀m∈set ms'. R σ' m"
    proof
      fix m
      assume "m ∈ set ms'"
      moreover have "set ms' ⊆ set ms"
        proof -
          from qr have "(ms, q) ∈ reachable qmsg TT" ..
          thus ?thesis using qtr
            by (auto dest!: step_invariantD [OF qmsg_queue_contents])
        qed
      ultimately have "R σ m" using ‹∀m∈set ms. R σ m› by auto
      with ‹∀j. U (σ j) (σ' j)› show "R σ' m"
        by (rule upreservesq)
    qed

    ultimately show "(σ', p') ∈ oreachable A ?owS (other U {i})
                  ∧ (ms', q') ∈ reachable qmsg (recvmsg (R σ'))
                  ∧ (∀m∈set ms'. R σ' m)" by simp
  qed
  with ‹pq = (p, (ms, q))› and ‹pq' = (p', (ms', q'))› show ?case
    by (simp_all del: Γ_QMSG_simps)
qed

lemma par_qmsg_oreachable_statelessassm:
  assumes "(σ, ζ) ∈ oreachable (A ⟨⟨_i qmsg)
                        (λσ _. orecvmsg (λ_. R) σ) (other (λ_ _. True) {i})"
    and ustutter: "⋀ξ. U ξ ξ"
  shows "(σ, fst ζ) ∈ oreachable A (λσ _. orecvmsg (λ_. R) σ) (other (λ_ _. True) {i})
       ∧ snd ζ ∈ reachable qmsg (recvmsg R)
       ∧ (∀m∈set (fst (snd ζ)). R m)"
proof -
  from assms(1)
    have "(σ, ζ) ∈ oreachable (A ⟨⟨_i qmsg)
                         (otherwith (λ_ _. True) {i} (orecvmsg (λ_. R)))
                         (other (λ_ _. True) {i})" by auto
  moreover
    have "A ⊨_A (otherwith (λ_ _. True) {i} (orecvmsg (λ_. R)),
              other (λ_ _. True) {i} →) globala (λ(σ, _, σ'). True)"
      by auto
  ultimately
    obtain "(σ, fst ζ) ∈ oreachable A
                    (otherwith (λ_ _. True) {i} (orecvmsg (λ_. R))) (other (λ_ _. True) {i})"
       and  *: "snd ζ ∈ reachable qmsg (recvmsg R)"
       and **: "(∀m∈set (fst (snd ζ)). R m)"
      by (auto dest!: par_qmsg_oreachable)
  from this(1)
    have "(σ, fst ζ) ∈ oreachable A (λσ _. orecvmsg (λ_. R) σ) (other (λ_ _. True) {i})"
      by rule auto
  thus ?thesis using * ** by simp
qed

lemma lift_into_qmsg:
  assumes "A ⊨ (otherwith S {i} (orecvmsg R), other U {i} →) global P"
    and "⋀ξ. U ξ ξ"
    and "⋀ξ ξ'. S ξ ξ' ⟹ U ξ ξ'"
    and "⋀σ σ' m. ⟦ ∀j. U (σ j) (σ' j); R σ m ⟧ ⟹ R σ' m"
    and "A ⊨_A (otherwith S {i} (orecvmsg R), other U {i} →)
```

```
              globala (λ(σ, _, σ'). U (σ i) (σ' i))"
    shows "A ⟨⟨i qmsg ⊨ (otherwith S {i} (orecvmsg R), other U {i} →) global P"
  proof (rule oinvariant_oreachableI)
    fix σ ζ
    assume "(σ, ζ) ∈ oreachable (A ⟨⟨i qmsg) (otherwith S {i} (orecvmsg R)) (other U {i})"
    then obtain s where "(σ, s) ∈ oreachable A (otherwith S {i} (orecvmsg R)) (other U {i})"
      by (auto dest!: par_qmsg_oreachable [OF _ assms(5,2-4)])
    with assms(1) show "global P (σ, ζ)"
      by (auto dest: oinvariant_weakenD [OF assms(1)])
  qed

lemma lift_step_into_qmsg:
  assumes inv: "A ⊨A (otherwith S {i} (orecvmsg R), other U {i} →) globala P"
      and ustutter: "⋀ξ. U ξ ξ"
      and sgivesu: "⋀ξ ξ'. S ξ ξ' ⟹ U ξ ξ'"
      and upreservesq: "⋀σ σ' m. ⟦ ∀j. U (σ j) (σ' j); R σ m ⟧ ⟹ R σ' m"
      and self_sync: "A ⊨A (otherwith S {i} (orecvmsg R), other U {i} →)
                          globala (λ(σ, _, σ'). U (σ i) (σ' i))"

      and recv_stutter:  "⋀σ σ' m. ⟦ ∀j. U (σ j) (σ' j); σ' i = σ i ⟧ ⟹ P (σ, receive m, σ')"
      and receive_right: "⋀σ σ' m.  P (σ, receive m, σ') ⟹ P (σ, τ, σ')"
    shows "A ⟨⟨i qmsg ⊨A (otherwith S {i} (orecvmsg R), other U {i} →) globala P"
      (is "_ ⊨A (?owS, ?U →) _")
  proof (rule ostep_invariantI)
    fix σ ζ a σ' ζ'
    assume or: "(σ, ζ) ∈ oreachable (A ⟨⟨i qmsg) ?owS ?U"
       and otr: "((σ, ζ), a, (σ', ζ')) ∈ trans (A ⟨⟨i qmsg)"
       and "?owS σ σ' a"
    from this(2) have "((σ, ζ), a, (σ', ζ')) ∈ oparp_sos i (trans A) (seqp_sos ΓQMSG)"
        by simp
    then obtain s msgs q s' msgs' q'
      where "ζ = (s, (msgs, q))" "ζ' = (s', (msgs', q'))"
        and "((σ, (s, (msgs, q))), a, (σ', (s', (msgs', q'))))
               ∈ oparp_sos i (trans A) (seqp_sos ΓQMSG)"
        by (metis prod_cases3)
    from this(1-2) and or
      obtain "(σ, s) ∈ oreachable A ?owS ?U"
             "(msgs, q) ∈ reachable qmsg (recvmsg (R σ))"
             "(∀m∈set msgs. R σ m)"
       by (auto dest: par_qmsg_oreachable [OF _ self_sync ustutter sgivesu]
               elim!: upreservesq)
    from otr ‹ζ = (s, (msgs, q))› ‹ζ' = (s', (msgs', q'))›
      have "((σ, (s, (msgs, q))), a, (σ', (s', (msgs', q'))))
              ∈ oparp_sos i (trans A) (seqp_sos ΓQMSG)"
        by simp
    hence "globala P ((σ, s), a, (σ', s'))"
    proof
      assume "((σ, s), a, (σ', s')) ∈ trans A"
      with ‹(σ, s) ∈ oreachable A ?owS ?U›
        show "globala P ((σ, s), a, (σ', s'))"
          using ‹?owS σ σ' a› by (rule ostep_invariantD [OF inv])
    next
      assume "((msgs, q), a, (msgs', q')) ∈ seqp_sos ΓQMSG"
         and "⋀m. a ≠ send m"
         and "σ' i = σ i"
      from this(3) and ustutter have "U (σ i) (σ' i)" by simp
      with ‹?owS σ σ' a› and sgivesu have "∀j. U (σ j) (σ' j)"
        by (clarsimp dest!: otherwith_syncD) metis
      moreover have "(∃m. a = receive m) ∨ (a = τ)"
      proof -
        from ‹(msgs, q) ∈ reachable qmsg (recvmsg (R σ))›
          have "(msgs, q) ∈ reachable qmsg TT" ..
        moreover from ‹((msgs, q), a, (msgs', q')) ∈ seqp_sos ΓQMSG›
          have "((msgs, q), a, (msgs', q')) ∈ trans qmsg" by simp
```

```
      ultimately show ?thesis
        using ‹⋀m. a ≠ send m›
        by (auto dest!: step_invariantD [OF qmsg_send_receive_or_tau])
    qed
    ultimately show "globala P ((σ, s), a, (σ', s'))"
      using ‹σ' i = σ i›
      by simp (metis receive_right recv_stutter step_seq_tau)
  next
    fix m
    assume "a = τ"
       and "((σ, s), receive m, (σ', s')) ∈ trans A"
       and "((msgs, q), send m, (msgs', q')) ∈ seqp_sos Γ_QMSG"

    from ‹(msgs, q) ∈ reachable qmsg (recvmsg (R σ))›
      have "(msgs, q) ∈ reachable qmsg TT" ..
    moreover from ‹((msgs, q), send m, (msgs', q')) ∈ seqp_sos Γ_QMSG›
      have "((msgs, q), send m, (msgs', q')) ∈ trans qmsg" by simp
    ultimately have "m∈set msgs"
      by (auto dest!: step_invariantD [OF qmsg_send_from_queue])

    with ‹∀m∈set msgs. R σ m› have "R σ m" ..
    with ‹?owS σ σ' a› have "?owS σ σ' (receive m)"
        by (auto dest!: otherwith_syncD)

    with ‹((σ, s), receive m, (σ', s')) ∈ trans A›
      have "globala P ((σ, s), receive m, (σ', s'))"
        using ‹(σ, s) ∈ oreachable A ?owS ?U›
        by - (rule ostep_invariantD [OF inv])
    hence "P (σ, receive m, σ')" by simp
    hence "P (σ, τ, σ')" by (rule receive_right)
    with ‹a = τ› show "globala P ((σ, s), a, (σ', s'))" by simp
  qed
  with ‹ζ = (s, (msgs, q))› and ‹ζ' = (s', (msgs', q'))› show "globala P ((σ, ζ), a, (σ', ζ'))"
    by simp
qed

lemma lift_step_into_qmsg_statelessassm:
  assumes "A ⊨_A (λσ _. orecvmsg (λ_. R) σ, other (λ_ _. True) {i} →) globala P"
      and "⋀σ σ' m. σ' i = σ i ⟹ P (σ, receive m, σ')"
      and "⋀σ σ' m. P (σ, receive m, σ') ⟹ P (σ, τ, σ')"
    shows "A ⟨⟨_i qmsg ⊨_A (λσ _. orecvmsg (λ_. R) σ, other (λ_ _. True) {i} →) globala P"
  proof -
    from assms(1) have *: "A ⊨_A (otherwith (λ_ _. True) {i} (orecvmsg (λ_. R)),
                                  other (λ_ _. True) {i} →) globala P"
      by rule auto
    hence "A ⟨⟨_i qmsg ⊨_A
             (otherwith (λ_ _. True) {i} (orecvmsg (λ_. R)), other (λ_ _. True) {i} →) globala P"
      by (rule lift_step_into_qmsg)
         (auto elim!: assms(2-3) simp del: step_seq_tau)
    thus ?thesis by rule auto
  qed

end
```

24 Transfer open results onto closed models

```
theory OClosed_Transfer
imports Closed OClosed_Lifting
begin

locale openproc =
  fixes np  :: "ip ⇒ ('s, ('m::msg) seq_action) automaton"
    and onp :: "ip ⇒ ((ip ⇒ 'g) × 'l, 'm seq_action) automaton"
    and sr  :: "'s ⇒ ('g × 'l)"
```

```
  assumes  init: "{ (σ, ζ) |σ ζ s. s ∈ init (np i)
                          ∧ (σ i, ζ) = sr s
                          ∧ (∀j. j≠i ⟶ σ j ∈ (fst o sr) ‘ init (np j)) } ⊆ init (onp i)"
      and init_notempty: "∀j. init (np j) ≠ {}"
      and trans: "⋀s a s’ σ σ’. ⟦ σ i = fst (sr s);
                               σ’ i = fst (sr s’);
                               (s, a, s’) ∈ trans (np i) ⟧
                  ⟹ ((σ, snd (sr s)), a, (σ’, snd (sr s’))) ∈ trans (onp i)"
begin

lemma init_pnet_p_NodeS:
  assumes "NodeS i s R ∈ init (pnet np p)"
    shows "p = ⟨i; R⟩"
  using assms by (cases p) (auto simp add: node_comps)

lemma init_pnet_p_SubnetS:
  assumes "SubnetS s1 s2 ∈ init (pnet np p)"
  obtains p1 p2 where "p = (p1 ∥ p2)"
                and "s1 ∈ init (pnet np p1)"
                and "s2 ∈ init (pnet np p2)"
  using assms by (cases p) (auto simp add: node_comps)

lemma init_pnet_fst_sr_netgmap:
  assumes "s ∈ init (pnet np p)"
      and "i ∈ net_ips s"
      and "wf_net_tree p"
    shows "the (fst (netgmap sr s) i) ∈ (fst ∘ sr) ‘ init (np i)"
  using assms proof (induction s arbitrary: p)
    fix ii s R_i p
    assume "NodeS ii s R_i ∈ init (pnet np p)"
      and "i ∈ net_ips (NodeS ii s R_i)"
      and "wf_net_tree p"
    note this(1)
    moreover then have "p = ⟨ii; R_i⟩"
      by (rule init_pnet_p_NodeS)
    ultimately have "s ∈ init (np ii)"
      by (clarsimp simp: node_comps)
    with ‹i ∈ net_ips (NodeS ii s R_i)›
      show "the (fst (netgmap sr (NodeS ii s R_i)) i) ∈ (fst ∘ sr) ‘ init (np i)"
        by clarsimp
  next
    fix s1 s2 p
    assume IH1: "⋀p. s1 ∈ init (pnet np p)
                ⟹ i ∈ net_ips s1
                ⟹ wf_net_tree p
                ⟹ the (fst (netgmap sr s1) i) ∈ (fst ∘ sr) ‘ init (np i)"
      and IH2: "⋀p. s2 ∈ init (pnet np p)
                ⟹ i ∈ net_ips s2
                ⟹ wf_net_tree p
                ⟹ the (fst (netgmap sr s2) i) ∈ (fst ∘ sr) ‘ init (np i)"
      and "SubnetS s1 s2 ∈ init (pnet np p)"
      and "i ∈ net_ips (SubnetS s1 s2)"
      and "wf_net_tree p"
    from this(3) obtain p1 p2 where "p = (p1 ∥ p2)"
                              and "s1 ∈ init (pnet np p1)"
                              and "s2 ∈ init (pnet np p2)"
      by (rule init_pnet_p_SubnetS)
    from this(1) and ‹wf_net_tree p› have "wf_net_tree p1"
                                    and "wf_net_tree p2"
                                    and "net_tree_ips p1 ∩ net_tree_ips p2 = {}"
      by auto
    from ‹i ∈ net_ips (SubnetS s1 s2)› have "i ∈ net_ips s1 ∨ i ∈ net_ips s2"
      by simp
    thus "the (fst (netgmap sr (SubnetS s1 s2)) i) ∈ (fst ∘ sr) ‘ init (np i)"
```

```
    proof
      assume "i ∈ net_ips s1"
      hence "i ∉ net_ips s2"
      proof -
        from ‹s1 ∈ init (pnet np p1)› and ‹i ∈ net_ips s1› have "i∈net_tree_ips p1" ..
        with ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}› have "i∉net_tree_ips p2" by auto
        with ‹s2 ∈ init (pnet np p2)› show ?thesis ..
      qed
      moreover from ‹s1 ∈ init (pnet np p1)›  ‹i ∈ net_ips s1› and ‹wf_net_tree p1›
        have "the (fst (netgmap sr s1) i) ∈ (fst ∘ sr) ‘ init (np i)"
          by (rule IH1)
      ultimately show ?thesis by simp
    next
      assume "i ∈ net_ips s2"
      moreover with ‹s2 ∈ init (pnet np p2)› have "the (fst (netgmap sr s2) i) ∈ (fst ∘ sr) ‘ init (np
i)"
        using ‹wf_net_tree p2› by (rule IH2)
      moreover from ‹s2 ∈ init (pnet np p2)› and ‹i ∈ net_ips s2› have "i∈net_tree_ips p2" ..
      ultimately show ?thesis by simp
    qed
  qed

lemma init_lifted:
  assumes "wf_net_tree p"
  shows "{ (σ, snd (netgmap sr s)) |σ s. s ∈ init (pnet np p)
                              ∧ (∀i. if i∈net_tree_ips p then σ i = the (fst (netgmap sr s) i)
                                     else σ i ∈ (fst o sr) ‘ init (np i)) } ⊆ init (opnet onp p)"
  using assms proof (induction p)
    fix i R
    assume "wf_net_tree ⟨i; R⟩"
    show "{(σ, snd (netgmap sr s)) |σ s. s ∈ init (pnet np ⟨i; R⟩)
          ∧ (∀j. if j ∈ net_tree_ips ⟨i; R⟩ then σ j = the (fst (netgmap sr s) j)
                 else σ j ∈ (fst ∘ sr) ‘ init (np j))} ⊆ init (opnet onp ⟨i; R⟩)"
      by (clarsimp simp add: node_comps onode_comps)
         (rule subsetD [OF init], auto)
  next
    fix p1 p2
    assume IH1: "wf_net_tree p1
               ⟹ {(σ, snd (netgmap sr s)) |σ s. s ∈ init (pnet np p1)
                     ∧ (∀i. if i ∈ net_tree_ips p1 then σ i = the (fst (netgmap sr s) i)
                            else σ i ∈ (fst ∘ sr) ‘ init (np i))} ⊆ init (opnet onp p1)"
               (is "_ ⟹ ?S1 ⊆ _")
       and IH2: "wf_net_tree p2
               ⟹ {(σ, snd (netgmap sr s)) |σ s. s ∈ init (pnet np p2)
                     ∧ (∀i. if i ∈ net_tree_ips p2 then σ i = the (fst (netgmap sr s) i)
                            else σ i ∈ (fst ∘ sr) ‘ init (np i))} ⊆ init (opnet onp p2)"
               (is "_ ⟹ ?S2 ⊆ _")
        and "wf_net_tree (p1 ∥ p2)"
    from this(3) have "wf_net_tree p1"
                 and "wf_net_tree p2"
                 and "net_tree_ips p1 ∩ net_tree_ips p2 = {}" by auto
    show "{(σ, snd (netgmap sr s)) |σ s. s ∈ init (pnet np (p1 ∥ p2))
          ∧ (∀i. if i ∈ net_tree_ips (p1 ∥ p2) then σ i = the (fst (netgmap sr s) i)
                 else σ i ∈ (fst ∘ sr) ‘ init (np i))} ⊆ init (opnet onp (p1 ∥ p2))"
    proof (rule, clarsimp simp only: split_paired_all pnet.simps automaton.simps)
      fix σ s1 s2
      assume σ_desc: "∀i. if i ∈ net_tree_ips (p1 ∥ p2)
                          then σ i = the (fst (netgmap sr (SubnetS s1 s2)) i)
                          else σ i ∈ (fst ∘ sr) ‘ init (np i)"
         and "s1 ∈ init (pnet np p1)"
         and "s2 ∈ init (pnet np p2)"
      from this(2-3) have "net_ips s1 = net_tree_ips p1"
                     and "net_ips s2 = net_tree_ips p2" by auto
      have "(σ, snd (netgmap sr s1)) ∈ ?S1"
```

```
proof -
  { fix i
    assume "i ∈ net_tree_ips p1"
    with ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}› have "i ∉ net_tree_ips p2" by auto
    with ‹s2 ∈ init (pnet np p2)› have "i ∉ net_ips s2" ..
    hence "the ((fst (netgmap sr s1) ++ fst (netgmap sr s2)) i) = the (fst (netgmap sr s1) i)"
      by simp
  }
  moreover
  { fix i
    assume "i ∉ net_tree_ips p1"
    have "σ i ∈ (fst ∘ sr) ‘ init (np i)"
    proof (cases "i ∈ net_tree_ips p2")
      assume "i ∉ net_tree_ips p2"
      with ‹i ∉ net_tree_ips p1› and σ_desc show ?thesis
        by (auto dest: spec [of _ i])
    next
      assume "i ∈ net_tree_ips p2"
      with ‹s2 ∈ init (pnet np p2)› have "i ∈ net_ips s2" ..
      with ‹s2 ∈ init (pnet np p2)› have "the (fst (netgmap sr s2) i) ∈ (fst ∘ sr) ‘ init (np i)"
        using ‹wf_net_tree p2› by (rule init_pnet_fst_sr_netgmap)
      with ‹i∈net_tree_ips p2› and ‹i∈net_ips s2› show ?thesis
        using σ_desc by simp
    qed
  }
  ultimately show ?thesis
    using ‹s1 ∈ init (pnet np p1)› and σ_desc by auto
qed
hence "(σ, snd (netgmap sr s1)) ∈ init (opnet onp p1)"
  by (rule subsetD [OF IH1 [OF ‹wf_net_tree p1›]])

have "(σ, snd (netgmap sr s2)) ∈ ?S2"
proof -
  { fix i
    assume "i ∈ net_tree_ips p2"
    with ‹s2 ∈ init (pnet np p2)› have "i ∈ net_ips s2" ..
    hence "the ((fst (netgmap sr s1) ++ fst (netgmap sr s2)) i) = the (fst (netgmap sr s2) i)"
      by simp
  }
  moreover
  { fix i
    assume "i ∉ net_tree_ips p2"
    have "σ i ∈ (fst ∘ sr) ‘ init (np i)"
    proof (cases "i ∈ net_tree_ips p1")
      assume "i ∉ net_tree_ips p1"
      with ‹i ∉ net_tree_ips p2› and σ_desc show ?thesis
        by (auto dest: spec [of _ i])
    next
      assume "i ∈ net_tree_ips p1"
      with ‹s1 ∈ init (pnet np p1)› have "i ∈ net_ips s1" ..
      with ‹s1 ∈ init (pnet np p1)› have "the (fst (netgmap sr s1) i) ∈ (fst ∘ sr) ‘ init (np i)"
        using ‹wf_net_tree p1› by (rule init_pnet_fst_sr_netgmap)
      moreover from ‹s2 ∈ init (pnet np p2)› and ‹i ∉ net_tree_ips p2› have "i∉net_ips s2" ..
      ultimately show ?thesis
        using ‹i∈net_tree_ips p1› ‹i∈net_ips s1› and ‹i∉net_tree_ips p2› σ_desc by simp
    qed
  }
  ultimately show ?thesis
    using ‹s2 ∈ init (pnet np p2)› and σ_desc by auto
qed
hence "(σ, snd (netgmap sr s2)) ∈ init (opnet onp p2)"
  by (rule subsetD [OF IH2 [OF ‹wf_net_tree p2›]])

with ‹(σ, snd (netgmap sr s1)) ∈ init (opnet onp p1)›
```

```
      show "(σ, snd (netgmap sr (SubnetS s1 s2))) ∈ init (opnet onp (p1 ∥ p2))"
      using ‹net_tree_ips p1 ∩ net_tree_ips p2 = {}›
            ‹net_ips s1 = net_tree_ips p1›
            ‹net_ips s2 = net_tree_ips p2› by simp
    qed
  qed

lemma init_pnet_opnet [elim]:
  assumes "wf_net_tree p"
      and "s ∈ init (pnet np p)"
    shows "netgmap sr s ∈ netmask (net_tree_ips p) ‘ init (opnet onp p)"
  proof -
    from ‹wf_net_tree p›
      have "{ (σ, snd (netgmap sr s)) |σ s. s ∈ init (pnet np p)
                              ∧ (∀i. if i∈net_tree_ips p then σ i = the (fst (netgmap sr s) i)
                                     else σ i ∈ (fst o sr) ‘ init (np i)) } ⊆ init (opnet onp p)"
        (is "?S ⊆ _")
      by (rule init_lifted)
    hence "netmask (net_tree_ips p) ‘ ?S ⊆ netmask (net_tree_ips p) ‘ init (opnet onp p)"
      by (rule image_mono)
    moreover have "netgmap sr s ∈ netmask (net_tree_ips p) ‘ ?S"
    proof -
      { fix i
        from init_notempty have "∃s. s ∈ (fst ∘ sr) ‘ init (np i)" by auto
        hence "(SOME x. x ∈ (fst ∘ sr) ‘ init (np i)) ∈ (fst ∘ sr) ‘ init (np i)" ..
      }
      with ‹s ∈ init (pnet np p)› and init_notempty
        have "(λi. if i ∈ net_tree_ips p
                    then the (fst (netgmap sr s) i)
                    else SOME x. x ∈ (fst ∘ sr) ‘ init (np i), snd (netgmap sr s)) ∈ ?S"
          (is "?s ∈ ?S") by auto
      moreover have "netgmap sr s = netmask (net_tree_ips p) ?s"
      proof (intro prod_eqI ext)
        fix i
        show "fst (netgmap sr s) i = fst (netmask (net_tree_ips p) ?s) i"
        proof (cases "i ∈ net_tree_ips p")
          assume "i ∈ net_tree_ips p"
          with ‹s∈init (pnet np p)› have "i∈net_ips s" ..
          hence "Some (the (fst (netgmap sr s) i)) = fst (netgmap sr s) i"
            by (rule some_the_fst_netgmap)
          with ‹i∈net_tree_ips p› show ?thesis
            by simp
        next
          assume "i ∉ net_tree_ips p"
          moreover with ‹s∈init (pnet np p)› have "i∉net_ips s" ..
          ultimately show ?thesis
            by simp
        qed
      qed simp
      ultimately show ?thesis
        by (rule rev_image_eqI)
    qed
    ultimately show ?thesis
      by (rule rev_subsetD [rotated])
  qed

lemma transfer_connect:
  assumes "(s, connect(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), connect(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 atomize_elim
    from assms have "((σ, snd (netgmap sr s)), connect(i, i'), (σ, snd (netgmap sr s'))) ∈ trans (opnet
onp n)
                  ∧ netgmap sr s' = netmask (net_tree_ips n) (σ, snd (netgmap sr s'))"
      proof (induction n arbitrary: s s' ζ)
        fix ii R_i ns ns' ζ
        assume "(ns, connect(i, i'), ns') ∈ trans (pnet np ⟨ii; R_i⟩)"
           and "netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)"
        from this(1) have "(ns, connect(i, i'), ns') ∈ node_sos (trans (np ii))"
          by (simp add: node_comps)
        moreover then obtain ni s s' R R' where "ns  = NodeS ni s R"
                                            and "ns' = NodeS ni s' R'" ..
        ultimately have "(NodeS ni s R, connect(i, i'), NodeS ni s' R') ∈ node_sos (trans (np ii))"
          by simp
        moreover then have "s' = s" by auto
        ultimately have "((σ, NodeS ni (snd (sr s)) R), connect(i, i'), (σ, NodeS ni (snd (sr s)) R'))
                                                      ∈ onode_sos (trans (onp ii))"
          by - (rule node_connectTE', auto intro!: onode_sos.intros [simplified])
        with ‹ns = NodeS ni s R› ‹ns' = NodeS ni s' R'› ‹s' = s›
             and ‹netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)›
          show "((σ, snd (netgmap sr ns)), connect(i, i'), (σ, snd (netgmap sr ns'))) ∈ trans (opnet onp
⟨ii; R_i⟩)
                ∧ netgmap sr ns' = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, snd (netgmap sr ns'))"
            by (simp add: onode_comps)
      next
        fix n1 n2 s s' ζ
        assume IH1: "⋀s s' ζ. (s, connect(i, i'), s') ∈ trans (pnet np n1)
                     ⟹ s ∈ reachable (pnet np n1) TT
                     ⟹ netgmap sr s = netmask (net_tree_ips n1) (σ, ζ)
                     ⟹ wf_net_tree n1
                     ⟹ ((σ, snd (netgmap sr s)), connect(i, i'), (σ, snd (netgmap sr s'))) ∈ trans (opnet
onp n1)
                        ∧ netgmap sr s' = netmask (net_tree_ips n1) (σ, snd (netgmap sr s'))"
           and IH2: "⋀s s' ζ. (s, connect(i, i'), s') ∈ trans (pnet np n2)
                     ⟹ s ∈ reachable (pnet np n2) TT
                     ⟹ netgmap sr s = netmask (net_tree_ips n2) (σ, ζ)
                     ⟹ wf_net_tree n2
                     ⟹ ((σ, snd (netgmap sr s)), connect(i, i'), (σ, snd (netgmap sr s'))) ∈ trans (opnet
onp n2)
                        ∧ netgmap sr s' = netmask (net_tree_ips n2) (σ, snd (netgmap sr s'))"
           and tr: "(s, connect(i, i'), s') ∈ trans (pnet np (n1 ∥ n2))"
           and sr: "s ∈ reachable (pnet np (n1 ∥ n2)) TT"
           and nm: "netgmap sr s = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
           and "wf_net_tree (n1 ∥ n2)"
        from this(3) have "(s, connect(i, i'), s') ∈ pnet_sos (trans (pnet np n1))
                                                            (trans (pnet np n2))"
          by simp
        then obtain s1 s1' s2 s2' where "s = SubnetS s1 s2"
                                    and "s' = SubnetS s1' s2'"
                                    and "(s1, connect(i, i'), s1') ∈ trans (pnet np n1)"
                                    and "(s2, connect(i, i'), s2') ∈ trans (pnet np n2)"
          by (rule partial_connectTE) auto
        from this(1) and nm have "netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
          by simp

        from ‹wf_net_tree (n1 ∥ n2)› have "wf_net_tree n1" and "wf_net_tree n2"
                                      and "net_tree_ips n1 ∩ net_tree_ips n2 = {}" by auto

        from sr ‹s = SubnetS s1 s2› have "s1 ∈ reachable (pnet np n1) TT" by (metis subnet_reachable(1))
        hence "net_ips s1 = net_tree_ips n1" by (rule pnet_net_ips_net_tree_ips)

        from sr ‹s = SubnetS s1 s2› have "s2 ∈ reachable (pnet np n2) TT" by (metis subnet_reachable(2))
        hence "net_ips s2 = net_tree_ips n2" by (rule pnet_net_ips_net_tree_ips)
```

```
      from nm ‹s = SubnetS s1 s2›
        have "netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)" by simp
      hence "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
        using ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}› ‹net_ips s1 = net_tree_ips n1›
              and ‹net_ips s2 = net_tree_ips n2› by (rule netgmap_subnet_split1)
      with ‹(s1, connect(i, i'), s1') ∈ trans (pnet np n1)›
       and ‹s1 ∈ reachable (pnet np n1) TT›
       have "((σ, snd (netgmap sr s1)), connect(i, i'), (σ, snd (netgmap sr s1'))) ∈ trans (opnet onp
n1)"
        and "netgmap sr s1' = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1'))"
         using ‹wf_net_tree n1› unfolding atomize_conj by (rule IH1)

      from ‹netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)›
           ‹net_ips s1 = net_tree_ips n1› and ‹net_ips s2 = net_tree_ips n2›
        have "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
          by (rule netgmap_subnet_split2)
      with ‹(s2, connect(i, i'), s2') ∈ trans (pnet np n2)›
       and ‹s2 ∈ reachable (pnet np n2) TT›
       have "((σ, snd (netgmap sr s2)), connect(i, i'), (σ, snd (netgmap sr s2'))) ∈ trans (opnet onp
n2)"
        and "netgmap sr s2' = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2'))"
         using ‹wf_net_tree n2› unfolding atomize_conj by (rule IH2)

      have "((σ, snd (netgmap sr s)), connect(i, i'), (σ, snd (netgmap sr s')))
                                           ∈ trans (opnet onp (n1 ∥ n2))"
      proof -
        from ‹((σ, snd (netgmap sr s1)), connect(i, i'), (σ, snd (netgmap sr s1'))) ∈ trans (opnet onp
n1)›
         and ‹((σ, snd (netgmap sr s2)), connect(i, i'), (σ, snd (netgmap sr s2'))) ∈ trans (opnet onp
n2)›
          have "((σ, SubnetS (snd (netgmap sr s1)) (snd (netgmap sr s2))), connect(i, i'),
                (σ, SubnetS (snd (netgmap sr s1')) (snd (netgmap sr s2'))))
                                         ∈ opnet_sos (trans (opnet onp n1)) (trans (opnet onp n2))"
            by (rule opnet_connect)
        with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› show ?thesis by simp
      qed

      moreover from ‹netgmap sr s1' = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1'))›
                    ‹netgmap sr s2' = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2'))›
                    ‹s' = SubnetS s1' s2'›
        have "netgmap sr s' = netmask (net_tree_ips (n1 ∥ n2)) (σ, snd (netgmap sr s'))" ..

      ultimately show "((σ, snd (netgmap sr s)), connect(i, i'), (σ, snd (netgmap sr s')))
                                                          ∈ trans (opnet onp (n1 ∥ n2))
                     ∧ netgmap sr s' = netmask (net_tree_ips (n1 ∥ n2)) (σ, snd (netgmap sr s'))" ..
    qed
  moreover from ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)› have "ζ = snd (netgmap sr s)" by simp
  ultimately show " ∃σ' ζ'. ((σ, ζ), connect(i, i'), (σ', ζ')) ∈ trans (opnet onp n)
                          ∧ (∀j. j ∉ net_ips ζ ⟶ σ' j = σ j)
                          ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')" by auto
qed

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 atomize_elim
    from assms have "((σ, snd (netgmap sr s)), disconnect(i, i'), (σ, snd (netgmap sr s'))) ∈ trans (opnet
onp n)
                   ∧ netgmap sr s' = netmask (net_tree_ips n) (σ, snd (netgmap sr s'))"
```

```
proof (induction n arbitrary: s s' ζ)
  fix ii Rᵢ ns ns' ζ
  assume "(ns, disconnect(i, i'), ns') ∈ trans (pnet np ⟨ii; Rᵢ⟩)"
     and "netgmap sr ns = netmask (net_tree_ips ⟨ii; Rᵢ⟩) (σ, ζ)"
  from this(1) have "(ns, disconnect(i, i'), ns') ∈ node_sos (trans (np ii))"
    by (simp add: node_comps)
  moreover then obtain ni s s' R R' where "ns  = NodeS ni s R"
                                      and "ns' = NodeS ni s' R'" ..
  ultimately have "(NodeS ni s R, disconnect(i, i'), NodeS ni s' R') ∈ node_sos (trans (np ii))"
    by simp
  moreover then have "s' = s" by auto
  ultimately have "((σ, NodeS ni (snd (sr s)) R), disconnect(i, i'), (σ, NodeS ni (snd (sr s)) R'))
                                                        ∈ onode_sos (trans (onp ii))"
    by - (rule node_disconnectTE', auto intro!: onode_sos.intros [simplified])
  with ‹ns = NodeS ni s R› ‹ns' = NodeS ni s' R'› ‹s' = s›
       and ‹netgmap sr ns = netmask (net_tree_ips ⟨ii; Rᵢ⟩) (σ, ζ)›
    show "((σ, snd (netgmap sr ns)), disconnect(i, i'), (σ, snd (netgmap sr ns'))) ∈ trans (opnet
onp ⟨ii; Rᵢ⟩)
          ∧ netgmap sr ns' = netmask (net_tree_ips ⟨ii; Rᵢ⟩) (σ, snd (netgmap sr ns'))"
      by (simp add: onode_comps)
next
  fix n1 n2 s s' ζ
  assume IH1: "⋀s s' ζ. (s, disconnect(i, i'), s') ∈ trans (pnet np n1)
              ⟹ s ∈ reachable (pnet np n1) TT
              ⟹ netgmap sr s = netmask (net_tree_ips n1) (σ, ζ)
              ⟹ wf_net_tree n1
              ⟹ ((σ, snd (netgmap sr s)), disconnect(i, i'), (σ, snd (netgmap sr s'))) ∈ trans
(opnet onp n1)
                 ∧ netgmap sr s' = netmask (net_tree_ips n1) (σ, snd (netgmap sr s'))"
     and IH2: "⋀s s' ζ. (s, disconnect(i, i'), s') ∈ trans (pnet np n2)
              ⟹ s ∈ reachable (pnet np n2) TT
              ⟹ netgmap sr s = netmask (net_tree_ips n2) (σ, ζ)
              ⟹ wf_net_tree n2
              ⟹ ((σ, snd (netgmap sr s)), disconnect(i, i'), (σ, snd (netgmap sr s'))) ∈ trans
(opnet onp n2)
                 ∧ netgmap sr s' = netmask (net_tree_ips n2) (σ, snd (netgmap sr s'))"
     and tr: "(s, disconnect(i, i'), s') ∈ trans (pnet np (n1 ∥ n2))"
     and sr: "s ∈ reachable (pnet np (n1 ∥ n2)) TT"
     and nm: "netgmap sr s = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
     and "wf_net_tree (n1 ∥ n2)"
  from this(3) have "(s, disconnect(i, i'), s') ∈ pnet_sos (trans (pnet np n1))
                                                            (trans (pnet np n2))"
    by simp
  then obtain s1 s1' s2 s2' where "s = SubnetS s1 s2"
                              and "s' = SubnetS s1' s2'"
                              and "(s1, disconnect(i, i'), s1') ∈ trans (pnet np n1)"
                              and "(s2, disconnect(i, i'), s2') ∈ trans (pnet np n2)"
    by (rule partial_disconnectTE) auto
  from this(1) and nm have "netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
    by simp

  from ‹wf_net_tree (n1 ∥ n2)› have "wf_net_tree n1" and "wf_net_tree n2"
                                and "net_tree_ips n1 ∩ net_tree_ips n2 = {}" by auto

  from sr ‹s = SubnetS s1 s2› have "s1 ∈ reachable (pnet np n1) TT" by (metis subnet_reachable(1))
  hence "net_ips s1 = net_tree_ips n1" by (rule pnet_net_ips_net_tree_ips)

  from sr ‹s = SubnetS s1 s2› have "s2 ∈ reachable (pnet np n2) TT" by (metis subnet_reachable(2))
  hence "net_ips s2 = net_tree_ips n2" by (rule pnet_net_ips_net_tree_ips)

  from nm ‹s = SubnetS s1 s2›
    have "netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)" by simp
  hence "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
    using ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}› ‹net_ips s1 = net_tree_ips n1›
```

```
               and ‹net_ips s2 = net_tree_ips n2› by (rule netgmap_subnet_split1)
        with ‹(s1, disconnect(i, i'), s1') ∈ trans (pnet np n1)›
         and ‹s1 ∈ reachable (pnet np n1) TT›
         have "((σ, snd (netgmap sr s1)), disconnect(i, i'), (σ, snd (netgmap sr s1'))) ∈ trans (opnet onp
n1)"
          and "netgmap sr s1' = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1'))"
           using ‹wf_net_tree n1› unfolding atomize_conj by (rule IH1)

        from ‹netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)›
             ‹net_ips s1 = net_tree_ips n1› and ‹net_ips s2 = net_tree_ips n2›
          have "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
            by (rule netgmap_subnet_split2)
        with ‹(s2, disconnect(i, i'), s2') ∈ trans (pnet np n2)›
         and ‹s2 ∈ reachable (pnet np n2) TT›
         have "((σ, snd (netgmap sr s2)), disconnect(i, i'), (σ, snd (netgmap sr s2'))) ∈ trans (opnet onp
n2)"
          and "netgmap sr s2' = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2'))"
           using ‹wf_net_tree n2› unfolding atomize_conj by (rule IH2)

        have "((σ, snd (netgmap sr s)), disconnect(i, i'), (σ, snd (netgmap sr s')))
                                          ∈ trans (opnet onp (n1 ∥ n2))"
        proof -
          from ‹((σ, snd (netgmap sr s1)), disconnect(i, i'), (σ, snd (netgmap sr s1'))) ∈ trans (opnet
onp n1)›
           and ‹((σ, snd (netgmap sr s2)), disconnect(i, i'), (σ, snd (netgmap sr s2'))) ∈ trans (opnet
onp n2)›
            have "((σ, SubnetS (snd (netgmap sr s1)) (snd (netgmap sr s2))), disconnect(i, i'),
                   (σ, SubnetS (snd (netgmap sr s1')) (snd (netgmap sr s2'))))
                                            ∈ opnet_sos (trans (opnet onp n1)) (trans (opnet onp n2))"
              by (rule opnet_disconnect)
          with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› show ?thesis by simp
        qed

        moreover from ‹netgmap sr s1' = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1'))›
                      ‹netgmap sr s2' = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2'))›
                      ‹s' = SubnetS s1' s2'›
          have "netgmap sr s' = netmask (net_tree_ips (n1 ∥ n2)) (σ, snd (netgmap sr s'))" ..

        ultimately show "((σ, snd (netgmap sr s)), disconnect(i, i'), (σ, snd (netgmap sr s')))
                                                              ∈ trans (opnet onp (n1 ∥ n2))
                         ∧ netgmap sr s' = netmask (net_tree_ips (n1 ∥ n2)) (σ, snd (netgmap sr s'))" ..
      qed
    moreover from ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)› have "ζ = snd (netgmap sr s)" by simp
    ultimately show "∃σ' ζ'. ((σ, ζ), disconnect(i, i'), (σ', ζ')) ∈ trans (opnet onp n)
                             ∧ (∀j. j ∉ net_ips ζ ⟶ σ' j = σ j)
                             ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')" by auto
  qed

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 atomize_elim
    from assms(4,2,1) obtain i where "i∈net_ips s"
                               and "∀j. j≠i ⟶ netmap s' j = netmap s j"
                               and "net_ip_action np τ i n s s'"
      by (metis pnet_tau_single_node)
    from this(2) have "∀j. j≠i ⟶ fst (netgmap sr s') j = fst (netgmap sr s) j"
      by (clarsimp intro!: netmap_is_fst_netgmap')
    from ‹(s, τ, s') ∈ trans (pnet np n)› have "net_ips s' = net_ips s"
```

```
    by (rule pnet_maintains_dom [THEN sym])
  define σ' where "σ' j = (if j = i then the (fst (netgmap sr s') i) else σ j)" for j
  from ‹∀j. j≠i ⟶ fst (netgmap sr s') j = fst (netgmap sr s) j›
       and ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)›
    have "∀j. j≠i ⟶ σ' j = σ j"
      unfolding σ'_def by clarsimp

  from assms(2) have "net_ips s = net_tree_ips n"
    by (rule pnet_net_ips_net_tree_ips)

  from ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)›
    have "ζ = snd (netgmap sr s)" by simp

  from ‹∀j. j≠i ⟶ fst (netgmap sr s') j = fst (netgmap sr s) j› ‹i ∈ net_ips s›
       ‹net_ips s = net_tree_ips n› ‹net_ips s' = net_ips s›
       ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)›
    have "fst (netgmap sr s') = fst (netmask (net_tree_ips n) (σ', snd (netgmap sr s')))"
      unfolding σ'_def [abs_def] by - (rule ext, clarsimp)

  hence "netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))"
    by (rule prod_eqI, simp)

  with assms(1, 3)
    have "((σ, snd (netgmap sr s)), τ, (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n)"
      using assms(2,4) ‹i∈net_ips s› and ‹net_ip_action np τ i n s s'›
  proof (induction n arbitrary: s s' ζ)
    fix ii R_i ns ns' ζ
    assume "(ns, τ, ns') ∈ trans (pnet np ⟨ii; R_i⟩)"
       and nsr: "ns ∈ reachable (pnet np ⟨ii; R_i⟩) TT"
       and "netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)"
       and "netgmap sr ns' = netmask (net_tree_ips ⟨ii; R_i⟩) (σ', snd (netgmap sr ns'))"
       and "i∈net_ips ns"
    from this(1) have "(ns, τ, ns') ∈ node_sos (trans (np ii))"
      by (simp add: node_comps)
    moreover with nsr obtain s s' R R' where "ns  = NodeS ii s R"
                                       and "ns' = NodeS ii s' R'"
       by (metis net_node_reachable_is_node node_tauTE')
    moreover from ‹i ∈ net_ips ns› and ‹ns  = NodeS ii s R› have "ii = i" by simp
    ultimately have ntr: "(NodeS i s R, τ, NodeS i s' R') ∈ node_sos (trans (np i))"
      by simp
    hence "R' = R" by (metis net_state.inject(1) node_tauTE')

    from ntr obtain a where "(s, a, s') ∈ trans (np i)"
                      and "(∃d. a = ¬unicast d ∧ d∉R) ∨ (a = τ)"
      by (rule node_tauTE') auto

    from ‹netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)› ‹ns  = NodeS ii s R› and ‹ii = i›
      have "σ i = fst (sr s)" by simp (metis map_upd_Some_unfold)

    moreover from ‹netgmap sr ns' = netmask (net_tree_ips ⟨ii; R_i⟩) (σ', snd (netgmap sr ns'))›
                ‹ns' = NodeS ii s' R'› and ‹ii = i›
      have "σ' i = fst (sr s')"
        unfolding σ'_def [abs_def] by clarsimp (hypsubst_thin,
                                   metis (full_types, lifting) fun_upd_same option.sel)
    ultimately have "((σ, snd (sr s)), a, (σ', snd (sr s'))) ∈ trans (onp i)"
      using ‹(s, a, s') ∈ trans (np i)› by (rule trans)

    from ‹(∃d. a = ¬unicast d ∧ d∉R) ∨ (a = τ)› ‹∀j. j≠i ⟶ σ' j = σ j› ‹R'=R›
         and ‹((σ, snd (sr s)), a, (σ', snd (sr s'))) ∈ trans (onp i)›
      have "((σ, NodeS i (snd (sr s)) R), τ, (σ', NodeS i (snd (sr s')) R')) ∈ onode_sos (trans (onp i))"
        by (metis onode_sos.onode_notucast onode_sos.onode_tau)

    with ‹ns  = NodeS ii s R› ‹ns' = NodeS ii s' R'› ‹ii = i›
      show "((σ, snd (netgmap sr ns)), τ, (σ', snd (netgmap sr ns'))) ∈ trans (opnet onp ⟨ii; R_i⟩)"
```

```
      by (simp add: onode_comps)
next
  fix n1 n2 s s' ζ
  assume IH1: "⋀s s' ζ. (s, τ, s') ∈ trans (pnet np n1)
              ⟹ netgmap sr s = netmask (net_tree_ips n1) (σ, ζ)
              ⟹ netgmap sr s' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s'))
              ⟹ s ∈ reachable (pnet np n1) TT
              ⟹ wf_net_tree n1
              ⟹ i∈net_ips s
              ⟹ net_ip_action np τ i n1 s s'
              ⟹ ((σ, snd (netgmap sr s)), τ, (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n1)"
     and IH2: "⋀s s' ζ. (s, τ, s') ∈ trans (pnet np n2)
              ⟹ netgmap sr s = netmask (net_tree_ips n2) (σ, ζ)
              ⟹ netgmap sr s' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s'))
              ⟹ s ∈ reachable (pnet np n2) TT
              ⟹ wf_net_tree n2
              ⟹ i∈net_ips s
              ⟹ net_ip_action np τ i n2 s s'
              ⟹ ((σ, snd (netgmap sr s)), τ, (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n2)"
     and tr: "(s, τ, s') ∈ trans (pnet np (n1 ∥ n2))"
     and sr: "s ∈ reachable (pnet np (n1 ∥ n2)) TT"
     and nm: "netgmap sr s = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
     and nm': "netgmap sr s' = netmask (net_tree_ips (n1 ∥ n2)) (σ', snd (netgmap sr s'))"
     and "wf_net_tree (n1 ∥ n2)"
     and "i∈net_ips s"
     and "net_ip_action np τ i (n1 ∥ n2) s s'"
  from tr have "(s, τ, s') ∈ pnet_sos (trans (pnet np n1)) (trans (pnet np n2))" by simp
  then obtain s1 s1' s2 s2' where "s = SubnetS s1 s2"
                              and "s' = SubnetS s1' s2'"
    by (rule partial_tauTE) auto
  from this(1) and nm have "netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
    by simp
  from ‹s' = SubnetS s1' s2'› and nm'
    have "netgmap sr (SubnetS s1' s2') = netmask (net_tree_ips (n1 ∥ n2)) (σ', snd (netgmap sr s'))"
      by simp

  from ‹wf_net_tree (n1 ∥ n2)› have "wf_net_tree n1"
                               and "wf_net_tree n2"
                               and "net_tree_ips n1 ∩ net_tree_ips n2 = {}" by auto

  from sr [simplified ‹s = SubnetS s1 s2›] have "s1 ∈ reachable (pnet np n1) TT"
    by (rule subnet_reachable(1))
  hence "net_ips s1 = net_tree_ips n1" by (rule pnet_net_ips_net_tree_ips)

  from sr [simplified ‹s = SubnetS s1 s2›] have "s2 ∈ reachable (pnet np n2) TT"
    by (rule subnet_reachable(2))
  hence "net_ips s2 = net_tree_ips n2" by (rule pnet_net_ips_net_tree_ips)

  from nm [simplified ‹s = SubnetS s1 s2›]
       ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}›
       ‹net_ips s1 = net_tree_ips n1›
       ‹net_ips s2 = net_tree_ips n2›
    have "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
      by (rule netgmap_subnet_split1)

  from nm [simplified ‹s = SubnetS s1 s2›]
       ‹net_ips s1 = net_tree_ips n1›
       ‹net_ips s2 = net_tree_ips n2›
    have "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
      by (rule netgmap_subnet_split2)

  from ‹i∈net_ips s› and ‹s = SubnetS s1 s2› have "i∈net_ips s1 ∨ i∈net_ips s2" by auto
    thus "((σ, snd (netgmap sr s)), τ, (σ', snd (netgmap sr s'))) ∈ trans (opnet onp (n1 ∥ n2))"
  proof
```

```
    assume "i∈net_ips s1"
    with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹net_ip_action np τ i (n1 ∥ n2) s s'›
      have "(s1, τ, s1') ∈ trans (pnet np n1)"
       and "net_ip_action np τ i n1 s1 s1'"
       and "s2' = s2" by simp_all

    from ‹net_ips s1 = net_tree_ips n1› and ‹(s1, τ, s1') ∈ trans (pnet np n1)›
      have "net_ips s1' = net_tree_ips n1" by (metis pnet_maintains_dom)

    from nm' [simplified ‹s' = SubnetS s1' s2'› ‹s2' = s2›]
                ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}›
                ‹net_ips s1' = net_tree_ips n1›
                ‹net_ips s2 = net_tree_ips n2›
      have "netgmap sr s1' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s1'))"
        by (rule netgmap_subnet_split1)

    from ‹(s1, τ, s1') ∈ trans (pnet np n1)›
         ‹netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))›
         ‹netgmap sr s1' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s1'))›
         ‹s1 ∈ reachable (pnet np n1) TT›
         ‹wf_net_tree n1›
         ‹i∈net_ips s1›
         ‹net_ip_action np τ i n1 s1 s1'›
      have "((σ, snd (netgmap sr s1)), τ, (σ', snd (netgmap sr s1'))) ∈ trans (opnet onp n1)"
         by (rule IH1)

    with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹s2' = s2› show ?thesis
      by (simp del: step_node_tau) (erule opnet_tau1)
  next
    assume "i∈net_ips s2"
    with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹net_ip_action np τ i (n1 ∥ n2) s s'›
      have "(s2, τ, s2') ∈ trans (pnet np n2)"
       and "net_ip_action np τ i n2 s2 s2'"
       and "s1' = s1" by simp_all

    from ‹net_ips s2 = net_tree_ips n2› and ‹(s2, τ, s2') ∈ trans (pnet np n2)›
      have "net_ips s2' = net_tree_ips n2" by (metis pnet_maintains_dom)

    from nm' [simplified ‹s' = SubnetS s1' s2'› ‹s1' = s1›]
                ‹net_ips s1 = net_tree_ips n1›
                ‹net_ips s2' = net_tree_ips n2›
      have "netgmap sr s2' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s2'))"
        by (rule netgmap_subnet_split2)

    from ‹(s2, τ, s2') ∈ trans (pnet np n2)›
         ‹netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))›
         ‹netgmap sr s2' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s2'))›
         ‹s2 ∈ reachable (pnet np n2) TT›
         ‹wf_net_tree n2›
         ‹i∈net_ips s2›
         ‹net_ip_action np τ i n2 s2 s2'›
      have "((σ, snd (netgmap sr s2)), τ, (σ', snd (netgmap sr s2'))) ∈ trans (opnet onp n2)"
         by (rule IH2)

    with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹s1' = s1› show ?thesis
      by (simp del: step_node_tau) (erule opnet_tau2)
  qed
qed
with ‹ζ = snd (netgmap sr s)› have "((σ, ζ), τ, (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n)"
  by simp
moreover from ‹∀j. j≠i ⟶ σ' j = σ j› ‹i ∈ net_ips s› ‹ζ = snd (netgmap sr s)›
  have "∀j. j∉net_ips ζ ⟶ σ' j = σ j" by (metis net_ips_netgmap)
ultimately have "((σ, ζ), τ, (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n)
                 ∧ (∀j. j∉net_ips ζ ⟶ σ' j = σ j)
```

```
                    ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))"
      using ‹netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))› by simp
    thus "∃σ' ζ'. ((σ, ζ), τ, (σ', ζ')) ∈ trans (opnet onp n)
                ∧ (∀j. j∉net_ips ζ ⟶ σ' j = σ j)
                ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')" by auto
  qed

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 atomize_elim
    from assms(4,2,1) obtain "i∈net_ips s"
                       and "∀j. j≠i ⟶ netmap s' j = netmap s j"
                       and "net_ip_action np (i:deliver(d)) i n s s'"
      by (metis delivered_to_net_ips pnet_deliver_single_node)
    from this(2) have "∀j. j≠i ⟶ fst (netgmap sr s') j = fst (netgmap sr s) j"
      by (clarsimp intro!: netmap_is_fst_netgmap')
    from ‹(s, i:deliver(d), s') ∈ trans (pnet np n)› have "net_ips s' = net_ips s"
      by (rule pnet_maintains_dom [THEN sym])
    define σ' where "σ' j = (if j = i then the (fst (netgmap sr s') i) else σ j)" for j
    from ‹∀j. j≠i ⟶ fst (netgmap sr s') j = fst (netgmap sr s) j›
         and ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)›
      have "∀j. j≠i ⟶ σ' j = σ j"
        unfolding σ'_def by clarsimp

    from assms(2) have "net_ips s = net_tree_ips n"
      by (rule pnet_net_ips_net_tree_ips)

    from ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)›
      have "ζ = snd (netgmap sr s)" by simp

    from ‹∀j. j≠i ⟶ fst (netgmap sr s') j = fst (netgmap sr s) j› ‹i ∈ net_ips s›
         ‹net_ips s = net_tree_ips n› ‹net_ips s' = net_ips s›
         ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)›
      have "fst (netgmap sr s') = fst (netmask (net_tree_ips n) (σ', snd (netgmap sr s')))"
        unfolding σ'_def [abs_def] by - (rule ext, clarsimp)

    hence "netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))"
      by (rule prod_eqI, simp)

    with assms(1, 3)
      have "((σ, snd (netgmap sr s)), i:deliver(d), (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n)"
        using assms(2,4) ‹i∈net_ips s› and ‹net_ip_action np (i:deliver(d)) i n s s'›
    proof (induction n arbitrary: s s' ζ)
      fix ii Ri ns ns' ζ
      assume "(ns, i:deliver(d), ns') ∈ trans (pnet np ⟨ii; Ri⟩)"
         and nsr: "ns ∈ reachable (pnet np ⟨ii; Ri⟩) TT"
         and "netgmap sr ns = netmask (net_tree_ips ⟨ii; Ri⟩) (σ, ζ)"
         and "netgmap sr ns' = netmask (net_tree_ips ⟨ii; Ri⟩) (σ', snd (netgmap sr ns'))"
         and "i∈net_ips ns"
      from this(1) have "(ns, i:deliver(d), ns') ∈ node_sos (trans (np ii))"
        by (simp add: node_comps)
      moreover with nsr obtain s s' R R' where "ns  = NodeS ii s R"
                                         and "ns' = NodeS ii s' R'"
         by (metis net_node_reachable_is_node node_sos_dest)
      moreover from ‹i ∈ net_ips ns› and ‹ns  = NodeS ii s R› have "ii = i" by simp
      ultimately have ntr: "(NodeS i s R, i:deliver(d), NodeS i s' R') ∈ node_sos (trans (np i))"
        by simp
      hence "R' = R" by (metis net_state.inject(1) node_deliverTE')
```

```
      from ntr have "(s, deliver d, s') ∈ trans (np i)"
        by (rule node_deliverTE') simp

      from ‹netgmap sr ns = netmask (net_tree_ips ⟨ii; Rᵢ⟩) (σ, ζ)› ‹ns  = NodeS ii s R› and ‹ii = i›
        have "σ i = fst (sr s)" by simp (metis map_upd_Some_unfold)

      moreover from ‹netgmap sr ns' = netmask (net_tree_ips ⟨ii; Rᵢ⟩) (σ', snd (netgmap sr ns'))›
                    ‹ns' = NodeS ii s' R'› and ‹ii = i›
        have "σ' i = fst (sr s')"
          unfolding σ'_def [abs_def] by clarsimp (hypsubst_thin,
                                      metis (lifting, full_types) fun_upd_same option.sel)
      ultimately have "((σ, snd (sr s)), deliver d, (σ', snd (sr s'))) ∈ trans (onp i)"
        using ‹(s, deliver d, s') ∈ trans (np i)› by (rule trans)

      with ‹∀j. j≠i ⟶ σ' j = σ j› ‹R'=R›
        have "((σ, NodeS i (snd (sr s)) R), i:deliver(d), (σ', NodeS i (snd (sr s')) R'))
                                                                  ∈ onode_sos (trans (onp i))"
          by (metis onode_sos.onode_deliver)

      with ‹ns  = NodeS ii s R› ‹ns' = NodeS ii s' R'› ‹ii = i›
        show "((σ, snd (netgmap sr ns)), i:deliver(d), (σ', snd (netgmap sr ns'))) ∈ trans (opnet onp ⟨ii;
Rᵢ⟩)"
          by (simp add: onode_comps)
    next
      fix n1 n2 s s' ζ
      assume IH1: "⋀s s' ζ. (s, i:deliver(d), s') ∈ trans (pnet np n1)
                   ⟹ netgmap sr s = netmask (net_tree_ips n1) (σ, ζ)
                   ⟹ netgmap sr s' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s'))
                   ⟹ s ∈ reachable (pnet np n1) TT
                   ⟹ wf_net_tree n1
                   ⟹ i∈net_ips s
                   ⟹ net_ip_action np (i:deliver(d)) i n1 s s'
                   ⟹ ((σ, snd (netgmap sr s)), i:deliver(d), (σ', snd (netgmap sr s'))) ∈ trans (opnet
onp n1)"
         and IH2: "⋀s s' ζ. (s, i:deliver(d), s') ∈ trans (pnet np n2)
                   ⟹ netgmap sr s = netmask (net_tree_ips n2) (σ, ζ)
                   ⟹ netgmap sr s' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s'))
                   ⟹ s ∈ reachable (pnet np n2) TT
                   ⟹ wf_net_tree n2
                   ⟹ i∈net_ips s
                   ⟹ net_ip_action np (i:deliver(d)) i n2 s s'
                   ⟹ ((σ, snd (netgmap sr s)), i:deliver(d), (σ', snd (netgmap sr s'))) ∈ trans (opnet
onp n2)"
         and tr: "(s, i:deliver(d), s') ∈ trans (pnet np (n1 ∥ n2))"
         and sr: "s ∈ reachable (pnet np (n1 ∥ n2)) TT"
         and nm: "netgmap sr s = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
         and nm': "netgmap sr s' = netmask (net_tree_ips (n1 ∥ n2)) (σ', snd (netgmap sr s'))"
         and "wf_net_tree (n1 ∥ n2)"
         and "i∈net_ips s"
         and "net_ip_action np (i:deliver(d)) i (n1 ∥ n2) s s'"
      from tr have "(s, i:deliver(d), s') ∈ pnet_sos (trans (pnet np n1)) (trans (pnet np n2))" by simp
      then obtain s1 s1' s2 s2' where "s = SubnetS s1 s2"
                                  and "s' = SubnetS s1' s2'"
        by (rule partial_deliverTE) auto
      from this(1) and nm have "netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
        by simp
      from ‹s' = SubnetS s1' s2'› and nm'
        have "netgmap sr (SubnetS s1' s2') = netmask (net_tree_ips (n1 ∥ n2)) (σ', snd (netgmap sr s'))"
          by simp

      from ‹wf_net_tree (n1 ∥ n2)› have "wf_net_tree n1"
                                   and "wf_net_tree n2"
                                   and "net_tree_ips n1 ∩ net_tree_ips n2 = {}" by auto
```

```
      from sr [simplified ‹s = SubnetS s1 s2›] have "s1 ∈ reachable (pnet np n1) TT"
        by (rule subnet_reachable(1))
      hence "net_ips s1 = net_tree_ips n1" by (rule pnet_net_ips_net_tree_ips)

      from sr [simplified ‹s = SubnetS s1 s2›] have "s2 ∈ reachable (pnet np n2) TT"
        by (rule subnet_reachable(2))
      hence "net_ips s2 = net_tree_ips n2" by (rule pnet_net_ips_net_tree_ips)

      from nm [simplified ‹s = SubnetS s1 s2›]
           ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}›
           ‹net_ips s1 = net_tree_ips n1›
           ‹net_ips s2 = net_tree_ips n2›
        have "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
          by (rule netgmap_subnet_split1)

      from nm [simplified ‹s = SubnetS s1 s2›]
           ‹net_ips s1 = net_tree_ips n1›
           ‹net_ips s2 = net_tree_ips n2›
        have "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
          by (rule netgmap_subnet_split2)

      from ‹i∈net_ips s› and ‹s = SubnetS s1 s2› have "i∈net_ips s1 ∨ i∈net_ips s2" by auto
        thus "((σ, snd (netgmap sr s)), i:deliver(d), (σ', snd (netgmap sr s'))) ∈ trans (opnet onp (n1
∥ n2))"
      proof
        assume "i∈net_ips s1"
        with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹net_ip_action np (i:deliver(d)) i (n1 ∥ n2) s s'›
          have "(s1, i:deliver(d), s1') ∈ trans (pnet np n1)"
           and "net_ip_action np (i:deliver(d)) i n1 s1 s1'"
           and "s2' = s2" by simp_all

        from ‹net_ips s1 = net_tree_ips n1› and ‹(s1, i:deliver(d), s1') ∈ trans (pnet np n1)›
          have "net_ips s1' = net_tree_ips n1" by (metis pnet_maintains_dom)

        from nm' [simplified ‹s' = SubnetS s1' s2'› ‹s2' = s2›]
                     ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}›
                     ‹net_ips s1' = net_tree_ips n1›
                     ‹net_ips s2 = net_tree_ips n2›
          have "netgmap sr s1' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s1'))"
            by (rule netgmap_subnet_split1)

        from ‹(s1, i:deliver(d), s1') ∈ trans (pnet np n1)›
             ‹netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))›
             ‹netgmap sr s1' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s1'))›
             ‹s1 ∈ reachable (pnet np n1) TT›
             ‹wf_net_tree n1›
             ‹i∈net_ips s1›
             ‹net_ip_action np (i:deliver(d)) i n1 s1 s1'›
          have "((σ, snd (netgmap sr s1)), i:deliver(d), (σ', snd (netgmap sr s1'))) ∈ trans (opnet onp
n1)"
             by (rule IH1)

        with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹s2' = s2› show ?thesis
          by simp (erule opnet_deliver1)
      next
        assume "i∈net_ips s2"
        with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹net_ip_action np (i:deliver(d)) i (n1 ∥ n2) s s'›
          have "(s2, i:deliver(d), s2') ∈ trans (pnet np n2)"
           and "net_ip_action np (i:deliver(d)) i n2 s2 s2'"
           and "s1' = s1" by simp_all

        from ‹net_ips s2 = net_tree_ips n2› and ‹(s2, i:deliver(d), s2') ∈ trans (pnet np n2)›
          have "net_ips s2' = net_tree_ips n2" by (metis pnet_maintains_dom)
```

```
      from nm' [simplified ‹s' = SubnetS s1' s2'› ‹s1' = s1›]
                ‹net_ips s1 = net_tree_ips n1›
                ‹net_ips s2' = net_tree_ips n2›
        have "netgmap sr s2' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s2'))"
          by (rule netgmap_subnet_split2)

      from ‹(s2, i:deliver(d), s2') ∈ trans (pnet np n2)›
           ‹netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))›
           ‹netgmap sr s2' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s2'))›
           ‹s2 ∈ reachable (pnet np n2) TT›
           ‹wf_net_tree n2›
           ‹i∈net_ips s2›
           ‹net_ip_action np (i:deliver(d)) i n2 s2 s2'›
        have "((σ, snd (netgmap sr s2)), i:deliver(d), (σ', snd (netgmap sr s2'))) ∈ trans (opnet onp
n2)"
           by (rule IH2)

      with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹s1' = s1› show ?thesis
        by simp (erule opnet_deliver2)
    qed
  qed
  with ‹ζ = snd (netgmap sr s)› have "((σ, ζ), i:deliver(d), (σ', snd (netgmap sr s'))) ∈ trans (opnet
onp n)"
    by simp
  moreover from ‹∀j. j≠i ⟶ σ' j = σ j› ‹i ∈ net_ips s› ‹ζ = snd (netgmap sr s)›
    have "∀j. j∉net_ips ζ ⟶ σ' j = σ j" by (metis net_ips_netgmap)
  ultimately have "((σ, ζ), i:deliver(d), (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n)
                   ∧ (∀j. j∉net_ips ζ ⟶ σ' j = σ j)
                   ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))"
    using ‹netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))› by simp
  thus "∃σ' ζ'. ((σ, ζ), i:deliver(d), (σ', ζ')) ∈ trans (opnet onp n)
                ∧ (∀j. j∉net_ips ζ ⟶ σ' j = σ j)
                ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')" by auto
qed

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 -

    from assms(2) have "net_ips s = net_tree_ips n" ..
    with assms(1) have "net_ips s' = net_tree_ips n"
      by (metis pnet_maintains_dom)

    from ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)›
      have "ζ = snd (netgmap sr s)" by simp

    from ‹netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')›
      have "ζ' = snd (netgmap sr s')"
       and "netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))"
        by simp_all

    from assms(1-3) ‹netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))› assms(5)
      have "((σ, snd (netgmap sr s)), H¬K:arrive(m), (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n)"
      proof (induction n arbitrary: s s' ζ H K)
        fix ii R_i ns ns' ζ H K
        assume "(ns, H¬K:arrive(m), ns') ∈ trans (pnet np ⟨ii; R_i⟩)"
           and nsr: "ns ∈ reachable (pnet np ⟨ii; R_i⟩) TT"
           and "netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)"
```

```
      and "netgmap sr ns' = netmask (net_tree_ips ⟨ii; R_i⟩) (σ', snd (netgmap sr ns'))"
    from this(1) have "(ns, H¬K:arrive(m), ns') ∈ node_sos (trans (np ii))"
      by (simp add: node_comps)
    moreover with nsr obtain s s' R where "ns  = NodeS ii s R"
                                      and "ns' = NodeS ii s' R"
      by (metis net_node_reachable_is_node node_arriveTE')
    ultimately have "(NodeS ii s R, H¬K:arrive(m), NodeS ii s' R) ∈ node_sos (trans (np ii))"
      by simp
    from this(1) have "((σ, NodeS ii (snd (sr s)) R), H¬K:arrive(m), (σ', NodeS ii (snd (sr s')) R))
                                                                ∈ onode_sos (trans (onp ii))"
    proof (rule node_arriveTE)
      assume "(s, receive m, s') ∈ trans (np ii)"
         and "H = {ii}"
         and "K = {}"
      from ‹netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)› and ‹ns  = NodeS ii s R›
        have "σ ii = fst (sr s)"
          by simp (metis map_upd_Some_unfold)
      moreover from ‹netgmap sr ns' = netmask (net_tree_ips ⟨ii; R_i⟩) (σ', snd (netgmap sr ns'))›
                and ‹ns' = NodeS ii s' R›
        have "σ' ii = fst (sr s')" by simp (metis map_upd_Some_unfold)
      ultimately have "((σ, snd (sr s)), receive m, (σ', snd (sr s'))) ∈ trans (onp ii)"
        using ‹(s, receive m, s') ∈ trans (np ii)› by (rule trans)
      hence "((σ, NodeS ii (snd (sr s)) R), {ii}¬{}:arrive(m), (σ', NodeS ii (snd (sr s')) R))
                                                                ∈ onode_sos (trans (onp ii))"
        by (rule onode_receive)
      with ‹H={ii}› and ‹K={}›
        show "((σ, NodeS ii (snd (sr s)) R), H¬K:arrive(m), (σ', NodeS ii (snd (sr s')) R))
                                                                ∈ onode_sos (trans (onp ii))"
          by simp
    next
      assume "H = {}"
         and "s' = s"
         and "K = {ii}"
      from ‹s' = s› ‹netgmap sr ns' = netmask (net_tree_ips ⟨ii; R_i⟩) (σ', snd (netgmap sr ns'))›
                    ‹netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)›
                    ‹ns = NodeS ii s R› and ‹ns' = NodeS ii s' R›
        have "σ' ii = σ ii" by simp (metis option.sel)
      hence "((σ, NodeS ii (snd (sr s)) R), {}¬{ii}:arrive(m), (σ', NodeS ii (snd (sr s)) R))
                                                                ∈ onode_sos (trans (onp ii))"
        by (rule onode_arrive)
      with ‹H={}› ‹K={ii}› and ‹s' = s›
      show "((σ, NodeS ii (snd (sr s)) R), H¬K:arrive(m), (σ', NodeS ii (snd (sr s')) R))
                                                               ∈ onode_sos (trans (onp ii))"
        by simp
    qed
  with ‹ns = NodeS ii s R› ‹ns' = NodeS ii s' R›
    show "((σ, snd (netgmap sr ns)), H¬K:arrive(m), (σ', snd (netgmap sr ns')))
                                                       ∈ trans (opnet onp ⟨ii; R_i⟩)"
      by (simp add: onode_comps)
next
  fix n1 n2 s s' ζ H K
  assume IH1: "⋀s s' ζ H K. (s, H¬K:arrive(m), s') ∈ trans (pnet np n1)
                   ⟹ s ∈ reachable (pnet np n1) TT
                   ⟹ netgmap sr s = netmask (net_tree_ips n1) (σ, ζ)
                   ⟹ netgmap sr s' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s'))
                   ⟹ wf_net_tree n1
                   ⟹ ((σ, snd (netgmap sr s)), H¬K:arrive(m), σ', snd (netgmap sr s'))
                                                                 ∈ trans (opnet onp n1)"
    and IH2: "⋀s s' ζ H K. (s, H¬K:arrive(m), s') ∈ trans (pnet np n2)
                   ⟹ s ∈ reachable (pnet np n2) TT
                   ⟹ netgmap sr s = netmask (net_tree_ips n2) (σ, ζ)
                   ⟹ netgmap sr s' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s'))
                   ⟹ wf_net_tree n2
                   ⟹ ((σ, snd (netgmap sr s)), H¬K:arrive(m), σ', snd (netgmap sr s'))
```

```
                                                      ∈ trans (opnet onp n2)"
  and "(s, H¬K:arrive(m), s') ∈ trans (pnet np (n1 ∥ n2))"
  and sr: "s ∈ reachable (pnet np (n1 ∥ n2)) TT"
  and nm: "netgmap sr s = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
  and nm': "netgmap sr s' = netmask (net_tree_ips (n1 ∥ n2)) (σ', snd (netgmap sr s'))"
  and "wf_net_tree (n1 ∥ n2)"
from this(3) have "(s, H¬K:arrive(m), s') ∈ pnet_sos (trans (pnet np n1))
                                               (trans (pnet np n2))"
  by simp
thus "((σ, snd (netgmap sr s)), H¬K:arrive(m), (σ', snd (netgmap sr s')))
                                                   ∈ trans (opnet onp (n1 ∥ n2))"
proof (rule partial_arriveTE)
  fix s1 s1' s2 s2' H1 H2 K1 K2
  assume "s = SubnetS s1 s2"
     and "s' = SubnetS s1' s2'"
     and tr1: "(s1, H1¬K1:arrive(m), s1') ∈ trans (pnet np n1)"
     and tr2: "(s2, H2¬K2:arrive(m), s2') ∈ trans (pnet np n2)"
     and "H = H1 ∪ H2"
     and "K = K1 ∪ K2"

  from ‹wf_net_tree (n1 ∥ n2)› have "wf_net_tree n1"
                                and "wf_net_tree n2"
                                and "net_tree_ips n1 ∩ net_tree_ips n2 = {}" by auto

  from sr [simplified ‹s = SubnetS s1 s2›] have "s1 ∈ reachable (pnet np n1) TT"
    by (rule subnet_reachable(1))
  hence "net_ips s1 = net_tree_ips n1" by (rule pnet_net_ips_net_tree_ips)
  with tr1 have "net_ips s1' = net_tree_ips n1" by (metis pnet_maintains_dom)

  from sr [simplified ‹s = SubnetS s1 s2›] have "s2 ∈ reachable (pnet np n2) TT"
    by (rule subnet_reachable(2))
  hence "net_ips s2 = net_tree_ips n2" by (rule pnet_net_ips_net_tree_ips)
  with tr2 have "net_ips s2' = net_tree_ips n2" by (metis pnet_maintains_dom)

  from ‹(s1, H1¬K1:arrive(m), s1') ∈ trans (pnet np n1)›
       ‹s1 ∈ reachable (pnet np n1) TT›
    have "((σ, snd (netgmap sr s1)), H1¬K1:arrive(m), (σ', snd (netgmap sr s1')))
                                                          ∈ trans (opnet onp n1)"
    proof (rule IH1 [OF _ _ _ _ ‹wf_net_tree n1›])
      from nm [simplified ‹s = SubnetS s1 s2›]
           ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}›
           ‹net_ips s1 = net_tree_ips n1›
           ‹net_ips s2 = net_tree_ips n2›
        show "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
          by (rule netgmap_subnet_split1)
    next
      from nm' [simplified ‹s' = SubnetS s1' s2'›]
           ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}›
           ‹net_ips s1' = net_tree_ips n1›
           ‹net_ips s2' = net_tree_ips n2›
        show "netgmap sr s1' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s1'))"
          by (rule netgmap_subnet_split1)
    qed

  moreover from ‹(s2, H2¬K2:arrive(m), s2') ∈ trans (pnet np n2)›
                ‹s2 ∈ reachable (pnet np n2) TT›
    have "((σ, snd (netgmap sr s2)), H2¬K2:arrive(m), (σ', snd (netgmap sr s2')))
                                                          ∈ trans (opnet onp n2)"
    proof (rule IH2 [OF _ _ _ _ ‹wf_net_tree n2›])
      from nm [simplified ‹s = SubnetS s1 s2›]
           ‹net_ips s1 = net_tree_ips n1›
           ‹net_ips s2 = net_tree_ips n2›
        show "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
          by (rule netgmap_subnet_split2)
```

```
        next
          from nm' [simplified ‹s' = SubnetS s1' s2'›]
               ‹net_ips s1' = net_tree_ips n1›
               ‹net_ips s2' = net_tree_ips n2›
            show "netgmap sr s2' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s2'))"
              by (rule netgmap_subnet_split2)
        qed
      ultimately show "((σ, snd (netgmap sr s)), H¬K:arrive(m), (σ', snd (netgmap sr s')))
                                                          ∈ trans (opnet onp (n1 ∥ n2))"
        using ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› ‹H = H1 ∪ H2› ‹K = K1 ∪ K2›
          by simp (rule opnet_sos.opnet_arrive)
    qed
  qed
  with ‹ζ = snd (netgmap sr s)› and ‹ζ' = snd (netgmap sr s')›
    show "((σ, ζ), H¬K:arrive(m), (σ', ζ')) ∈ trans (opnet onp n)"
      by simp
qed

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 atomize_elim
  define σ' where "σ' i = (if i∈net_tree_ips n then the (fst (netgmap sr s') i) else σ i)" for i

  from assms(2) have "net_ips s = net_tree_ips n"
    by (rule pnet_net_ips_net_tree_ips)
  with assms(1) have "net_ips s' = net_tree_ips n"
    by (metis pnet_maintains_dom)

  have "netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))"
  proof (rule prod_eqI)
    from ‹net_ips s' = net_tree_ips n›
      show "fst (netgmap sr s') = fst (netmask (net_tree_ips n) (σ', snd (netgmap sr s')))"
        unfolding σ'_def [abs_def] by - (rule ext, clarsimp)
  qed simp

  moreover with assms(1-3)
  have "((σ, ζ), H¬K:arrive(m), (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n)"
    using ‹wf_net_tree n› by (rule transfer_arrive')

  moreover have "∀j. j∉net_ips ζ ⟶ σ' j = σ j"
  proof -
    have "∀j. j∉net_tree_ips n ⟶ σ' j = σ j" unfolding σ'_def by simp
    with assms(3) and ‹net_ips s = net_tree_ips n›
      show ?thesis
        by clarsimp (metis (mono_tags) net_ips_netgmap snd_conv)
  qed

  ultimately show "∃σ' ζ'. ((σ, ζ), H¬K:arrive(m), (σ', ζ')) ∈ trans (opnet onp n)
                         ∧ (∀j. j∉net_ips ζ ⟶ σ' j = σ j)
                         ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')" by auto
qed

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 atomize_elim
  define σ' where "σ' i = (if i∈net_tree_ips n then the (fst (netgmap sr s') i) else σ i)" for i

  from assms(2) have "net_ips s = net_tree_ips n" ..
  with assms(1) have "net_ips s' = net_tree_ips n"
    by (metis pnet_maintains_dom)
  have "netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))"
  proof (rule prod_eqI)
    from ‹net_ips s' = net_tree_ips n›
      show "fst (netgmap sr s') = fst (netmask (net_tree_ips n) (σ', snd (netgmap sr s')))"
    unfolding σ'_def [abs_def] by - (rule ext, clarsimp simp add: some_the_fst_netgmap)
  qed simp

  from ‹net_ips s' = net_tree_ips n› and ‹net_ips s = net_tree_ips n›
    have "∀j. j∉net_ips (snd (netgmap sr s)) ⟶ σ' j = σ j"
      unfolding σ'_def by simp

  from ‹netgmap sr s = netmask (net_tree_ips n) (σ, ζ)›
    have "ζ = snd (netgmap sr s)" by simp

  from assms(1-3) ‹netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))› assms(4)
    have "((σ, snd (netgmap sr s)), mR:*cast(m), (σ', snd (netgmap sr s'))) ∈ trans (opnet onp n)"
    proof (induction n arbitrary: s s' ζ mR)
      fix ii R_i ns ns' ζ mR
      assume "(ns, mR:*cast(m), ns') ∈ trans (pnet np ⟨ii; R_i⟩)"
         and nsr: "ns ∈ reachable (pnet np ⟨ii; R_i⟩) TT"
         and "netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)"
         and "netgmap sr ns' = netmask (net_tree_ips ⟨ii; R_i⟩) (σ', snd (netgmap sr ns'))"
      from this(1) have "(ns, mR:*cast(m), ns') ∈ node_sos (trans (np ii))"
        by (simp add: node_comps)
      moreover with nsr obtain s s' R where "ns  = NodeS ii s R"
                                        and "ns' = NodeS ii s' R"
        by (metis net_node_reachable_is_node node_castTE')
      ultimately have "(NodeS ii s R, mR:*cast(m), NodeS ii s' R) ∈ node_sos (trans (np ii))"
        by simp

      from ‹netgmap sr ns = netmask (net_tree_ips ⟨ii; R_i⟩) (σ, ζ)› and ‹ns  = NodeS ii s R›
        have "σ ii = fst (sr s)"
          by simp (metis map_upd_Some_unfold)
      from ‹netgmap sr ns' = netmask (net_tree_ips ⟨ii; R_i⟩) (σ', snd (netgmap sr ns'))›
          and ‹ns' = NodeS ii s' R›
        have "σ' ii = fst (sr s')" by simp (metis map_upd_Some_unfold)

      from ‹(NodeS ii s R, mR:*cast(m), NodeS ii s' R) ∈ node_sos (trans (np ii))›
        have "((σ, NodeS ii (snd (sr s)) R), mR:*cast(m), (σ', NodeS ii (snd (sr s')) R))
                                                              ∈ onode_sos (trans (onp ii))"
      proof (rule node_castTE)
        assume "(s, broadcast m, s') ∈ trans (np ii)"
           and "mR = R"
        from ‹σ ii = fst (sr s)› ‹σ' ii = fst (sr s')› and this(1)
          have "((σ, snd (sr s)), broadcast m, (σ', snd (sr s'))) ∈ trans (onp ii)"
            by (rule trans)
        hence "((σ, NodeS ii (snd (sr s)) R), R:*cast(m), (σ', NodeS ii (snd (sr s')) R))
                                                              ∈ onode_sos (trans (onp ii))"
          by (rule onode_bcast)
        with ‹mR = R› show "((σ, NodeS ii (snd (sr s)) R), mR:*cast(m), (σ', NodeS ii (snd (sr s')) R))
                                                              ∈ onode_sos (trans (onp ii))"
            by simp
      next
        fix D
        assume "(s, groupcast D m, s') ∈ trans (np ii)"
           and "mR = R ∩ D"
```

```
        from ‹σ ii = fst (sr s)› ‹σ' ii = fst (sr s')› and this(1)
          have "((σ, snd (sr s)), groupcast D m, (σ', snd (sr s'))) ∈ trans (onp ii)"
            by (rule trans)
        hence "((σ, NodeS ii (snd (sr s)) R), (R ∩ D):*cast(m), (σ', NodeS ii (snd (sr s')) R))
                                                                ∈ onode_sos (trans (onp ii))"
          by (rule onode_gcast)
        with ‹mR = R ∩ D› show "((σ, NodeS ii (snd (sr s)) R), mR:*cast(m), (σ', NodeS ii (snd (sr s'))
R))
                                                                ∈ onode_sos (trans (onp ii))"
            by simp
      next
        fix d
        assume "(s, unicast d m, s') ∈ trans (np ii)"
           and "d ∈ R"
           and "mR = {d}"
        from ‹σ ii = fst (sr s)› ‹σ' ii = fst (sr s')› and this(1)
          have "((σ, snd (sr s)), unicast d m, (σ', snd (sr s'))) ∈ trans (onp ii)"
            by (rule trans)
        hence "((σ, NodeS ii (snd (sr s)) R), {d}:*cast(m), (σ', NodeS ii (snd (sr s')) R))
                                                                ∈ onode_sos (trans (onp ii))"
          using ‹d∈R› by (rule onode_ucast)
        with ‹mR={d}› show "((σ, NodeS ii (snd (sr s)) R), mR:*cast(m), (σ', NodeS ii (snd (sr s')) R))
                                                                ∈ onode_sos (trans (onp ii))"
          by simp
      qed
    with ‹ns = NodeS ii s R› ‹ns' = NodeS ii s' R›
      show "((σ, snd (netgmap sr ns)), mR:*cast(m), (σ', snd (netgmap sr ns')))
                                                       ∈ trans (opnet onp ⟨ii; R_i⟩)"
        by (simp add: onode_comps)
  next
    fix n1 n2 s s' ζ mR
    assume IH1: "⋀s s' ζ mR. (s, mR:*cast(m), s') ∈ trans (pnet np n1)
                     ⟹ s ∈ reachable (pnet np n1) TT
                     ⟹ netgmap sr s = netmask (net_tree_ips n1) (σ, ζ)
                     ⟹ netgmap sr s' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s'))
                     ⟹ wf_net_tree n1
                     ⟹ ((σ, snd (netgmap sr s)), mR:*cast(m), σ', snd (netgmap sr s'))
                                                                  ∈ trans (opnet onp n1)"
      and IH2: "⋀s s' ζ mR. (s, mR:*cast(m), s') ∈ trans (pnet np n2)
                     ⟹ s ∈ reachable (pnet np n2) TT
                     ⟹ netgmap sr s = netmask (net_tree_ips n2) (σ, ζ)
                     ⟹ netgmap sr s' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s'))
                     ⟹ wf_net_tree n2
                     ⟹ ((σ, snd (netgmap sr s)), mR:*cast(m), σ', snd (netgmap sr s'))
                                                                  ∈ trans (opnet onp n2)"
      and "(s, mR:*cast(m), s') ∈ trans (pnet np (n1 ∥ n2))"
      and sr: "s ∈ reachable (pnet np (n1 ∥ n2)) TT"
      and nm: "netgmap sr s = netmask (net_tree_ips (n1 ∥ n2)) (σ, ζ)"
      and nm': "netgmap sr s' = netmask (net_tree_ips (n1 ∥ n2)) (σ', snd (netgmap sr s'))"
      and "wf_net_tree (n1 ∥ n2)"
    from this(3) have "(s, mR:*cast(m), s') ∈ pnet_sos (trans (pnet np n1)) (trans (pnet np n2))"
      by simp
    then obtain s1 s1' s2 s2' H K
      where "s  = SubnetS s1  s2"
        and "s' = SubnetS s1' s2'"
        and "H ⊆ mR"
        and "K ∩ mR = {}"
        and trtr: "((s1, mR:*cast(m), s1') ∈ trans (pnet np n1)
                          ∧ (s2, H¬K:arrive(m), s2') ∈ trans (pnet np n2))
                  ∨ ((s1, H¬K:arrive(m), s1') ∈ trans (pnet np n1)
                          ∧ (s2, mR:*cast(m), s2') ∈ trans (pnet np n2))"
        by (rule partial_castTE) metis+

    from ‹wf_net_tree (n1 ∥ n2)› have "wf_net_tree n1"
```

```
                              and "wf_net_tree n2"
                              and "net_tree_ips n1 ∩ net_tree_ips n2 = {}" by auto

    from sr [simplified ‹s = SubnetS s1 s2›] have "s1 ∈ reachable (pnet np n1) TT"
      by (rule subnet_reachable(1))
    hence "net_ips s1 = net_tree_ips n1" by (rule pnet_net_ips_net_tree_ips)
    with trtr have "net_ips s1' = net_tree_ips n1" by (metis pnet_maintains_dom)

    from sr [simplified ‹s = SubnetS s1 s2›] have "s2 ∈ reachable (pnet np n2) TT"
      by (rule subnet_reachable(2))
    hence "net_ips s2 = net_tree_ips n2" by (rule pnet_net_ips_net_tree_ips)
    with trtr have "net_ips s2' = net_tree_ips n2" by (metis pnet_maintains_dom)

    from nm [simplified ‹s = SubnetS s1 s2›]
         ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}›
         ‹net_ips s1 = net_tree_ips n1›
         ‹net_ips s2 = net_tree_ips n2›
      have "netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"
        by (rule netgmap_subnet_split1)

    from nm' [simplified ‹s' = SubnetS s1' s2'›]
         ‹net_tree_ips n1 ∩ net_tree_ips n2 = {}›
         ‹net_ips s1' = net_tree_ips n1›
         ‹net_ips s2' = net_tree_ips n2›
      have "netgmap sr s1' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s1'))"
        by (rule netgmap_subnet_split1)

    from nm [simplified ‹s = SubnetS s1 s2›]
         ‹net_ips s1 = net_tree_ips n1›
         ‹net_ips s2 = net_tree_ips n2›
      have "netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))"
        by (rule netgmap_subnet_split2)

    from nm' [simplified ‹s' = SubnetS s1' s2'›]
         ‹net_ips s1' = net_tree_ips n1›
         ‹net_ips s2' = net_tree_ips n2›
      have "netgmap sr s2' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s2'))"
        by (rule netgmap_subnet_split2)

    from trtr show "((σ, snd (netgmap sr s)), mR:*cast(m), (σ', snd (netgmap sr s')))
                                                        ∈ trans (opnet onp (n1 ∥ n2))"
    proof (elim disjE conjE)
      assume "(s1, mR:*cast(m), s1') ∈ trans (pnet np n1)"
         and "(s2, H¬K:arrive(m), s2') ∈ trans (pnet np n2)"
      from ‹(s1, mR:*cast(m), s1') ∈ trans (pnet np n1)›
           ‹s1 ∈ reachable (pnet np n1) TT›
           ‹netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))›
           ‹netgmap sr s1' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s1'))›
           ‹wf_net_tree n1›
        have "((σ, snd (netgmap sr s1)), mR:*cast(m), (σ', snd (netgmap sr s1'))) ∈ trans (opnet onp n1)"
          by (rule IH1)

      moreover from ‹(s2, H¬K:arrive(m), s2') ∈ trans (pnet np n2)›
           ‹s2 ∈ reachable (pnet np n2) TT›
           ‹netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))›
           ‹netgmap sr s2' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s2'))›
           ‹wf_net_tree n2›
        have "((σ, snd (netgmap sr s2)), H¬K:arrive(m), (σ', snd (netgmap sr s2'))) ∈ trans (opnet onp
n2)"
          by (rule transfer_arrive')

      ultimately have "((σ, SubnetS (snd (netgmap sr s1)) (snd (netgmap sr s2))), mR:*cast(m),
                       (σ', SubnetS (snd (netgmap sr s1')) (snd (netgmap sr s2'))))
                          ∈ opnet_sos (trans (opnet onp n1)) (trans (opnet onp n2))"
```

```
        using ‹H ⊆ mR› and ‹K ∩ mR = {}› by (rule opnet_sos.intros(1))
      with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› show ?thesis by simp
    next
      assume "(s1, H¬K:arrive(m), s1') ∈ trans (pnet np n1)"
         and "(s2, mR:*cast(m), s2') ∈ trans (pnet np n2)"
      from ‹(s1, H¬K:arrive(m), s1') ∈ trans (pnet np n1)›
           ‹s1 ∈ reachable (pnet np n1) TT›
           ‹netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))›
           ‹netgmap sr s1' = netmask (net_tree_ips n1) (σ', snd (netgmap sr s1'))›
           ‹wf_net_tree n1›
        have "((σ, snd (netgmap sr s1)), H¬K:arrive(m), (σ', snd (netgmap sr s1'))) ∈ trans (opnet onp
n1)"
          by (rule transfer_arrive')

      moreover from ‹(s2, mR:*cast(m), s2') ∈ trans (pnet np n2)›
           ‹s2 ∈ reachable (pnet np n2) TT›
           ‹netgmap sr s2 = netmask (net_tree_ips n2) (σ, snd (netgmap sr s2))›
           ‹netgmap sr s2' = netmask (net_tree_ips n2) (σ', snd (netgmap sr s2'))›
           ‹wf_net_tree n2›
        have "((σ, snd (netgmap sr s2)), mR:*cast(m), (σ', snd (netgmap sr s2'))) ∈ trans (opnet onp n2)"
          by (rule IH2)

      ultimately have "((σ, SubnetS (snd (netgmap sr s1)) (snd (netgmap sr s2))), mR:*cast(m),
                        (σ', SubnetS (snd (netgmap sr s1')) (snd (netgmap sr s2'))))
                          ∈ opnet_sos (trans (opnet onp n1)) (trans (opnet onp n2))"
        using ‹H ⊆ mR› and ‹K ∩ mR = {}› by (rule opnet_sos.intros(2))
      with ‹s = SubnetS s1 s2› ‹s' = SubnetS s1' s2'› show ?thesis by simp
    qed
  qed
  with ‹ζ = snd (netgmap sr s)› have "((σ, ζ), mR:*cast(m), (σ', snd (netgmap sr s'))) ∈ trans (opnet
onp n)"
    by simp
  moreover from ‹∀j. j∉net_ips (snd (netgmap sr s)) ⟶ σ' j = σ j› ‹ζ = snd (netgmap sr s)›
    have "∀j. j∉net_ips ζ ⟶ σ' j = σ j" by simp
  moreover note ‹netgmap sr s' = netmask (net_tree_ips n) (σ', snd (netgmap sr s'))›
  ultimately show "∃σ' ζ'. ((σ, ζ), mR:*cast(m), (σ', ζ')) ∈ trans (opnet onp n)
                          ∧ (∀j. j∉net_ips ζ ⟶ σ' j = σ j)
                          ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
    by auto
qed

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 atomize_elim
    show "∃σ' ζ'. ((σ, ζ), a, (σ', ζ')) ∈ trans (opnet onp n)
                 ∧ (∀j. j∉net_ips ζ ⟶ σ' j = σ j)
                 ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
    proof (cases a)
      case node_cast
      with assms(4) show ?thesis
        by (auto elim!: transfer_cast [OF _ assms(1-3)])
    next
      case node_deliver
      with assms(4) show ?thesis
        by (auto elim!: transfer_deliver [OF _ assms(1-3)])
    next
      case node_arrive
      with assms(4) show ?thesis
```

```
        by (auto elim!: transfer_arrive [OF _ assms(1-3)])
    next
      case node_connect
      with assms(4) show ?thesis
        by (auto elim!: transfer_connect [OF _ assms(1-3)])
    next
      case node_disconnect
      with assms(4) show ?thesis
        by (auto elim!: transfer_disconnect [OF _ assms(1-3)])
    next
      case node_newpkt
      with assms(4) have False by (metis pnet_never_newpkt)
      thus ?thesis ..
    next
      case node_tau
      with assms(4) show ?thesis
        by (auto elim!: transfer_tau [OF _ assms(1-3), simplified])
    qed
  qed

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 (atomize_elim)
    from assms have "(s, a, s') ∈ cnet_sos (trans (pnet np n))" by simp
    thus "∃a'. (s, a', s') ∈ trans (pnet np n)
               ∧ (∀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 cases
      case (cnet_cast R m) thus ?thesis
      by (auto intro!: exI [where x="R:*cast(m)"] dest!: ocnet_cast)
    qed (auto intro!: ocnet_sos.intros [simplified])
  qed

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 atomize_elim
    from assms(1) have "s ∈ reachable (pnet np n) TT" ..
    from assms(4)
      show "∃σ' ζ'. ((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))
                   ∧ netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
        by (cases a)
           ((elim transfer_action_pnet_closed
                  transfer_pnet_action [OF ‹s ∈ reachable (pnet np n) TT› assms(2-3)])?,
            (auto intro!: exI)[1])+
  qed

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")
  using assms(2) proof induction
    fix s
```

```
    assume "s ∈ init (closed (pnet np n))"
    hence "s ∈ init (pnet np n)" by simp
    with ‹wf_net_tree n› have "netgmap sr s ∈ netmask (net_tree_ips n) ` init (opnet onp n)"
      by (rule init_pnet_opnet)
    hence "netgmap sr s ∈ netmask (net_tree_ips n) ` init (oclosed (opnet onp n))"
      by simp
    moreover have "netmask (net_tree_ips n) ` init (oclosed (opnet onp n))
                                  ⊆ netmask (net_tree_ips n) ` ?oreachable n"
      by (intro image_mono subsetI) (rule oreachable_init)
    ultimately show "netgmap sr s ∈ netmask (net_tree_ips n) ` ?oreachable n"
      by (rule rev_subsetD)
  next
    fix s a s'
    assume "s ∈ reachable (closed (pnet np n)) TT"
       and "netgmap sr s ∈ netmask (net_tree_ips n) ` ?oreachable n"
       and "(s, a, s') ∈ trans (closed (pnet np n))"
    from this(2) obtain σ ζ where "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
                              and "(σ, ζ) ∈ ?oreachable n"
      by clarsimp
    from ‹s ∈ reachable (closed (pnet np n)) TT› this(1) ‹wf_net_tree n›
         and ‹(s, a, s') ∈ trans (closed (pnet np n))›
      obtain σ' ζ' where "((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))"
                     and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
        by (rule transfer_action)
    from ‹(σ, ζ) ∈ ?oreachable n› and this(1) have "(σ', ζ') ∈ ?oreachable n"
      by (rule oreachable_local) simp
    with ‹netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')›
      show "netgmap sr s' ∈ netmask (net_tree_ips n) ` ?oreachable n" by (rule image_eqI)
  qed

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)"
  by (auto simp add: initmissing_def default_def)

lemma netmask_initmissing_netgmap:
  "netmask (net_ips s) (initmissing (netgmap sr s)) = netgmap sr s"
  proof (intro prod_eqI ext)
    fix i
    show "fst (netmask (net_ips s) (initmissing (netgmap sr s))) i = fst (netgmap sr s) i"
      unfolding initmissing_def by (clarsimp split: option.split)
  qed (simp add: initmissing_def)

lemma snd_initmissing [simp]:
  "snd (initmissing x)= snd x"
  unfolding initmissing_def by simp

lemma initmnissing_snd_netgmap [simp]:
  assumes "initmissing (netgmap sr s) = (σ, ζ)"
    shows "snd (netgmap sr s) = ζ"
  using assms unfolding initmissing_def by simp


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)"
  using assms unfolding initmissing_def by (clarsimp split: option.split)

lemma not_in_net_ips_fst_init_missing [simp]:
  assumes "i ∉ net_ips s"
    shows "fst (initmissing (netgmap sr s)) i = someinit i"
  using assms unfolding initmissing_def by (clarsimp split: option.split)

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 -
    obtain σ ζ where "initmissing (netgmap sr s) = (σ, ζ)" by (metis surj_pair)
    with assms(1) have "(σ, ζ) ∈ ?oreachable n" by simp

    have "netgmap sr s = netmask (net_ips s) (σ, ζ)"
    proof (intro prod_eqI ext)
      fix i
      show "fst (netgmap sr s) i = fst (netmask (net_ips s) (σ, ζ)) i"
      proof (cases "i∈net_ips s")
        assume "i∈net_ips s"
        hence "fst (initmissing (netgmap sr s)) i = the (fst (netgmap sr s) i)"
          by (rule in_net_ips_fst_init_missing)
        moreover from ‹i∈net_ips s› have "Some (the (fst (netgmap sr s) i)) = fst (netgmap sr s) i"
          by (rule some_the_fst_netgmap)
        ultimately show ?thesis
          using ‹initmissing (netgmap sr s) = (σ, ζ)› by simp
      qed simp
    next
      from ‹initmissing (netgmap sr s) = (σ, ζ)›
        show "snd (netgmap sr s) = snd (netmask (net_ips s) (σ, ζ))"
          by simp
    qed
    with assms(2) have "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)" by simp
    moreover from ‹(σ, ζ) ∈ ?oreachable n›
      have "netmask (net_ips s) (σ, ζ) ∈ netmask (net_ips s) ‘ ?oreachable n"
        by (rule imageI)
    ultimately show ?thesis
      by (simp only: assms(2))
  qed

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")
  using assms(2) proof induction
    fix s
    assume "s ∈ init (closed (pnet np n))"
    hence "s ∈ init (pnet np n)" by simp

    from ‹wf_net_tree n› have "initmissing (netgmap sr s) ∈ init (opnet onp n)"
    proof (rule init_lifted [THEN subsetD], intro CollectI exI conjI allI)
      show "initmissing (netgmap sr s) = (fst (initmissing (netgmap sr s)), snd (netgmap sr s))"
        by (metis snd_initmissing surjective_pairing)
    next
      from ‹s ∈ init (pnet np n)› show "s ∈ init (pnet np n)" ..
    next
      fix i
      show "if i ∈ net_tree_ips n
             then (fst (initmissing (netgmap sr s))) i = the (fst (netgmap sr s) i)
             else (fst (initmissing (netgmap sr s))) i ∈ (fst ∘ sr) ‘ init (np i)"
```

```
    proof (cases "i ∈ net_tree_ips n", simp_all only: if_True if_False)
      assume "i ∈ net_tree_ips n"
      with ‹s ∈ init (pnet np n)› have "i ∈ net_ips s" ..
      thus "fst (initmissing (netgmap sr s)) i = the (fst (netgmap sr s) i)" by simp
    next
      assume "i ∉ net_tree_ips n"
      with ‹s ∈ init (pnet np n)› have "i ∉ net_ips s" ..
      hence "fst (initmissing (netgmap sr s)) i = someinit i" by simp
      moreover have "someinit i ∈ (fst ∘ sr) ` init (np i)"
      unfolding someinit_def proof (rule someI_ex)
        from init_notempty show "∃x. x ∈ (fst o sr) ` init (np i)" by auto
      qed
      ultimately show "fst (initmissing (netgmap sr s)) i ∈ (fst ∘ sr) ` init (np i)"
        by simp
    qed
  qed
  hence "initmissing (netgmap sr s) ∈ init (oclosed (opnet onp n))" by simp
  thus "initmissing (netgmap sr s) ∈ ?oreachable n" ..
next
  fix s a s'
  assume "s ∈ reachable (closed (pnet np n)) TT"
     and "(s, a, s') ∈ trans (closed (pnet np n))"
     and "initmissing (netgmap sr s) ∈ ?oreachable n"
  from this(1) have "s ∈ reachable (pnet np n) TT" ..
  hence "net_ips s = net_tree_ips n" by (rule pnet_net_ips_net_tree_ips)
  with ‹initmissing (netgmap sr s) ∈ ?oreachable n›
    have "netgmap sr s ∈ netmask (net_tree_ips n) ` ?oreachable n"
      by (rule initmissing_oreachable_netmask)

  obtain σ ζ where "(σ, ζ) = initmissing (netgmap sr s)" by (metis surj_pair)
  with ‹initmissing (netgmap sr s) ∈ ?oreachable n›
    have "(σ, ζ) ∈ ?oreachable n" by simp
  from ‹(σ, ζ) = initmissing (netgmap sr s)› and ‹net_ips s = net_tree_ips n› [symmetric]
    have "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
      by (clarsimp simp add: netmask_initmissing_netgmap)

  with ‹s ∈ reachable (closed (pnet np n)) TT›
    obtain σ' ζ' where "((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))"
                 and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
      using ‹wf_net_tree n› and ‹(s, a, s') ∈ trans (closed (pnet np n))›
      by (rule transfer_action)

  from ‹(σ, ζ) ∈ ?oreachable n› have "net_ips ζ = net_tree_ips n"
    by (rule opnet_net_ips_net_tree_ips [OF oclosed_oreachable_oreachable])
  with ‹((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))›
    have "∀j. j∉net_tree_ips n ⟶ σ' j = σ j"
      by (clarsimp elim!: ocomplete_no_change)
  have "initmissing (netgmap sr s') = (σ', ζ')"
  proof (intro prod_eqI ext)
    fix i
    from ‹netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')›
         ‹∀j. j∉net_tree_ips n ⟶ σ' j = σ j›
         ‹(σ, ζ) = initmissing (netgmap sr s)›
         ‹net_ips s = net_tree_ips n›
    show "fst (initmissing (netgmap sr s')) i = fst (σ', ζ') i"
      unfolding initmissing_def by simp
  next
    from ‹netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')›
      show "snd (initmissing (netgmap sr s')) = snd (σ', ζ')" by simp
  qed
  moreover from ‹(σ, ζ) ∈ ?oreachable n› ‹((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))›
    have "(σ', ζ') ∈ ?oreachable n"
      by (rule oreachable_local) (rule TrueI)
```

```
    ultimately show "initmissing (netgmap sr s') ∈ ?oreachable n"
      by simp
  qed

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)))"
  unfolding netglobal_def by simp

lemma netglobalE [elim]:
  assumes "netglobal P s"
      and "⋀σ. ⟦ P σ; fst (initmissing (netgmap sr s)) = σ ⟧ ⟹ Q σ"
    shows "netglobal Q s"
  using assms by simp

lemma netglobal_weakenE [elim]:
  assumes "p ⊫ netglobal P"
      and "⋀σ. P σ ⟹ Q σ"
    shows "p ⊫ netglobal Q"
  using assms(1) proof (rule invariant_weakenE)
    fix s
    assume "netglobal P s"
    thus "netglobal Q s"
      by (rule netglobalE) (erule assms(2))
  qed

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"
  unfolding invariant_def proof
    fix s
    assume "s ∈ reachable (closed (pnet np n)) TT"
    with assms(1)
      have "initmissing (netgmap sr s) ∈ oreachable (oclosed (opnet onp n)) (λ_ _ _. True) U"
        by (rule pnet_reachable_transfer)
    with assms(2) have "global P (initmissing (netgmap sr s))" ..
    thus "netglobal P s" by simp
  qed

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 unfold_locales
    fix i
    show "{ (σ, ζ) |σ ζ s. s ∈ init (np i ⟨⟨ qp)
                         ∧ (σ i, ζ) = ((λ(p, q). (fst (sr p), (snd (sr p), q))) s)
                         ∧ (∀j. j≠i ⟶ σ j ∈ (fst o (λ(p, q). (fst (sr p), (snd (sr p), q))))
                                                   ` init (np j ⟨⟨ qp)) } ⊆ init (onp i ⟨⟨_i qp)"
      (is "?S ⊆ _")
    proof
      fix s
      assume "s ∈ ?S"
```

```
    then obtain σ p lq
      where "s = (σ, (snd (sr p), lq))"
        and "lq ∈ init qp"
        and "p ∈ init (np i)"
        and "σ i = fst (sr p)"
        and "∀j. j ≠ i ⟶ σ j ∈ (fst ∘ (λ(p, q). (fst (sr p), snd (sr p), q)))
                                                                          ` (init (np j) × init qp)"
      by auto
    from this(5) have "∀j. j ≠ i ⟶ σ j ∈ (fst ∘ sr) ` init (np j)"
      by auto
    with ‹p ∈ init (np i)› and ‹σ i = fst (sr p)› have "(σ, snd (sr p)) ∈ init (onp i)"
      by - (rule init [THEN subsetD], auto)
    with ‹lq∈ init qp› have "((σ, snd (sr p)), lq) ∈ init (onp i) × init qp"
      by simp
    hence "(σ, (snd (sr p), lq)) ∈ extg ` (init (onp i) × init qp)"
      by (rule rev_image_eqI) simp
    with ‹s = (σ, (snd (sr p), lq))› show "s ∈ init (onp i ⟨⟨i qp)"
      by simp
  qed
next
  fix i s a s' σ σ'
  assume "σ i = fst ((λ(p, q). (fst (sr p), (snd (sr p), q))) s)"
     and "σ' i = fst ((λ(p, q). (fst (sr p), (snd (sr p), q))) s')"
     and "(s, a, s') ∈ trans (np i ⟨⟨ qp)"
  then obtain p q p' q' where "s  = (p, q)"
                          and "s' = (p', q')"
                          and "σ i  = fst (sr p)"
                          and "σ' i = fst (sr p')"
    by (clarsimp split: prod.split_asm)
  from this(1-2) and ‹(s, a, s') ∈ trans (np i ⟨⟨ qp)›
    have "((p, q), a, (p', q')) ∈ parp_sos (trans (np i)) (trans qp)" by simp
  hence "((σ, (snd (sr p), q)), a, (σ', (snd (sr p'), q'))) ∈ trans (onp i ⟨⟨i qp)"
  proof cases
    assume "q' = q"
       and "(p, a, p') ∈ trans (np i)"
       and "⋀m. a ≠ receive m"
    from ‹σ i = fst (sr p)› and ‹σ' i = fst (sr p')› this(2)
      have "((σ, snd (sr p)), a, (σ', snd (sr p'))) ∈ trans (onp i)" by (rule trans)
    with ‹q' = q› and ‹⋀m. a ≠ receive m›
      show "((σ, snd (sr p), q), a, (σ', (snd (sr p'), q'))) ∈ trans (onp i ⟨⟨i qp)"
        by (auto elim!: oparleft)
  next
    assume "p' = p"
       and "(q, a, q') ∈ trans qp"
       and "⋀m. a ≠ send m"
    with ‹σ i = fst (sr p)› and ‹σ' i = fst (sr p')›
      show "((σ, snd (sr p), q), a, (σ', (snd (sr p'), q'))) ∈ trans (onp i ⟨⟨i qp)"
        by (auto elim!: oparright)
  next
    fix m
    assume "a = τ"
       and "(p, receive m, p') ∈ trans (np i)"
       and "(q, send m, q') ∈ trans qp"
    from ‹σ i = fst (sr p)› and ‹σ' i = fst (sr p')› this(2)
      have "((σ, snd (sr p)), receive m, (σ', snd (sr p'))) ∈ trans (onp i)"
        by (rule trans)
    with ‹(q, send m, q') ∈ trans qp› and ‹a = τ›
      show "((σ, snd (sr p), q), a, (σ', (snd (sr p'), q'))) ∈ trans (onp i ⟨⟨i qp)"
        by (simp del: step_seq_tau) (rule oparboth)
  qed
  with ‹s = (p, q)› ‹s' = (p', q')›
    show "((σ, snd ((λ(p, q). (fst (sr p), (snd (sr p), q))) s)), a,
               (σ', snd ((λ(p, q). (fst (sr p), (snd (sr p), q))) s'))) ∈ trans (onp i ⟨⟨i qp)"
      by simp
```

```
  next
    show "∀j. init (np j ⟨⟨ qp) ≠ {}"
      by (clarsimp simp add: init_notempty init_qp_notempty)
  qed

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 ≡ λ(d,did). Newpkt d did"
definition eq_newpkt_def: "eq_newpkt m ≡ case m of Newpkt d did  ⇒ True | _ ⇒ False"

instance by intro_classes (simp add: eq_newpkt_def)
end

definition pkt :: "nat × nat ⇒ msg"
where "pkt ≡ λ(no, sid). Pkt no sid"

lemma pkt_simp [simp]:
  "pkt(no, sid) = Pkt no sid"
  unfolding pkt_def by simp

lemma not_eq_newpkt_pkt [simp]: "¬eq_newpkt (Pkt no sid)"
  unfolding eq_newpkt_def by simp
```

26.2 Protocol model

```
record state =
  id    :: "nat"
  no    :: "nat"
  nhid  :: "nat"

  msg    :: "msg"
  num    :: "nat"
  sid    :: "nat"

abbreviation toy_init :: "ip ⇒ state"
where "toy_init i ≡ (|
         id = i,
         no = 0,
         nhid = i,
```

```
        msg    = (SOME x. True),
        num    = (SOME x. True),
        sid    = (SOME x. True)
      ⦈"

lemma some_neq_not_eq [simp]: "¬((SOME x :: nat. x ≠ i) = i)"
  by (subst some_eq_ex) (metis zero_neq_numeral)

definition clear_locals :: "state ⇒ state"
where "clear_locals ξ = ξ ⦇
    msg    := (SOME x. True),
    num    := (SOME x. True),
    sid    := (SOME x. True)
  ⦈"

lemma clear_locals_but_not_globals [simp]:
  "id (clear_locals ξ) = id ξ"
  "no (clear_locals ξ) = no ξ"
  "nhid (clear_locals ξ) = nhid ξ"
  unfolding clear_locals_def by auto

definition is_newpkt
where "is_newpkt ξ ≡ case msg ξ of
                      Newpkt data did ⇒ { ξ⦇num := data⦈ }
                    | _ ⇒ {}"

definition is_pkt
where "is_pkt ξ ≡ case msg ξ of
                   Pkt num' sid' ⇒ { ξ⦇ num := num', sid := sid' ⦈ }
                 | _ ⇒ {}"

lemmas is_msg_defs =
  is_pkt_def is_newpkt_def

lemma is_msg_inv_id [simp]:
  "ξ' ∈ is_pkt ξ    ⟹ id ξ' = id ξ"
  "ξ' ∈ is_newpkt ξ ⟹ id ξ' = id ξ"
  unfolding is_msg_defs
  by (cases "msg ξ", clarsimp+)+

lemma is_msg_inv_sid [simp]:
  "ξ' ∈ is_newpkt ξ ⟹ sid ξ' = sid ξ"
  unfolding is_msg_defs
  by (cases "msg ξ", clarsimp+)+

lemma is_msg_inv_no [simp]:
  "ξ' ∈ is_pkt ξ    ⟹ no ξ' = no ξ"
  "ξ' ∈ is_newpkt ξ ⟹ no ξ' = no ξ"
  unfolding is_msg_defs
  by (cases "msg ξ", clarsimp+)+

lemma is_msg_inv_nhid [simp]:
  "ξ' ∈ is_pkt ξ    ⟹ nhid ξ' = nhid ξ"
  "ξ' ∈ is_newpkt ξ ⟹ nhid ξ' = nhid ξ"
  unfolding is_msg_defs
  by (cases "msg ξ", clarsimp+)+

lemma is_msg_inv_msg [simp]:
  "ξ' ∈ is_pkt ξ    ⟹ msg ξ' = msg ξ"
  "ξ' ∈ is_newpkt ξ ⟹ msg ξ' = msg ξ"
  unfolding is_msg_defs
  by (cases "msg ξ", clarsimp+)+

datatype pseqp =
```

```
    PToy

fun nat_of_seqp :: "pseqp ⇒ nat"
where
  "nat_of_seqp PToy = 1"

instantiation "pseqp" :: ord
begin
definition less_eq_seqp [iff]: "l1 ≤ l2 = (nat_of_seqp l1 ≤ nat_of_seqp l2)"
definition less_seqp [iff]:    "l1 < l2 = (nat_of_seqp l1 < nat_of_seqp l2)"
instance ..
end

abbreviation Toy
where
  "Toy ≡ λ_. ⟦clear_locals⟧ call(PToy)"

fun Γ_TOY :: "(state, msg, pseqp, pseqp label) seqp_env"
where
  "Γ_TOY PToy = labelled PToy (
     receive(λmsg' ξ. ξ ⦇ msg := msg' ⦈).
     ⟦ξ. ξ ⦇nhid := id ξ⦈⟧
     (   ⟨is_newpkt⟩
        (
            ⟦ξ. ξ ⦇no := max (no ξ) (num ξ)⦈⟧
            broadcast(λξ. pkt(no ξ, id ξ)). Toy()
        )
      ⊕ ⟨is_pkt⟩
      (
           ⟨ξ. num ξ > no ξ⟩
              ⟦ξ. ξ ⦇no := num ξ⦈⟧
              ⟦ξ. ξ ⦇nhid := sid ξ⦈⟧
              broadcast(λξ. pkt(no ξ, id ξ)). Toy()
        ⊕ ⟨ξ. num ξ ≤ no ξ⟩
              Toy()
      )
     ))"

declare Γ_TOY.simps [simp del, code del]
lemmas Γ_TOY_simps [simp, code] = Γ_TOY.simps [simplified]

fun Γ_TOY_skeleton
where "Γ_TOY_skeleton PToy = seqp_skeleton (Γ_TOY PToy)"

lemma Γ_TOY_skeleton_wf [simp]:
  "wellformed Γ_TOY_skeleton"
  proof (rule, intro allI)
    fix pn pn'
    show "call(pn') ∉ stermsl (Γ_TOY_skeleton pn)"
      by (cases pn) simp_all
  qed

declare Γ_TOY_skeleton.simps [simp del, code del]
lemmas Γ_TOY_skeleton_simps [simp, code] = Γ_TOY_skeleton.simps [simplified Γ_TOY_simps seqp_skeleton.simps

lemma toy_proc_cases [dest]:
  fixes p pn
  assumes "p ∈ ctermsl (Γ_TOY pn)"
    shows "p ∈ ctermsl (Γ_TOY PToy)"
  using assms
  by (cases pn) simp_all

definition σ_TOY :: "ip ⇒ (state × (state, msg, pseqp, pseqp label) seqp) set"
where "σ_TOY i ≡ {(toy_init i, Γ_TOY PToy)}"
```

```
abbreviation ptoy
  :: "ip ⇒ (state × (state, msg, pseqp, pseqp label) seqp, msg seq_action) automaton"
where
  "ptoy i ≡ (| init = σ_TOY i, trans = seqp_sos Γ_TOY |)"

lemma toy_trans: "trans (ptoy i) = seqp_sos Γ_TOY"
  by simp

lemma toy_control_within [simp]: "control_within Γ_TOY (init (ptoy i))"
  unfolding σ_TOY_def by (rule control_withinI) (auto simp del: Γ_TOY_simps)

lemma toy_wf [simp]:
  "wellformed Γ_TOY"
  proof (rule, intro allI)
    fix pn pn'
    show "call(pn') ∉ stermsl (Γ_TOY pn)"
      by (cases pn) simp_all
  qed

lemmas toy_labels_not_empty [simp] = labels_not_empty [OF toy_wf]

lemma toy_ex_label [intro]: "∃l. l∈labels Γ_TOY p"
  by (metis toy_labels_not_empty all_not_in_conv)

lemma toy_ex_labelE [elim]:
  assumes "∀l∈labels Γ_TOY p. P l p"
      and "∃p l. P l p ⟹ Q"
    shows "Q"
 using assms by (metis toy_ex_label)

lemma toy_simple_labels [simp]: "simple_labels Γ_TOY"
  proof
    fix pn p
    assume "p∈subterms(Γ_TOY pn)"
    thus "∃!l. labels Γ_TOY p = {l}"
    by (cases pn) (simp_all cong: seqp_congs | elim disjE)+
  qed

lemma σ_TOY_labels [simp]: "(ξ, p) ∈ σ_TOY i ⟹  labels Γ_TOY p = {PToy-:0}"
  unfolding σ_TOY_def by simp
```

By default, we no longer let the simplifier descend into process terms.

```
declare seqp_congs [cong]


declare
  Γ_TOY_simps [cterms_env]
  toy_proc_cases [ctermsl_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 σ_OTOY :: "((ip ⇒ state) × ((state, msg, pseqp, pseqp label) seqp)) set"
where "σ_OTOY ≡ {(toy_init, Γ_TOY PToy)}"

abbreviation optoy
  :: "ip ⇒ ((ip ⇒ state) × (state, msg, pseqp, pseqp label) seqp, msg seq_action) automaton"
where
  "optoy i ≡ (| init = σ_OTOY, trans = oseqp_sos Γ_TOY i |)"

lemma initiali_toy [intro!, simp]: "initiali i (init (optoy i)) (init (ptoy i))"
  unfolding σ_TOY_def σ_OTOY_def by rule simp_all
```

lemma *oaodv_control_within [simp]: "control_within* Γ_{TOY} *(init (optoy i))"*
  unfolding σ_{OTOY}*_def* **by** *(rule control_withinI) (auto simp del:* Γ_{TOY}*_simps)*

lemma σ_{OTOY}*_labels [simp]: "(σ, p) ∈* σ_{OTOY} *⟹ labels* Γ_{TOY} *p = {PToy-:0}"*
  unfolding σ_{OTOY}*_def* **by** *simp*

lemma *otoy_trans: "trans (optoy i) = oseqp_sos* Γ_{TOY} *i"*
  by *simp*

declare
  oseq_invariant_ctermsI [OF toy_wf oaodv_control_within toy_simple_labels otoy_trans, cterms_intros]
  oseq_step_invariant_ctermsI [OF toy_wf oaodv_control_within toy_simple_labels otoy_trans, cterms_intros]

26.4 Predicates

definition *msg_sender :: "msg ⇒ ip"*
where *"msg_sender m ≡ case m of Pkt _ ipc ⇒ ipc"*

lemma *msg_sender_simps [simp]:*
  "⋀d did sid. msg_sender (Pkt d sid) = sid"
  unfolding *msg_sender_def* **by** *simp_all*

abbreviation *not_Pkt :: "msg ⇒ bool"*
where *"not_Pkt m ≡ case m of Pkt _ _ ⇒ False | _ ⇒ True"*

definition *nos_inc :: "state ⇒ state ⇒ bool"*
where *"nos_inc ξ ξ' ≡ (no ξ ≤ no ξ')"*

definition *msg_ok :: "(ip ⇒ state) ⇒ msg ⇒ bool"*
where *"msg_ok σ m ≡ case m of Pkt num' sid' ⇒ num' ≤ no (σ sid') | _ ⇒ True"*

lemma *msg_okI [intro]:*
  assumes *"⋀num' sid'. m = Pkt num' sid' ⟹ num' ≤ no (σ sid')"*
    shows *"msg_ok σ m"*
  using *assms* **unfolding** *msg_ok_def*
  by *(auto split: msg.split)*

lemma *msg_ok_Pkt [simp]:*
  "msg_ok σ (Pkt data src) = (data ≤ no (σ src))"
  unfolding *msg_ok_def* **by** *simp*

lemma *msg_ok_pkt [simp]:*
  "msg_ok σ (pkt(data, src)) = (data ≤ no (σ src))"
  unfolding *msg_ok_def* **by** *simp*

lemma *msg_ok_Newpkt [simp]:*
  "msg_ok σ (Newpkt data dst)"
  unfolding *msg_ok_def* **by** *simp*

lemma *msg_ok_newpkt [simp]:*
  "msg_ok σ (newpkt(data, dst))"
  unfolding *msg_ok_def* **by** *simp*

26.5 Sequential Invariants

lemma *seq_no_leq_num:*
  "ptoy i ⊫ onl Γ_{TOY} *(λ(ξ, l). l∈{PToy-:7..PToy-:8} ⟶ no ξ ≤ num ξ)"*
  by *inv_cterms*

lemma *seq_nos_incs:*
  "ptoy i $\models_A$ *onll* Γ_{TOY} *(λ((ξ, _), _, (ξ', _)). nos_inc ξ ξ')"*
  unfolding *nos_inc_def*

```
  by (inv_cterms inv add: onl_invariant_sterms [OF toy_wf seq_no_leq_num])

lemma seq_nos_incs':
  "ptoy i ⊫A (λ((ξ, _), _, (ξ', _)). nos_inc ξ ξ')"
  by (rule step_invariant_weakenE [OF seq_nos_incs]) (auto dest!: onllD)

lemma sender_ip_valid:
  "ptoy i ⊫A onll ΓTOY (λ((ξ, _), a, _). anycast (λm. msg_sender m = id ξ) a)"
  by inv_cterms

lemma id_constant:
  "ptoy i ⊫ onl ΓTOY (λ(ξ, _). id ξ = i)"
  by inv_cterms (simp add: σTOY_def)

lemma nhid_eq_id:
  "ptoy i ⊫ onl ΓTOY (λ(ξ, l). l∈{PToy-:2..PToy-:8} ⟶ nhid ξ = id ξ)"
  by inv_cterms

lemma seq_msg_ok:
  "ptoy i ⊫A onll ΓTOY (λ((ξ, _), a, _).
              anycast (λm. case m of Pkt num' sid' ⇒ num' = no ξ ∧ sid' = i | _ ⇒ True) a)"
  by (inv_cterms inv add: onl_invariant_sterms [OF toy_wf id_constant])

lemma nhid_eq_i:
  "ptoy i ⊫ onl ΓTOY (λ(ξ, l). l∈{PToy-:2..PToy-:8} ⟶ nhid ξ = i)"
  proof (rule invariant_arbitraryI, clarify intro!: onlI impI)
    fix ξ p l n
    assume "(ξ, p) ∈ reachable (ptoy i) TT"
       and "l ∈ labels ΓTOY p"
       and "l ∈ {PToy-:2..PToy-:8}"
    from this(1-3) have "nhid ξ = id ξ"
      by - (drule invariantD [OF nhid_eq_id], auto)
    moreover with ‹(ξ, p) ∈ reachable (ptoy i) TT› and ‹l ∈ labels ΓTOY p› have "id ξ = i"
      by (auto dest: invariantD [OF id_constant])
    ultimately show "nhid ξ = i"
      by simp
  qed
```

26.6 Global Invariants

```
lemma nos_incD [dest]:
  assumes "nos_inc ξ ξ'"
    shows "no ξ ≤ no ξ'"
  using assms unfolding nos_inc_def .

lemma nos_inc_simp [simp]:
  "nos_inc ξ ξ' = (no ξ ≤ no ξ')"
  unfolding nos_inc_def ..

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 ⊫A (otherwith nos_inc {i} S,
                      other nos_inc {i} →)
                       onll ΓTOY (λ((σ, _), a, (σ', _)). (∀j. nos_inc (σ j) (σ' j)))"
  proof -
    have *: "⋀σ σ' a. ⟦ otherwith nos_inc {i} S σ σ' a; no (σ i) ≤ no (σ' i) ⟧
                       ⟹ ∀j. no (σ j) ≤ no (σ' j)"
```

```
      by (auto dest!: otherwith_syncD)
    show ?thesis
      by (inv_cterms
            inv add: oseq_step_invariant_sterms [OF oseq_nos_incs [THEN oinvariant_step_anyact]
                                                                    toy_wf otoy_trans]
            simp add: seqllsimp) (auto elim!: *)
  qed

lemma oreceived_msg_inv:
  assumes other: "⋀σ σ' m. ⟦ P σ m; other Q {i} σ σ' ⟧ ⟹ P σ' m"
      and local: "⋀σ m. P σ m ⟹ P (σ(i := σ i⦇msg := m⦈)) m"
    shows "optoy i ⊨ (otherwith Q {i} (orecvmsg P), other Q {i} →)
                      onl Γ_TOY (λ(σ, l). l ∈ {PToy-:1} ⟶ P σ (msg (σ i)))"
  proof (inv_cterms, intro impI)
    fix σ σ' l
    assume "l = PToy-:1 ⟶ P σ (msg (σ i))"
       and "l = PToy-:1"
       and "other Q {i} σ σ'"
    from this(1-2) have "P σ (msg (σ i))" ..
    hence "P σ' (msg (σ i))" using ‹other Q {i} σ σ'›
      by (rule other)
    moreover from ‹other Q {i} σ σ'› have "σ' i = σ i" ..
    ultimately show "P σ' (msg (σ' i))" by simp
  next
    fix σ σ' msg
    assume "otherwith Q {i} (orecvmsg P) σ σ' (receive msg)"
       and "σ' i = σ i⦇msg := msg⦈"
    from this(1) have "P σ msg"
                  and "∀j. j≠i ⟶ Q (σ j) (σ' j)" by auto
    from this(1) have "P (σ(i := σ i⦇msg := msg⦈)) msg" by (rule local)
    thus "P σ' msg"
    proof (rule other)
      from ‹σ' i = σ i⦇msg := msg⦈› and ‹∀j. j≠i ⟶ Q (σ j) (σ' j)›
        show "other Q {i} (σ(i := σ i⦇msg := msg⦈)) σ'"
          by - (rule otherI, auto)
    qed
  qed

lemma msg_ok_other_nos_inc [elim]:
  assumes "msg_ok σ m"
      and "other nos_inc {i} σ σ'"
    shows "msg_ok σ' m"
  proof (cases m)
    fix num sid
    assume "m = Pkt num sid"
    with ‹msg_ok σ m› have "num ≤ no (σ sid)" by simp
    also from ‹other nos_inc {i} σ σ'› have "no (σ sid) ≤ no (σ' sid)"
      by (rule otherE) (metis eq_iff nos_incD)
    finally have "num ≤ no (σ' sid)" .
    with ‹m = Pkt num sid› show ?thesis
      by simp
  qed simp

lemma msg_ok_no_leq_no [simp, elim]:
  assumes "msg_ok σ m"
      and "∀j. no (σ j) ≤ no (σ' j)"
    shows "msg_ok σ' m"
  using assms(1) proof (cases m)
    fix num sid
    assume "m = Pkt num sid"
    with ‹msg_ok σ m› have "num ≤ no (σ sid)" by simp
    also from ‹∀j. no (σ j) ≤ no (σ' j)› have "no (σ sid) ≤ no (σ' sid)"
      by simp
    finally have "num ≤ no (σ' sid)" .
```

```
    with ‹m = Pkt num sid› show ?thesis
      by simp
  qed (simp add: assms(1))

lemma oreceived_msg_ok:
  "optoy i ⊨ (otherwith nos_inc {i} (orecvmsg msg_ok),
             other nos_inc {i} →)
            onl Γ_TOY (λ(σ, l). l∈{PToy-:1..} ⟶ msg_ok σ (msg (σ i)))"
  (is "_ ⊨ (?S, ?U →) _")
  proof (inv_cterms inv add: oseq_step_invariant_sterms [OF all_nos_inc toy_wf otoy_trans],
         intro impI, elim impE)
    fix σ σ'
    assume "msg_ok σ (msg (σ i))"
       and other: "other nos_inc {i} σ σ'"
    moreover from other have "msg (σ' i) = msg (σ i)"
      by (clarsimp elim!: otherE)
    ultimately show "msg_ok σ' (msg (σ' i))"
      by (auto)
  next
    fix p l σ a q l' σ' pp p' m
    assume a1: "(σ', p') ∈ oreachable (optoy i) ?S ?U"
       and a2: "PToy-:1 ∈ labels Γ_TOY p'"
       and a3: "σ' i = σ i⦇msg := m⦈"
    have inv: "optoy i ⊨ (?S, ?U →) onl Γ_TOY (λ(σ, l). l ∈ {PToy-:1} ⟶ msg_ok σ (msg (σ i)))"
    proof (rule oreceived_msg_inv)
      fix σ σ' m
      assume "msg_ok σ m"
         and "other nos_inc {i} σ σ'"
      thus "msg_ok σ' m" ..
    next
      fix σ m
      assume "msg_ok σ m"
      thus "msg_ok (σ(i := σ i⦇msg := m⦈)) m"
        by (cases m) auto
    qed
    from a1 a2 a3 show "msg_ok σ' m"
      by (clarsimp dest!: oinvariantD [OF inv] onlD)
  qed simp

lemma is_pkt_handler_num_leq_no:
  shows "optoy i ⊨ (otherwith nos_inc {i} (orecvmsg msg_ok),
                   other nos_inc {i} →)
                  onl Γ_TOY (λ(σ, l). l∈{PToy-:6..PToy-:10} ⟶ num (σ i) ≤ no (σ (sid (σ i))))"
  proof -
    { fix σ σ'
      assume "∀j. no (σ j) ≤ no (σ' j)"
         and "num (σ i) ≤ no (σ (sid (σ i)))"
      have "num (σ i) ≤ no (σ' (sid (σ i)))"
      proof -
        note ‹num (σ i) ≤ no (σ (sid (σ i)))›
        also from ‹∀j. no (σ j) ≤ no (σ' j)› have "no (σ (sid (σ i))) ≤ no (σ' (sid (σ i)))"
          by auto
        finally show ?thesis .
      qed
    } note solve_step = this
    show ?thesis
    proof (inv_cterms inv add: oseq_step_invariant_sterms [OF all_nos_inc toy_wf otoy_trans]
                              onl_oinvariant_sterms [OF toy_wf oreceived_msg_ok]
                        solve: solve_step, intro impI, elim impE)
      fix σ σ'
      assume *: "num (σ i) ≤ no (σ (sid (σ i)))"
         and "other nos_inc {i} σ σ'"
      from this(2) obtain "∀i∈{i}. σ' i = σ i"
                      and "∀j. j ∉ {i} ⟶ nos_inc (σ j) (σ' j)" ..
```

```
      show "num (σ' i) ≤ no (σ' (sid (σ' i)))"
      proof (cases "sid (σ i) = i")
        assume "sid (σ i) = i"
        with * ‹∀i∈{i}. σ' i = σ i›
        show ?thesis by simp
      next
        assume "sid (σ i) ≠ i"
        with ‹∀j. j ∉ {i} ⟶ nos_inc (σ j) (σ' j)›
          have "no (σ (sid (σ i))) ≤ no (σ' (sid (σ i)))" by simp
        with * ‹∀i∈{i}. σ' i = σ i›
        show ?thesis by simp
      qed
    next
      fix p l σ a q l' σ' pp p'
      assume "msg_ok σ (msg (σ i))"
         and "∀j. no (σ j) ≤ no (σ' j)"
         and "σ' i ∈ is_pkt (σ i)"
      show "num (σ' i) ≤ no (σ' (sid (σ' i)))"
      proof (cases "msg (σ i)")
        fix num' sid'
        assume "msg (σ i) = Pkt num' sid'"
        with ‹σ' i ∈ is_pkt (σ i)› obtain "num (σ' i) = num'"
                                      and "sid (σ' i) = sid'"
          unfolding is_pkt_def by auto
        with ‹msg (σ i) = Pkt num' sid'› and ‹msg_ok σ (msg (σ i))›
          have "num (σ' i) ≤ no (σ (sid (σ' i)))"
            by simp
        also from ‹∀j. no (σ j) ≤ no (σ' j)› have "no (σ (sid (σ' i))) ≤ no (σ' (sid (σ' i)))" ..
        finally show ?thesis .
      next
        fix num' sid'
        assume "msg (σ i) = Newpkt num' sid'"
        with ‹σ' i ∈ is_pkt (σ i)› have False
          unfolding is_pkt_def by simp
        thus ?thesis ..
      qed
    qed
  qed

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 ⊨ (otherwith nos_inc {i} (orecvmsg msg_ok),
                    other nos_inc {i} →) global (λσ. no (σ i) ≤ no (σ (nhid (σ i))))"
    (is "_ ⊨ (?S, ?U →) ?P")
  proof -
    have nhidinv: "optoy i ⊨ (?S, ?U →)
                            onl Γ_TOY (λ(σ, l). l∈{PToy-:2..PToy-:8}
                                                ⟶ nhid (σ i) = id (σ i))"
      by (rule oinvariant_weakenE [OF oseq_nhid_eq_id]) (auto simp: seqlsimp)
    have idinv: "optoy i ⊨ (?S, ?U →) onl Γ_TOY (λ(σ, l). id (σ i) = i)"
      by (rule oinvariant_weakenE [OF oseq_id_constant]) (auto simp: seqlsimp)
    { fix σ σ' a
      assume "no (σ i) ≤ no (σ (nhid (σ i)))"
```

```
      and "∀j. nos_inc (σ j) (σ' j)"
    note this(1)
    also from ‹∀j. nos_inc (σ j) (σ' j)› have "no (σ (nhid (σ i))) ≤ no (σ' (nhid (σ i)))"
      by auto
    finally have "no (σ i) ≤ no (σ' (nhid (σ i)))" ..
  } note * = this
  have "optoy i ⊨ (otherwith nos_inc {i} (orecvmsg msg_ok),
                   other nos_inc {i} →)
                  onl Γ_TOY (λ(σ, l). no (σ i) ≤ no (σ (nhid (σ i))))"
  proof (inv_cterms
         inv add: onl_oinvariant_sterms [OF toy_wf oseq_no_leq_num [THEN oinvariant_anyact]]
                  oseq_step_invariant_sterms [OF all_nos_inc toy_wf otoy_trans]
                  onl_oinvariant_sterms [OF toy_wf is_pkt_handler_num_leq_no]
                  onl_oinvariant_sterms [OF toy_wf nhidinv]
                  onl_oinvariant_sterms [OF toy_wf idinv]
         simp add: seqlsimp seqllsimp
         simp del: nos_inc_simp
            solve: *)
    fix σ p l
    assume "(σ, p) ∈ σ_OTOY"
    thus "no (σ i) ≤ no (σ (nhid (σ i)))"
      by (simp add: σ_OTOY_def)
  next
    fix σ σ' p l
    assume or: "(σ, p) ∈ oreachable (optoy i) ?S ?U"
       and "l ∈ labels Γ_TOY p"
       and "no (σ i) ≤ no (σ (nhid (σ i)))"
       and "other nos_inc {i} σ σ'"
    show "no (σ' i) ≤ no (σ' (nhid (σ' i)))"
    proof (cases "nhid (σ' i) = i")
      assume "nhid (σ' i) = i"
      with ‹no (σ i) ≤ no (σ (nhid (σ i)))› show ?thesis
        by simp
    next
      assume "nhid (σ' i) ≠ i"
      moreover from ‹other nos_inc {i} σ σ'› [THEN other_localD] have "σ' i = σ i"
        by simp
      ultimately have "no (σ (nhid (σ i))) ≤ no (σ' (nhid (σ' i)))"
        using ‹other nos_inc {i} σ σ'› and ‹σ' i = σ i› by (auto)
      with ‹no (σ i) ≤ no (σ (nhid (σ i)))› and ‹σ' i = σ i› show ?thesis
        by simp
    qed
  next
    fix p l σ a q l' σ' pp p'
    assume "no (σ i) ≤ num (σ i)"
       and "num (σ i) ≤ no (σ (sid (σ i)))"
       and "∀j. nos_inc (σ j) (σ' j)"
    from this(1-2) have "no (σ i) ≤ no (σ (sid (σ i)))"
      by (rule le_trans)
    also from ‹∀j. nos_inc (σ j) (σ' j)›
      have "no (σ (sid (σ i))) ≤ no (σ' (sid (σ i)))"
        by auto
    finally show "no (σ i) ≤ no (σ' (sid (σ i)))" ..
  qed
  thus ?thesis
    by (rule oinvariant_weakenE)
       (auto simp: onl_def)
qed

lemma anycast_weakenE [elim]:
  assumes "anycast P a"
      and "⋀m. P m ⟹ Q m"
  shows "anycast Q a"
  using assms unfolding anycast_def
```

```
  by (auto split: seq_action.split)

lemma oseq_msg_ok:
  "optoy i ⊨A (act TT, other U {i} →) globala (λ(σ, a, _). anycast (msg_ok σ) a)"
  by (rule ostep_invariant_weakenE [OF open_seq_step_invariant
          [OF seq_msg_ok initiali_toy otoy_trans toy_trans, simplified seql_onl_swap]])
     (auto simp: seqllsimp dest!: onllD elim!: anycast_weakenE intro!: msg_okI)
```

26.7 Lifting

```
lemma opar_bigger_than_next:
  shows "optoy i ⟨⟨i qmsg ⊨ (otherwith nos_inc {i} (orecvmsg msg_ok),
                    other nos_inc {i} →) global (λσ. no (σ i) ≤ no (σ (nhid (σ i))))"
  proof (rule lift_into_qmsg [OF oseq_bigger_than_next])
    fix σ σ' m
    assume "∀j. nos_inc (σ j) (σ' j)"
       and "msg_ok σ m"
    from this(2) show "msg_ok σ' m"
    proof (cases m, simp only: msg_ok_Pkt)
      fix num' sid'
      assume "num' ≤ no (σ sid')"
      also from ‹∀j. nos_inc (σ j) (σ' j)› have "no (σ sid') ≤ no (σ' sid')"
        by simp
      finally show "num' ≤ no (σ' sid')" .
    qed simp
  next
    show "optoy i ⊨A (otherwith nos_inc {i} (orecvmsg msg_ok), other nos_inc {i} →)
                    globala (λ(σ, _, σ'). nos_inc (σ i) (σ' i))"
      by (rule ostep_invariant_weakenE [OF open_seq_step_invariant
                                    [OF seq_nos_incs initiali_toy otoy_trans toy_trans]])
         (auto simp: seqllsimp dest!: onllD)
  qed simp

lemma onode_bigger_than_next:
  "⟨i : optoy i ⟨⟨i qmsg : R_i⟩o
    ⊨ (otherwith nos_inc {i} (oarrivemsg msg_ok), other nos_inc {i} →)
      global (λσ. no (σ i) ≤ no (σ (nhid (σ i))))"
  by (rule node_lift [OF opar_bigger_than_next])

lemma node_local_nos_inc:
  "⟨i : optoy i ⟨⟨i qmsg : R_i⟩o ⊨A (λσ _. oarrivemsg (λ_ _. True) σ, other (λ_ _. True) {i} →)
                              globala (λ(σ, _, σ'). nos_inc (σ i) (σ' i))"
  proof (rule node_lift_step_statelessassm)
    have "optoy i ⊨A (λσ _. orecvmsg (λ_ _. True) σ, other (λ_ _. True) {i} →)
                    globala (λ(σ, _, σ'). nos_inc (σ i) (σ' i))"
      by (rule ostep_invariant_weakenE [OF oseq_nos_incs])
         (auto simp: seqllsimp dest!: onllD)
    thus "optoy i ⟨⟨i qmsg ⊨A (λσ _. orecvmsg (λ_ _. True) σ, other (λ_ _. True) {i} →)
                          globala (λ(σ, _, σ'). nos_inc (σ i) (σ' i))"
      by (rule lift_step_into_qmsg_statelessassm) auto
  qed simp

lemma opnet_bigger_than_next:
  "opnet (λi. optoy i ⟨⟨i qmsg) n
    ⊨ (otherwith nos_inc (net_tree_ips n) (oarrivemsg msg_ok),
       other nos_inc (net_tree_ips n) →)
      global (λσ. ∀i∈net_tree_ips n. no (σ i) ≤ no (σ (nhid (σ i))))"
  proof (rule pnet_lift [OF onode_bigger_than_next])
    fix i R_i
    have "⟨i : optoy i ⟨⟨i qmsg : R_i⟩o ⊨A (λσ _. oarrivemsg msg_ok σ, other (λ_ _. True) {i} →)
                                    globala (λ(σ, a, _). castmsg (msg_ok σ) a)"
    proof (rule node_lift_anycast_statelessassm)
      have "optoy i ⊨A (λσ _. orecvmsg (λ_ _. True) σ, other (λ_ _. True) {i} →)
                      globala (λ(σ, a, _). anycast (msg_ok σ) a)"
```

```
        by (rule ostep_invariant_weakenE [OF oseq_msg_ok]) auto
      hence "optoy i ⟨⟨i qmsg ⊨A (λσ _. orecvmsg (λ_ _. True) σ, other (λ_ _. True) {i} →)
                                   globala (λ(σ, a, _). anycast (msg_ok σ) a)"
        by (rule lift_step_into_qmsg_statelessassm) auto
      thus "optoy i ⟨⟨i qmsg ⊨A (λσ _. orecvmsg msg_ok σ, other (λ_ _. True) {i} →)
                                  globala (λ(σ, a, _). anycast (msg_ok σ) a)"
        by (rule ostep_invariant_weakenE) auto
    qed
    thus "⟨i : optoy i ⟨⟨i qmsg : Ri⟩o ⊨A (λσ _. oarrivemsg msg_ok σ, other nos_inc {i} →)
                                           globala (λ(σ, a, _). castmsg (msg_ok σ) a)"
      by (rule ostep_invariant_weakenE) auto
  next
    fix i Ri
    show "⟨i : optoy i ⟨⟨i qmsg : Ri⟩o ⊨A (λσ _. oarrivemsg msg_ok σ,
                                           other nos_inc {i} →)
           globala (λ(σ, a, σ'). a ≠ τ ∧ (∀d. a ≠ i:deliver(d)) ⟶ nos_inc (σ i) (σ' i))"
      by (rule ostep_invariant_weakenE [OF node_local_nos_inc]) auto
  next
    fix i R
    show "⟨i : optoy i ⟨⟨i qmsg : R⟩o ⊨A (λσ _. oarrivemsg msg_ok σ,
                                          other nos_inc {i} →)
           globala (λ(σ, a, σ'). a = τ ∨ (∃d. a = i:deliver(d)) ⟶ nos_inc (σ i) (σ' i))"
      by (rule ostep_invariant_weakenE [OF node_local_nos_inc]) auto
  qed simp_all

lemma ocnet_bigger_than_next:
  "oclosed (opnet (λi. optoy i ⟨⟨i qmsg) n)
     ⊨ (λ_ _ _. True, other nos_inc (net_tree_ips n) →)
        global (λσ. ∀i∈net_tree_ips n. no (σ i) ≤ no (σ (nhid (σ i))))"
  proof (rule inclosed_closed)
    show "opnet (λi. optoy i ⟨⟨i qmsg) n
            ⊨ (otherwith (=) (net_tree_ips n) inoclosed, other nos_inc (net_tree_ips n) →)
               global (λσ. ∀i∈net_tree_ips n. no (σ i) ≤ no (σ (nhid (σ i))))"
    proof (rule oinvariant_weakenE [OF opnet_bigger_than_next])
      fix s s':: "nat ⇒ state" and a :: "msg node_action"
      assume "otherwith (=) (net_tree_ips n) inoclosed s s' a"
      thus "otherwith nos_inc (net_tree_ips n) (oarrivemsg msg_ok) s s' a"
      proof (rule otherwithE, intro otherwithI)
        assume "inoclosed s a"
           and "∀j. j ∉ net_tree_ips n ⟶ s j = s' j"
           and "otherwith ((=)) (net_tree_ips n) inoclosed s s' a"
        thus "oarrivemsg msg_ok s a"
          by (cases a) auto
      qed auto
    qed simp
  qed
```

26.8 Transfer

```
definition
  initmissing :: "(nat ⇒ state option) × 'a ⇒ (nat ⇒ state) × 'a"
where
  "initmissing σ = (λi. case (fst σ) i of None ⇒ toy_init i | Some s ⇒ s, snd σ)"

lemma not_in_net_ips_fst_init_missing [simp]:
  assumes "i ∉ net_ips σ"
    shows "fst (initmissing (netgmap fst σ)) i = toy_init i"
  using assms unfolding initmissing_def by simp

lemma fst_initmissing_netgmap_pair_fst [simp]:
  "fst (initmissing (netgmap (λ(p, q). (fst (Fun.id p), snd (Fun.id p), q)) s))
                                          = fst (initmissing (netgmap fst s))"
  unfolding initmissing_def by auto
```

```
interpretation toy_openproc: openproc ptoy optoy Fun.id
  rewrites "toy_openproc.initmissing = initmissing"
  proof -
    show "openproc ptoy optoy Fun.id"
    proof unfold_locales
      fix i :: ip
      have "{(σ, ζ). (σ i, ζ) ∈ σ_TOY i ∧ (∀j. j ≠ i ⟶ σ j ∈ fst ` σ_TOY j)} ⊆ σ_OTOY"
        unfolding σ_TOY_def σ_OTOY_def
        proof (rule equalityD1)
          show "⋀f p. {(σ, ζ). (σ i, ζ) ∈ {(f i, p)} ∧ (∀j. j ≠ i
                      ⟶ σ j ∈ fst ` {(f j, p)})} = {(f, p)}"
            by (rule set_eqI) auto
        qed
      thus "{ (σ, ζ) |σ ζ s. s ∈ init (ptoy i)
                             ∧ (σ i, ζ) = Fun.id s
                             ∧ (∀j. j≠i ⟶ σ j ∈ (fst o Fun.id) ` init (ptoy j)) } ⊆ init (optoy i)"
        by simp
    next
      show "∀j. init (ptoy j) ≠ {}"
        unfolding σ_TOY_def by simp
    next
      fix i s a s' σ σ'
      assume "σ i = fst (Fun.id s)"
         and "σ' i = fst (Fun.id s')"
         and "(s, a, s') ∈ trans (ptoy i)"
      then obtain q q' where "s = (σ i, q)"
                         and "s' = (σ' i, q')"
                         and "((σ i, q), a, (σ' i, q')) ∈ trans (ptoy i)"
         by (cases s, cases s') auto
      from this(3) have "((σ, q), a, (σ', q')) ∈ trans (optoy i)"
        by simp (rule open_seqp_action [OF toy_wf])

      with ‹s = (σ i, q)› and ‹s' = (σ' i, q')›
        show "((σ, snd (Fun.id s)), a, (σ', snd (Fun.id s'))) ∈ trans (optoy i)"
          by simp
    qed
    then interpret op0: openproc ptoy optoy Fun.id .
    have [simp]: "⋀i. (SOME x. x ∈ (fst o Fun.id) ` init (ptoy i)) = toy_init i"
      unfolding σ_TOY_def by simp
    hence "⋀i. openproc.initmissing ptoy Fun.id i = initmissing i"
      unfolding op0.initmissing_def op0.someinit_def initmissing_def
      by (auto split: option.split)
    thus "openproc.initmissing ptoy Fun.id = initmissing" ..
  qed

lemma fst_initmissing_netgmap_default_toy_init_netlift:
  "fst (initmissing (netgmap sr s)) = default toy_init (netlift sr s)"
  unfolding initmissing_def default_def
  by (simp add: fst_netgmap_netlift del: One_nat_def)

definition
  netglobal :: "((nat ⇒ state) ⇒ bool) ⇒ ((state × 'b) × 'c) net_state ⇒ bool"
where
  "netglobal P ≡ (λs. 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 -
    show "openproc_parq ptoy optoy Fun.id qmsg"
      by (unfold_locales) simp
    then interpret opq: openproc_parq ptoy optoy Fun.id qmsg .

    have im: "⋀σ. openproc.initmissing (λi. ptoy i ⟨⟨ qmsg) (λ(p, q). (fst (Fun.id p), snd (Fun.id p),
```

```
q)) σ
                                                                                = initmissing σ"
      unfolding opq.initmissing_def opq.someinit_def initmissing_def
      unfolding σTOY_def σQMSG_def by (clarsimp cong: option.case_cong)
    thus "openproc.initmissing (λi. ptoy i ⟨⟨ qmsg) (λ(p, q). (fst (Fun.id p), snd (Fun.id p), q)) = initmiss
      by (rule ext)

    have "⋀P σ. openproc.netglobal (λi. ptoy i ⟨⟨ qmsg) (λ(p, q). (fst (Fun.id p), snd (Fun.id p), q))
P σ
                                                                            = netglobal P σ"
      unfolding opq.netglobal_def netglobal_def opq.initmissing_def initmissing_def opq.someinit_def
      unfolding σTOY_def σQMSG_def
      by (clarsimp cong: option.case_cong
                   simp del: One_nat_def
                   simp add: fst_initmissing_netgmap_default_toy_init_netlift
                                          [symmetric, unfolded initmissing_def])
    thus "openproc.netglobal (λi. ptoy i ⟨⟨ qmsg) (λ(p, q). (fst (Fun.id p), snd (Fun.id p), q)) = netglobal
      by auto
  qed
```

26.9 Final result

```
lemma bigger_than_next:
  assumes "wf_net_tree n"
  shows "closed (pnet (λi. ptoy i ⟨⟨ qmsg) n) ⊫ netglobal (λσ. ∀i. no (σ i) ≤ no (σ (nhid (σ i))))"
        (is "_ ⊫ netglobal (λσ. ∀i. ?inv σ i)")
  proof -
    from ‹wf_net_tree n›
      have proto: "closed (pnet (λi. ptoy i ⟨⟨ qmsg) n)
                    ⊫ netglobal (λσ. ∀i∈net_tree_ips n. no (σ i) ≤ no (σ (nhid (σ i))))"
        by (rule toy_openproc_par_qmsg.close_opnet [OF _ ocnet_bigger_than_next])
    show ?thesis
    unfolding invariant_def opnet_sos.opnet_tau1
    proof (rule, simp only: toy_openproc_par_qmsg.netglobalsimp
                            fst_initmissing_netgmap_pair_fst, rule allI)
      fix σ i
      assume sr: "σ ∈ reachable (closed (pnet (λi. ptoy i ⟨⟨ qmsg) n)) TT"
      hence "∀i∈net_tree_ips n. ?inv (fst (initmissing (netgmap fst σ))) i"
        by - (drule invariantD [OF proto],
              simp only: toy_openproc_par_qmsg.netglobalsimp
                         fst_initmissing_netgmap_pair_fst)
      thus "?inv (fst (initmissing (netgmap fst σ))) i"
      proof (cases "i∈net_tree_ips n")
        assume "i∉net_tree_ips n"
        from sr have "σ ∈ reachable (pnet (λi. ptoy i ⟨⟨ qmsg) n) TT" ..
        hence "net_ips σ = net_tree_ips n" ..
        with ‹i∉net_tree_ips n› have "i∉net_ips σ" by simp
        hence "(fst (initmissing (netgmap fst σ))) i = toy_init i"
          by simp
        thus ?thesis by simp
      qed metis
    qed
  qed

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.