Mechanization of the Algebra for Wireless Networks (AWN)

Timothy Bourke*

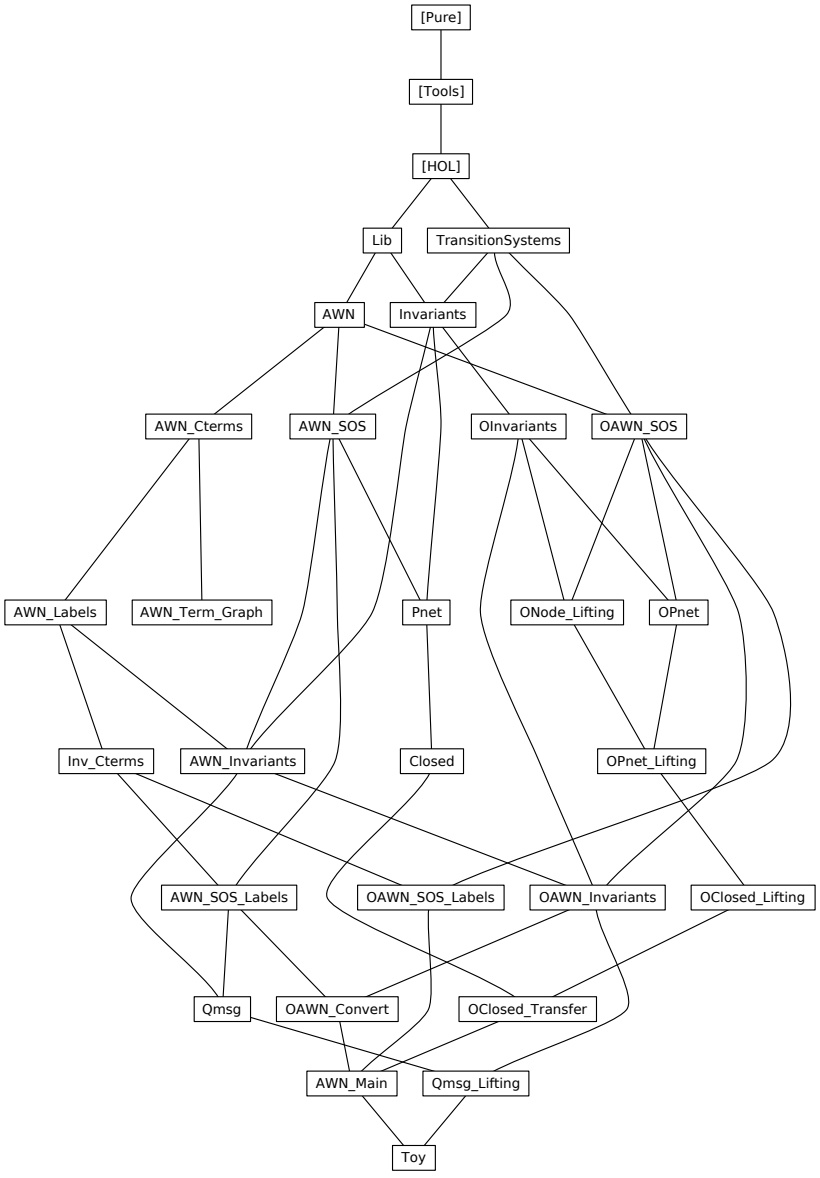
March 17, 2025

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

theory *Lib*
imports *Main*
begin

definition
  TT :: "'a ⇒ bool"
where
  "TT = (λ_. True)"

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

lemma *in_set_tl: "x ∈ set (tl xs) ⟹ x ∈ set xs"*
  ⟨*proof*⟩

lemma *nat_le_eq_or_lt [elim]:*
    fixes *x :: nat*
  assumes *"x ≤ y"*
      and *eq: "x = y ⟹ P x y"*
      and *lt: "x < y ⟹ P x y"*
    shows *"P x y"*
  ⟨*proof*⟩

lemma *disjoint_commute:*
  "(A ∩ B = {}) ⟹ (B ∩ A = {})"
  ⟨*proof*⟩

definition
  default :: "('i ⇒ 's) ⇒ ('i ⇒ 's option) ⇒ ('i ⇒ 's)"
where
  "default df f = (λi. case f i of None ⇒ df i | Some s ⇒ s)"

end

2 Transition systems (automata)

theory *TransitionSystems*
imports *Main*
begin

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

record *('s, 'a) automaton =*
  init :: "'s set"
  trans :: "('s, 'a) transition set"

end

3 Reachability and Invariance

theory *Invariants*
imports *Lib TransitionSystems*
begin

3.1 Reachability

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

inductive_set *reachable*
  for *A :: "('s, 'a) automaton"*

```
  and I :: "'a ⇒ bool"
where
    reachable_init: "s ∈ init A ⟹ s ∈ reachable A I"
  | reachable_step: "⟦ s ∈ reachable A I; (s, a, s') ∈ trans A; I a ⟧ ⟹ s' ∈ reachable A I"

inductive_cases reachable_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"
  ⟨proof⟩

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

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

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

3.2 Invariance

```
definition invariant
  :: "('s, 'a) automaton ⇒ ('a ⇒ bool) ⇒ ('s ⇒ bool) ⇒ bool"
  (<_ ⊨ (1'(_ →')/ _)> [100, 0, 9] 8)
where
  "(A ⊨ (I →) P) = (∀s∈reachable A I. P s)"

abbreviation
  any_invariant
  :: "('s, 'a) automaton ⇒ ('s ⇒ bool) ⇒ bool"
  (<_ ⊨ _> [100, 9] 8)
where
  "(A ⊨ P) ≡ (A ⊨ (TT →) P)"

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

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

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

lemma *invariantD [dest]:*
assumes *"A ⊨ (I →) P"*
and *"s ∈ reachable A I"*
shows *"P s"*
⟨*proof*⟩

lemma *invariant_initE [elim]:*
assumes *invP: "A ⊨ (I →) P"*
and *init: "s ∈ init A"*
shows *"P s"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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)"
⟨*proof*⟩

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"*
⟨*proof*⟩

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')"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *step_invariant_pairI [intro]:*
assumes **: "⋀ξ p ξ' p' a.*
⟦ (ξ, p) ∈ reachable A I; ((ξ, p), a, (ξ', p')) ∈ trans A; I a ⟧
⟹ P ((ξ, p), a, (ξ', p'))"
shows *"A ⊨A (I →) P"*
⟨*proof*⟩

lemma *step_invariant_arbitraryI:*
assumes *"⋀ξ p a ξ' p'. ⟦ (ξ, p) ∈ reachable A I; ((ξ, p), a, (ξ', p')) ∈ trans A; I a ⟧*
⟹ P ((ξ, p), a, (ξ', p'))"
shows *"A ⊨A (I →) P"*
⟨*proof*⟩

lemma *step_invariant_weakenE [elim!]:*
fixes *T σ P Q*
assumes *invP: "A ⊨A (PI →) P"*
and *PQ: "⋀t. P t ⟹ Q t"*
and *QIPI: "⋀a. QI a ⟹ PI a"*
shows *"A ⊨A (QI →) Q"*
⟨*proof*⟩

lemma *step_invariant_weaken_with_invariantE [elim]:*
assumes *pinv: "A ⊨ (I →) P"*
and *qinv: "A ⊨A (I →) Q"*
and *wr: "⋀s a s'. ⟦ P s; P s'; Q (s, a, s'); I a ⟧ ⟹ R (s, a, s')"*
shows *"A ⊨A (I →) R"*
⟨*proof*⟩

lemma *step_to_invariantI:*
assumes *sinv: "A ⊨A (I →) Q"*
and *init: "⋀s. s ∈ init A ⟹ P s"*
and *step: "⋀s s' a.*
⟦ s ∈ reachable A I;
P s;
Q (s, a, s');
I a ⟧ ⟹ P s'"
shows *"A ⊨ (I →) P"*
⟨*proof*⟩

end

4 Open reachability and invariance

theory *OInvariants*
imports *Invariants*
begin

4.1 Open reachability

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

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

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

```
  for A  :: "('g × 'l, 'a) automaton"
  and S  :: "'g ⇒ 'g ⇒ 'a ⇒ bool"
  and U  :: "'g ⇒ 'g ⇒ bool"
where
    oreachable_init: "s ∈ init A ⟹ s ∈ oreachable A S U"
  | oreachable_local: "⟦ s ∈ oreachable A S U; (s, a, s') ∈ trans A; S (fst s) (fst s') a ⟧
                       ⟹ s' ∈ oreachable A S U"
  | oreachable_other: "⟦ s ∈ oreachable A S U; U (fst s) σ' ⟧
                       ⟹ (σ', snd s) ∈ oreachable A S U"

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

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

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

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

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

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

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

4.2 Open Invariance

```
definition oinvariant
  :: "('g × 'l, 'a) automaton
      ⇒ ('g ⇒ 'g ⇒ 'a ⇒ bool) ⇒ ('g ⇒ 'g ⇒ bool)
      ⇒ (('g × 'l) ⇒ bool) ⇒ bool"
  (<_ ⊨ (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"
⟨*proof*⟩

lemma *oinvariant_oreachableI:*
assumes "⋀σ s. (σ, s)∈oreachable A S U ⟹ P (σ, s)"
shows "A ⊨ (S, U →) P"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *oinvariantD [dest]:*
assumes "A ⊨ (S, U →) P"
and "s ∈ oreachable A S U"
shows "P s"
⟨*proof*⟩

lemma *oinvariant_initD [dest, elim]:*
assumes *invP:* "A ⊨ (S, U →) P"
and *init:* "s ∈ init A"
shows "P s"
⟨*proof*⟩

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*⟩

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*⟩

lemma *close_open_invariant:*
assumes *oinv:* "A ⊨ (act I, U →) P"
shows "A ⊫ (I →) P"
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *other_stepsE [elim]:*
```
  assumes "other_steps U J"
      and "U σ σ'"
    shows "∀j∈J. σ' j = σ j"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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*⟩

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*⟩

lemma *oinvariant_anyact:*
```
  assumes "A ⊨ (act TT, U →) P"
```

shows *"A ⊨ (S, U →) P"*
⟨*proof*⟩

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')))"
⟨*proof*⟩

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"*
⟨*proof*⟩

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'))"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *oinvariant_step_anyact:*
```
  assumes "p ⊨_A (act TT, U →) P"
    shows "p ⊨_A (S, U →) P"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *otherwith_actionD [dest]:*
```
  assumes "otherwith Q I P σ σ' a"
    shows "P σ a"
```
⟨*proof*⟩

lemma *otherwith_syncD [dest]:*
```
  assumes "otherwith Q I P σ σ' a"
    shows "∀j. j∉I ⟶ Q (σ j) (σ' j)"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *all_but_eq [dest]:*
  assumes *"σ' i = σ i"*
    and *"∀j. j ≠ i ⟶ σ j = σ' j"*
  shows *"σ = σ'"*
⟨*proof*⟩

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'"*
⟨*proof*⟩

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'"*
⟨*proof*⟩

lemma *other_localD [dest]:*
  "other P {i} σ σ' ⟹ σ' i = σ i"
⟨*proof*⟩

lemma *other_otherD [dest]:*
  "other P {i} σ σ' ⟹ ∀j. j≠i ⟶ P (σ j) (σ' j)"
⟨*proof*⟩

lemma *other_bothE [elim]:*
  assumes *"other P {i} σ σ'"*
  obtains *"σ' i = σ i"* **and** *"∀j. j≠i ⟶ P (σ j) (σ' j)"*
⟨*proof*⟩

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'"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)))"*
⟨*proof*⟩

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)))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

```
lemma subterms_trans [elim]:
  assumes "q ∈ subterms p"
      and "r ∈ subterms q"
    shows "r ∈ subterms p"
```
⟨*proof*⟩

```
lemma root_in_subterms [simp]:
   "⋀Γ pn. ∃pn'. Γ pn ∈ subterms (Γ pn')"
```
⟨*proof*⟩

```
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"
```
⟨*proof*⟩

5.2 Actions

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

```
class tau =
  fixes tau :: "'a" (<τ>)
```

5.2.1 Sequential Actions (and related predicates)

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

instantiation "seq_action" :: (type) tau
begin
definition step_seq_tau [simp]: "τ ≡ τ_s"
instance ⟨proof⟩
end

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

lemma recvmsg_simps[simp]:
  "⋀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"
```
⟨*proof*⟩

lemma *recvmsgTT [simp]: "recvmsg TT a"*
⟨*proof*⟩

lemma *recvmsgE [elim]:*
assumes *"recvmsg (R σ) a"*
and *"⋀m. R σ m ⟹ R σ' m"*
shows *"recvmsg (R σ') a"*
⟨*proof*⟩

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

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

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

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

lemma *orecvmsgEI [elim]:*
"⟦ orecvmsg P σ a; ⋀σ a. P σ a ⟹ Q σ a ⟧ ⟹ orecvmsg Q σ a"
⟨*proof*⟩

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

lemma *orecvmsg_recv_weaken [elim]:*
"⟦ orecvmsg P σ a; ⋀σ a. P σ a ⟹ Q a ⟧ ⟹ recvmsg Q a"
⟨*proof*⟩

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

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

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

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 ⟨*proof*⟩
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"
```
⟨*proof*⟩

lemma *arrivemsgTT [simp]: "arrivemsg i TT = TT"*
⟨*proof*⟩

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"
```
⟨*proof*⟩

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

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 τ_n = True"
⟨*proof*⟩

5.3 Networks

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

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

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

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

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

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

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

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

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

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

lemma *netmap_none_not_in_net_ips:*
assumes *"netmap ns i = None"*

shows *"i∉net_ips ns"*
⟨*proof*⟩

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

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

lemma *netmap_subnets_same:*
assumes *"netmap s1 i = x"*
and *"netmap s2 i = x"*
shows *"netmap (SubnetS s1 s2) i = x"*
⟨*proof*⟩

lemma *netmap_subnets_samef:*
assumes *"netmap s1 = f"*
and *"netmap s2 = f"*
shows *"netmap (SubnetS s1 s2) = f"*
⟨*proof*⟩

lemma *netmap_add_disjoint [elim]:*
assumes *"∀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*⟩

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"*
⟨*proof*⟩

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*⟩

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"
⟨*proof*⟩

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)"

⟨*proof*⟩

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 s_1 s_2) = (let (σ_1, ss) = netgmap sr s_1 in
                                    let (σ_2, tt) = netgmap sr s_2 in
                                    (σ_1 ++ σ_2, SubnetS ss tt))"

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

lemma *netgmap_pair_dom [elim]:*
  obtains σ ζ **where** *"netgmap sr n = (σ, ζ)"*
                and *"dom σ = net_ips n"*
    ⟨*proof*⟩

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

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

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

lemma *fst_netgmap_subnet [simp]:*
  "fst (case netgmap sr s1 of (σ_1, ss) ⇒
        case netgmap sr s2 of (σ_2, tt) ⇒
        (σ_1 ++ σ_2, SubnetS ss tt)) = (fst (netgmap sr s1) ++ fst (netgmap sr s2))"
  ⟨*proof*⟩

lemma *snd_netgmap_subnet [simp]:*
  "snd (case netgmap sr s1 of (σ_1, ss) ⇒
        case netgmap sr s2 of (σ_2, tt) ⇒
        (σ_1 ++ σ_2, SubnetS ss tt)) = (SubnetS (snd (netgmap sr s1)) (snd (netgmap sr s2)))"
  ⟨*proof*⟩

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

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

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

lemma *netgmap_netmask_subnets [elim]:*
  assumes *"netgmap sr s1 = netmask (net_tree_ips n1) (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*⟩

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))"*
⟨*proof*⟩

lemma *netgmap_subnet_split1:*
assumes *"netgmap sr (SubnetS s1 s2) = netmask (net_tree_ips (n1 ‖ n2)) (σ, ζ)"*
and *"net_tree_ips n1 ∩ net_tree_ips n2 = {}"*
and *"net_ips s1 = net_tree_ips n1"*
and *"net_ips s2 = net_tree_ips n2"*
shows *"netgmap sr s1 = netmask (net_tree_ips n1) (σ, snd (netgmap sr s1))"*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

end

6 Semantics of the Algebra of Wireless Networks

theory *AWN_SOS*
imports *TransitionSystems AWN*
begin

6.1 Table 1: Structural operational semantics for sequential process expressions

inductive_set
  seqp_sos
  :: "('s, 'm, 'p, 'l) seqp_env ⇒ ('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"
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *seqp_assignTE [elim]:*
```
"⟦((ξ, {l}⟦u⟧ p), a, (ξ', q)) ∈ seqp_sos Γ; ⟦a = τ; ξ' = u ξ; q = p⟧ ⟹ P⟧ ⟹ P"
```
⟨*proof*⟩

lemma *seqp_guardTE [elim]:*
```
"⟦((ξ, {l}⟨g⟩ p), a, (ξ', q)) ∈ seqp_sos Γ; ⟦a = τ; ξ' ∈ g ξ; q = p⟧ ⟹ P⟧ ⟹ P"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *par_sendTE [elim]:*
```
"⟦((s, t), send m, (s', t')) ∈ parp_sos S T;
  ⟦(s, send m, s') ∈ S; t' = t⟧ ⟹ P⟧ ⟹ P"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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

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

lemmas *parpTEs =*
par_broadcastTE
par_groupcastTE
par_unicastTE
par_notunicastTE
par_sendTE
par_deliverTE
par_receiveTE

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

definition
```
par_comp :: "('s1, 'm seq_action) automaton
             ⇒ ('s2, 'm seq_action) automaton
             ⇒ ('s1 × 's2, 'm seq_action) automaton"
(<(_ ⟨⟨ _)> [102, 103] 102)
```
where
```
"s ⟨⟨ t ≡ (| init = init s × init t, trans = parp_sos (trans s) (trans t) |)"
```

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

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

6.3 Table 3: Structural operational semantics for node expressions

inductive_set
```
node_sos :: "('s, 'm seq_action) transition set ⇒ ('s net_state, 'm node_action) transition set"
for S :: "('s, 'm seq_action) transition set"
```
where
```
  node_bcast:
  "(s, broadcast m, s') ∈ S ⟹ (NodeS i s R, R:*cast(m), NodeS i s' R) ∈ node_sos S"
| node_gcast:
  "(s, groupcast D m, s') ∈ S ⟹ (NodeS i s R, (R∩D):*cast(m), NodeS i s' R) ∈ node_sos S"
| node_ucast:
  "⟦ (s, unicast d m, s') ∈ S; d∈R ⟧ ⟹ (NodeS i s R, {d}:*cast(m), NodeS i s' R) ∈ node_sos S"
| node_notucast:
```

```
    "⟦ (s, ¬unicast d, s') ∈ S; d∉R ⟧ ⟹ (NodeS i s R, τ, NodeS i s' R) ∈ node_sos S"
  | node_deliver:
    "(s, deliver d, s') ∈ S ⟹ (NodeS i s R, i:deliver(d), NodeS i s' R) ∈ node_sos S"
  | node_receive:
    "(s, receive m, s') ∈ S ⟹ (NodeS i s R, {i}¬{}: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)"
⟨proof⟩

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

definition
  node_comp :: "ip ⇒ ('s, 'm seq_action) automaton ⇒ ip set
                   ⇒ ('s net_state, 'm node_action) automaton"
    (‹(⟨_ : (_) : _⟩)› [0, 0, 0] 104)
where
  "⟨i : np : R_i⟩ ≡ (| init = {NodeS i s R_i|s. s ∈ init np}, trans = node_sos (trans np) |)"

lemma trans_node_comp:
  "trans (⟨i : np : R_i⟩) = node_sos (trans np)"
⟨proof⟩

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

lemmas node_comps = trans_node_comp init_node_comp

lemma trans_par_node_comp [simp]:
```

```
  "trans (⟨i : s ⟨⟨ t : R⟩) = node_sos (parp_sos (trans s) (trans t))"
  ⟨proof⟩

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

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

lemma node_sos_dest:
  assumes "(NodeS i p R, a, s') ∈ node_sos S"
    shows "∃P' R'. s' = NodeS i P' R'"
  ⟨proof⟩

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⟩

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"
  ⟨proof⟩
```

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)"
  ⟨proof⟩

fun pnet :: "(ip ⇒ ('s, 'm seq_action) automaton)
             ⇒ net_tree ⇒ ('s net_state, 'm node_action) automaton"
where
    "pnet np (⟨i; R_i⟩)  =  ⟨i : np i : R_i⟩"
  | "pnet np (p_1 ‖ p_2) = (| init = {SubnetS s_1 s_2 |s_1 s_2. s_1 ∈ init (pnet np p_1)
                                                          ∧ s_2 ∈ init (pnet np p_2)},
                              trans = pnet_sos (trans (pnet np p_1)) (trans (pnet np p_2)) |)"

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

lemma pnet_node_init' [elim]:
 assumes "s ∈ init (pnet np ⟨i; R⟩)"
 obtains ns where "s = NodeS i ns R"
             and "ns ∈ init (np i)"
   ⟨proof⟩

lemma pnet_node_trans [elim, simp]:
  assumes "(s, a, s') ∈ trans (pnet np ⟨i; R⟩)"
    shows "(s, a, s') ∈ node_sos (trans (np i))"
  ⟨proof⟩

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

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

6.5 Table 5: Structural operational semantics for complete network expressions

```
inductive_set
  cnet_sos :: "('s, ('m::msg) node_action) transition set
                  ⇒ ('s, 'm node_action) transition set"
  for S :: "('s, 'm node_action) transition set"
```

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

inductive_cases *connect_completeTE: "(s, connect(i, i'), s') ∈ cnet_sos S"*
and *disconnect_completeTE: "(s, disconnect(i, i'), s') ∈ cnet_sos S"*
and *tau_completeTE: "(s, 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*⟩

lemma *complete_no_arrive [simp]:*
"(s, ii¬ni:arrive(m), s') ∉ cnet_sos T"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *microstep_choiceE [elim]:*
"⟦ (p1 ⊕ p2) ⇝Γ p;
(p1 ⊕ p2) ⇝Γ p1 ⟹ P;
(p1 ⊕ p2) ⇝Γ p2 ⟹ P ⟧ ⟹ P"
⟨*proof*⟩

lemma *microstep_callD [dest]:*
"(call(pn)) ⇝Γ p ⟹ p = Γ pn"
⟨*proof*⟩

lemma *microstep_callE [elim]:*
"⟦ (call(pn)) ⇝Γ p; p = Γ(pn) ⟹ P ⟧ ⟹ P"
⟨*proof*⟩

lemma *no_microstep_guard:* *"¬ (({l}⟨g⟩ p) ⇝Γ q)"*
⟨*proof*⟩

lemma *no_microstep_assign:* *"¬ ({l}⟦f⟧ p) ⇝Γ q"*
⟨*proof*⟩

lemma *no_microstep_unicast:* *"¬ (({l}unicast(s_{ip}, s_{msg}).p ▷ q) ⇝Γ r)"*
⟨*proof*⟩

lemma *no_microstep_broadcast:* *"¬ (({l}broadcast(s_{msg}).p) ⇝Γ q)"*
⟨*proof*⟩

lemma *no_microstep_groupcast:* *"¬ (({l}groupcast(s_{ips}, s_{msg}).p) ⇝Γ q)"*
⟨*proof*⟩

lemma *no_microstep_send:* *"¬ (({l}send(s_{msg}).p) ⇝Γ q)"*
⟨*proof*⟩

lemma *no_microstep_deliver:* *"¬ (({l}deliver(s_{data}).p) ⇝Γ q)"*
⟨*proof*⟩

lemma *no_microstep_receive:* *"¬ (({l}receive(u_{msg}).p) ⇝Γ q)"*
⟨*proof*⟩

lemma *microstep_call_or_choice [dest]:*
assumes *"p ⇝Γ q"*

shows *"(∃pn. p = call(pn)) ∨ (∃p1 p2. p = p1 ⊕ p2)"*
⟨*proof*⟩

lemmas *no_microstep [intro,simp] =*
no_microstep_guard
no_microstep_assign
no_microstep_unicast
no_microstep_broadcast
no_microstep_groupcast
no_microstep_send
no_microstep_deliver
no_microstep_receive

7.2 Wellformed process specifications

A process specification Γ is wellformed if its *microstep* Γ relation is free of loops and infinite chains.
For example, these specifications are not wellformed: 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)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

7.3 Start terms (sterms)

Formulate sets of local subterms from which an action is directly possible. Since the process specification Γ is not considered, only choice terms *p1*

⊕

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"*
⟨*proof*⟩

lemma *stermsl_no_choice[simp]: "p1 ⊕ p2 ∉ stermsl p"*
⟨*proof*⟩

lemma *stermsl_choice_disj[simp]:*
"p ∈ stermsl (p1 ⊕ p2) = (p ∈ stermsl p1 ∨ p ∈ stermsl p2)"
⟨*proof*⟩

lemma *stermsl_in_branch[elim]:*
"⟦p ∈ stermsl (p1 ⊕ p2); p ∈ stermsl p1 ⟹ P; p ∈ stermsl p2 ⟹ P⟧ ⟹ P"
⟨*proof*⟩

lemma *stermsl_commute:*
"stermsl (p1 ⊕ p2) = stermsl (p2 ⊕ p1)"
⟨*proof*⟩

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

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

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

lemma *nocall_stermsl_max:*
assumes *"r ∈ stermsl p"*
and *"not_call r"*
shows *"¬ (r ⇝Γ q)"*
⟨*proof*⟩

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

7.4 Start terms

The start terms are those terms, relative to a wellformed process specification 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}"
```
⟨*proof*⟩

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

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

declare *sterms.psimps [simp]*

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

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

lemma *sterms_choice_disj [simp]:*
```
  assumes "wellformed Γ"
    shows "p ∈ sterms Γ (p1 ⊕ p2) = (p ∈ sterms Γ p1 ∨ p ∈ sterms Γ p2)"
```
⟨*proof*⟩

lemma *sterms_no_choice [simp]:*
```
  assumes "wellformed Γ"
    shows "p1 ⊕ p2 ∉ sterms Γ p"
```
⟨*proof*⟩

lemma *sterms_not_choice [simp]:*
```
  assumes "wellformed Γ"
      and "q ∈ sterms Γ p"
    shows "not_choice q"
```
⟨*proof*⟩

lemma *sterms_no_call [simp]:*
```
  assumes "wellformed Γ"
    shows "call(pn) ∉ sterms Γ p"
```
⟨*proof*⟩

lemma *sterms_not_call [simp]:*
```
  assumes "wellformed Γ"
      and "q ∈ sterms Γ p"
    shows "not_call q"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *sterms_commute:*
```
  assumes "wellformed Γ"
    shows "sterms Γ (p1 ⊕ p2) = sterms Γ (p2 ⊕ p1)"
```

⟨*proof*⟩

lemma *sterms_not_empty:*
assumes *"wellformed Γ"*
shows *"sterms Γ p ≠ {}"*
⟨*proof*⟩

lemma *sterms_sterms [simp]:*
assumes *"wellformed Γ"*
shows *"(⋃x∈sterms Γ p. sterms Γ x) = sterms Γ p"*
⟨*proof*⟩

lemma *sterms_stermsl:*
assumes *"ps ∈ sterms Γ p"*
and *"wellformed Γ"*
shows *"ps ∈ stermsl p ∨ (∃pn. ps ∈ stermsl (Γ pn))"*
⟨*proof*⟩

lemma *stermsl_sterms [elim]:*
assumes *"q ∈ stermsl p"*
and *"not_call q"*
and *"wellformed Γ"*
shows *"q ∈ sterms Γ p"*
⟨*proof*⟩

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

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

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

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

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

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

theorem *sterms_maximal_microstep:*
assumes *"wellformed Γ"*

shows *"sterms Γ p = {q. p ⇝Γ* q ∧ ¬(∃q'. q ⇝Γ q')}"*
⟨*proof*⟩

7.5 Derivative terms

The derivatives of a term are those *sterm*s 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)"
⟨*proof*⟩

lemma *dterms_dom_basic [simp]:*
assumes *"not_call p"*
and *"not_choice p"*
shows *"dterms_dom (Γ, p)"*
⟨*proof*⟩

lemma *dterms_termination:*
assumes *"wellformed Γ"*
shows *"dterms_dom (Γ, p)"*
⟨*proof*⟩

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

lemma *sterms_after_dterms [simp]:*
assumes *"wellformed Γ"*
shows *"(⋃x∈dterms Γ p. sterms Γ x) = dterms Γ p"*
⟨*proof*⟩

lemma *sterms_before_dterms [simp]:*
assumes *"wellformed Γ"*
shows *"(⋃x∈sterms Γ p. dterms Γ x) = dterms Γ p"*
⟨*proof*⟩

lemma *dterms_choice_disj [simp]:*
assumes *"wellformed Γ"*
shows *"p ∈ dterms Γ (p1 ⊕ p2) = (p ∈ dterms Γ p1 ∨ p ∈ dterms Γ p2)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *dterms_no_choice:*
assumes *"wellformed Γ"*
shows *"p1 ⊕ p2 ∉ dterms Γ p"*

⟨*proof*⟩

lemma *dterms_not_choice [simp]:*
assumes *"wellformed Γ"*
and *"q ∈ dterms Γ p"*
shows *"not_choice q"*
⟨*proof*⟩

lemma *dterms_no_call:*
assumes *"wellformed Γ"*
shows *"call(pn) ∉ dterms Γ p"*
⟨*proof*⟩

lemma *dterms_not_call [simp]:*
assumes *"wellformed Γ"*
and *"q ∈ dterms Γ p"*
shows *"not_call q"*
⟨*proof*⟩

lemma *dterms_subterms:*
assumes *wf: "wellformed Γ"*
and *"∃pn. p ∈ subterms (Γ pn)"*
and *"q ∈ dterms Γ p"*
shows *"∃pn. q ∈ subterms (Γ pn)"*
⟨*proof*⟩

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 *dterm*s 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"*
⟨*proof*⟩

lemma *cterms_no_choice [simp]:*
assumes *"wellformed Γ"*
shows *"p1 ⊕ p2 ∉ cterms Γ"*
⟨*proof*⟩

lemma *cterms_not_call [simp]:*
assumes *"wellformed Γ"*
and *"p ∈ cterms Γ"*
shows *"not_call p"*
⟨*proof*⟩

lemma *cterms_no_call [simp]:*

assumes *"wellformed Γ"*
shows *"call(pn) ∉ cterms Γ"*
⟨*proof*⟩

lemma *sterms_cterms [elim]:*
assumes *"p ∈ cterms Γ"*
and *"q ∈ sterms Γ p"*
and *"wellformed Γ"*
shows *"q ∈ cterms Γ"*
⟨*proof*⟩

lemma *dterms_cterms [elim]:*
assumes *"p ∈ cterms Γ"*
and *"q ∈ dterms Γ p"*
and *"wellformed Γ"*
shows *"q ∈ cterms Γ"*
⟨*proof*⟩

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"
⟨*proof*⟩

7.7 Local control terms

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

function
ctermsl :: "('s, 'm, 'p, 'l) seqp ⇒ ('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)}"
⟨*proof*⟩
termination ⟨*proof*⟩

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

lemma *ctermsl_refl [intro]: "not_choice p ⟹ p ∈ ctermsl p"*
⟨*proof*⟩

lemma *ctermsl_subterms:*
"ctermsl p = {q. q ∈ subterms p ∧ not_choice q }" (**is** *"?lhs = ?rhs"*)
⟨*proof*⟩

lemma *ctermsl_trans [elim]:*
assumes "q ∈ ctermsl p"
and "r ∈ ctermsl q"
shows "r ∈ ctermsl p"
⟨*proof*⟩

lemma *ctermsl_ex_trans [elim]:*
assumes "∃q ∈ ctermsl p. r ∈ ctermsl q"
shows "r ∈ ctermsl p"
⟨*proof*⟩

lemma *call_ctermsl_empty [elim]:*
"⟦ p ∈ ctermsl p'; not_call p ⟧ ⟹ not_call p'"
⟨*proof*⟩

lemma *stermsl_ctermsl_choice1 [simp]:*
assumes "q ∈ stermsl p1"
shows "q ∈ ctermsl (p1 ⊕ p2)"
⟨*proof*⟩

lemma *stermsl_ctermsl_choice2 [simp]:*
assumes "q ∈ stermsl p2"
shows "q ∈ ctermsl (p1 ⊕ p2)"
⟨*proof*⟩

lemma *stermsl_ctermsl [elim]:*
assumes "q ∈ stermsl p"
shows "q ∈ ctermsl p"
⟨*proof*⟩

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

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

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

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

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

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

7.8 Local deriviative terms

We define local *dterm*s 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))                      = {}"
```
⟨*proof*⟩
termination ⟨*proof*⟩

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

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

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

lemma *dtermsl_choice_disj [simp]:*
```
"p ∈ dtermsl (p1 ⊕ p2) = (p ∈ dtermsl p1 ∨ p ∈ dtermsl p2)"
```
⟨*proof*⟩

lemma *dtermsl_in_branch [elim]:*
```
"⟦p ∈ dtermsl (p1 ⊕ p2); p ∈ dtermsl p1 ⟹ P; p ∈ dtermsl p2 ⟹ P⟧ ⟹ P"
```
⟨*proof*⟩

lemma *ctermsl_dtermsl [elim]:*
```
assumes "q ∈ dtermsl p"
  shows "q ∈ ctermsl p"
```
⟨*proof*⟩

lemma *dtermsl_dterms [elim]:*
```
assumes "q ∈ dtermsl p"
    and "not_call q"
    and "wellformed Γ"
  shows "q ∈ dterms Γ p"
```
⟨*proof*⟩

lemma *ctermsl_stermsl_or_dtermsl:*
```
assumes "q ∈ ctermsl p"
  shows "q ∈ stermsl p ∨ (∃p'∈dtermsl p. q ∈ ctermsl p')"
```
⟨*proof*⟩

lemma *dtermsl_add_stermsl_beforeD:*
```
assumes "q ∈ dtermsl p"
  shows "∃ps∈stermsl p. q ∈ dtermsl ps"
```
⟨*proof*⟩

lemma *call_dtermsl_empty [elim]:*
```
"q ∈ dtermsl p ⟹ not_call p"
```
⟨*proof*⟩

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_reac able, 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*⟩

lemma *ctermsE [elim]:*
assumes *"wellformed Γ"*
and *"p ∈ cterms Γ"*
obtains *pn* **where** *"p ∈ ctermsl (Γ pn)"*
and *"not_call p"*
⟨*proof*⟩

corollary *cterms_subterms:*
assumes *"wellformed Γ"*
shows *"cterms Γ = {p|p pn. p∈subterms (Γ pn) ∧ not_call p ∧ not_choice p}"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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

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

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

end

8 Labelling sequential processes

theory *AWN_Labels*
imports *AWN AWN_Cterms*
begin

8.1 Labels

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

function *(domintros) labels*
:: "('s, 'm, 'p, 'l) seqp_env ⇒ ('s, 'm, 'p, 'l) seqp ⇒ 'l set"
where
"labels Γ ({l}⟨fg⟩ p) = {l}"
| "labels Γ ({l}⟦fa⟧ p) = {l}"
| "labels Γ (p1 ⊕ p2) = labels Γ p1 ∪ labels Γ p2"

```
  | "labels Γ ({l}unicast(fip, fmsg).p ▷ q)  = {l}"
  | "labels Γ ({l}broadcast(fmsg). p)         = {l}"
  | "labels Γ ({l}groupcast(fips, fmsg). p)   = {l}"
  | "labels Γ ({l}send(fmsg).p)               = {l}"
  | "labels Γ ({l}deliver(fdata).p)           = {l}"
  | "labels Γ ({l}receive(fmsg).p)            = {l}"
  | "labels Γ (call(pn))                      = labels Γ (Γ pn)"
```
⟨*proof*⟩

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

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

declare *labels.psimps[simp]*

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

lemma *labels_not_empty:*
```
    fixes Γ p
  assumes "wellformed Γ"
    shows "labels Γ p ≠ {}"
```
⟨*proof*⟩

lemma *has_label [dest]:*
```
    fixes Γ p
  assumes "wellformed Γ"
    shows "∃l. l ∈ labels Γ p"
```
⟨*proof*⟩

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')"
```
⟨*proof*⟩

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'"
```
⟨*proof*⟩

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"*
⟨*proof*⟩

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 *l₁ = l₂*.
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 *l₁* 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'"*
⟨*proof*⟩

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'"*
⟨*proof*⟩

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"*
⟨*proof*⟩

primrec *labelfrom :: "int ⇒ int ⇒ ('s, 'm, 'p, 'a) seqp ⇒ int × ('s, 'm, 'p, int) seqp"*
where
"labelfrom n nn ({_}⟨f⟩ p) =
(let (nn', p') = labelfrom nn (nn + 1) p in
(nn', {n}⟨f⟩ p'))"
| "labelfrom n nn ({_}⟦f⟧ p) =
(let (nn', p') = labelfrom nn (nn + 1) p in
(nn', {n}⟦f⟧ p'))"

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

datatype 'pn label =
    LABEL 'pn int  (<_-:_> [1000, 1000] 999)

instantiation "label" :: (ord) ord
begin

fun less_eq_label :: "'a label ⇒ 'a label ⇒ bool"
where "(l1-:n1) ≤ (l2-:n2) = (l1 = l2 ∧ n1 ≤ n2)"

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

instance ⟨proof⟩
end

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

end
```

9 A custom tactic for showing invariants via control terms

theory *Inv_Cterms*
imports *AWN_Labels*
begin

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

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

To configure the tactic for a set of transition rules:

1. add elimination rules: declare seqpTEs [cterms_seqte]

2. add rules to replace derivative terms: declare elimders [cterms_elimders]

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

1. add simp rules: declare Γ.simps [cterms_env]
2. add case rules: declare 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"* ⟨*proof*⟩

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

⟨*ML*⟩

declare
insert_iff [cterms_env]
Un_insert_right [cterms_env]
sup_bot_right [cterms_env]
Product_Type.prod_cases [cterms_env]
ctermsl.simps [cterms_env]

end

10 Configure the inv-cterms tactic for sequential processes

theory *AWN_SOS_Labels*
imports *AWN_SOS Inv_Cterms*
begin

lemma *elimder_guard:*
assumes *"p = {l}⟨fg⟩ qq"*
and *"l' ∈ labels Γ q"*
and *"((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"*
obtains *p'* **where** *"p = {l}⟨fg⟩ p'"*
and *"l' ∈ labels Γ qq"*
⟨*proof*⟩

lemma *elimder_assign:*
assumes *"p = {l}⟦fa⟧ qq"*
and *"l' ∈ labels Γ q"*
and *"((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"*
obtains *p'* **where** *"p = {l}⟦fa⟧ p'"*
and *"l' ∈ labels Γ qq"*
⟨*proof*⟩

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

lemma *elimder_bcast:*
assumes *"p = {l}broadcast(fmsg).qq"*
and *"l' ∈ labels Γ q"*
and *"((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"*
obtains *p'* **where** *"p = {l}broadcast(fmsg). p'"*
and *"l' ∈ labels Γ qq"*
⟨*proof*⟩

lemma *elimder_gcast:*
assumes *"p = {l}groupcast(fips, fmsg).qq"*
and *"l' ∈ labels Γ q"*
and *"((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"*
obtains *p'* **where** *"p = {l}groupcast(fips, fmsg). p'"*
and *"l' ∈ labels Γ qq"*
⟨*proof*⟩

lemma *elimder_send:*
assumes *"p = {l}send(fmsg).qq"*
and *"l' ∈ labels Γ q"*
and *"((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"*
obtains *p'* **where** *"p = {l}send(fmsg). p'"*
and *"l' ∈ labels Γ qq"*
⟨*proof*⟩

lemma *elimder_deliver:*
assumes *"p = {l}deliver(fdata).qq"*
and *"l' ∈ labels Γ q"*
and *"((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"*
obtains *p'* **where** *"p = {l}deliver(fdata).p'"*
and *"l' ∈ labels Γ qq"*
⟨*proof*⟩

lemma *elimder_receive:*
assumes *"p = {l}receive(fmsg).qq"*
and *"l' ∈ labels Γ q"*
and *"((ξ, p), a, (ξ', q)) ∈ seqp_sos Γ"*
obtains *p'* **where** *"p = {l}receive(fmsg).p'"*
and *"l' ∈ labels Γ qq"*
⟨*proof*⟩

lemmas *elimders =*
elimder_guard
elimder_assign
elimder_ucast
elimder_bcast
elimder_gcast
elimder_send
elimder_deliver
elimder_receive

declare
seqpTEs [cterms_seqte]
elimders [cterms_elimders]

end

11 Lemmas for partial networks

theory *Pnet*
imports *AWN_SOS Invariants*
begin

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

lemma *pnet_maintains_dom:*

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

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

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

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

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

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

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

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

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

lemma *reachable_par_subnet_induct [consumes, case_names init step]:*
assumes *"SubnetS s t ∈ reachable (pnet np (p1 ‖ p2)) I"*
and *init: "⋀s t. SubnetS s t ∈ init (pnet np (p1 ‖ p2)) ⟹ P s t"*
and *step: "⋀s t s' t' a. ⟦*
SubnetS s t ∈ reachable (pnet np (p1 ‖ p2)) I;
P s t; (SubnetS s t, a, SubnetS s' t') ∈ (trans (pnet np (p1 ‖ p2))); I a ⟧
⟹ P s' t'"
shows *"P s t"*
⟨*proof*⟩

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

lemma *delivered_to_node [elim]:*

assumes *"s ∈ reachable (pnet np ⟨ii; R_i⟩) TT"*
and *"(s, i:deliver(d), s') ∈ trans (pnet np ⟨ii; R_i⟩)"*
shows *"i = ii"*
⟨*proof*⟩

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

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

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

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

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

fun *net_ip_action :: "(ip ⇒ ('s, 'm seq_action) automaton)*
⇒ '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')"
⟨*proof*⟩

lemma *pnet_deliver_single_node [elim]:*
assumes *"wf_net_tree p"*
and *"s ∈ reachable (pnet np p) TT"*
and *"(s, i:deliver(d), s') ∈ trans (pnet np p)"*
shows *"(∀j. j≠i ⟶ netmap s' j = netmap s j) ∧ net_ip_action np (i:deliver(d)) i p s s'"*
(**is** *"?P p s s'"*)
⟨*proof*⟩

end

12 Lemmas for closed networks

theory *Closed*
imports *Pnet*
begin

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

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

lemma *closed_reachable_par_subnet_induct [consumes, case_names init step]:*
assumes "SubnetS s t ∈ reachable (closed (pnet np (p1 ‖ p2))) I"
and *init:* "⋀s t. SubnetS s t ∈ init (closed (pnet np (p1 ‖ p2))) ⟹ P s t"
and *step:* "⋀s t s' t' a. ⟦
SubnetS s t ∈ reachable (closed (pnet np (p1 ‖ p2))) I;
P s t; (SubnetS s t, a, SubnetS s' t') ∈ trans (closed (pnet np (p1 ‖ p2))); I a ⟧
⟹ P s' t'"
shows "P s t"
⟨*proof*⟩

lemma *reachable_closed_reachable_pnet [elim]:*
assumes "s ∈ reachable (closed (pnet np n)) TT"
shows "s ∈ reachable (pnet np n) TT"
⟨*proof*⟩

lemma *closed_node_net_state [elim]:*
assumes "st ∈ reachable (closed (pnet np ⟨ii; R_i⟩)) TT"
obtains ξ p q R **where** "st = NodeS ii ((ξ, p), q) R"
⟨*proof*⟩

lemma *closed_subnet_net_state [elim]:*
assumes "st ∈ reachable (closed (pnet np (p1 ‖ p2))) TT"
obtains s t **where** "st = SubnetS s t"
⟨*proof*⟩

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

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

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

end

13 Open semantics of the Algebra of Wireless Networks

theory *OAWN_SOS*
imports *TransitionSystems AWN*

begin

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

13.1 Open structural operational semantics for sequential process expressions

inductive_set
  oseqp_sos
  :: "('s, 'm, 'p, 'l) seqp_env ⇒ ip
      ⇒ ((ip ⇒ 's) × ('s, 'm, 'p, 'l) seqp, 'm seq_action) transition set"
  for 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_s
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"
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

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"
⟨proof⟩
```

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"
⟨proof⟩
```

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"
⟨proof⟩
```

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"
⟨proof⟩
```

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"
⟨proof⟩
```

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"
⟨proof⟩
```

lemmas *oseqpTEs =*
oseq_broadcastTE
oseq_groupcastTE
oseq_unicastTE
oseq_sendTE
oseq_deliverTE
oseq_receiveTE
oseq_assignTE
oseq_callTE
oseq_choiceTE
oseq_guardTE

declare *oseqp_sos.intros [intro]*

13.2 Open structural operational semantics for parallel process expressions

inductive_set

```
  oparp_sos :: "ip
              ⇒ ((ip ⇒ 's) × 's1, 'm seq_action) transition set
              ⇒ ('s2, 'm seq_action) transition set
              ⇒ ((ip ⇒ 's) × ('s1 × 's2), 'm seq_action) transition set"
  for i :: ip
  and S :: "((ip ⇒ 's) × 's1, 'm seq_action) transition set"
  and T :: "('s2, 'm seq_action) transition set"
where
    oparleft:  "⟦ ((σ, s), a, (σ', s')) ∈ S; ⋀m. a ≠ receive m ⟧ ⟹
                ((σ, (s, t)), a, (σ', (s', t))) ∈ oparp_sos i S T"
  | oparright: "⟦ (t, a, t') ∈ T; ⋀m. a ≠ send m; σ' i = σ i ⟧ ⟹
                ((σ, (s, t)), a, (σ', (s, t'))) ∈ oparp_sos i S T"
  | oparboth:  "⟦ ((σ, s), receive m, (σ', s')) ∈ S; (t, send m, t') ∈ T ⟧ ⟹
                ((σ, (s, t)), τ, (σ', (s', t'))) ∈ oparp_sos i S T"
```

lemma *opar_broadcastTE [elim]:*

```
"⟦((σ, (s, t)), broadcast m, (σ', (s', t'))) ∈ oparp_sos i S T;
  ⟦((σ, s), broadcast m, (σ', s')) ∈ S; t' = t⟧ ⟹ P;
  ⟦(t, broadcast m, t') ∈ T; s' = s; σ' i = σ i⟧ ⟹ P⟧ ⟹ P"
```

⟨*proof*⟩

lemma *opar_groupcastTE [elim]:*
"⟦((σ, (s, t)), groupcast ips m, (σ', (s', t'))) ∈ oparp_sos i S T;
⟦((σ, s), groupcast ips m, (σ', s')) ∈ S; t' = t⟧ ⟹ P;
⟦(t, groupcast ips m, t') ∈ T; s' = s; σ' i = σ i⟧ ⟹ P⟧ ⟹ P"
⟨*proof*⟩

lemma *opar_unicastTE [elim]:*
"⟦((σ, (s, t)), unicast i m, (σ', (s', t'))) ∈ oparp_sos i S T;
⟦((σ, s), unicast i m, (σ', s')) ∈ S; t' = t⟧ ⟹ P;
⟦(t, unicast i m, t') ∈ T; s' = s; σ' i = σ i⟧ ⟹ P⟧ ⟹ P"
⟨*proof*⟩

lemma *opar_notunicastTE [elim]:*
"⟦((σ, (s, t)), notunicast i, (σ', (s', t'))) ∈ oparp_sos i S T;
⟦((σ, s), notunicast i, (σ', s')) ∈ S; t' = t⟧ ⟹ P;
⟦(t, notunicast i, t') ∈ T; s' = s; σ' i = σ i⟧ ⟹ P⟧ ⟹ P"
⟨*proof*⟩

lemma *opar_sendTE [elim]:*
"⟦((σ, (s, t)), send m, (σ', (s', t'))) ∈ oparp_sos i S T;
⟦((σ, s), send m, (σ', s')) ∈ S; t' = t⟧ ⟹ P⟧ ⟹ P"
⟨*proof*⟩

lemma *opar_deliverTE [elim]:*
"⟦((σ, (s, t)), deliver d, (σ', (s', t'))) ∈ oparp_sos i S T;
⟦((σ, s), deliver d, (σ', s')) ∈ S; t' = t⟧ ⟹ P;
⟦(t, deliver d, t') ∈ T; s' = s; σ' i = σ i⟧ ⟹ P⟧ ⟹ P"
⟨*proof*⟩

lemma *opar_receiveTE [elim]:*
"⟦((σ, (s, t)), receive m, (σ', (s', t'))) ∈ oparp_sos i S T;
⟦(t, receive m, t') ∈ T; s' = s; σ' i = σ i⟧ ⟹ P⟧ ⟹ P"
⟨*proof*⟩

inductive_cases *opar_tauTE: "((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"
⟨*proof*⟩

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))"
⟨*proof*⟩

lemma *extg_range_prod: "extg ‘ (i1 × i2) = {(σ, (s1, s2))|σ s1 s2. (σ, s1) ∈ i1 ∧ s2 ∈ i2}"*
⟨*proof*⟩

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) |)"
```
⟨*proof*⟩

lemma *trans_opar_comp [simp]:*
```
  "trans (s ⟨⟨i t) = oparp_sos i (trans s) (trans t)"
```
⟨*proof*⟩

lemma *init_opar_comp [simp]:*
```
  "init (s ⟨⟨i t) = extg ‘ (init s × init t)"
```
⟨*proof*⟩

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})"
  ⟨proof⟩

definition
  onode_comp :: "ip
                 ⇒ ((ip ⇒ 's) × 'l, 'm seq_action) automaton
                 ⇒ ip set
                 ⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) automaton"
    (‹(⟨_ : (_) : _⟩o)› [0, 0, 0] 104)
where
  "⟨i : onp : R_i⟩o ≡ (| init = {(σ, NodeS i s R_i)|σ s. (σ, s) ∈ init onp},
                       trans = onode_sos (trans onp) |)"

lemma trans_onode_comp:
  "trans (⟨i : S : R⟩o) = onode_sos (trans S)"
  ⟨proof⟩

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

lemmas onode_comps = trans_onode_comp init_onode_comp

lemma fst_par_onode_comp [simp]:
  "trans (⟨i : s ⟨⟨I t : R⟩o) = onode_sos (oparp_sos I (trans s) (trans t))"
  ⟨proof⟩

lemma init_par_onode_comp [simp]:
  "init (⟨i : s ⟨⟨I t : R⟩o) = {(σ, NodeS i (s1, s2) R)|σ s1 s2. ((σ, s1), s2) ∈ init s × init t}"
  ⟨proof⟩
```

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

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

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

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

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

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

13.4 Open structural operational semantics for partial network expressions

inductive_set
opnet_sos :: "((ip ⇒ 's) × 'l net_state, 'm node_action) transition set
⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) transition set
⇒ ((ip ⇒ 's) × 'l net_state, 'm node_action) transition set"
for *S :: "((ip ⇒ 's) × 'l net_state, 'm node_action) transition set"*
and *T :: "((ip ⇒ 's) × 'l net_state, 'm node_action) transition set"*
where
opnet_cast1:
"⟦ ((σ, s), R:*cast(m), (σ', s')) ∈ S; ((σ, t), H¬K:arrive(m), (σ', t')) ∈ T; H ⊆ R; K ∩ R = {} ⟧
⟹ ((σ, SubnetS s t), R:*cast(m), (σ', SubnetS s' t')) ∈ opnet_sos S T"

| *opnet_cast2:*
"⟦ ((σ, s), H¬K:arrive(m), (σ', s')) ∈ S; ((σ, t), R:*cast(m), (σ', t')) ∈ T; H ⊆ R; K ∩ R = {} ⟧
⟹ ((σ, SubnetS s t), R:*cast(m), (σ', SubnetS s' t')) ∈ opnet_sos S T"

| *opnet_arrive:*

```
    "⟦ ((σ, s), H¬K:arrive(m), (σ', s')) ∈ S; ((σ, t), H'¬K':arrive(m), (σ', t')) ∈ T ⟧
      ⟹ ((σ, SubnetS s t),  (H ∪ H')¬(K ∪ K'):arrive(m), (σ', SubnetS s' t')) ∈ opnet_sos S T"

  | opnet_deliver1:
    "((σ, s), i:deliver(d), (σ', s')) ∈ S
      ⟹ ((σ, SubnetS s t), i:deliver(d), (σ', SubnetS s' t)) ∈ opnet_sos S T"

  | opnet_deliver2:
    "⟦ ((σ, t), i:deliver(d), (σ', t')) ∈ T ⟧
      ⟹ ((σ, SubnetS s t), i:deliver(d), (σ', SubnetS s t')) ∈ opnet_sos S T"

  | opnet_tau1:
    "((σ, s), τ, (σ', s')) ∈ S ⟹ ((σ, SubnetS s t), τ, (σ', SubnetS s' t)) ∈ opnet_sos S T"

  | opnet_tau2:
    "((σ, t), τ, (σ', t')) ∈ T ⟹ ((σ, SubnetS s t), τ, (σ', SubnetS s t')) ∈ opnet_sos S T"

  | opnet_connect:
    "⟦ ((σ, s), connect(i, i'), (σ', s')) ∈ S; ((σ, t), connect(i, i'), (σ', t')) ∈ T ⟧
      ⟹ ((σ, SubnetS s t), connect(i, i'), (σ', SubnetS s' t')) ∈ opnet_sos S T"

  | opnet_disconnect:
    "⟦ ((σ, s), disconnect(i, i'), (σ', s')) ∈ S; ((σ, t), disconnect(i, i'), (σ', t')) ∈ T ⟧
      ⟹ ((σ, SubnetS s t), disconnect(i, i'), (σ', SubnetS s' t')) ∈ opnet_sos S T"

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

fun opnet :: "(ip ⇒ ((ip ⇒ 's) × 'l, 'm seq_action) automaton)
              ⇒ 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 (p_1 ‖ p_2) = (| init = {(σ, SubnetS s_1 s_2) |σ s_1 s_2.
                                        (σ, s_1) ∈ init (opnet onp p_1)
                                      ∧ (σ, s_2) ∈ init (opnet onp p_2)
                                      ∧ net_ips s_1 ∩ net_ips s_2 = {}},
                           trans = opnet_sos (trans (opnet onp p_1)) (trans (opnet onp p_2)) |)"

lemma opnet_node_init [elim, simp]:
  assumes "(σ, s) ∈ init (opnet onp ⟨i; R⟩)"
    shows "(σ, s) ∈ { (σ, NodeS i ns R) |σ ns. (σ, ns) ∈ init (onp i)}"
  ⟨proof⟩

lemma opnet_node_init' [elim]:
 assumes "(σ, s) ∈ init (opnet onp ⟨i; R⟩)"
 obtains ns where "s = NodeS i ns R"
             and "(σ, ns) ∈ init (onp i)"
   ⟨proof⟩

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

13.5 Open structural operational semantics for complete network expressions

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

```
  for S :: "((ip ⇒ 's) × 'l net_state, 'm node_action) transition set"
where
    ocnet_connect:
    "⟦ ((σ, s), connect(i, i'), (σ', s')) ∈ S; ∀j. j ∉ net_ips s ⟶ (σ' j = σ j) ⟧
     ⟹ ((σ, s), connect(i, i'), (σ', s')) ∈ ocnet_sos S"

  | ocnet_disconnect:
    "⟦ ((σ, s), disconnect(i, i'), (σ', s')) ∈ S; ∀j. j ∉ net_ips s ⟶ (σ' j = σ j) ⟧
     ⟹ ((σ, s), disconnect(i, i'), (σ', s')) ∈ ocnet_sos S"

  | ocnet_cast:
    "⟦ ((σ, s), R:*cast(m), (σ', s')) ∈ S; ∀j. j ∉ net_ips s ⟶ (σ' j = σ j) ⟧
     ⟹ ((σ, s), τ, (σ', s')) ∈ ocnet_sos S"

  | ocnet_tau:
    "⟦ ((σ, s), τ, (σ', s')) ∈ S; ∀j. j ∉ net_ips s ⟶ (σ' j = σ j) ⟧
     ⟹ ((σ, s), τ, (σ', s')) ∈ ocnet_sos S"

  | ocnet_deliver:
    "⟦ ((σ, s), i:deliver(d), (σ', s')) ∈ S; ∀j. j ∉ net_ips s ⟶ (σ' j = σ j) ⟧
     ⟹ ((σ, s), i:deliver(d), (σ', s')) ∈ ocnet_sos S"

  | ocnet_newpkt:
    "⟦ ((σ, s), {i}¬K:arrive(newpkt(d, di)), (σ', s')) ∈ S; ∀j. j ∉ net_ips s ⟶ (σ' j = σ j) ⟧
     ⟹ ((σ, s), i:newpkt(d, di), (σ', s')) ∈ ocnet_sos S"

inductive_cases oconnect_completeTE: "((σ, s), connect(i, i'), (σ', s')) ∈ ocnet_sos S"
            and odisconnect_completeTE: "((σ, s), disconnect(i, i'), (σ', s')) ∈ ocnet_sos S"
            and otau_completeTE: "((σ, s), τ, (σ', s')) ∈ ocnet_sos S"
            and odeliver_completeTE: "((σ, s), i:deliver(d), (σ', s')) ∈ ocnet_sos S"
            and onewpkt_completeTE: "((σ, s), i:newpkt(d, di), (σ', s')) ∈ ocnet_sos S"

lemmas ocompleteTEs = oconnect_completeTE
                      odisconnect_completeTE
                      otau_completeTE
                      odeliver_completeTE
                      onewpkt_completeTE

lemma ocomplete_no_cast [simp]:
  "((σ, s), R:*cast(m), (σ', s')) ∉ ocnet_sos T"
  ⟨proof⟩

lemma ocomplete_no_arrive [simp]:
  "((σ, s), ii¬ni:arrive(m), (σ', s')) ∉ ocnet_sos T"
  ⟨proof⟩

lemma ocomplete_no_change [elim]:
  assumes "((σ, s), a, (σ', s')) ∈ ocnet_sos T"
      and "j ∉ net_ips s"
    shows "σ' j = σ j"
  ⟨proof⟩

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)"
  ⟨proof⟩

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"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

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

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

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

lemma *oelimder_receive:*

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

lemmas *oelimders =*
oelimder_guard
oelimder_assign
oelimder_ucast
oelimder_bcast
oelimder_gcast
oelimder_send
oelimder_deliver
oelimder_receive

declare
oseqpTEs [cterms_seqte]
oelimders [cterms_elimders]

end

15 Lemmas for open partial networks

theory *OPnet*
imports *OAWN_SOS OInvariants*
begin

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

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

lemma *opnet_net_ips_net_tree_ips:*
assumes *"(σ, ns) ∈ oreachable (opnet np p) S U"*
shows *"net_ips ns = net_tree_ips p"*
⟨*proof*⟩

lemma *opnet_net_ips_net_tree_ips_init:*
assumes *"(σ, ns) ∈ init (opnet np p)"*
shows *"net_ips ns = net_tree_ips p"*
⟨*proof*⟩

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'"*
⟨*proof*⟩

lemma *net_par_oreachable_is_subnet:*
assumes *"(σ, st) ∈ oreachable (opnet np (p1 ∥ p2)) S U"*
shows *"∃s t. st = SubnetS s t"*
⟨*proof*⟩

end

16 Lifting rules for (open) nodes

theory *ONode_Lifting*
imports *AWN OAWN_SOS OInvariants*
begin

lemma *node_net_state':*
assumes "s ∈ oreachable (⟨i : T : R_i⟩$_o$) S U"
shows "∃σ ζ R. s = (σ, NodeS i ζ R)"
⟨*proof*⟩

lemma *node_net_state:*
assumes "(σ, s) ∈ oreachable (⟨i : T : R_i⟩$_o$) S U"
shows "∃ζ R. s = NodeS i ζ R"
⟨*proof*⟩

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*⟩

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)"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *node_addressD [dest, simp]:*
assumes "(σ, NodeS i ζ R) ∈ oreachable (⟨ii : T : R_i⟩$_o$) S U"
shows "i = ii"
⟨*proof*⟩

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*⟩

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})"
⟨*proof*⟩

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 : Rᵢ⟩ₒ ⊨ (otherwith S {ii} (oarrivemsg I), other U {ii} →) global P"*
⟨*proof*⟩

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 : Rᵢ⟩ₒ ⊨ₐ (otherwith S {i} (oarrivemsg I), other U {i} →)*
                globala (λ(σ, _, σ'). Q σ σ')"
  (**is** *"_ ⊨ₐ (?S, ?U →) _"*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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

lemma *node_tau_deliver_unchanged:*
  "⟨i : ζᵢ : Rᵢ⟩ₒ ⊨ₐ (S, U →) globala (λ(σ, a, σ'). a = τ ∨ (∃i d. a = i:deliver(d))
                                                ⟶ (∀j. j≠i ⟶ σ' j = σ j))"
⟨*proof*⟩

end

17 Lifting rules for (open) partial networks

theory *OPnet_Lifting*
imports *ONode_Lifting OAWN_SOS OPnet*
begin

lemma *oreachable_par_subnet_induct [consumes, case_names init other local]:*
  assumes *"(σ, SubnetS s t) ∈ oreachable (opnet onp (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"
⟨proof⟩

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⟩

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⟩

lemma ostep_arrive_invariantD [elim]:
  assumes "p ⊨A (λσ _. oarrivemsg I σ, U →) P"
      and "(σ, s) ∈ oreachable p (otherwith S IPS (oarrivemsg I)) U"
      and "((σ, s), a, (σ', s')) ∈ trans p"
      and "oarrivemsg I σ a"
  shows "P ((σ, s), a, (σ', s'))"
⟨proof⟩

lemma opnet_sync_action_subnet_oreachable:
  assumes "(σ, SubnetS s t) ∈ oreachable (opnet onp (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 = {}"
⟨proof⟩
```

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

```
lemma subnet_oreachable_true_true:
  assumes "(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")
⟨proof⟩
```

It may also be tempting to try splitting from the assumption `(σ, SubnetS s₁ s₂) ∈ oreachable (opnet onp (p₁ ∥ p₂)) (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⟩

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 = {}"
  ⟨proof⟩

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⟩
```

end

18 Lifting rules for (open) closed networks

theory *OClosed_Lifting*
imports *OPnet_Lifting*
begin

lemma *trans_fst_oclosed_fst1 [dest]:*
"(s, connect(i, i'), s') ∈ ocnet_sos (trans p) ⟹ (s, connect(i, i'), s') ∈ trans p"
⟨*proof*⟩

lemma *trans_fst_oclosed_fst2 [dest]:*
"(s, disconnect(i, i'), s') ∈ ocnet_sos (trans p) ⟹ (s, disconnect(i, i'), s') ∈ trans p"
⟨*proof*⟩

lemma *trans_fst_oclosed_fst3 [dest]:*
"(s, i:deliver(d), s') ∈ ocnet_sos (trans p) ⟹ (s, i:deliver(d), s') ∈ trans p"
⟨*proof*⟩

lemma *oclosed_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 _"*)
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

end

19 Generic invariants on sequential AWN processes

theory *AWN_Invariants*
imports *Invariants AWN_SOS AWN_Labels*
begin

19.1 Invariants via labelled control terms

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

definition
control_within :: "('s, 'm, 'p, 'l) seqp_env ⇒ ('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"*
⟨*proof*⟩

lemma *control_withinD [dest]:*
assumes *"control_within Γ σ"*
and *"(ξ, p) ∈ σ"*

shows "∃pn. p ∈ subterms (Γ pn)"
⟨*proof*⟩

lemma *control_within_topI [intro]:*
assumes "⋀p. p ∈ Range σ ⟹ ∃pn. p = Γ pn"
shows "control_within Γ σ"
⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

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

lemma *onlD [dest]:*
assumes "onl Γ P (ξ, p)"
shows "∀l∈labels Γ p. P (ξ, l)"
⟨*proof*⟩

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*⟩

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)"
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *seq_invariant_ctermI:*
assumes *wf: "wellformed Γ"*
and *cw: "control_within Γ (init A)"*
and *sl: "simple_labels Γ"*
and *sp: "trans A = seqp_sos Γ"*
and *init: "⋀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*⟩

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"*
⟨*proof*⟩

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'))"*
⟨*proof*⟩

lemma *onllIl [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'))"*
⟨*proof*⟩

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'))"*
⟨*proof*⟩

lemma *onl_weaken [elim!]: "⋀Γ P Q s. ⟦ onl Γ P s; ⋀s. P s ⟹ Q s ⟧ ⟹ onl Γ Q s"*
⟨*proof*⟩

lemma *onll_weaken [elim!]: "⋀Γ P Q s. ⟦ onll Γ P s; ⋀s. P s ⟹ Q s ⟧ ⟹ onll Γ Q s"*
⟨*proof*⟩

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'))"
⟨*proof*⟩

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*⟩

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*⟩

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'))"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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'))"*
⟨*proof*⟩

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'))"*
⟨*proof*⟩

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⟩

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"
⟨proof⟩

end
```

20 Generic open invariants on sequential AWN processes

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

20.1 Open invariants via labelled control terms

```
lemma oseqp_sos_subterms:
  assumes "wellformed Γ"
    and "∃pn. p ∈ subterms (Γ pn)"
    and "((σ, p), a, (σ', p')) ∈ oseqp_sos Γ i"
  shows "∃pn. p' ∈ subterms (Γ pn)"
⟨proof⟩

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)"
⟨proof⟩

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*⟩

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*⟩

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)"
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *oseq_invariant_ctermI:*
assumes *wf: "wellformed Γ"*
and *cw: "control_within Γ (init A)"*
and *sl: "simple_labels Γ"*
and *sp: "trans A = oseqp_sos Γ i"*
and *init: "⋀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*⟩

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*⟩

20.2 Open step invariants via labelled control terms

lemma *onll_ostep_invariantI [intro]:*
assumes *∗: "⋀σ p l a σ' p' l'. ⟦ (σ, p)∈oreachable A S U; ((σ, p), a, (σ', p')) ∈ trans A; S σ σ' a; l ∈labels Γ p; l'∈labels Γ p' ⟧ ⟹ P ((σ, l), a, (σ', l'))"*
shows *"A ⊨A (S, U →) onll Γ P"*
⟨*proof*⟩

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

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

lemma *onll_ostep_invariant_weakenD [dest]:*
assumes "A $\models_A$ (S', U' $\rightarrow$) onll Γ P"
and "(σ, p) $\in$ oreachable A S U"
and "((σ, p), a, (σ', p')) $\in$ trans A"
and "S' σ σ' a"
and weakenS: "$\bigwedge$s s' a. S s s' a $\Longrightarrow$ S' s s' a"
and weakenU: "$\bigwedge$s s'. U s s' $\Longrightarrow$ U' s s'"
shows "$\forall$l$\in$labels Γ p. $\forall$l'$\in$labels Γ p'. P ((σ, l), a, (σ', l'))"
⟨*proof*⟩

lemma *onll_ostep_to_invariantI [intro]:*
assumes sinv: "A $\models_A$ (S, U $\rightarrow$) onll Γ Q"
and wf: "wellformed Γ"
and init: "$\bigwedge\sigma$ l p. ⟦ (σ, p) $\in$ init A; l$\in$labels Γ p ⟧ $\Longrightarrow$ P (σ, l)"
and other: "$\bigwedge\sigma$ σ' p l.
⟦ (σ, p) $\in$ oreachable A S U;
l$\in$labels Γ p;
P (σ, l);
U σ σ' ⟧ $\Longrightarrow$ P (σ', l)"
and local: "$\bigwedge\sigma$ p l σ' l' a.
⟦ (σ, p) $\in$ oreachable A S U;
l$\in$labels Γ p;
P (σ, l);
Q ((σ, l), a, (σ', l'));
S σ σ' a⟧ $\Longrightarrow$ P (σ', l')"
shows "A $\models$ (S, U $\rightarrow$) onl Γ P"
⟨*proof*⟩

lemma *onll_ostep_invariant_sterms:*
assumes wf: "wellformed Γ"
and si: "A $\models_A$ (S, U $\rightarrow$) onll Γ P"
and sr: "(σ, p) $\in$ oreachable A S U"
and sos: "((σ, p), a, (σ', q)) $\in$ trans A"
and "S σ σ' a"
and "l'$\in$labels Γ q"
and "p'$\in$sterms Γ p"
and "l$\in$labels Γ p'"
shows "P ((σ, l), a, (σ', l'))"
⟨*proof*⟩

lemma *oseq_step_invariant_sterms:*
assumes inv: "A $\models_A$ (S, U $\rightarrow$) onll Γ P"
and wf: "wellformed Γ"
and sp: "trans A = oseqp_sos Γ i"
and "l'$\in$labels Γ q"
and sr: "(σ, p) $\in$ oreachable A S U"
and tr: "((σ, p'), a, (σ', q)) $\in$ trans A"
and "S σ σ' a"
and "p'$\in$sterms Γ p"
shows "$\forall$l$\in$labels Γ p'. P ((σ, l), a, (σ', l'))"
⟨*proof*⟩

lemma *oseq_step_invariant_sterms_weaken:*
assumes inv: "A $\models_A$ (S, U $\rightarrow$) onll Γ P"
and wf: "wellformed Γ"
and sp: "trans A = oseqp_sos Γ i"
and "l'$\in$labels Γ q"
and sr: "(σ, p) $\in$ oreachable A S' U'"
and tr: "((σ, p'), a, (σ', q)) $\in$ trans A"
and "S' σ σ' a"
and "p'$\in$sterms Γ p"
and weakenS: "$\bigwedge\sigma$ σ' a. S' σ σ' a $\Longrightarrow$ S σ σ' a"
and weakenU: "$\bigwedge\sigma$ σ'. U' σ σ' $\Longrightarrow$ U σ σ'"

shows "∀l∈labels Γ p'. P ((σ, l), a, (σ', l'))"
⟨*proof*⟩

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'))"
⟨*proof*⟩

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*⟩

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"
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *open_from_initialiD [dest]:*
assumes *"initiali i OI CI"*
and *"(σ, p) ∈ OI"*
shows *"∃ξ. σ i = ξ ∧ (ξ, p) ∈ CI"*
⟨*proof*⟩

lemma *closed_from_initialiD [dest]:*
assumes *"initiali i OI CI"*
and *"(ξ, p) ∈ CI"*
shows *"∃σ. σ i = ξ ∧ (σ, p) ∈ OI"*
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *same_seql [elim]:*
assumes *"∀j∈{i}. σ' j = σ j"*
and *"seql i P (σ', s)"*
shows *"seql i P (σ, s)"*
⟨*proof*⟩

lemma *seqlsimp:*
"seql i P (σ, p) = P (σ i, p)"
⟨*proof*⟩

lemma *other_steps_resp_local [intro!, simp]: "other_steps (other A I) I"*
⟨*proof*⟩

lemma *seql_onl_swap:*
"seql i (onl Γ P) = onl Γ (seql i P)"
⟨*proof*⟩

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*⟩

lemma *oseqp_sos_subreachable [intro!, simp]:*
assumes *"trans OA = oseqp_sos Γ i"*
shows *"subreachable OA (other ANY {i}) {i}"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

Any invariant shown in the (simpler) closed semantics can be transferred to an invariant in the open semantics.

theorem *open_seq_invariant [intro]:*
assumes *"A ⊨ (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*⟩

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'))"*
⟨*proof*⟩

lemma *seqllI [intro!]:*
assumes *"P ((σ i, p), a, (σ' i, p'))"*
shows *"seqll i P ((σ, p), a, (σ', p'))"*
⟨*proof*⟩

lemma *seqllD [dest]:*
assumes *"seqll i P ((σ, p), a, (σ', p'))"*
shows *"P ((σ i, p), a, (σ' i, p'))"*
⟨*proof*⟩

lemma *seqllsimp:*
"seqll i P ((σ, p), a, (σ', p')) = P ((σ i, p), a, (σ' i, p'))"
⟨*proof*⟩

lemma *seqll_onll_swap:*
"seqll i (onll Γ P) = onll Γ (seqll i P)"
⟨*proof*⟩

theorem *open_seq_step_invariant [intro]:*
assumes *"A ⊨_A (I →) P"*
and *"initiali i (init OA) (init A)"*
and *spo: "trans OA = oseqp_sos Γ i"*

and *sp: "trans A = seqp_sos Γ"*
shows *"OA ⊨A (act I, other ANY {i} →) (seqll i P)"*
⟨*proof*⟩

end

22 Model the standard queuing model

theory *Qmsg*
imports *AWN_SOS_Labels AWN_Invariants*
begin

Define the queue process

fun Γ_{QMSG} *:: "('m list, 'm, unit, unit label) seqp_env"*
where
"Γ_{QMSG} *() = 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 σ_{QMSG} *:: "(('m::msg) list × ('m list, 'm, unit, unit label) seqp) set"*
where *"*σ_{QMSG} *≡ {([], Γ*$_{QMSG}$ *())}"*

abbreviation *qmsg*
:: "(('m::msg) list × ('m list, 'm, unit, unit label) seqp, 'm seq_action) automaton"
where
"qmsg ≡ (| init = σ_{QMSG}*, trans = seqp_sos* Γ_{QMSG} *|)"*

declare Γ_{QMSG}*.simps [simp del, code del]*
lemmas Γ_{QMSG}*_simps [simp, code] =* Γ_{QMSG}*.simps [simplified]*

lemma σ_{QMSG}*_not_empty [simp, intro]: "*σ_{QMSG} *≠ {}"*
⟨*proof*⟩

lemma σ_{QMSG}*_exists [simp]: "∃qmsg q. (qmsg, q) ∈* σ_{QMSG}*"*
⟨*proof*⟩

lemma *qmsg_wf [simp]: "wellformed* Γ_{QMSG}*"*
⟨*proof*⟩

lemmas *qmsg_labels_not_empty [simp] = labels_not_empty [OF qmsg_wf]*

lemma *qmsg_control_within [simp]: "control_within* Γ_{QMSG} *(init qmsg)"*
⟨*proof*⟩

lemma *qmsg_simple_labels [simp]: "simple_labels* Γ_{QMSG}*"*
⟨*proof*⟩

lemma *qmsg_trans: "trans qmsg = seqp_sos* Γ_{QMSG}*"*
⟨*proof*⟩

lemma σ_{QMSG}*_labels [simp]: "(ξ, q) ∈* σ_{QMSG} *⟹ labels* Γ_{QMSG} *q = {()-:0}"*
⟨*proof*⟩

lemma *qmsg_proc_cases [dest]:*
fixes *p pn*
shows *"p ∈ ctermsl (*Γ_{QMSG} *pn) ⟹ p ∈ ctermsl (*Γ_{QMSG} *())"*
⟨*proof*⟩

declare
Γ_{QMSG}*_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"
```
⟨*proof*⟩

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*⟩

lemma *qmsg_msgs_not_empty:*
```
  "qmsg ⊨ onl Γ_QMSG (λ(msgs, l). l = ()-:1 ⟶ msgs ≠ [])"
```
⟨*proof*⟩

lemma *qmsg_send_from_queue:*
```
  "qmsg ⊨_A (λ((msgs, q), a, _). sendmsg (λm. m∈set msgs) a)"
```
⟨*proof*⟩

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*⟩

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

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)"
```
⟨*proof*⟩

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⟩

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⟩

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⟩

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⟩

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⟩"
⟨proof⟩
```

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"
⟨*proof*⟩

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*⟩

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*⟩

lemma *transfer_disconnect:*
assumes *"(s, disconnect(i, i’), s’) ∈ trans (pnet np n)"*
and *"s ∈ reachable (pnet np n) TT"*
and *"netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"*
and *"wf_net_tree n"*
obtains *σ’ ζ’* **where** *"((σ, ζ), disconnect(i, i’), (σ’, ζ’)) ∈ trans (opnet onp n)"*
and *"∀ j. j∉net_ips ζ ⟶ σ’ j = σ j"*
and *"netgmap sr s’ = netmask (net_tree_ips n) (σ’, ζ’)"*
⟨*proof*⟩

lemma *transfer_tau:*
assumes *"(s, τ, s’) ∈ trans (pnet np n)"*
and *"s ∈ reachable (pnet np n) TT"*
and *"netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"*
and *"wf_net_tree n"*
obtains *σ’ ζ’* **where** *"((σ, ζ), τ, (σ’, ζ’)) ∈ trans (opnet onp n)"*
and *"∀ j. j∉net_ips ζ ⟶ σ’ j = σ j"*
and *"netgmap sr s’ = netmask (net_tree_ips n) (σ’, ζ’)"*
⟨*proof*⟩

lemma *transfer_deliver:*
assumes *"(s, i:deliver(d), s’) ∈ trans (pnet np n)"*
and *"s ∈ reachable (pnet np n) TT"*
and *"netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"*
and *"wf_net_tree n"*
obtains *σ’ ζ’* **where** *"((σ, ζ), i:deliver(d), (σ’, ζ’)) ∈ trans (opnet onp n)"*

```
                and "∀j. j∉net_ips ζ ⟶ σ' j = σ j"
                and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
```
⟨*proof*⟩

lemma *transfer_arrive':*
```
  assumes "(s, H¬K:arrive(m), s') ∈ trans (pnet np n)"
      and "s ∈ reachable (pnet np n) TT"
      and "netgmap sr s  = netmask (net_tree_ips n) (σ, ζ)"
      and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
      and "wf_net_tree n"
  shows "((σ, ζ), H¬K:arrive(m), (σ', ζ')) ∈ trans (opnet onp n)"
```
⟨*proof*⟩

lemma *transfer_arrive:*
```
  assumes "(s, H¬K:arrive(m), s') ∈ trans (pnet np n)"
      and "s ∈ reachable (pnet np n) TT"
      and "netgmap sr s  = netmask (net_tree_ips n) (σ, ζ)"
      and "wf_net_tree n"
  obtains σ' ζ' where "((σ, ζ), H¬K:arrive(m), (σ', ζ')) ∈ trans (opnet onp n)"
                and "∀j. j∉net_ips ζ ⟶ σ' j = σ j"
                and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
```
⟨*proof*⟩

lemma *transfer_cast:*
```
  assumes "(s, mR:*cast(m), s') ∈ trans (pnet np n)"
      and "s ∈ reachable (pnet np n) TT"
      and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
      and "wf_net_tree n"
  obtains σ' ζ' where "((σ, ζ), mR:*cast(m), (σ', ζ')) ∈ trans (opnet onp n)"
                and "∀j. j∉net_ips ζ ⟶ σ' j = σ j"
                and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
```
⟨*proof*⟩

lemma *transfer_pnet_action:*
```
  assumes "s ∈ reachable (pnet np n) TT"
      and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
      and "wf_net_tree n"
      and "(s, a, s') ∈ trans (pnet np n)"
  obtains σ' ζ' where "((σ, ζ), a, (σ', ζ')) ∈ trans (opnet onp n)"
                and "∀j. j∉net_ips ζ ⟶ σ' j = σ j"
                and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
```
⟨*proof*⟩

lemma *transfer_action_pnet_closed:*
```
  assumes "(s, a, s') ∈ trans (closed (pnet np n))"
  obtains a' where "(s, a', s') ∈ trans (pnet np n)"
              and "⋀σ ζ σ' ζ'. ⟦ ((σ, ζ), a', (σ', ζ')) ∈ trans (opnet onp n);
                                 (∀j. j∉net_ips ζ ⟶ σ' j = σ j) ⟧
                           ⟹ ((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))"
```
⟨*proof*⟩

lemma *transfer_action:*
```
  assumes "s ∈ reachable (closed (pnet np n)) TT"
      and "netgmap sr s = netmask (net_tree_ips n) (σ, ζ)"
      and "wf_net_tree n"
      and "(s, a, s') ∈ trans (closed (pnet np n))"
  obtains σ' ζ' where "((σ, ζ), a, (σ', ζ')) ∈ trans (oclosed (opnet onp n))"
                and "netgmap sr s' = netmask (net_tree_ips n) (σ', ζ')"
```
⟨*proof*⟩

lemma *pnet_reachable_transfer':*
```
  assumes "wf_net_tree n"
      and "s ∈ reachable (closed (pnet np n)) TT"
    shows "netgmap sr s ∈ netmask (net_tree_ips n) ‘ oreachable (oclosed (opnet onp n)) (λ_ _ _. True)
```

```
U"
          (is " _ ∈ ?f ‘ ?oreachable n")
```
⟨proof⟩

definition
```
  someinit :: "nat ⇒ 'g"
```
where
```
  "someinit i ≡ SOME x. x ∈ (fst o sr) ‘ init (np i)"
```

definition
```
  initmissing :: "((nat ⇒ 'g option) × 'a) ⇒ (nat ⇒ 'g) × 'a"
```
where
```
  "initmissing σ = (λi. case (fst σ) i of None ⇒ someinit i | Some s ⇒ s, snd σ)"
```

lemma *initmissing_def':*
```
  "initmissing = apfst (default someinit)"
```
⟨proof⟩

lemma *netmask_initmissing_netgmap:*
```
  "netmask (net_ips s) (initmissing (netgmap sr s)) = netgmap sr s"
```
⟨proof⟩

lemma *snd_initmissing [simp]:*
```
  "snd (initmissing x)= snd x"
```
⟨proof⟩

lemma *initmnissing_snd_netgmap [simp]:*
```
  assumes "initmissing (netgmap sr s) = (σ, ζ)"
    shows "snd (netgmap sr s) = ζ"
```
⟨proof⟩

lemma *in_net_ips_fst_init_missing [simp]:*
```
  assumes "i ∈ net_ips s"
    shows "fst (initmissing (netgmap sr s)) i = the (fst (netgmap sr s) i)"
```
⟨proof⟩

lemma *not_in_net_ips_fst_init_missing [simp]:*
```
  assumes "i ∉ net_ips s"
    shows "fst (initmissing (netgmap sr s)) i = someinit i"
```
⟨proof⟩

lemma *initmissing_oreachable_netmask [elim]:*
```
  assumes "initmissing (netgmap sr s) ∈ oreachable (oclosed (opnet onp n)) (λ_ _ _. True) U"
          (is "_ ∈ ?oreachable n")
     and "net_ips s = net_tree_ips n"
    shows "netgmap sr s ∈ netmask (net_tree_ips n) ‘ ?oreachable n"
```
⟨proof⟩

lemma *pnet_reachable_transfer:*
```
  assumes "wf_net_tree n"
     and "s ∈ reachable (closed (pnet np n)) TT"
    shows "initmissing (netgmap sr s) ∈ oreachable (oclosed (opnet onp n)) (λ_ _ _. True) U"
          (is " _ ∈ ?oreachable n")
```
⟨proof⟩

definition
```
  netglobal :: "((nat ⇒ 'g) ⇒ bool) ⇒ 's net_state ⇒ bool"
```
where
```
  "netglobal P ≡ (λs. P (fst (initmissing (netgmap sr s))))"
```

lemma *netglobalsimp [simp]:*
```
  "netglobal P s = P (fst (initmissing (netgmap sr s)))"
```
⟨proof⟩

lemma *netglobalE [elim]:*
  assumes *"netglobal P s"*
      and *"⋀σ. ⟦ P σ; fst (initmissing (netgmap sr s)) = σ ⟧ ⟹ Q σ"*
    shows *"netglobal Q s"*
  ⟨*proof*⟩

lemma *netglobal_weakenE [elim]:*
  assumes *"p ⊨ netglobal P"*
      and *"⋀σ. P σ ⟹ Q σ"*
    shows *"p ⊨ netglobal Q"*
  ⟨*proof*⟩

lemma *close_opnet:*
  assumes *"wf_net_tree n"*
      and *"oclosed (opnet onp n) ⊨ (λ_ _ _. True, U →) global P"*
    shows *"closed (pnet np n) ⊨ netglobal P"*
  ⟨*proof*⟩

end

locale *openproc_parq =*
  op?: openproc np onp sr **for** *np :: "ip ⇒ ('s, ('m::msg) seq_action) automaton"* **and** *onp sr*
  + **fixes** *qp :: "('t, 'm seq_action) automaton"*
    assumes *init_qp_notempty: "init qp ≠ {}"*

sublocale *openproc_parq ⊆ openproc "λi. np i ⟨⟨ qp"*
                                  "λi. onp i ⟨⟨$_i$ qp"
                                  "λ(p, q). (fst (sr p), (snd (sr p), q))"
  ⟨*proof*⟩

end

25 Import all AWN-related theories

theory *AWN_Main*
imports *AWN_SOS AWN_SOS_Labels OAWN_SOS_Labels AWN_Invariants*
        OAWN_Convert OClosed_Transfer
begin

end

26 Simple toy example

theory *Toy*
imports *Main AWN_Main Qmsg_Lifting*
begin

26.1 Messages used in the protocol

datatype *msg =*
    Pkt data ip
  | Newpkt data ip

instantiation *msg :: msg*
begin
definition *newpkt_def [simp]: "newpkt ≡ λ(d,did). Newpkt d did"*
definition *eq_newpkt_def: "eq_newpkt m ≡ case m of Newpkt d did ⇒ True | _ ⇒ False"*

instance ⟨*proof*⟩
end

definition *pkt :: "nat × nat ⇒ msg"*

where *"pkt ≡ λ(no, sid). Pkt no sid"*

lemma *pkt_simp [simp]:*
"pkt(no, sid) = Pkt no sid"
⟨*proof*⟩

lemma *not_eq_newpkt_pkt [simp]: "¬eq_newpkt (Pkt no sid)"*
⟨*proof*⟩

26.2 Protocol model

record *state =*
id :: "nat"
no :: "nat"
nhid :: "nat"

msg :: "msg"
num :: "nat"
sid :: "nat"

abbreviation *toy_init :: "ip ⇒ 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)"*
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

```
lemma is_msg_inv_sid [simp]:
  "ξ' ∈ is_newpkt ξ ⟹ sid ξ' = sid ξ"
  ⟨proof⟩

lemma is_msg_inv_no [simp]:
  "ξ' ∈ is_pkt ξ    ⟹ no ξ' = no ξ"
  "ξ' ∈ is_newpkt ξ ⟹ no ξ' = no ξ"
  ⟨proof⟩

lemma is_msg_inv_nhid [simp]:
  "ξ' ∈ is_pkt ξ    ⟹ nhid ξ' = nhid ξ"
  "ξ' ∈ is_newpkt ξ ⟹ nhid ξ' = nhid ξ"
  ⟨proof⟩

lemma is_msg_inv_msg [simp]:
  "ξ' ∈ is_pkt ξ    ⟹ msg ξ' = msg ξ"
  "ξ' ∈ is_newpkt ξ ⟹ msg ξ' = msg ξ"
  ⟨proof⟩

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 ⟨proof⟩
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*⟩

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)"*
⟨*proof*⟩

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}*"*
⟨*proof*⟩

lemma *toy_control_within [simp]: "control_within* Γ_{TOY} *(init (ptoy i))"*
⟨*proof*⟩

lemma *toy_wf [simp]:*
"wellformed Γ_{TOY}*"*
⟨*proof*⟩

lemmas *toy_labels_not_empty [simp] = labels_not_empty [OF toy_wf]*

lemma *toy_ex_label [intro]: "∃l. l∈labels* Γ_{TOY} *p"*
⟨*proof*⟩

lemma *toy_ex_labelE [elim]:*
assumes *"∀l∈labels* Γ_{TOY} *p. P l p"*
and *"∃p l. P l p ⟹ Q"*
shows *"Q"*
⟨*proof*⟩

lemma *toy_simple_labels [simp]: "simple_labels* Γ_{TOY}*"*
⟨*proof*⟩

lemma σ_{TOY}*_labels [simp]: "(ξ, p) ∈* σ_{TOY} *i ⟹ labels* Γ_{TOY} *p = {PToy-:0}"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *oaodv_control_within [simp]: "control_within Γ_{TOY} (init (optoy i))"*
⟨*proof*⟩

lemma *σ_{OTOY}_labels [simp]: "(σ, p) ∈ σ_{OTOY} ⟹ labels Γ_{TOY} p = {PToy-:0}"*
⟨*proof*⟩

lemma *otoy_trans: "trans (optoy i) = oseqp_sos Γ_{TOY} i"*
⟨*proof*⟩

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"
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *msg_ok_Pkt [simp]:*
"msg_ok σ (Pkt data src) = (data ≤ no (σ src))"
⟨*proof*⟩

lemma *msg_ok_pkt [simp]:*
"msg_ok σ (pkt(data, src)) = (data ≤ no (σ src))"
⟨*proof*⟩

lemma *msg_ok_Newpkt [simp]:*
"msg_ok σ (Newpkt data dst)"
⟨*proof*⟩

lemma *msg_ok_newpkt [simp]:*
"msg_ok σ (newpkt(data, dst))"
⟨*proof*⟩

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)"
⟨proof⟩

lemma *seq_nos_incs:*
"ptoy i ⊨$_A$ onll Γ_{TOY} (λ((ξ, _), _, (ξ', _)). nos_inc ξ ξ')"
⟨proof⟩

lemma *seq_nos_incs':*
"ptoy i ⊨$_A$ (λ((ξ, _), _, (ξ', _)). nos_inc ξ ξ')"
⟨proof⟩

lemma *sender_ip_valid:*
"ptoy i ⊨$_A$ onll Γ_{TOY} (λ((ξ, _), a, _). anycast (λm. msg_sender m = id ξ) a)"
⟨proof⟩

lemma *id_constant:*
"ptoy i ⊨ onl Γ_{TOY} (λ(ξ, _). id ξ = i)"
⟨proof⟩

lemma *nhid_eq_id:*
"ptoy i ⊨ onl Γ_{TOY} (λ(ξ, l). l∈{PToy-:2..PToy-:8} ⟶ nhid ξ = id ξ)"
⟨proof⟩

lemma *seq_msg_ok:*
"ptoy i ⊨$_A$ onll Γ_{TOY} (λ((ξ, _), a, _).
anycast (λm. case m of Pkt num' sid' ⇒ num' = no ξ ∧ sid' = i | _ ⇒ True) a)"
⟨proof⟩

lemma *nhid_eq_i:*
"ptoy i ⊨ onl Γ_{TOY} (λ(ξ, l). l∈{PToy-:2..PToy-:8} ⟶ nhid ξ = i)"
⟨proof⟩

26.6 Global Invariants

lemma *nos_incD [dest]:*
assumes "nos_inc ξ ξ'"
shows "no ξ ≤ no ξ'"
⟨proof⟩

lemma *nos_inc_simp [simp]:*
"nos_inc ξ ξ' = (no ξ ≤ no ξ')"
⟨proof⟩

lemmas *oseq_nos_incs =*
open_seq_step_invariant [OF seq_nos_incs initiali_toy otoy_trans toy_trans,
simplified seqll_onll_swap]

lemmas *oseq_no_leq_num =*
open_seq_invariant [OF seq_no_leq_num initiali_toy otoy_trans toy_trans,
simplified seql_onl_swap]

lemma *all_nos_inc:*
shows "optoy i ⊨$_A$ (otherwith nos_inc {i} S,
other nos_inc {i} →)
onll Γ_{TOY} (λ((σ, _), a, (σ', _)). (∀j. nos_inc (σ j) (σ' j)))"
⟨proof⟩

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⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *anycast_weakenE [elim]:*
assumes *"anycast P a"*
and *"⋀m. P m ⟹ Q m"*
shows *"anycast Q a"*
⟨*proof*⟩

lemma *oseq_msg_ok:*
"optoy i ⊨_A (act TT, other U {i} →) globala (λ(σ, a, _). anycast (msg_ok σ) a)"
⟨*proof*⟩

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*⟩

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))))"

⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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"`
⟨*proof*⟩

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))"
```
⟨*proof*⟩

interpretation *toy_openproc: openproc ptoy optoy Fun.id*
rewrites `"toy_openproc.initmissing = initmissing"`
⟨*proof*⟩

lemma *fst_initmissing_netgmap_default_toy_init_netlift:*
```
"fst (initmissing (netgmap sr s)) = default toy_init (netlift sr s)"
```
⟨*proof*⟩

definition
```
netglobal :: "((nat ⇒ 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*⟩

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*⟩

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.