Refinement-based Specification and Security Analysis of Separation Kernels

Yongwang Zhao
School of Computer Science and Engineering, Nanyang Technological University, Singapore
School of Computer Science and Engineering, Beihang University, China
zhaoyongwang@gmail.com, zhaoyw@buaa.edu.cn

December 20, 2016

Contents

1 Security Model of Separation Kernels

```
theory SK-SecurityModel
imports Main
begin
```

1.1 Security State Machine

```
locale SM =
  fixes s0 :: 's
  fixes step :: 'e ⇒ ('s × 's) set
  fixes domain :: 's ⇒ 'e ⇒ ('d option)
  fixes sched :: 'd
  fixes vpeq :: 's ⇒ 'd ⇒ 's ⇒ bool  ((- ∼ - ∼ -))
  fixes interference :: 'd ⇒ 'd ⇒ bool ((- ⇝ -))
  assumes
    vpeq-transitive-lemma : ∀ s t r d. (s ∼ d ∼ t) ∧ (t ∼ d ∼ r) ⟶ (s ∼ d ∼ r) and
    vpeq-symmetric-lemma : ∀ s t d. (s ∼ d ∼ t) ⟶ (t ∼ d ∼ s) and
    vpeq-reflexive-lemma : ∀ s d. (s ∼ d ∼ s) and
    sched-vpeq : ∀ s t a. (s ∼ sched ∼ t) ⟶ (domain s a) = (domain t a) and
    sched-intf-all : ∀ d. (sched ⇝ d) and
    no-intf-sched : ∀ d. (d ⇝ sched) ⟶ d = sched and
    interf-reflexive: ∀ d. (d ⇝ d)
begin

  definition non-interference :: 'd ⇒ 'd ⇒ bool ((- \⇝ -))
    where (u \⇝  v) ≡ ¬ (u ⇝ v)

  definition ivpeq :: 's ⇒ 'd set ⇒ 's ⇒ bool ((- ≈ - ≈ -))
  where ivpeq s D t ≡ ∀ d ∈ D. (s ∼ d ∼ t)

  primrec run :: 'e list ⇒ ('s × 's) set
    where run-Nil: run [] = Id |
          run-Cons: run (a#as) = step a O run as

  definition next-states :: 's ⇒ 'e ⇒ 's set
    where next-states s a ≡ {Q. (s,Q)∈step a}

  definition execute  :: 'e list ⇒ 's ⇒ 's set
    where execute as s = Image (run as) {s}
```

```
definition reachable :: 's ⇒ 's ⇒ bool ((- ↪ -) [70,71] 60) where
  reachable s1 s2 ≡  (∃ as. (s1,s2) ∈ run as)

definition reachable0 :: 's ⇒ bool  where
  reachable0 s ≡ reachable s0 s

declare non-interference-def[cong] and ivpeq-def[cong] and next-states-def[cong]
        execute-def[cong] and reachable-def[cong] and reachable0-def[cong] and run.simps(1)[cong] and
        run.simps(2)[cong]

lemma reachable-s0 : reachable0 s0
  by (metis SM.reachable-def SM-axioms pair-in-Id-conv reachable0-def run.simps(1))

lemma reachable-self : reachable s s
  using reachable-def run.simps(1) by fastforce

lemma reachable-step : (s,s')∈step a ⟹ reachable s s'
  proof−
    assume a0: (s,s')∈step a
    then have (s,s')∈run [a] by auto
    then show ?thesis using reachable-def by blast
  qed

lemma run-trans : ∀ C T V as bs. (C,T)∈run as ∧ (T,V)∈run bs ⟶ (C,V)∈run (as@bs)
  proof −
  {
    fix T V as bs
    have ∀ C. (C,T)∈run as ∧ (T,V)∈run bs ⟶ (C,V)∈run (as@bs)
      proof(induct as)
        case Nil show ?case by simp
      next
        case (Cons c cs)
        assume a0: ∀ C. (C, T) ∈ run cs ∧ (T, V) ∈ run bs ⟶ (C, V) ∈ run (cs @ bs)
        show ?case
          proof−
          {
            fix C
            have (C, T) ∈ run (c # cs) ∧ (T, V) ∈ run bs ⟶ (C, V) ∈ run ((c # cs) @ bs)
              proof
                assume b0: (C, T) ∈ run (c # cs) ∧ (T, V) ∈ run bs
                from b0 obtain C' where b2: (C,C')∈step c ∧ (C',T)∈run cs by auto
                with a0 b0 have (C',V)∈run (cs@bs) by blast
                with b2 show (C, V) ∈ run ((c # cs) @ bs)
```

```
          using append-Cons relcomp.relcompI run-Cons by auto
        qed
      }
      then show ?thesis by auto
      qed
    qed
  }
  then show ?thesis by auto
  qed

lemma reachable-trans : ⟦reachable C T; reachable T V⟧ ⟹ reachable C V
  proof−
    assume a0: reachable C T
    assume a1: reachable T V
    from a0 have C = T ∨ (∃ as. (C,T)∈run as) by simp
    then show ?thesis
      proof
        assume b0: C = T
        show ?thesis
          proof −
            from a1 have T = V ∨ (∃ as. (T,V)∈run as) by simp
            then show ?thesis
              proof
                assume c0: T=V
                with a0 show ?thesis by auto
              next
                assume c0: (∃ as. (T,V)∈run as)
                then show ?thesis using a1 b0 by auto
              qed
          qed
      next
        assume b0: ∃ as. (C,T)∈run as
        show ?thesis
          proof −
            from a1 have T = V ∨ (∃ as. (T,V)∈run as) by simp
            then show ?thesis
              proof
                assume c0: T=V
                then show ?thesis using a0 by auto
              next
                assume c0: (∃ as. (T,V)∈run as)
                from b0 obtain as where d0: (C,T)∈run as by auto
                from c0 obtain bs where d1: (T,V)∈run bs by auto
```

```
          then show ?thesis using d0  run-trans by fastforce
        qed
      qed
    qed
  qed

lemma reachableStep : ⟦reachable0 C; (C,C')∈step a⟧ ⟹ reachable0 C'
  proof −
    assume a0: reachable0 C
    assume a1: (C,C')∈step a
    from a0 have (C = s0) ∨ (∃ as. (s0,C) ∈ run as)  by auto
    then show reachable0 C'
      proof
        assume b0: C = s0
        show reachable0 C'
          using a1 b0  reachable-step by auto
      next
        assume b0: ∃ as. (s0,C) ∈ run as
        show reachable0 C'
          using a0 a1 reachable-step reachable0-def reachable-trans by blast
      qed
  qed

lemma reachable0-reach : ⟦reachable0 C; reachable C C'⟧ ⟹ reachable0 C'
  using  reachable-trans by fastforce

declare reachable-def[cong del] and reachable0-def[cong del]

end
```

1.2 Information flow security properties

```
locale SM-enabled = SM s0 step domain sched vpeq interference
  for s0 :: 's and
      step :: 'e ⇒ ('s × 's) set and
      domain :: 's ⇒ 'e ⇒ ('d option) and
      sched :: 'd and
      vpeq :: 's ⇒ 'd ⇒ 's ⇒ bool  ((- ∼ - ∼ -)) and
      interference :: 'd ⇒ 'd ⇒ bool ((- ⇝ -))
  +
  assumes enabled0: ∀ s a. reachable0 s ⟶ (∃ s'. (s,s') ∈ step a)
begin
  lemma enabled : reachable0 s ⟹ (∃ s'. (s,s') ∈ step a)
    using enabled0 by simp
```

```
lemma enabled-ex: ∀ s es. reachable0 s ⟶ (∃ s′. s′ ∈ execute es s)
  proof −
  {
    fix es
    have ∀ s. reachable0 s ⟶ (∃ s′. s′ ∈ execute es s)
      proof(induct es)
        case Nil show ?case by auto
      next
        case (Cons a as)
        assume a0: ∀ s. reachable0 s ⟶ (∃ s′. s′ ∈ execute as s)
        show ?case
          proof−
          {
            fix s
            assume b0: reachable0 s
            have b1: ∃ s1. (s,s1)∈step a using enabled b0 by simp
            then obtain s1 where b2: (s,s1)∈step a by auto
            with a0 b0 have b3: ∃ s′. s′ ∈ execute as s1
              using reachableStep by blast
            then obtain s2 where b4: s2∈execute as s1 by auto
∞           then have s2 ∈ execute (a # as) s
              using Image-singleton-iff SM-axioms b2 relcomp.simps run-Cons by fastforce
            then have ∃ s′. s′ ∈ execute (a # as) s by auto
          }
          then show ?thesis by auto
          qed
      qed
  }
  then show ?thesis by auto
  qed

lemma enabled-ex2: reachable0 s ⟹ (∃ s′. s′ ∈ execute es s)
  using enabled-ex by auto

primrec sources :: ′e list ⇒ ′s ⇒ ′d ⇒ ′d set where
  sources-Nil:sources [] s d = {d} |
  sources-Cons:sources (a # as) s d = (⋃{sources as s′ d| s′. (s,s′) ∈ step a}) ∪
                         {w . w = the (domain s a) ∧ (∃ v s′. (w ⇝ v) ∧ (s,s′)∈step a ∧ v∈sources as s′ d)}
declare sources-Nil [simp del]
declare sources-Cons [simp del]

primrec ipurge :: ′e list ⇒ ′d ⇒ ′s set ⇒ ′e list where
```

```
ipurge-Nil:   ipurge [] u ss = [] |
ipurge-Cons:  ipurge (a#as) u ss = (if ∃ s∈ss. the (domain s a) ∈ (sources (a#as) s u) then
                                        a # ipurge as u (⋃ s∈ss. {s'. (s,s') ∈ step a})
                                      else
                                        ipurge as u ss
                                     )

definition observ-equivalence :: 's ⇒ 'e list ⇒ 's ⇒
      'e list ⇒ 'd ⇒ bool ((- ◁ - ≅ - ◁ - @ -))
  where observ-equivalence s as t bs d ≡
          ∀ s' t'. ((s,s')∈ run as ∧ (t,t')∈ run bs) ⟶(s' ∼ d ∼ t')

declare observ-equivalence-def[cong]

lemma observ-equiv-sym:
  (s ◁ as ≅ t ◁ bs @ d) ⟹ (t ◁ bs ≅ s ◁ as @ d)
  using observ-equivalence-def vpeq-symmetric-lemma by blast

lemma observ-equiv-trans:
  ⟦reachable0 t; (s ◁ as ≅ t ◁ bs @ d); (t ◁ bs ≅ x ◁ cs @ d)⟧ ⟹ (s ◁ as ≅ x ◁ cs @ d)
  apply clarsimp
  apply(cut-tac s=t and es=bs in enabled-ex2)
  apply simp
  apply auto[1]
  by (metis (no-types, hide-lams) vpeq-transitive-lemma)


lemma exec-equiv-leftI:
⟦reachable0 C; ∀ C'. (C,C')∈step a ⟶ (C' ◁ as ≅ D ◁ bs @ d)⟧ ⟹ (C ◁ (a # as) ≅ D ◁ bs @ d)
  proof −
    assume a0: reachable0 C
    assume a1: ∀ C'. (C,C')∈step a ⟶ (C' ◁ as ≅ D ◁ bs @ d)
    have ∀ S' T'. ((C,S')∈ run (a#as) ∧ (D,T')∈ run bs) ⟶(S' ∼ d ∼ T')
      proof −
      {
        fix S' T'
        assume b0: (C, S') ∈ run (a # as) ∧ (D, T') ∈ run bs
        then obtain C' where b1: (C,C')∈step a ∧ (C',S')∈run as
          using relcompEpair run-Cons by auto
        with a1 have b2: (C' ◁ as ≅ D ◁ bs @ d) by auto
        with b0 b1 have S' ∼ d ∼ T' by auto
      }
      then show ?thesis by auto
```

qed
then show *?thesis* **by** *fastforce*
qed

lemma *exec-equiv-both*:
$[\![reachable0\ C1;\ reachable0\ C2;\ \forall\ C1'\ C2'.\ (C1,C1') \in step\ a\ \wedge (C2,C2') \in step\ b \longrightarrow (C1' \lhd as \cong C2' \lhd bs\ @\ u)]\!]$
$\Longrightarrow (C1 \lhd (a\ \#\ as) \cong C2 \lhd (b\ \#\ bs)\ @\ u)$
proof −
assume *a0*: $reachable0\ C1$
assume *a1*: $reachable0\ C2$
assume *a2*: $\forall\ C1'\ C2'.\ (C1,C1') \in step\ a\ \wedge (C2,C2') \in step\ b \longrightarrow (C1' \lhd as \cong C2' \lhd bs\ @\ u)$
then have $\forall\, S'\ T'.\ ((C1,S') \in run\ (a\ \#\ as) \wedge (C2,T') \in run\ (b\ \#\ bs)) \longrightarrow (S' \sim u \sim T')$
using *relcompEpair run-Cons* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *sources-refl*:$reachable0\ s \Longrightarrow u \in sources\ as\ s\ u$
apply(*induct as arbitrary*: *s*)
apply(*simp add*: *sources-Nil*)
apply(*simp add*: *sources-Cons*)
using *enabled reachableStep*
by *metis*

lemma *scheduler-in-sources-Cons*:
$reachable0\ s \Longrightarrow the\ (domain\ s\ a) = sched \Longrightarrow the\ (domain\ s\ a) \in sources\ (a\#as)\ s\ u$
apply(*unfold sources-Cons*)
apply(*erule ssubst*)
apply(*rule UnI2*)
apply(*clarsimp*)
apply(*rule-tac x=u* **in** *exI*)
apply(*safe*)
apply (*simp add*: *sched-intf-all*)
using *enabled reachableStep sources-refl*
by *blast*

definition *noninterference-r* :: *bool*
where $noninterference\text{-}r \equiv \forall\, d\ as\ s.\ reachable0\ s \longrightarrow (s \lhd as \cong s \lhd (ipurge\ as\ d\ \{s\})\ @\ d)$

definition *noninterference* :: *bool*
where $noninterference \equiv \forall\, d\ as.\ (s0 \lhd as \cong s0 \lhd (ipurge\ as\ d\ \{s0\})\ @\ d)$

definition *weak-noninterference* :: *bool*

where $\textit{weak-noninterference} \equiv \forall\, d\ as\ bs.\ ipurge\ as\ d\ \{s0\} = ipurge\ bs\ d\ \{s0\}$
$\longrightarrow (s0 \lhd as \cong s0 \lhd bs\ @\ d)$

definition *weak-noninterference-r* :: *bool*
where $\textit{weak-noninterference-r} \equiv \forall\, d\ as\ bs\ s.\ reachable0\ s \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{s\}$
$\longrightarrow (s \lhd as \cong s \lhd bs\ @\ d)$

definition *noninfluence*::*bool*
where $\textit{noninfluence} \equiv \forall\ d\ as\ s\ t\ .\ reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge (s \sim sched \sim t) \longrightarrow (s \lhd as \cong t \lhd (ipurge\ as\ d\ \{t\})\ @\ d)$

definition *noninfluence-gen*::*bool*
where $\textit{noninfluence-gen} \equiv \forall\ d\ as\ s\ ts\ .\ reachable0\ s \wedge (\forall\, t{\in}ts.\ reachable0\ t)$
$\wedge (\forall\, t{\in}ts.\ (s \approx (sources\ as\ s\ d) \approx t))$
$\wedge (\forall\, t{\in}ts.\ (s \sim sched \sim t))$
$\longrightarrow (\forall\, t{\in}ts.\ (s \lhd as \cong t \lhd (ipurge\ as\ d\ ts)\ @\ d))$

definition *weak-noninfluence* ::*bool*
where $\textit{weak-noninfluence} \equiv \forall\ d\ as\ bs\ s\ t\ .\ reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge (s \sim sched \sim t) \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{s\}$
$\longrightarrow (s \lhd as \cong t \lhd bs\ @\ d)$

definition *weak-noninfluence2* ::*bool*
where $\textit{weak-noninfluence2} \equiv \forall\ d\ as\ bs\ s\ t\ .\ reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge (s \sim sched \sim t) \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{t\}$
$\longrightarrow (s \lhd as \cong t \lhd bs\ @\ d)$

definition *nonleakage* :: *bool*
where $\textit{nonleakage} \equiv \forall\, d\ as\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim sched \sim t)$
$\wedge (s \approx (sources\ as\ s\ d) \approx t) \longrightarrow (s \lhd as \cong t \lhd as\ @\ d)$
declare *noninterference-r-def*[*cong*] **and** *noninterference-def*[*cong*] **and** *weak-noninterference-def*[*cong*] **and** *weak-noninterference-r-def*[*cong*] **and** *noninfluence-def*[*cong*] **and** *noninfluence-gen-def*[*cong*] **and** *weak-noninfluence-def*[*cong*] **and** *weak-noninfluence2-def*[*cong*] **and** *nonleakage-def*[*cong*]

1.3 Unwinding conditions

definition *step-consistent* :: *bool* **where**
$\textit{step-consistent} \equiv \forall\, a\ d\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim d \sim t) \wedge (s \sim sched \sim t) \wedge$
$(((the\ (domain\ s\ a)) \rightsquigarrow d) \longrightarrow (s \sim (the\ (domain\ s\ a)) \sim t)) \longrightarrow$
$(\forall\ s'\ t'.\ (s,s') \in step\ a \wedge (t,t') \in step\ a \longrightarrow (s' \sim d \sim t'))$

definition *weak-step-consistent* :: *bool* **where**
$\textit{weak-step-consistent} \equiv \forall\, a\ d\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim d \sim t) \wedge (s \sim sched \sim t) \wedge$
$((the\ (domain\ s\ a)) \rightsquigarrow d) \wedge (s \sim (the\ (domain\ s\ a)) \sim t) \longrightarrow$

$(\forall\ s'\ t'.\ (s,s') \in \mathit{step}\ a \wedge (t,t') \in \mathit{step}\ a \longrightarrow (s' \sim d \sim t'))$

definition *step-consistent-e* :: *'e* $\Rightarrow$ *bool* **where**
step-consistent-e $a \equiv \forall\, d\ s\ t.\ \mathit{reachable0}\ s \wedge \mathit{reachable0}\ t \wedge (s \sim d \sim t) \wedge (s \sim \mathit{sched} \sim t) \wedge$
$(((\mathit{the}\ (\mathit{domain}\ s\ a)) \rightsquigarrow d) \longrightarrow (s \sim (\mathit{the}\ (\mathit{domain}\ s\ a)) \sim t)) \longrightarrow$
$(\forall\ s'\ t'.\ (s,s') \in \mathit{step}\ a \wedge (t,t') \in \mathit{step}\ a \longrightarrow (s' \sim d \sim t'))$

definition *weak-step-consistent-e* :: *'e* $\Rightarrow$ *bool* **where**
weak-step-consistent-e $a \equiv \forall\, d\ s\ t.\ \mathit{reachable0}\ s \wedge \mathit{reachable0}\ t \wedge (s \sim d \sim t) \wedge (s \sim \mathit{sched} \sim t) \wedge$
$((\mathit{the}\ (\mathit{domain}\ s\ a)) \rightsquigarrow d) \wedge (s \sim (\mathit{the}\ (\mathit{domain}\ s\ a)) \sim t) \longrightarrow$
$(\forall\ s'\ t'.\ (s,s') \in \mathit{step}\ a \wedge (t,t') \in \mathit{step}\ a \longrightarrow (s' \sim d \sim t'))$

definition *local-respect* :: *bool* **where**
local-respect $\equiv \forall\ a\ d\ s\ s'.\ \mathit{reachable0}\ s \wedge ((\mathit{the}\ (\mathit{domain}\ s\ a))\ \backslash\!\rightsquigarrow d) \wedge (s,s') \in \mathit{step}\ a \longrightarrow (s \sim d \sim s')$

definition *local-respect-e* :: *'e* $\Rightarrow$ *bool* **where**
local-respect-e $a \equiv \forall\, d\ s\ s'.\ \mathit{reachable0}\ s \wedge ((\mathit{the}\ (\mathit{domain}\ s\ a))\ \backslash\!\rightsquigarrow d) \wedge (s,s') \in \mathit{step}\ a \longrightarrow (s \sim d \sim s')$

lemma *local-respect-all-evt* : *local-respect* $=$ ($\forall\, a.$ *local-respect-e* a)
by (*simp add*: *local-respect-def local-respect-e-def*)

declare *step-consistent-def* [*cong*] **and** *weak-step-consistent-def* [*cong*] **and** *step-consistent-e-def* [*cong*] **and** *weak-step-consistent-e-def* [*cong*] **and** *local-respect-def* [*cong*] **and** *local-respect-e-def* [*cong*]

lemma *step-consistent-all-evt* : *step-consistent* $=$ ($\forall\, a.$ *step-consistent-e* a)
by *simp*

lemma *weak-step-consistent-all-evt* : *weak-step-consistent* $=$ ($\forall\, a.$ *weak-step-consistent-e* a)
by *simp*

lemma *step-cons-impl-weak* : *step-consistent* $\Longrightarrow$ *weak-step-consistent*
using *step-consistent-def weak-step-consistent-def* **by** *blast*

lemma *weak-with-step-cons*:
assumes *p1*:*weak-step-consistent*
and *p2*:*local-respect*
shows *step-consistent*
proof –
{
fix $d\ a\ s\ t\ s'\ t'$

```
    have reachable0 s ∧ reachable0 t ⟶ (s ∼ d ∼ t) ∧ (s ∼ sched ∼ t) ∧
        (((the (domain s a)) ⇝ d) ⟶ (s ∼ (the (domain s a)) ∼ t)) ⟶ (s,s′)∈step a ∧ (t,t′)∈step a
        ⟶ (s′ ∼ d ∼ t′)
      proof −
      {
        assume aa:reachable0 s ∧ reachable0 t
        assume a0:s ∼ d ∼ t
        assume a1:s ∼ sched ∼ t
        assume a2:((the (domain s a)) ⇝ d) ⟶ (s ∼ (the (domain s a)) ∼ t)
        assume a3: (s,s′)∈step a ∧ (t,t′)∈step a
        have s′ ∼ d ∼ t′
         proof(cases (the (domain s a)) ⇝ d)
           assume b0:(the (domain s a)) ⇝ d
           show ?thesis using aa a0 a1 a2 b0 p1 weak-step-consistent-def a3 by blast
           next
           assume b1:¬((the (domain s a)) ⇝ d)
           have b2:(domain s a) = (domain t a) by (simp add: a1 sched-vpeq)
           with b1 have b3:¬((the (domain t a)) ⇝ d) by auto
           then have b4:s∼d∼s′ using aa b1  p2 a3  by fastforce
           then have b5:t∼d∼t′ using aa b3 p2 a3 by fastforce
           then show ?thesis using a0 b4 vpeq-symmetric-lemma vpeq-transitive-lemma by blast
         qed
      }
      then show ?thesis by auto
      qed
  }
  then show ?thesis using step-consistent-def by blast
  qed
```

1.4 Lemmas for the inference framework

```
lemma sched-equiv-preserved:
  assumes 1:step-consistent
  and     2:s ∼sched∼ t
  and     3:(s,s′)∈step a
  and     4:(t,t′)∈step a
  and     5:reachable0 s ∧ reachable0 t
shows s′ ∼sched∼ t′
  apply(case-tac the (domain s a) = sched)
  using 1 2 3 4 5 step-consistent-def apply blast
  using 1 2 3 4 5 no-intf-sched step-consistent-def by blast

lemma sched-equiv-preserved-left:
  ⟦local-respect; reachable0 s; (s ∼sched∼ t); the (domain s a) ≠ sched; (s,s′)∈ step a⟧
```

$\Longrightarrow\ (s' \sim sched\sim\ t)$
using *local-respect-def no-intf-sched non-interference-def*
vpeq-symmetric-lemma vpeq-transitive-lemma **by** *blast*

lemma *un-eq*:
$[\![S = S';\ T = T']\!] \Longrightarrow S \cup T = S' \cup T'$
by *auto*

lemma *Un-eq*:
$[\![\bigwedge x\ y.\ [\![x \in xs;\ y \in ys]\!] \Longrightarrow P\ x = Q\ y;\ \exists\ x.\ x \in xs;\ \exists\ y.\ y \in ys]\!] \Longrightarrow (\bigcup x{\in}xs.\ P\ x) = (\bigcup y{\in}ys.\ Q\ y)$
by *auto*

declare *step-consistent-def* [*cong del*]

lemma *sources-eq0*: $step\text{-}consistent \wedge (s \sim sched \sim t) \wedge reachable0\ s \wedge reachable0\ t$
$\longrightarrow sources\ as\ s\ d = sources\ as\ t\ d$
proof (*induct as arbitrary*: *s t*)
case *Nil* **show** *?case*
by (*simp add*: *sources-Nil*)
next
case (*Cons a as*) **show** *?case*
using *sources-Cons* **apply**(*clarsimp simp*: *sources-Cons*)
apply(*rule un-eq*)
apply(*simp only*: *Union-eq*, *simp only*: *UNION-eq*[*symmetric*])
apply(*rule Un-eq*, *clarsimp*)
apply (*meson Cons.hyps reachable0-reach reachableStep reachable-s0 sched-equiv-preserved*)
using *enabled* **apply** *simp*
using *enabled* **apply** *simp*
apply(*clarsimp simp*: *sched-vpeq*)
apply(*rule Collect-cong*)
apply(*rule conj-cong*, *rule refl*)
apply(*rule iff-exI*)
apply (*metis* (*no-types*, *hide-lams*) *Cons.hyps enabled reachableStep sched-equiv-preserved*)
done
qed

lemma *sources-eq*:
$[\![step\text{-}consistent;\ s \sim sched \sim t;\ reachable0\ s;\ reachable0\ t]\!] \Longrightarrow sources\ as\ s\ d = sources\ as\ t\ d$
using *sources-eq0* **by** *blast*

lemma *same-sources-dom*:
$[\![s \approx(sources\ (a\#as)\ s\ d)\approx\ t;\ (the\ (domain\ s\ a)) \leadsto x;\ x \in sources\ as\ s'\ d;$

$(s,s')\in step\ a] \Longrightarrow (s \sim(the\ (domain\ s\ a))\sim t)$
apply *simp*
apply(*erule bspec*)
apply(*subst sources-Cons*)
apply(*rule UnI2*)
apply(*blast*)
done

lemma *sources-step*:
$[\![reachable0\ s;\ (the\ (domain\ s\ a))\ \backslash\!\leadsto d]\!] \Longrightarrow sources\ [a]\ s\ d = \{d\}$
by (*auto simp*: *sources-Cons sources-Nil enabled dest*: *enabled*)

lemma *sources-step2*:
$[\![reachable0\ s;\ (the\ (domain\ s\ a)) \leadsto d]\!] \Longrightarrow sources\ [a]\ s\ d = \{the\ (domain\ s\ a), d\}$
apply(*auto simp*: *sources-Cons sources-Nil enabled dest*: *enabled*)
done

lemma *sources-unwinding-step*:
$[\![s \approx(sources\ (a\#as)\ s\ d)\approx t;\ (s\sim sched\sim t);\ step\text{-}consistent;$
$(s,s')\in step\ a;\ (t,t')\in step\ a;\ reachable0\ s;\ reachable0\ t]\!] \Longrightarrow (s' \approx(sources\ as\ s'\ d)\approx t')$
apply(*clarsimp simp*: *ivpeq-def sources-Cons*)
using *UnionI step-consistent-def* **by** *blast*

lemma *sources-eq-step*:
$[\![local\text{-}respect;\ step\text{-}consistent;(s,s')\in step\ a;$
$(the\ (domain\ s\ a)) \neq sched;\ reachable0\ s]\!] \Longrightarrow$
$(sources\ as\ s'\ d) = (sources\ as\ s\ d)$
using *reachableStep sched-equiv-preserved-left sources-eq0 vpeq-reflexive-lemma* **by** *blast*

lemma *sources-equiv-preserved-left*: $[\![local\text{-}respect;\ step\text{-}consistent;\ s\sim sched\sim t;$
$the\ (domain\ s\ a) \notin sources\ (a\#as)\ s\ d;\ s \approx sources\ (a\#as)\ s\ d\approx t;\ (s,s')\in step\ a;$
$(the\ (domain\ s\ a)) \neq sched;\ reachable0\ s;\ reachable0\ t]\!] \Longrightarrow (s' \approx sources\ as\ s'\ d\approx t)$
apply(*clarsimp simp*: *ivpeq-def cong del*: *local-respect-def*)
apply(*rename-tac v*)
apply(*case-tac* $(the\ (domain\ s\ a)) \leadsto v$)
apply(*fastforce simp*: *sources-Cons cong del*: *local-respect-def*)
proof −
fix $v :: 'd$
assume *a1*: *local-respect*
assume *a2*: *step-consistent*
assume *a3*: $s \sim sched \sim t$
assume *a4*: $\forall\, d\in sources\ (a\ \#\ as)\ s\ d.\ (s \sim d \sim t)$

```
    assume a5: (s, s') ∈ step a
    assume a6: reachable0 s
    assume a7: reachable0 t
    assume a8: v ∈ sources as s' d
    assume a9: ¬ ((the (domain s a)) ⇝ v)
    obtain ss :: 's ⇒ 'e ⇒ 's where
      f10: ∀ e. (t, ss t e) ∈ step e
      using a7 by (meson enabled)
    have ∀ e. domain s e = domain t e
      using a3 by (meson sched-vpeq)
    then have f11: ∀ d sa e. (t, sa) ∉ step e ∨ (t ∼ d ∼ sa) ∨ ((the (domain s e)) ⇝ d)
      using a7 a1 local-respect-def non-interference-def by force
    have s' ∼ v ∼ (ss t a)
      using f10 a8 a7 a6 a5 a4 a3 a2 by (metis (no-types) ivpeq-def sources-unwinding-step)
    then show s' ∼ v ∼ t
      using f11 f10 a9 by (meson vpeq-symmetric-lemma vpeq-transitive-lemma)
  qed

lemma ipurge-eq'-helper:
  ⟦s ∈ ss; the (domain s a) ∈ sources (a # as) s u; ∀ s∈ts. the (domain s a) ∉ sources (a # as) s u;
   (∀ s t. s ∈ ss ∧ t ∈ ts ⟶ (s ∼sched∼ t) ∧ reachable0 s ∧ reachable0 t);  t ∈ ts; step-consistent⟧ ⟹
  False
  apply(cut-tac s=s and t=t and as=as and d=u in sources-eq, (simp cong del: step-consistent-def)+)
  apply(clarsimp  simp: sources-Cons | safe)+
  apply(rename-tac s')
   apply(drule-tac x=t in bspec, simp)
   apply (clarsimp cong del: step-consistent-def)
   apply(cut-tac s=t in enabled, simp)
   apply(erule exE, rename-tac t')
   apply(drule-tac x=sources as t' u in spec)
   apply(cut-tac s=s' and t=t' and d=u in sources-eq, simp+)
      apply(fastforce elim: sched-equiv-preserved)
     apply(fastforce intro: reachableStep)
    apply(fastforce intro: reachableStep)
   apply(fastforce simp: sched-vpeq )
  apply(drule-tac x=t in bspec, simp)
  apply (clarsimp )
  apply(rename-tac v s')
  apply(drule-tac x=v in spec, erule impE, fastforce simp: sched-vpeq)
  apply(cut-tac s=t in enabled[where a=a], simp, clarsimp, rename-tac t')
  apply(cut-tac s=s' and t=t' and d=u in sources-eq, simp+)
    apply(fastforce elim: sched-equiv-preserved )
   apply(fastforce intro: reachableStep )
```

apply(*fastforce intro*: *reachableStep*)
apply(*fastforce simp*: *sched-vpeq*)
done

lemma *ipurge-eq′*:
$(\forall\ s\ t.\ s{\in}ss \wedge t{\in}ts \longrightarrow (s \sim sched\sim t) \wedge reachable0\ s \wedge reachable0\ t) \wedge$
$(\exists\ s.\ s \in ss) \wedge (\exists\ t.\ t \in ts) \wedge step\text{-}consistent \longrightarrow ipurge\ as\ u\ ss = ipurge\ as\ u\ ts$
proof (*induct as arbitrary*: *ss ts*)
case *Nil* **show** *?case*
apply *simp*
done
next
case (*Cons a as*) **show** *?case*
apply(*clarsimp simp*: *sched-vpeq*)
apply(*intro conjI impI*)
apply(*rule Cons.hyps*[*rule-format*])
apply (*clarsimp*)
apply(*metis sched-equiv-preserved reachableStep enabled*)
apply (*clarsimp*)
apply(*drule ipurge-eq′-helper*, *simp*+)[*1*]
apply (*clarsimp*)
apply(*drule ipurge-eq′-helper*, (*simp add*: *vpeq-symmetric-lemma*)+)[*1*]
apply(*rule Cons.hyps*[*rule-format*], *auto*)
done
qed

lemma *ipurge-eq*:$[\![step\text{-}consistent;\ s \sim sched \sim t;\ reachable0\ s \wedge reachable0\ t]\!]$
$\Longrightarrow ipurge\ as\ d\ \{s\} = ipurge\ as\ d\ \{t\}$
by (*simp add*: *ipurge-eq′*)

declare *step-consistent-def* [*cong*]

1.5 Inference framework of information flow security properties

theorem *nonintf-impl-weak*: *noninterference* $\Longrightarrow$ *weak-noninterference*
by (*metis noninterference-def observ-equiv-sym observ-equiv-trans reachable-s0 weak-noninterference-def*)

theorem *wk-nonintf-r-impl-wk-nonintf*: *weak-noninterference-r* $\Longrightarrow$ *weak-noninterference*
using *reachable-s0* **by** *auto*

theorem *nonintf-r-impl-noninterf*: *noninterference-r* $\Longrightarrow$ *noninterference*
using *noninterference-def noninterference-r-def reachable-s0* **by** *auto*

theorem *nonintf-r-impl-wk-nonintf-r*: *noninterference-r* $\Longrightarrow$ *weak-noninterference-r*

by (*metis noninterference-r-def observ-equiv-sym observ-equiv-trans weak-noninterference-r-def*)

lemma *noninf-impl-nonintf-r*: *noninfluence* $\Longrightarrow$ *noninterference-r*
using *ivpeq-def noninfluence-def noninterference-r-def vpeq-reflexive-lemma* **by** *blast*

lemma *noninf-impl-nonlk*: *noninfluence* $\Longrightarrow$ *nonleakage*
using *noninterference-r-def nonleakage-def observ-equiv-sym*
observ-equiv-trans noninfluence-def noninf-impl-nonintf-r **by** *blast*

lemma *wk-noninfl-impl-nonlk*: *weak-noninfluence* $\Longrightarrow$ *nonleakage*
using *weak-noninfluence-def nonleakage-def* **by** *blast*

lemma *wk-noninfl-impl-wk-nonintf-r*: *weak-noninfluence* $\Longrightarrow$ *weak-noninterference-r*
using *ivpeq-def weak-noninfluence-def vpeq-reflexive-lemma weak-noninterference-r-def* **by** *blast*

lemma *noninf-gen-impl-noninfl*: *noninfluence-gen* $\Longrightarrow$ *noninfluence*
using *noninfluence-gen-def noninfluence-def*
by (*metis empty-iff insert-iff*)

lemma *nonlk-imp-sc*: *nonleakage* $\Longrightarrow$ *step-consistent*
proof –
assume *p0*: *nonleakage*
then have *p1*[*rule-format*]: $\forall$ *as d s t*. *reachable0 s* $\wedge$ *reachable0 t* $\longrightarrow$ ($s \sim$ *sched* $\sim t$)
$\longrightarrow$ ($s \approx$ (*sources as s d*) $\approx t$) $\longrightarrow$ ($s \lhd$ *as* $\cong t \lhd$ *as* @ *d*)
using *nonleakage-def* **by** *blast*

have $\forall$ *a d s t*. *reachable0 s* $\wedge$ *reachable0 t* $\longrightarrow$ ($s \sim d \sim t$) $\wedge$ ($s \sim$ *sched* $\sim t$) $\wedge$
(((*the* (*domain s a*)) $\leadsto d$) $\longrightarrow$ ($s \sim$ (*the* (*domain s a*)) $\sim t$)) $\longrightarrow$
($\forall$ s' t'. (s,s') $\in$ *step a* $\wedge$ (t,t') $\in$ *step a* $\longrightarrow$ ($s' \sim d \sim t'$))
proof –
{
fix *a d s t*
assume *a0*: *reachable0 s* $\wedge$ *reachable0 t*
and *a1*: ($s \sim d \sim t$) $\wedge$ ($s \sim$ *sched* $\sim t$)
and *a2*: ((*the* (*domain s a*)) $\leadsto d$) $\longrightarrow$ ($s \sim$ (*the* (*domain s a*)) $\sim t$)
have $\forall$ s' t'. (s,s') $\in$ *step a* $\wedge$ (t,t') $\in$ *step a* $\longrightarrow$ ($s' \sim d \sim t'$)
proof –
{
fix s' t'
assume *b0*: (s,s') $\in$ *step a* $\wedge$ (t,t') $\in$ *step a*
have $s' \sim d \sim t'$
proof(*cases* (*the* (*domain s a*)) $\leadsto d$)
assume *c0*: (*the* (*domain s a*)) $\leadsto d$

```
          with a2 have s ∼ (the (domain s a)) ∼ t by simp
          with a0 a1 c0 have s ≈ (sources [a] s d) ≈ t
            using sources-step2[of s a d]
              insert-iff singletonD by auto
          then have s ⊲ [a] ≅ t ⊲ [a] @ d
            using p1[of s t [a] d] a0 a1 by blast
          with b0 show ?thesis
           by auto
        next
          assume c0: ¬((the (domain s a)) ⇝ d)
          with a0 a1 have s ≈ (sources [a] s d) ≈ t
            using sources-step[of s a d]  by auto
          then have s ⊲ [a] ≅ t ⊲ [a] @ d
            using p1[of s t [a] d] a0 a1 by auto
          with b0 show ?thesis
            by auto
        qed
      }
      then show ?thesis by auto
      qed
    }
    then show ?thesis by blast
    qed
  then show step-consistent using step-consistent-def by blast
qed


lemma sc-imp-nonlk: step-consistent ⟹ nonleakage
 proof −
  assume p0: step-consistent
  have ∀ d as s t. reachable0 s ∧ reachable0 t ⟶ (s ∼ sched ∼ t)
                ⟶ (s ≈ (sources as s d) ≈ t) ⟶ (s ⊲ as ≅ t ⊲ as @ d)
    proof −
    {
      fix as
      have ∀ d s t. reachable0 s ∧ reachable0 t ⟶ (s ∼ sched ∼ t)
                ⟶ (s ≈ (sources as s d) ≈ t) ⟶ (s ⊲ as ≅ t ⊲ as @ d)
        proof(induct as)
          case Nil show ?case using sources-refl by auto
        next
          case (Cons b bs)
          assume a0: ∀ d s t. reachable0 s ∧ reachable0 t ⟶ (s ∼ sched ∼ t)
                    ⟶ (s ≈ sources bs s d ≈ t) ⟶ (s ⊲ bs ≅ t ⊲ bs @ d)
```

```
        show ?case
          proof −
          {
            fix d s t
            assume b0: reachable0 s ∧ reachable0 t
              and  b1: s ∼ sched ∼ t
              and  b2: s ≈ sources (b # bs) s d ≈ t
            then have s ◁ b # bs ≅ t ◁ b # bs @ d
              using exec-equiv-both sources-unwinding-step p0 a0
                by (meson reachableStep SM-axioms sched-equiv-preserved)
          }
          then show ?thesis by blast
          qed
        qed
      }
      then show ?thesis by blast
    qed

    then show nonleakage using nonleakage-def by blast
  qed

theorem sc-eq-nonlk: step-consistent = nonleakage
  using nonlk-imp-sc sc-imp-nonlk by blast

lemma noninf-imp-lr: noninfluence ⟹ local-respect
  proof −
    assume p0: noninfluence
    then have p1[rule-format]: ∀ d as s t . reachable0 s ∧ reachable0 t ⟶ (s ≈ (sources as s d) ≈ t)
                          ⟶ (s ∼ sched ∼ t) ⟶ (s ◁ as ≅ t ◁ (ipurge as d {t}) @ d)
      using noninfluence-def by blast

    have ∀ a d s s'. reachable0 s ⟶ ((the (domain s a)) \⇝ d) ∧ (s,s')∈step a ⟶ (s ∼ d ∼ s')
      proof −
      {
        fix a d s s'
        assume a0: reachable0 s
          and  a1: ((the (domain s a)) \⇝ d) ∧ (s,s')∈step a
        then have a2: the (domain s a) ≠ d using  interf-reflexive by auto
        from a0 a1 p1[of s s [a] d] have a3: s ◁ [a] ≅ s ◁ (ipurge [a] d {s}) @ d
          using  vpeq-reflexive-lemma by auto
        from a0 a1 a2 have ipurge [a] d {s} = []
          using sources-step SM-enabled-axioms  by fastforce
        with a1 a3 have s ∼ d ∼ s'
```

```
        by (metis IdI R-O-Id observ-equiv-sym observ-equivalence-def run-Cons run-Nil)
      }
      then show ?thesis by auto
      qed
    then show local-respect using local-respect-def by blast
  qed

lemma noninf-imp-sc: noninfluence ⟹ step-consistent
  using nonlk-imp-sc noninf-impl-nonlk by blast

theorem UnwindingTheorem : ⟦step-consistent; local-respect⟧ ⟹ noninfluence-gen
proof −
  assume p1:step-consistent
  assume p2:local-respect
  {
    fix as d
    have ∀ s ts. reachable0 s ∧ (∀ t∈ts. reachable0 t)
              ⟶ (∀ t∈ts. (s ≈ (sources as s d) ≈ t))
              ⟶ (∀ t∈ts. (s ∼ sched ∼ t))
              ⟶ (∀ t∈ts. (s ◁ as ≅ t ◁ (ipurge as d ts) @ d))
      proof(induct as)
        case Nil show ?case using sources-refl by auto
      next
        case (Cons b bs)
        assume a0: ∀ s ts. reachable0 s ∧ (∀ t∈ts. reachable0 t)
                  ⟶ (∀ t∈ts. (s ≈ (sources bs s d) ≈ t))
                  ⟶ (∀ t∈ts. (s ∼ sched ∼ t))
                  ⟶ (∀ t∈ts. (s ◁ bs ≅ t ◁ (ipurge bs d ts) @ d))
        show ?case
          proof −
          {
            fix s ts
            assume b0: reachable0 s ∧ (∀ t∈ts. reachable0 t)
              and  b1: ∀ t∈ts. (s ≈ (sources (b # bs) s d) ≈ t)
              and  b2: ∀ t∈ts. (s ∼ sched ∼ t)
            {
              fix t
              assume c0: t∈ts
              have c1: sources (b#bs) s d = sources (b#bs) t d
                using b0 b2 c0 p1 sources-eq0 by blast
              have c2: domain s b = domain t b
                by (simp add: b2 c0 sched-vpeq)
              have s ◁ b # bs ≅ t ◁ ipurge (b # bs) d ts @ d
```

```
proof(cases the (domain s b) ∈ sources (b#bs) s d)
  assume d0:the (domain s b) ∈ sources (b#bs) s d
  have d1: ipurge (b # bs) d ts = b # ipurge bs d (⋃s∈ts. {s'. (s,s') ∈ step b})
    using c0 c1 c2 d0 by auto
  let ?ts' = ⋃s∈ts. {s'. (s,s') ∈ step b}
  let ?bs' = ipurge bs d (⋃s∈ts. {s'. (s,s') ∈ step b})
  {
    fix s' t'
    assume e0: (s,s')∈ run (b#bs) ∧ (t,t')∈ run (b # ?bs')
    then have e1: ∃s'' t''. (s,s'')∈step b ∧ (s'',s')∈run bs ∧ (t,t'')∈step b ∧ (t'',t')∈run ?bs'
      using relcompEpair run-Cons by auto
    then obtain s'' and t'' where e2: (s,s'')∈step b ∧ (s'',s')∈run bs ∧ (t,t'')∈step b ∧ (t'',t')∈run ?bs'
      by auto
    have ∀t∈?ts'. reachable0 t using b0 reachableStep by auto
    moreover
    have ∀t∈?ts'. (s'' ≈ (sources bs s'' d) ≈ t)
      using b0 b1 b2 e2 p1 sources-unwinding-step by blast
    moreover
    have ∀t∈?ts'. (s'' ∼ sched ∼ t)
      using SM-enabled.sched-equiv-preserved SM-enabled-axioms b0 b2 e2 p1 by fast
    ultimately
    have e3: ∀t∈?ts'. (s'' ⊲ bs ≅ t ⊲ (ipurge bs d ?ts') @ d) using a0
      by (metis b0 e2 reachableStep)
    then have s' ∼ d ∼ t'
      using UN-iff c0 e2 mem-Collect-eq  by auto
  }
  then have ∀s' t'. ((s,s')∈ run (b#bs) ∧ (t,t')∈ run (b # ?bs')) ⟶(s' ∼ d ∼ t')
    by simp
  with d1 show ?thesis by auto
next
  assume d0:¬(the (domain s b) ∈ sources (b#bs) s d)
  have d1: ipurge (b # bs) d ts = ipurge bs d ts
    using b0 b2 d0 p1 sched-vpeq sources-eq by (auto cong del: step-consistent-def)
  let ?bs' = ipurge bs d ts
  {
    fix s' t'
    assume e0: (s,s')∈ run (b#bs) ∧ (t,t')∈ run ?bs'
    then have e1: ∃s'' t''. (s,s'')∈step b ∧ (s'',s')∈run bs
      using relcompEpair run-Cons by auto
    then obtain s'' where e2: (s,s'')∈step b ∧ (s'',s')∈run bs
      by auto
    have ∀t∈ts. (s'' ≈ (sources bs s'' d) ≈ t)
      using b0 b1 b2 d0 e2 p1 p2 scheduler-in-sources-Cons
```

sources-equiv-preserved-left **by** *blast*
moreover
have $\forall\, t{\in}ts.\ (s'' \sim sched \sim t)$
using *b0 b2 d0 e2 p2 sched-equiv-preserved-left*
scheduler-in-sources-Cons **by** *blast*
ultimately
have *e3*: $\forall\, t{\in}ts.\ (s'' \lhd bs \cong t \lhd (ipurge\ bs\ d\ ts)\ @\ d)$ **using** *a0*
by (*metis b0 e2 reachableStep*)
then have $s' \sim d \sim t'$
using *c0 e0 e2* **by** *auto*
}
then have $\forall\, s'\ t'.\ ((s,s'){\in}\ run\ (b\#bs) \wedge (t,t'){\in}\ run\ ?bs') \longrightarrow (s' \sim d \sim t')$
by *simp*
with *d1* **show** *?thesis* **by** *auto*
qed

}

}
then show *?thesis* **by** *auto*
qed
qed
}
then show *?thesis* **using** *noninfluence-gen-def* **by** *blast*
qed

theorem *UnwindingTheorem1* : $[\![$*weak-step-consistent*; *local-respect*$]\!] \Longrightarrow$ *noninfluence-gen*
using *UnwindingTheorem weak-with-step-cons* **by** *blast*

theorem *noninf-eq-noninf-gen*: *noninfluence* = *noninfluence-gen*
using *UnwindingTheorem noninf-imp-lr noninf-imp-sc noninf-gen-impl-noninfl* **by** *blast*

theorem *uc-eq-noninf* : (*step-consistent* $\wedge$ *local-respect*) = *noninfluence*
using *UnwindingTheorem1 step-cons-impl-weak noninf-eq-noninf-gen*
noninf-imp-lr noninf-imp-sc **by** *blast*

theorem *noninf-impl-weak*:*noninfluence* $\Longrightarrow$ *weak-noninfluence*
by (*smt observ-equiv-sym observ-equiv-trans ipurge-eq weak-noninfluence-def*
noninterference-r-def noninf-imp-sc noninfluence-def noninf-impl-nonintf-r)

lemma *wk-nonintf-r-and-nonlk-impl-noninfl*: $[\![$*weak-noninterference-r*; *nonleakage*$]\!] \Longrightarrow$ *weak-noninfluence*
proof –
assume *p0*: *weak-noninterference-r*

and $p1$: *nonleakage*
then have $a0$: $\forall\, d\ as\ bs\ s.\ reachable0\ s \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{s\}$
$\longrightarrow (s \lhd as \cong s \lhd bs\ @\ d)$
using *weak-noninterference-r-def* **by** *blast*
from $p1$ **have** $a1$: $\forall\, d\ as\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim sched \sim t)$
$\wedge\ (s \approx (sources\ as\ s\ d) \approx t) \longrightarrow (s \lhd as \cong t \lhd as\ @\ d)$
using *nonleakage-def* **by** *blast*

then have $\forall\ d\ as\ bs\ s\ t\ .\ reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge\ (s \sim sched \sim t) \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{s\}$
$\longrightarrow (s \lhd as \cong t \lhd bs\ @\ d)$
proof −
{
fix $d\ as\ bs\ s\ t$
assume $b0$: $reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge\ (s \sim sched \sim t) \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{s\}$
with $a1$ **have** $b1$: $s \lhd as \cong t \lhd as\ @\ d$ **by** *blast*
from $b0$ **have** $b2$: $ipurge\ as\ d\ \{s\} = ipurge\ as\ d\ \{t\}$
using *ipurge-eq nonlk-imp-sc p1* **by** *blast*
from $b0$ **have** $b3$: $ipurge\ bs\ d\ \{s\} = ipurge\ bs\ d\ \{t\}$
using *ipurge-eq nonlk-imp-sc p1* **by** *blast*
from $a0\ b0\ b2\ b3$ **have** $b4$: $s \lhd as \cong s \lhd bs\ @\ d$ **by** *blast*
from $a0\ b0\ b2\ b3$ **have** $b5$: $t \lhd as \cong t \lhd bs\ @\ d$ **by** *auto*
from $b1\ b4\ b5$ **have** $s \lhd as \cong t \lhd bs\ @\ d$
using *b0 observ-equiv-trans* **by** *blast*
}
then show *?thesis* **by** *blast*
qed
then show *?thesis* **using** *weak-noninfluence-def* **by** *blast*
qed

lemma *nonintf-r-and-nonlk-impl-noninfl*: $[\![noninterference\text{-}r;\ nonleakage]\!] \Longrightarrow noninfluence$
proof −
assume $p0$: *noninterference-r*
and $p1$: *nonleakage*
then have $a0$: $\forall\, d\ as\ s.\ reachable0\ s \longrightarrow (s \lhd as \cong s \lhd (ipurge\ as\ d\ \{s\})\ @\ d)$
using *noninterference-r-def* **by** *blast*
from $p1$ **have** $a1$: $\forall\, d\ as\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim sched \sim t)$
$\wedge\ (s \approx (sources\ as\ s\ d) \approx t) \longrightarrow (s \lhd as \cong t \lhd as\ @\ d)$
using *nonleakage-def* **by** *blast*

then have $\forall\ d\ as\ s\ t\ .\ reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge\ (s \sim sched \sim t) \longrightarrow (s \lhd as \cong t \lhd (ipurge\ as\ d\ \{t\})\ @\ d)$

```
    proof –
    {
      fix d as bs s t
      assume b0: reachable0 s ∧ reachable0 t ∧ (s ≈ (sources as s d) ≈ t)
                  ∧ (s ∼ sched ∼ t)
      with a1 have b1: s ⊲ as ≅ t ⊲ as @ d by blast
      from b0 a0 have b2: s ⊲ as ≅ s ⊲ (ipurge as d {s}) @ d by fast
      from b0 a0 have b3: t ⊲ as ≅ t ⊲ (ipurge as d {t}) @ d by fast

      from b1 b2 b3 have s ⊲ as ≅ t ⊲ (ipurge as d {t}) @ d
        using b0 observ-equiv-trans by blast
    }
    then show ?thesis by blast
    qed
  then show ?thesis using noninfluence-def by blast
qed

lemma noninfl-impl-noninfl2: weak-noninfluence ⟹ weak-noninfluence2
  using ipurge-eq wk-noninfl-impl-nonlk weak-noninfluence2-def
    weak-noninfluence-def nonlk-imp-sc by metis

lemma noninf2-imp-lr: weak-noninfluence2 ⟹ local-respect
  proof –
    assume p0: weak-noninfluence2
    then have p1[rule-format]: ∀ d as bs s t . reachable0 s ∧ reachable0 t ∧ (s ≈ (sources as s d) ≈ t)
                                   ∧ (s ∼ sched ∼ t) ∧ ipurge as d {s} = ipurge bs d {t}
                                   ⟶ (s ⊲ as ≅ t ⊲ bs @ d)
      using weak-noninfluence2-def by blast

    have ∀ a d s s′. reachable0 s ⟶ ((the (domain s a)) \⇝ d) ∧ (s,s′)∈step a ⟶ (s ∼ d ∼ s′)
      proof –
      {
        fix a d s s′
        assume a0: reachable0 s
          and  a1: ((the (domain s a)) \⇝ d) ∧ (s,s′)∈step a
        then have a2: the (domain s a) ≠ d using non-interference-def interf-reflexive by auto
        from a0 a1 a2 have ipurge [a] d {s} = ipurge [] d {s}
          using sources-step SM-enabled-axioms by fastforce
        with a0 have s ⊲ [a] ≅ s ⊲ [] @ d
          using p1[of s s [a] d []] ivpeq-def vpeq-reflexive-lemma by blast
        with a1 have s ∼ d ∼ s′
          by (metis IdI R-O-Id observ-equiv-sym observ-equivalence-def run-Cons run-Nil)
      }
```

then show *?thesis* **by** *auto*
qed
then show *local-respect* **using** *local-respect-def* **by** *blast*
qed

lemma *noninf2-imp-sc*: *weak-noninfluence2* $\Longrightarrow$ *step-consistent*
proof –
assume *p0*: *weak-noninfluence2*
then have *p1*[*rule-format*]: $\forall$ *d as bs s t* . *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ (*s* $\approx$ (*sources as s d*) $\approx$ *t*)
$\wedge$ (*s* $\sim$ *sched* $\sim$ *t*) $\wedge$ *ipurge as d* {*s*} = *ipurge bs d* {*t*}
$\longrightarrow$ (*s* $\lhd$ *as* $\cong$ *t* $\lhd$ *bs* @ *d*)
using *weak-noninfluence2-def* **by** *blast*

have $\forall$ *a d s t*. *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ (*s* $\sim$ *d* $\sim$ *t*) $\wedge$ (*s* $\sim$ *sched* $\sim$ *t*) $\wedge$
(((*the* (*domain s a*)) $\leadsto$ *d*) $\longrightarrow$ (*s* $\sim$ (*the* (*domain s a*)) $\sim$ *t*)) $\longrightarrow$
($\forall$ *s' t'*. (*s*,*s'*) $\in$ *step a* $\wedge$ (*t*,*t'*) $\in$ *step a* $\longrightarrow$ (*s'* $\sim$*d*$\sim$ *t'*))
proof –
{
fix *a d s t*
assume *a0*: *reachable0 s* $\wedge$ *reachable0 t*
and *a1*: (*s* $\sim$ *d* $\sim$ *t*) $\wedge$ (*s* $\sim$ *sched* $\sim$ *t*)
and *a2*: ((*the* (*domain s a*)) $\leadsto$ *d*) $\longrightarrow$ (*s* $\sim$ (*the* (*domain s a*)) $\sim$ *t*)
then have *a3*: *domain s a* = *domain t a* **by** (*simp add*: *sched-vpeq*)

have $\forall$ *s' t'*. (*s*,*s'*) $\in$ *step a* $\wedge$ (*t*,*t'*) $\in$ *step a* $\longrightarrow$ (*s'* $\sim$*d*$\sim$ *t'*)
proof –
{
fix *s' t'*
assume *b0*: (*s*,*s'*) $\in$ *step a* $\wedge$ (*t*,*t'*) $\in$ *step a*
have *s'* $\sim$*d*$\sim$ *t'*
proof(*cases* (*the* (*domain s a*)) $\leadsto$ *d*)
assume *c0*: (*the* (*domain s a*)) $\leadsto$ *d*
with *a2* **have** *c1*: *s* $\sim$ (*the* (*domain s a*)) $\sim$ *t* **by** *simp*
with *a0 a1 c0* **have** *c2*: *s* $\approx$ (*sources* [*a*] *s d*) $\approx$ *t*
using *sources-step2*[*of s a d*] **by** *auto*
from *a0 c0 a3* **have** *c4*: *ipurge* [*a*] *d* {*s*} = *ipurge* [*a*] *d* {*t*}
using *sources-step2*[*of s a d*] *sources-step2*[*of t a d*]
ipurge-Cons[*of a* [] *d* {*s*}] *ipurge-Cons*[*of a* [] *d* {*t*}]
ipurge-Nil **by** *auto*
then have *s* $\lhd$ [*a*] $\cong$ *t* $\lhd$ [*a*] @ *d*
using *p1*[*of s t* [*a*] *d*] *a0 a1 c2* **by** *blast*
with *b0* **show** *?thesis*
by *auto*

```
          next
            assume c0: ¬((the (domain s a)) ⇝ d)
            then have c1: the (domain s a) ≠ d using interf-reflexive by auto
            from c0 a0 a1 have c2: s ≈ (sources [a] s d) ≈ t
              using sources-step[of s a d]  by auto
            from a0 c0 c1 a3 have c4: ipurge [a] d {s} = ipurge [a] d {t}
              using sources-step[of s a d] sources-step[of t a d]
                ipurge-Cons[of a [] d {s}] ipurge-Cons[of a [] d {t}]
                ipurge-Nil by auto
            then have s ◁ [a] ≅ t ◁ [a] @ d
              using p1[of s t [a] d] a0 a1 c2 by blast
            with b0 show ?thesis
              by auto
          qed
        }
        then show ?thesis by auto
        qed
      }
      then show ?thesis by blast
      qed
    then show step-consistent using step-consistent-def by blast
   qed

  theorem noninfl-eq-noninfl2: weak-noninfluence = weak-noninfluence2
    using noninf2-imp-lr noninf2-imp-sc noninf-impl-weak noninfl-impl-noninfl2 uc-eq-noninf  by blast

  theorem nonintf-r-and-nonlk-eq-strnoninfl: (noninterference-r ∧ nonleakage) = noninfluence
    using nonintf-r-and-nonlk-impl-noninfl noninf-impl-nonintf-r noninf-impl-nonlk by blast

  theorem wk-nonintf-r-and-nonlk-eq-noninfl: (weak-noninterference-r ∧ nonleakage) = weak-noninfluence
    using wk-noninfl-impl-nonlk wk-noninfl-impl-wk-nonintf-r wk-nonintf-r-and-nonlk-impl-noninfl by blast

 end
end
```

2 Top-level Specification and security proofs

```
theory SK-TopSpec
imports SK-SecurityModel
begin

declare [[ smt-timeout = 90 ]]
```

2.1 Definitions

2.1.1 Data type, basic components, and system configuration

type-synonym *max-buffer-size* = *nat*
type-synonym *buffer-size* = *nat*
typedecl *Message*

type-synonym *partition-id* = *nat*
type-synonym *partition-name* = *string*
type-synonym *domain-id* = *nat*
type-synonym *channel-id* = *nat*
type-synonym *channel-name* = *string*
datatype *port-direction* = *SOURCE* | *DESTINATION*
type-synonym *port-name* = *string*
type-synonym *port-id* = *nat*
datatype *Port-Type* = *Queuing port-id port-name max-buffer-size port-direction Message set*
    | *Sampling port-id port-name port-direction Message option*

datatype *Channel-Type* = *Channel-Sampling channel-name port-name port-name set*
    | *Channel-Queuing channel-name port-name port-name*

record *Communication-Config* =
    ports-conf :: *Port-Type set*
    channels-conf :: *Channel-Type set*

datatype *partition-type* = *USER-PARTITION* | *SYSTEM-PARTITION*
datatype *partition-mode-type* = *IDLE* | *WARM-START* | *COLD-START* | *NORMAL*

datatype *Partition-Conf* = *PartConf partition-id partition-name partition-type port-name set*

type-synonym *Partitions* = *partition-id* $\rightharpoonup$ *Partition-Conf*

record *Sys-Config* = *partconf* :: *Partitions*
    commconf :: *Communication-Config*
    scheduler :: *domain-id*
    transmitter :: *domain-id*

2.1.2 System state

type-synonym *Ports* = *port-id* $\rightharpoonup$ *Port-Type*

type-synonym *Channels* = *channel-id* $\rightharpoonup$ *Channel-Type*

record *Communication-State* =
  ports :: *Ports*

record *Partition-State-Type* =
  part-mode :: *partition-mode-type*

type-synonym *Partitions-State* = *partition-id* $\rightharpoonup$ *Partition-State-Type*

record *State* =
  current :: *domain-id*
  partitions :: *Partitions-State*
  comm :: *Communication-State*
  part-ports :: *port-id* $\rightharpoonup$ *partition-id*

2.1.3 Events

datatype *Hypercall* = *Create-Sampling-Port port-name*
  | *Write-Sampling-Message port-id Message*
  | *Read-Sampling-Message port-id*
  | *Get-Sampling-Portid port-name*
  | *Get-Sampling-Portstatus port-id*
  | *Create-Queuing-Port port-name*
  | *Send-Queuing-Message port-id Message*
  | *Receive-Queuing-Message port-id*
  | *Get-Queuing-Portid port-name*
  | *Get-Queuing-Portstatus port-id*
  | *Clear-Queuing-Port port-id*
  | *Set-Partition-Mode partition-mode-type*
  | *Get-Partition-Status*

typedecl *Exception*
typedecl *PartitionAction*
datatype *System-Event* = *Schedule*
  | *Transfer-Sampling-Message Channel-Type*
  | *Transfer-Queuing-Message Channel-Type*

datatype *Event* = *hyperc Hypercall* | *sys System-Event*

2.1.4 Utility Functions used for Event Specification

primrec *get-partname-by-type* :: *Partition-Conf* $\Rightarrow$ *partition-name*
  where *get-partname-by-type* (*PartConf* - *pn* - -) = *pn*

primrec *get-partid-by-type* :: *Partition-Conf* $\Rightarrow$ *partition-id*

```
  where get-partid-by-type (PartConf pid - - -) = pid

definition is-a-samplingport :: State ⇒ port-id ⇒ bool
  where is-a-samplingport s pid ≡ case ((ports (comm s)) pid) of
                                     Some (Sampling - - - -) ⇒ True |
                                     - ⇒ False

definition is-a-queuingport :: State ⇒ port-id ⇒ bool
  where is-a-queuingport s pid ≡ case ((ports (comm s)) pid) of
                                     Some (Queuing - - - - -) ⇒ True |
                                     - ⇒ False

definition is-source-port :: State ⇒ port-id ⇒ bool
  where is-source-port s pid ≡
          case ((ports (comm s)) pid) of
              Some (Queuing - - - SOURCE -) ⇒ True |
              Some (Sampling - - SOURCE -) ⇒ True |
              - ⇒ False

definition is-dest-port :: State ⇒ port-id ⇒ bool
  where is-dest-port s pid ≡
            case ((ports (comm s)) pid) of
                Some (Queuing - - - DESTINATION -) ⇒ True |
                Some (Sampling - - DESTINATION -) ⇒ True |
                - ⇒ False

definition is-a-port-of-partition :: State ⇒ port-id ⇒ partition-id ⇒ bool
  where is-a-port-of-partition s port part ≡ (part-ports s) port = Some part

definition is-samplingport-name :: Port-Type ⇒ port-name ⇒ bool
where
is-samplingport-name p n
   ≡ case p of
       (Queuing - name - - -)        ⇒ False
     | (Sampling - name - -) ⇒ name=n


definition is-queuingport-name :: Port-Type ⇒ port-name ⇒ bool
where
is-queuingport-name p n
   ≡ case p of
       (Queuing - name - - -) ⇒ name = n
     | (Sampling - name - -) ⇒ False
```

```
definition is-port-name :: Port-Type ⇒ port-name ⇒ bool
where
is-port-name p n
  ≡ case p of
      (Queuing - name - - -) ⇒ name = n
    | (Sampling - name - -) ⇒ name = n


definition get-samplingport-conf :: Sys-Config ⇒ port-name ⇒ Port-Type option
  where get-samplingport-conf sc pname ≡
                (if (∃p. p ∈ ports-conf (commconf sc) ∧ is-samplingport-name p pname)
                  then Some (SOME p . p ∈ ports-conf (commconf sc) ∧ is-samplingport-name p pname)
                  else None)

definition get-queuingport-conf :: Sys-Config ⇒ port-name ⇒ Port-Type option
  where get-queuingport-conf sc pname ≡
            (if (∃p. p ∈ ports-conf (commconf sc) ∧ is-queuingport-name p pname)
              then Some (SOME p . p ∈ ports-conf (commconf sc) ∧ is-queuingport-name p pname)
              else None)

definition get-portid-by-name :: State ⇒ port-name ⇒ port-id option
  where get-portid-by-name s pn ≡
                (if (∃pid. is-port-name (the (ports (comm s) pid)) pn)
                  then Some (SOME pid . is-port-name (the (ports (comm s) pid)) pn)
                  else None)

definition get-portids-by-names :: State ⇒ port-name set ⇒ (port-id option) set
  where get-portids-by-names s ps ≡ {x. (∃y. y ∈ ps ∧ x = get-portid-by-name s y)}

definition get-portname-by-id :: State ⇒ port-id ⇒ port-name option
  where get-portname-by-id s pid ≡
          let p = ports (comm s) pid in
            case p of Some (Queuing - name - - -) ⇒ Some name
                    | Some (Sampling - name - -) ⇒ Some name
                    | None ⇒ None

definition get-portname-by-type :: Port-Type ⇒ port-name
  where get-portname-by-type pt ≡ case pt of (Queuing - name - - -) ⇒ name
                                             | (Sampling - name - -) ⇒ name

definition get-portid-in-type :: Port-Type ⇒ port-id
```

```
  where get-portid-in-type pt ≡ case pt of (Queuing pid - - - -) ⇒ pid
                                          | (Sampling pid - - -) ⇒ pid

primrec get-partition-cfg-ports :: Partition-Conf ⇒ port-name set
  where get-partition-cfg-ports (PartConf - - - pset) = pset


definition get-partition-cfg-ports-byid :: Sys-Config ⇒ partition-id ⇒ port-name set
  where get-partition-cfg-ports-byid sc p ≡
            if (partconf sc) p = None
            then {}
            else get-partition-cfg-ports (the ((partconf sc) p) )

definition get-ports-of-partition :: State ⇒ partition-id ⇒ port-id set
  where get-ports-of-partition s p = {x. (part-ports s) x = Some p}

primrec get-msg-from-samplingport :: Port-Type ⇒ Message option
  where get-msg-from-samplingport (Sampling - - - msg) = msg |
        get-msg-from-samplingport (Queuing - - - - -) = None

primrec get-msgs-from-queuingport :: Port-Type ⇒ (Message set) option
  where get-msgs-from-queuingport (Sampling - - - -) = None |
        get-msgs-from-queuingport (Queuing - - - - msgs) = Some msgs

definition get-port-byid :: State ⇒ port-id ⇒ Port-Type option
  where get-port-byid s pid ≡  ports (comm s) pid

definition get-the-msg-of-samplingport :: State ⇒ port-id ⇒ Message option
  where get-the-msg-of-samplingport s pid ≡
          let ps = get-port-byid s pid in
              if ps = None then None else get-msg-from-samplingport (the ps)

definition get-the-msgs-of-queuingport :: State ⇒ port-id ⇒ (Message set) option
  where get-the-msgs-of-queuingport s pid ≡
          let ps = get-port-byid s pid in
              if ps = None then None else get-msgs-from-queuingport (the ps)

definition get-port-conf-byid :: Sys-Config ⇒ State ⇒ port-id ⇒ Port-Type option
  where get-port-conf-byid sc s pid ≡ ports (comm s) pid

primrec is-channel-srcname :: Channel-Type ⇒ port-name ⇒ bool
  where is-channel-srcname (Channel-Sampling - n -) name = (name = n) |
        is-channel-srcname (Channel-Queuing - n -) name = (name = n)
```

```
primrec is-channel-destname :: Channel-Type ⇒ port-name ⇒ bool
  where is-channel-destname (Channel-Sampling - - ns) name = (name ∈ ns) |
        is-channel-destname (Channel-Queuing - - n) name = (name = n)

definition get-channel-bysrcport-id :: Sys-Config ⇒ State ⇒ port-id ⇒ Channel-Type option
  where get-channel-bysrcport-id sc s pid ≡
          let nm = get-portname-by-id s pid in
              if ∃x. x∈ channels-conf (commconf sc) ∧ is-channel-srcname x (the nm) then
                let c' = SOME c. c∈channels-conf (commconf sc) ∧ is-channel-srcname c (the nm) in
                  Some c'
              else None

definition get-destports-bysrcport :: Sys-Config ⇒ State ⇒ port-id ⇒ (port-id option) set
  where get-destports-bysrcport sc s pid ≡
          let c = get-channel-bysrcport-id sc s pid in
            case c of Some (Channel-Sampling - - ps) ⇒ get-portids-by-names s ps|
                      Some (Channel-Queuing - - p) ⇒ insert (get-portid-by-name s p) {}|
                      None ⇒ {}

definition update-sampling-port-msg :: State ⇒ port-id ⇒ Message ⇒ State
  where update-sampling-port-msg s pid m ≡
            case ((ports (comm s)) pid) of
              Some (Sampling spid name d msg) ⇒
                (let cs = comm s;
                     pts = ports cs
                 in s(|comm :=
                       cs(| ports := pts(pid := Some (Sampling spid name d (Some m))) |)
                     |)
                )
              |
              - ⇒ s

definition st-msg-destspl-ports :: (port-id ⇒ Port-Type option) ⇒
                          (port-id option) set ⇒ Message ⇒
                          (port-id ⇒ Port-Type option)
  where st-msg-destspl-ports f a b ≡
        % x. (case f x of Some (Sampling spid name d msg) ⇒ Some (Sampling spid name d (Some b)) |
                       - ⇒ f x)

definition update-sampling-ports-msg :: State ⇒ (port-id option) set ⇒ Message ⇒ State
  where update-sampling-ports-msg s st m =
            (let cs = comm s;
```

```
          pts = ports cs
        in s(|comm :=
          cs(| ports := st-msg-destspl-ports pts st m |)
          |)
        )

definition insert-msg2queuing-port :: State ⇒ port-id
    ⇒ Message ⇒ State
  where insert-msg2queuing-port s pid m ≡
        case ((ports (comm s)) pid) of
          Some (Queuing spid name maxs d msgs)
            ⇒ (let cs = comm s;
                  pts = ports cs
                  in s(|comm :=
                      cs(| ports := pts( pid := Some (Queuing spid name maxs d (insert m msgs))) |)
                    |)
                )
          | - ⇒ s

definition replace-msg2queuing-port :: State ⇒ port-id ⇒ Message ⇒ State
  where replace-msg2queuing-port s pid m ≡ s

definition remove-msg-from-queuingport :: State ⇒ port-id ⇒ (State × Message option)
  where remove-msg-from-queuingport s pid ≡
        case ((ports (comm s)) pid) of
          Some (Queuing spid name maxs d msgs)
            ⇒ (let cs = comm s;
                    pts = ports cs;
                    m = SOME x. x∈msgs
                  in (s(|comm :=
                      cs(| ports := pts( pid := Some (Queuing spid name maxs d (msgs − {m}))) |)
                    |),Some m)
                )
          | - ⇒ (s,None)

definition clear-msg-queuingport :: Port-Type ⇒ Port-Type
  where clear-msg-queuingport pt ≡ (case pt of (Queuing spid name maxs d -) ⇒ (Queuing spid name maxs d {}) |
                                          - ⇒ pt)

definition is-a-partition :: Sys-Config ⇒ domain-id ⇒ bool
  where is-a-partition sc nid ≡ (partconf sc) nid ≠ None

definition is-a-transmitter :: Sys-Config ⇒ domain-id ⇒ bool
```

```
  where is-a-transmitter sc nid ≡ (transmitter sc) = nid

definition is-a-scheduler :: Sys-Config ⇒ domain-id ⇒ bool
  where is-a-scheduler sc nid ≡ (scheduler sc) = nid

definition is-a-syspart :: Sys-Config ⇒ partition-id ⇒ bool
  where is-a-syspart sc pid ≡ let p = (partconf sc) pid in
                                  case p of Some (PartConf - - SYSTEM-PARTITION -) ⇒ True |
                                            - ⇒ False

definition is-a-normpart :: Sys-Config ⇒ partition-id ⇒ bool
  where is-a-normpart sc pid ≡ let p = (partconf sc) pid in
                                  case p of Some (PartConf - - USER-PARTITION -) ⇒ True |
                                            - ⇒ False

definition is-there-a-channel-2parts :: Sys-Config ⇒ partition-id ⇒ partition-id ⇒ bool
  where is-there-a-channel-2parts sc p1 p2 ≡
            let ps1 = get-partition-cfg-ports-byid sc p1;
                ps2 = get-partition-cfg-ports-byid sc p2 in
                  (∃ c. c∈ channels-conf (commconf sc) ∧
                       (case c of (Channel-Sampling - sp dps) ⇒ sp∈ps1 ∧ ps2∩dps≠{} |
                                  (Channel-Queuing - sp dp) ⇒ sp∈ps1 ∧ dp∈ps2
                       )
                  )

definition part-intf-transmitter :: Sys-Config ⇒ partition-id ⇒ bool
  where part-intf-transmitter sc p ≡ (let pns = get-partition-cfg-ports (the ((partconf sc) p)) in
                                    (∃ ch pn. ch∈channels-conf (commconf sc) ∧ pn∈pns ⟶
                                         is-channel-srcname ch pn ∨ is-channel-destname ch pn))

definition transmitter-intf-part :: Sys-Config ⇒ partition-id ⇒ bool
  where transmitter-intf-part sc p ≡ (let pns = get-partition-cfg-ports (the ((partconf sc) p)) in
                                    (∃ ch pn. ch∈channels-conf (commconf sc) ∧
                                            pn∈pns ⟶ is-channel-destname ch pn ))

primrec get-max-bufsize-of-port :: Port-Type ⇒ max-buffer-size
  where get-max-bufsize-of-port (Queuing - - n - -) = n |
        get-max-bufsize-of-port (Sampling - - - -) = 1

primrec get-current-bufsize-port :: Port-Type ⇒ buffer-size
  where get-current-bufsize-port (Queuing - - - - ms) = card ms |
        get-current-bufsize-port (Sampling - - - m) = (if m = None then 0 else 1)
```

```
definition is-full-portqueuing :: Sys-Config ⇒ State ⇒ port-id ⇒ bool
  where is-full-portqueuing sc s p ≡
        let conf = get-port-conf-byid sc s p;
              st = get-port-byid s p in
                get-max-bufsize-of-port (the conf) = get-current-bufsize-port (the st)

definition is-empty-port :: State ⇒ port-id ⇒ bool
  where is-empty-port s p ≡
        let st = get-port-byid s p in
              get-current-bufsize-port (the st) = 0

definition get-port-buf-size :: State ⇒ port-id ⇒ buffer-size
  where get-port-buf-size s p ≡
        let st = get-port-byid s p in
              get-current-bufsize-port (the st)

definition is-empty-portqueuing :: State ⇒ port-id ⇒ bool
  where is-empty-portqueuing s p ≡
      let st = get-port-byid s p in
            get-current-bufsize-port (the st) = 0

definition is-empty-portsampling :: State ⇒ port-id ⇒ bool
  where is-empty-portsampling s p ≡
            let st = get-port-byid s p in
                get-current-bufsize-port (the st) = 0

definition has-msg-inportqueuing :: State ⇒ port-id ⇒ bool
  where has-msg-inportqueuing s pid ≡
            case ((ports (comm s)) pid) of
                Some (Queuing - - - - msgs)
                  ⇒ card msgs ≠ 0
               | - ⇒ False

definition get-partconf-byid :: Sys-Config ⇒ partition-id ⇒ Partition-Conf option
  where get-partconf-byid sc pid ≡ (partconf sc) pid

definition get-partstate-byid :: State ⇒ partition-id ⇒ Partition-State-Type option
  where get-partstate-byid s pid ≡ (partitions s) pid
```

2.1.5 Event specification

```
definition create-sampling-port :: Sys-Config ⇒ State ⇒ port-name ⇒ (State × port-id option) where
  create-sampling-port sc s p ≡
            if (get-samplingport-conf sc p = None
```

```
      ∨ get-portid-by-name s p ≠ None
      ∨ p ∉ get-partition-cfg-ports-byid sc (current s))
    then (s,None)
    else
      let cs = comm s;
          pts = ports cs;
          partpts = part-ports s;
          part = current s;
          newid = get-portid-in-type (the (get-samplingport-conf sc p)) in
      (s(|comm := cs(| ports := pts(newid := get-samplingport-conf sc p)|),
        part-ports := partpts(newid := Some part)
       |), Some newid)
```

```
definition write-sampling-message :: State ⇒ port-id ⇒ Message ⇒ (State × bool) where
  write-sampling-message s p m ≡
        (if(¬ is-a-samplingport s p
          ∨ ¬ is-source-port s p
          ∨ ¬ is-a-port-of-partition s p (current s))
        then (s, False)
        else (update-sampling-port-msg s p m, True))
```

```
definition read-sampling-message :: State ⇒ port-id ⇒ (State × Message option) where
  read-sampling-message s pid ≡
        (if (¬ is-a-samplingport s pid
          ∨ ¬ is-a-port-of-partition s pid (current s)
          ∨ ¬ is-dest-port s pid)
        then (s, None)
        else if is-empty-portsampling s pid then
          (s, None)
        else
          (s, get-the-msg-of-samplingport s pid)
        )
```

```
definition get-sampling-port-id :: Sys-Config ⇒ State ⇒ port-name ⇒ (State × port-id option) where
  get-sampling-port-id sc s pname ≡
      (if (pname ∉ get-partition-cfg-ports-byid sc (current s))
      then (s, None)
      else (s, get-portid-by-name s pname))
```

```
definition get-sampling-port-status :: Sys-Config ⇒ State ⇒ port-id ⇒ (State × Port-Type option) where
  get-sampling-port-status sc s pid ≡ (s, get-port-conf-byid sc s pid)
```

```
definition create-queuing-port :: Sys-Config ⇒ State ⇒ port-name ⇒ (State × port-id option) where
  create-queuing-port sc s p ≡
          if (get-queuingport-conf sc p = None
               ∨ get-portid-by-name s p ≠ None
             ∨ p ∉ get-partition-cfg-ports-byid sc (current s))
          then (s,None)
          else
            let cs = comm s;
                pts = ports cs;
                part = current s;
                partpts = part-ports s;
                newid = get-portid-in-type (the (get-queuingport-conf sc p)) in
            (s(|comm :=
              cs(| ports := pts(newid := get-queuingport-conf sc p)|),
               part-ports := partpts(newid := Some part)
             |), Some newid)

definition send-queuing-message-maylost :: Sys-Config ⇒ State ⇒ port-id ⇒ Message ⇒ (State × bool) where
  send-queuing-message-maylost sc s p m ≡
            (if(¬ is-a-queuingport s p
              ∨ ¬ is-source-port s p
              ∨ ¬ is-a-port-of-partition s p (current s))
            then (s, False)
            else if is-full-portqueuing sc s p then
              (replace-msg2queuing-port s p m, True)
            else
              (insert-msg2queuing-port s p m, True))

definition receive-queuing-message :: State ⇒ port-id ⇒ (State × Message option) where
  receive-queuing-message s pid ≡
            (if (¬ is-a-queuingport s pid
              ∨ ¬ is-a-port-of-partition s pid (current s)
              ∨ ¬ is-dest-port s pid
              ∨ is-empty-portqueuing s pid)
            then (s, None)
            else remove-msg-from-queuingport s pid
            )

definition get-queuing-port-id :: Sys-Config ⇒ State ⇒ port-name ⇒ (State × port-id option) where
  get-queuing-port-id sc s pname ≡
          (if (pname ∉ get-partition-cfg-ports-byid sc (current s))
          then (s, None)
          else (s, get-portid-by-name s pname))
```

```
definition get-queuing-port-status :: Sys-Config ⇒ State ⇒ port-id ⇒ (State × Port-Type option) where
  get-queuing-port-status sc s pid ≡ (s, get-port-conf-byid sc s pid)

definition clear-queuing-port :: State ⇒ port-id ⇒ State where
  clear-queuing-port s pid ≡
          (if (¬ is-a-queuingport s pid
                ∨ ¬ is-a-port-of-partition s pid (current s)
                ∨ ¬ is-dest-port s pid)
          then s
          else
            let cs = comm s;
                pts = ports cs;
                pt = (ports cs) pid
              in s(|comm :=
                    cs(| ports := pts( pid := Some (clear-msg-queuingport (the pt))) |)
                  |)

          )

definition schedule :: Sys-Config ⇒ State ⇒ State set where
  schedule sc s ≡ {s(|current:= SOME p. p∈{x. (partconf sc) x ≠ None ∨ x = transmitter sc}|)}

definition get-partition-status :: Sys-Config ⇒ State ⇒ (State × (Partition-Conf option) × (Partition-State-Type option)) where
  get-partition-status sc s ≡ (s, (get-partconf-byid sc (current s), get-partstate-byid s (current s)))

definition set-partition-mode :: Sys-Config ⇒ State ⇒ partition-mode-type ⇒ State where
  set-partition-mode sc s m ≡
      (if (partconf sc) (current s) ≠ None ∧ (partitions s) (current s) ≠ None ∧
         ¬ (part-mode (the ((partitions s) (current s))) = COLD-START ∧ m = WARM-START) then
        let pts = partitions s;
            pstate = the (pts (current s))
        in s(|partitions := pts(current s := Some (pstate(|part-mode := m|)))|)
      else
        s)


primrec transf-sampling-msg :: State ⇒ Channel-Type ⇒ State where
  transf-sampling-msg s (Channel-Sampling - sn dns) =
          (let sp = get-portid-by-name s sn;
              dps = get-portids-by-names s dns in
                if sp≠None ∧ card dps = card dns then
                  let m = the (get-the-msg-of-samplingport s (the sp)) in
```

```
              update-sampling-ports-msg s dps m
            else
              s
        ) |
  transf-sampling-msg s (Channel-Queuing - - -) = s

primrec transf-queuing-msg-maylost :: Sys-Config ⇒ State ⇒ Channel-Type ⇒ State where
  transf-queuing-msg-maylost sc s (Channel-Queuing - sn dn) =
        (let sp = get-portid-by-name s sn;
             dp = get-portid-by-name s dn in
              if sp ≠ None ∧ dp ≠ None ∧ has-msg-inportqueuing s (the sp) then
                let sm = remove-msg-from-queuingport s (the sp) in
                  if is-full-portqueuing sc (fst sm) (the dp) then
                    replace-msg2queuing-port (fst sm) (the dp) (the (snd sm))
                  else
                    insert-msg2queuing-port (fst sm) (the dp) (the (snd sm))
              else s
            ) |
  transf-queuing-msg-maylost sc s (Channel-Sampling - - -) = s

definition system-init :: Sys-Config ⇒ State
  where system-init sc ≡ (|current=(SOME x. (partconf sc) x≠None),
                          partitions=(λ p. (
                                        case ((partconf sc) p) of
                                          None ⇒ None |
                                          (Some (PartConf - - - -)) ⇒ Some ((|part-mode=IDLE|))
                                        )
                                     ),
                          comm = (|ports=(λ x. None)|),
                          part-ports = (λ x. None)
                         |)
```

2.2 Instantiation and Its Proofs of Security Model

```
consts sysconf :: Sys-Config

definition sys-config-witness :: Sys-Config
where
sys-config-witness ≡ (| partconf = Map.empty,
                        commconf = (| ports-conf = {}, channels-conf = {}|),
                        scheduler = 0,
                        transmitter = 1 |)

specification (sysconf)
```

part-id-conf:∀ *p*. (*partconf sysconf*) *p* ≠ *None* ⟶ *get-partid-by-type* (*the* ((*partconf sysconf*) *p*)) = *p*
part-not-sch:(*partconf sysconf*) *x* ≠ *None* ⟹ *x*≠*scheduler sysconf*
part-not-trans : (*partconf sysconf*) *x* ≠ *None* ⟹ *x*≠*transmitter sysconf*
sch-not-part : *scheduler sysconf* = *x* ⟹ (*partconf sysconf*) *x* = *None*
trans-not-part : *transmitter sysconf* = *x* ⟹ (*partconf sysconf*) *x* = *None*
sch-not-trans : *scheduler sysconf* ≠ *transmitter sysconf*
port-name-diff:∀ *p1 p2*. *p1* ∈ *ports-conf* (*commconf sysconf*) ∧ *p2* ∈ *ports-conf* (*commconf sysconf*)
⟶ *get-portname-by-type p1* ≠ *get-portname-by-type p2*
port-partition:∀ *p1 p2*. *get-partition-cfg-ports-byid sysconf p1* ∩ *get-partition-cfg-ports-byid sysconf p2* = {}
apply (*intro exI*[*of - sys-config-witness*] *allI impI*, *simp*)
apply (*rule conjI*, *simp add*: *sys-config-witness-def*)+
by (*simp add*: *get-partition-cfg-ports-byid-def sys-config-witness-def*)

consts *s0t* :: *State*
definition *s0t-witness* :: *State*
where *s0t-witness* ≡ *system-init sysconf*

specification (*s0t*)
s0t-init: *s0t* = *system-init sysconf*
by *simp*

primrec *event-enabled* :: *State* ⇒ *Event* ⇒ *bool*
where *event-enabled s* (*hyperc h*) = (*is-a-partition sysconf* (*current s*)
∧ *part-mode* (*the* (*partitions s* (*current s*))) ≠ *IDLE*) |
event-enabled s (*sys h*) = (*case h of Schedule* ⇒ *True* |
Transfer-Sampling-Message c ⇒ (*current s* = *transmitter sysconf*) |
Transfer-Queuing-Message c ⇒ (*current s* = *transmitter sysconf*))

definition *exec-event* :: *Event* ⇒ (*State* × *State*) *set* **where**
exec-event e = {(*s*,*s*′). *s*′∈ (*if event-enabled s e then* (
case e of hyperc (*Create-Sampling-Port pname*) ⇒ {*fst* (*create-sampling-port sysconf s pname*)} |
hyperc (*Write-Sampling-Message p m*) ⇒ {*fst* (*write-sampling-message s p m*)} |
hyperc (*Read-Sampling-Message p*) ⇒ {*fst* (*read-sampling-message s p*)} |
hyperc (*Get-Sampling-Portid pname*) ⇒ {*fst* (*get-sampling-port-id sysconf s pname*)} |
hyperc (*Get-Sampling-Portstatus p*) ⇒ {*fst* (*get-sampling-port-status sysconf s p*)} |
hyperc (*Create-Queuing-Port pname*) ⇒ {*fst* (*create-queuing-port sysconf s pname*)} |
hyperc (*Send-Queuing-Message p m*) ⇒ {*fst* (*send-queuing-message-maylost sysconf s p m*)} |
hyperc (*Receive-Queuing-Message p*) ⇒ {*fst* (*receive-queuing-message s p*)} |
hyperc (*Get-Queuing-Portid pname*) ⇒ {*fst* (*get-queuing-port-id sysconf s pname*)} |
hyperc (*Get-Queuing-Portstatus p*) ⇒ {*fst* (*get-queuing-port-status sysconf s p*)} |
hyperc (*Clear-Queuing-Port p*) ⇒ {*clear-queuing-port s p*} |
hyperc (*Set-Partition-Mode m*) ⇒ {*set-partition-mode sysconf s m*} |
hyperc (*Get-Partition-Status*) ⇒ {*fst* (*get-partition-status sysconf s*)} |

sys Schedule ⇒ *schedule sysconf s* |
sys (*Transfer-Sampling-Message c*) ⇒ {*transf-sampling-msg s c*} |
sys (*Transfer-Queuing-Message c*) ⇒ {*transf-queuing-msg-maylost sysconf s c*})
else {*s*})}

primrec *domain-of-event* :: *State* ⇒ *Event* ⇒ *domain-id option*
where *domain-of-event s* (*hyperc h*) = *Some* (*current s*) |
domain-of-event s (*sys h*) = (*case h of Schedule* ⇒ *Some* (*scheduler sysconf*) |
Transfer-Sampling-Message c ⇒ *Some* (*transmitter sysconf*) |
Transfer-Queuing-Message c ⇒ *Some* (*transmitter sysconf*))

definition *interference1* :: *domain-id* ⇒ *domain-id* ⇒ *bool* ((- ⇝ -))
where
interference1 d1 d2 ≡
if d1 = *d2 then True*
else if is-a-scheduler sysconf d1 then True
else if ¬(*is-a-scheduler sysconf d1*) ∧ (*is-a-scheduler sysconf d2*) *then False*
else if is-a-partition sysconf d1 ∧ *is-a-transmitter sysconf d2 then part-intf-transmitter sysconf d1*
else if is-a-transmitter sysconf d1 ∧ *is-a-partition sysconf d2 then transmitter-intf-part sysconf d2*
else False

definition *non-interference1* :: *domain-id* ⇒ *domain-id* ⇒ *bool* ((- \⇝ -))
where (*u* \⇝ *v*) ≡ ¬ (*u* ⇝ *v*)

declare *non-interference1-def* [*cong*] **and** *interference1-def* [*cong*] **and** *domain-of-event-def*[*cong*] **and**
event-enabled-def[*cong*] **and** *is-a-partition-def*[*cong*] **and** *is-a-transmitter-def*[*cong*] **and**
is-a-scheduler-def[*cong*] **and** *is-a-syspart-def*[*cong*] **and** *is-a-normpart-def*[*cong*] **and**
transmitter-intf-part-def[*cong*] **and** *part-intf-transmitter-def*[*cong*]

lemma *nintf-neq*: *u* \⇝ *v* ⟹ *u* ≠ *v* **by** *auto*

lemma *nintf-reflx*: *interference1 u u* **by** *auto*

definition *vpeq-part-comm* :: *State* ⇒ *partition-id* ⇒ *State* ⇒ *bool*
where *vpeq-part-comm s d t* ≡
(*let ps1* = *get-ports-of-partition s d*;
ps2 = *get-ports-of-partition t d in*
(*ps1* = *ps2*) ∧
(∀ *p*. *p*∈*ps1* ⟶
(*is-a-queuingport s p* = *is-a-queuingport t p*) ∧
(*is-dest-port s p* = *is-dest-port t p*) ∧
(*if is-dest-port s p then*
get-port-buf-size s p = *get-port-buf-size t p*

else True)
)
)

definition *vpeq-part* :: *State* ⇒ *partition-id* ⇒ *State* ⇒ *bool*
where *vpeq-part s d t* ≡ (*partitions s*) *d* = (*partitions t*) *d* ∧ *vpeq-part-comm s d t*

definition *vpeq-transmitter* :: *State* ⇒ *domain-id* ⇒ *State* ⇒ *bool*
where *vpeq-transmitter s d t* ≡ *comm s* = *comm t* ∧ *part-ports s* = *part-ports t*

definition *vpeq-sched* :: *State* ⇒ *domain-id* ⇒ *State* ⇒ *bool*
where *vpeq-sched s d t* ≡ *current s* = *current t*

definition *vpeq1* :: *State* ⇒ *domain-id* ⇒ *State* ⇒ *bool* ((- ∼ - ∼ -))
where *vpeq1 s d t* ≡
(*if d* = *scheduler sysconf then*
(*vpeq-sched s d t*)
else if d = *transmitter sysconf then*
(*vpeq-transmitter s d t*)
else if is-a-partition sysconf d then
(*vpeq-part s d t*)
else True)

declare *vpeq-part-comm-def* [*cong*] **and**
vpeq-part-def [*cong*] **and** *Let-def* [*cong*] **and** *vpeq-transmitter-def*[*cong*] **and**
vpeq-sched-def[*cong*] **and** *vpeq1-def*[*cong*]

lemma *vpeq-part-comm-transitive-lemma* :
∀ *s t r d*. *vpeq-part-comm s d t* ∧ *vpeq-part-comm t d r* ⟶ *vpeq-part-comm s d r*
by *auto*

lemma *vpeq-part-comm-symmetric-lemma*:∀ *s t d*. *vpeq-part-comm s d t* ⟶ *vpeq-part-comm t d s*
by *auto*

lemma *vpeq-part-comm-reflexive-lemma*:∀ *s d*. *vpeq-part-comm s d s*
by *auto*

lemma *vpeq-part-transitive-lemma* : ∀ *s t r d*. *vpeq-part s d t* ∧ *vpeq-part t d r* ⟶ *vpeq-part s d r*
by *auto*

lemma *vpeq-part-symmetric-lemma*:∀ *s t d*. *vpeq-part s d t* ⟶ *vpeq-part t d s*
by *auto*

lemma *vpeq-part-reflexive-lemma*:$\forall$ *s d*. *vpeq-part s d s*
by *auto*

lemma *vpeq-transmitter-transitive-lemma* :
$\forall$ *s t r d*. *vpeq-transmitter s d t* $\wedge$ *vpeq-transmitter t d r*
$\longrightarrow$ *vpeq-transmitter s d r*
by *simp*

lemma *vpeq-transmitter-symmetric-lemma*:$\forall$ *s t d*. *vpeq-transmitter s d t* $\longrightarrow$ *vpeq-transmitter t d s*
by *simp*

lemma *vpeq-transmitter-reflexive-lemma*:$\forall$ *s d*. *vpeq-transmitter s d s*
by *auto*

lemma *vpeq-scheduler-transitive-lemma* : $\forall$ *s t r d*. *vpeq-sched s d t* $\wedge$ *vpeq-sched t d r* $\longrightarrow$ *vpeq-sched s d r*
by *simp*

lemma *vpeq-scheduler-symmetric-lemma*:$\forall$ *s t d*. *vpeq-sched s d t* $\longrightarrow$ *vpeq-sched t d s*
by *simp*

lemma *vpeq-scheduler-reflexive-lemma*:$\forall$ *s d*. *vpeq-sched s d s*
by *simp*

lemma *vpeq1-transitive-lemma* : $\forall$ *s t r d*. (*vpeq1 s d t*) $\wedge$ (*vpeq1 t d r*) $\longrightarrow$ (*vpeq1 s d r*)
by *auto*

lemma *vpeq1-symmetric-lemma* : $\forall$ *s t d*. (*vpeq1 s d t*) $\longrightarrow$ (*vpeq1 t d s*)
by *auto*

lemma *vpeq1-reflexive-lemma* : $\forall$ *s d*. (*vpeq1 s d s*)
by *auto*

lemma *sched-current-lemma* : $\forall$ *s t a*. *vpeq1 s* (*scheduler sysconf*) *t* $\longrightarrow$ (*domain-of-event s a*) = (*domain-of-event t a*)
by (*metis* (*full-types*) *Event.exhaust domain-of-event.simps*(*1*) *domain-of-event.simps*(*2*) *vpeq1-def vpeq-sched-def*)

lemma *schedeler-intf-all-help* : $\forall$ *d*. *interference1* (*scheduler sysconf*) *d*
by *auto*

lemma *no-intf-sched-help* : $\forall$ *d*. *interference1 d* (*scheduler sysconf*) $\longrightarrow$ *d* = (*scheduler sysconf*)
by *auto*

lemma *reachable-top*: $\forall$ *s a*. (*SM.reachable0 s0t exec-event*) *s* $\longrightarrow$ ($\exists$ *s'*. (*s*, *s'*) $\in$ *exec-event a*)
proof –

{
fix *s a*
assume *p0*: (*SM.reachable0 s0t exec-event*) *s*
have ∃ *s'*. (*s*, *s'*) ∈ *exec-event a*
proof(*induct a*)
case (*hyperc x*) **show** *?case*
apply (*induct x*)
by (*simp add*:*exec-event-def*)+
next
case (*sys x*) **then show** *?case*
apply (*induct x*)
by (*simp add*:*exec-event-def schedule-def*)+
qed
}
then show *?thesis* **by** *auto*
qed

interpretation *SM-enabled*
s0t exec-event domain-of-event scheduler sysconf vpeq1 interference1
using *vpeq1-transitive-lemma vpeq1-symmetric-lemma vpeq1-reflexive-lemma sched-current-lemma*
schedeler-intf-all-help no-intf-sched-help reachable-top nintf-reflx
SM.intro[*of vpeq1 scheduler sysconf domain-of-event interference1*]
SM-enabled-axioms.intro [*of s0t exec-event*]
SM-enabled.intro[*of domain-of-event scheduler sysconf vpeq1 interference1 s0t exec-event*] **by** *blast*

2.3 Correctness for top-level specification

2.3.1 Correctness lemmas

lemma *create-sampling-port-cor*:⟦*r* = *create-sampling-port sysconf s p*; (*snd r*) ≠ *None*⟧
⟹ (*ports* (*comm* (*fst r*))) (*the* (*snd r*)) ≠ *None*
by (*metis create-sampling-port-def get-samplingport-conf-def port-name-diff snd-conv*)

lemma *create-sampling-port-prepost*:
assumes *p1*:*get-samplingport-conf sysconf p* ≠ *None*
and *p2*:*get-portid-by-name s p* = *None*
and *p3*:*p* ∈ *get-partition-cfg-ports-byid sysconf* (*current s*)
and *p4*:*r* = *create-sampling-port sysconf s p*
shows (*ports* (*comm* (*fst r*))) (*the* (*snd r*)) ≠ *None*
by (*metis create-sampling-port-cor create-sampling-port-def option.distinct*(*2*) *p1 p2 p3 p4 sndI*)

lemma *write-sampling-message-cor*:
assumes *p1*:*r* = *write-sampling-message s pid m*
and *p2*:(*snd r*) ≠ *False*

```
    shows the (get-the-msg-of-samplingport (fst r) pid) = m
  proof −
    let ?sp = (ports (comm (fst r))) pid
    let ?s′ = fst r
    have a1:r = (update-sampling-port-msg s pid m, True)
      by (metis eq-snd-iff p1 p2 write-sampling-message-def)
    have a2:is-a-samplingport s pid using p1 p2 write-sampling-message-def by fastforce
    then have a3:get-port-byid s pid ≠ None
      unfolding is-a-samplingport-def get-port-byid-def by (metis option.case-eq-if)
    show ?thesis
      proof(induct (ports (comm s)) pid)
        case None show ?thesis using None.hyps a3 get-port-byid-def by auto
      next
        case (Some pt)
        have b0:(ports (comm s)) pid = Some pt by (simp add: Some.hyps)
        show ?thesis
        proof(induct the ((ports (comm s)) pid))
          case (Queuing x1 x2 x3 x4 x5)
          have the ((ports (comm s)) pid) = Queuing x1 x2 x3 x4 x5 by (simp add: Queuing.hyps)
              with a2 show ?thesis by (simp add: b0 is-a-samplingport-def)
        next
          case (Sampling x1 x2 x3 x4)
            have c0:the (ports (comm ?s′) pid) = Sampling x1 x2 x3 (Some m)
              by (smt Communication-State.select-convs(1) Communication-State.surjective
                Communication-State.update-convs(1) Port-Type.simps(6) Sampling.hyps
                State.select-convs(3) State.surjective State.update-convs(3) a1 b0
                fstI fun-upd-same option.case-eq-if option.distinct(2) option.sel update-sampling-port-msg-def)
            then have c1:get-the-msg-of-samplingport ?s′ pid = get-msg-from-samplingport (the (get-port-byid ?s′ pid))
              unfolding get-the-msg-of-samplingport-def get-port-byid-def
                by (smt Communication-State.select-convs(1) Communication-State.surjective
                  Communication-State.update-convs(1) Port-Type.simps(6) Sampling.hyps
                  State.select-convs(3) State.surjective State.update-convs(3) a1 b0
                  fstI fun-upd-same option.case-eq-if option.distinct(2) update-sampling-port-msg-def)
            then show ?thesis by (simp add: c0 get-port-byid-def)
        qed
      qed
    qed

lemma write-sampling-message-prepost:
  assumes p1:r = write-sampling-message s pid m
    and   p2:is-a-samplingport s pid
    and   p3:is-source-port s pid
    and   p4:is-a-port-of-partition s pid (current s)
```

shows *the* (*get-the-msg-of-samplingport* (*fst r*) *pid*) = *m*
by (*metis p1 p2 p3 p4 sndI write-sampling-message-cor write-sampling-message-def*)

lemma *read-sampling-message-cor*:
assumes *p1*:*r* = *read-sampling-message s pid*
shows *fst r* = *s* $\wedge$ (((*snd r*) $\neq$ *None*) $\longrightarrow$ (*snd r*) = *get-the-msg-of-samplingport s pid*)
by (*simp add*: *p1 read-sampling-message-def*)

lemma *read-sampling-message-prepost*:
assumes *p1*:*r* = *read-sampling-message s pid*
and *p2*:*is-a-samplingport s pid*
and *p3*:*is-dest-port s pid*
and *p4*:*is-a-port-of-partition s pid* (*current s*)
and *p5*:$\neg$ *is-empty-portsampling s pid*
shows (*snd r*) = *get-the-msg-of-samplingport s pid*
by (*simp add*: *p1 p2 p3 p4 p5 read-sampling-message-def*)

lemma *get-sampling-port-id-cor*:
assumes *p1*:*r* = *get-sampling-port-id sysconf s pname*
shows *fst r* = *s* $\wedge$ (((*snd r*) $\neq$ *None*) $\longrightarrow$ (*snd r*) = *get-portid-by-name s pname*)
proof(*rule conjI*)
show *fst r* = *s* **by** (*simp add*: *get-sampling-port-id-def p1*)
show ((*snd r*) $\neq$ *None*) $\longrightarrow$ (*snd r*) = *get-portid-by-name s pname*
by (*simp add*: *get-sampling-port-id-def p1*)
qed

lemma *get-sampling-port-id-prepost*:
assumes *p1*:*r* = *get-sampling-port-id sysconf s pname*
and *p2*:*pname* $\in$ *get-partition-cfg-ports-byid sysconf* (*current s*)
shows (*snd r*) = *get-portid-by-name s pname*
by (*simp add*: *get-sampling-port-id-def p1 p2*)

lemma *get-sampling-port-status-prepost*:
assumes *p1*:*r* = *get-sampling-port-status sysconf s pid*
shows *fst r* = *s* $\wedge$ ((*snd r*) = *get-port-conf-byid sc s pid*)
using *get-port-conf-byid-def get-sampling-port-status-def p1* **by** *auto*

lemma *create-queuing-ports-cor*:$[\![$*r* = *create-queuing-port sysconf s p*; (*snd r*) $\neq$ *None*$]\!]$
$\Longrightarrow$ (*ports* (*comm* (*fst r*))) (*the* (*snd r*)) $\neq$ *None*
by (*metis create-queuing-port-def get-queuingport-conf-def port-name-diff snd-conv*)

lemma *create-queuing-ports-prepost*:

```
  assumes p1:r = create-queuing-port sysconf s p
    and p2:get-queuingport-conf sysconf p ≠ None
    and p3:get-portid-by-name s p = None
    and p4:p ∈ get-partition-cfg-ports-byid sysconf (current s)
   shows (ports (comm (fst r))) (the (snd r)) ≠ None
   by (metis create-queuing-port-def create-queuing-ports-cor option.distinct(1) p1 p2 p3 p4 sndI)

lemma send-queuing-message-maylost-cor:
 assumes p1:r = send-queuing-message-maylost sysconf s pid m
  and   p2:(snd r) ≠ False
  shows (is-full-portqueuing sysconf s pid ⟶ ((fst r) = s))
      ∧ (¬ is-full-portqueuing sysconf s pid ⟶ m ∈ (the (get-the-msgs-of-queuingport (fst r) pid)) )
proof(rule conjI)
  show is-full-portqueuing sysconf s pid ⟶ ((fst r) = s)
    using p1 p2 unfolding send-queuing-message-maylost-def
    by (simp add: replace-msg2queuing-port-def)
  show ¬ is-full-portqueuing sysconf s pid ⟶ m ∈ (the (get-the-msgs-of-queuingport (fst r) pid))
  proof −
  {
    assume a0:¬ is-full-portqueuing sysconf s pid
    have a1:is-a-queuingport s pid using p1 p2 send-queuing-message-maylost-def by fastforce
    let ?s′ = fst r
    have m ∈ (the (get-the-msgs-of-queuingport (fst r) pid))
    proof −
      have b0:?s′ = insert-msg2queuing-port s pid m
        using p1 p2 a0 unfolding send-queuing-message-maylost-def by auto
      then show ?thesis
      proof(induct (ports (comm s)) pid)
        case None show ?thesis using None.hyps a1 is-a-queuingport-def by auto
      next
        case (Some pt)
        have c0:(ports (comm s)) pid = Some pt by (simp add: Some.hyps)
        show ?thesis
        proof(induct the ((ports (comm s)) pid))
          case (Sampling x1 x2 x3 x4)
          have the ((ports (comm s)) pid) = Sampling x1 x2 x3 x4 by (simp add: Sampling.hyps)
          with a1 show ?thesis by (simp add: c0 is-a-queuingport-def)
        next
          case (Queuing x1 x2 x3 x4 x5)
          have a1:the ((ports (comm s)) pid) = Queuing x1 x2 x3 x4 x5 by (simp add: Queuing.hyps)
          with b0 c0 have a2:(ports (comm ?s′)) pid = Some (Queuing x1 x2 x3 x4 (insert m x5))
            unfolding insert-msg2queuing-port-def by (smt Communication-State.select-convs(1)
              Communication-State.surjective Communication-State.update-convs(1) Port-Type.simps(5)
```

```
            State.ext-inject State.surjective State.update-convs(3) fun-upd-same option.sel option.simps(5))
          then show ?thesis unfolding get-the-msgs-of-queuingport-def Let-def get-port-byid-def by simp
        qed
      qed
    qed
  }
  then show ?thesis by auto
  qed
qed

lemma send-queuing-message-maylost-prepost:
  assumes p1:r = send-queuing-message-maylost sysconf s pid m
  and   p2:is-a-queuingport s pid
  and   p3:is-source-port s pid
  and   p4:is-a-port-of-partition s pid (current s)
  and   p5:¬ is-full-portqueuing sysconf s pid
  shows m ∈ (the (get-the-msgs-of-queuingport (fst r) pid))
  by (metis p1 p2 p3 p4 p5 send-queuing-message-maylost-cor send-queuing-message-maylost-def sndI)

lemma receive-queuing-message-cor:
  assumes p1:r = receive-queuing-message s pid
  and   p2:(snd r) ≠ None
  shows the (snd r) ∉ (the (get-the-msgs-of-queuingport (fst r) pid))
 proof −
  from p1 p2 have a1:is-a-queuingport s pid by (metis receive-queuing-message-def snd-conv)
  from p2 have a2:r = remove-msg-from-queuingport s pid using p1 receive-queuing-message-def by auto[1]
  then show ?thesis
  proof(induct (ports (comm s)) pid)
    case None show ?thesis using None.hyps a1 is-a-queuingport-def by auto
  next
    case (Some pt)
    have b1:(ports (comm s)) pid = Some pt by (simp add: Some.hyps)
    show ?thesis
    proof(induct the ((ports (comm s)) pid))
      case (Sampling x1 x2 x3 x4)
      have c1:pt = Sampling x1 x2 x3 x4 by (simp add: Sampling.hyps b1)
      then show ?thesis using a1 b1 is-a-queuingport-def by auto
    next
      case (Queuing x1 x2 x3 x4 x5)
      have c1:pt = Queuing x1 x2 x3 x4 x5 by (simp add: Queuing.hyps b1)
      then have (ports (comm (fst r))) pid = Some (Queuing x1 x2 x3 x4 (x5 − {the (snd r)}))
        by (smt Communication-State.select-convs(1) Communication-State.surjective
          Communication-State.update-convs(1) Port-Type.case(1) Queuing.hyps State.ext-inject
```

```
      State.surjective State.update-convs(3) a1 a2 fstI fun-upd-same is-a-queuingport-def
      option.case-eq-if option.sel remove-msg-from-queuingport-def sndI)
    then show ?thesis by (simp add: get-port-byid-def get-the-msgs-of-queuingport-def)
   qed
  qed
qed

lemma receive-queuing-message-prepost:
  assumes p1:r = receive-queuing-message s pid
   and  p2:is-a-queuingport s pid
   and  p3:is-dest-port s pid
   and  p4:is-a-port-of-partition s pid (current s)
   and  p5:¬ is-empty-portqueuing s pid
   shows the (snd r) ∉ (the (get-the-msgs-of-queuingport (fst r) pid))
proof −
  from p1 p2 p3 p4 p5 have a0:r = remove-msg-from-queuingport s pid
   by (simp add: receive-queuing-message-def)
  then show ?thesis
  proof(induct (ports (comm s)) pid)
   case None show ?thesis using None.hyps is-dest-port-def p3 by auto
  next
   case (Some x)
   have a1:x = the ((ports (comm s)) pid) by (metis Some.hyps option.sel)
   then show ?thesis
   proof(induct the ((ports (comm s)) pid))
    case (Queuing x1 x2 x3 x4 x5)
    have b1:x = Queuing x1 x2 x3 x4 x5 by (simp add: Queuing.hyps a1)
    then have (ports (comm (fst r))) pid = Some (Queuing x1 x2 x3 x4 (x5 − {the (snd r)}))
    proof −
     have Some (Queuing x1 x2 x3 x4 x5) = ports (comm s) pid
      using Some.hyps b1 by blast
     hence remove-msg-from-queuingport s pid = (case Some (Queuing x1 x2 x3 x4 x5) of None
          ⇒ (s, None) | Some (Queuing n cs na p M) ⇒
            (s(|comm := comm s (|ports := ports (comm s)(pid ↦ Queuing n cs na p (M − {SOME m. m ∈ M}))|)|),
            Some (SOME m. m ∈ M)) | Some (Sampling n cs p z) ⇒ (s, None))
      by (metis remove-msg-from-queuingport-def)
     thus ?thesis
      by (simp add: Some(2))
    qed
    then show ?thesis by (simp add: get-port-byid-def get-the-msgs-of-queuingport-def)
   next
    case (Sampling x1 x2 x3 x4)
    have c1:x = Sampling x1 x2 x3 x4 by (simp add: Sampling.hyps a1)
```

```
        then show ?thesis
        proof −
          have case Some x of None ⇒ False | Some (Queuing n cs na p M) ⇒ True
              | Some (Sampling n cs p z) ⇒ False
            using Some.hyps is-a-queuingport-def p2 by presburger
          thus ?thesis
            using c1 by force
        qed
      qed
    qed
  qed

  lemma get-queuing-port-id-cor:
    assumes p1:r = get-queuing-port-id sysconf s pname
      shows fst r = s ∧ (((snd r) ≠ None) ⟶ (snd r) = get-portid-by-name s pname)
  proof(rule conjI)
    show fst r = s by (simp add: get-queuing-port-id-def p1)
    show ((snd r) ≠ None) ⟶ (snd r) = get-portid-by-name s pname
      by (simp add: get-queuing-port-id-def p1)
  qed

  lemma get-queuing-port-id-prepost:
    assumes p1:r = get-queuing-port-id sysconf s pname
        and p2:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
      shows (snd r) = get-portid-by-name s pname
      by (simp add: get-queuing-port-id-def p1 p2)

  lemma get-queuing-port-status-prepost:
    assumes p1:r = get-queuing-port-status sysconf s pid
      shows fst r = s ∧ ((snd r) = get-port-conf-byid sc s pid)
      using get-port-conf-byid-def get-queuing-port-status-def p1 by auto

lemma clear-queuing-port-cor:
   assumes p1:r = clear-queuing-port s pid
    and   p2:r ≠ s
    shows the (get-the-msgs-of-queuingport r pid) = {}
proof −
  from p1 p2 have a1:is-a-queuingport s pid by (metis clear-queuing-port-def)
  then show ?thesis
  proof(induct (ports (comm s)) pid)
    case None show ?thesis using None.hyps a1 is-a-queuingport-def by auto
   next
    case (Some pt)
```

```
      have b1:(ports (comm s)) pid = Some pt by (simp add: Some.hyps)
      show ?thesis
      proof(induct the ((ports (comm s)) pid))
        case (Sampling x1 x2 x3 x4)
        have c1:pt = Sampling x1 x2 x3 x4 by (simp add: Sampling.hyps b1)
        then show ?thesis using a1 b1 is-a-queuingport-def by auto
      next
        case (Queuing x1 x2 x3 x4 x5)
        have c1:pt = Queuing x1 x2 x3 x4 x5 by (simp add: Queuing.hyps b1)
        with p1 a1 b1 c1 have (ports (comm r)) pid = Some (Queuing x1 x2 x3 x4 {})
          unfolding clear-queuing-port-def clear-msg-queuingport-def
            by (smt Communication-State.select-convs(1)
            Communication-State.surjective Communication-State.update-convs(1)
            Port-Type.simps(5) Queuing.hyps State.select-convs(3) State.surjective
            State.update-convs(3) fun-upd-same p2)
        then show ?thesis by (simp add: get-port-byid-def get-the-msgs-of-queuingport-def)
      qed
    qed
  qed

  lemma clear-queuing-port-prepost:
     assumes p1:r = clear-queuing-port s pid
      and  p2:is-a-queuingport s pid
      and  p3:is-dest-port s pid
      and  p4:is-a-port-of-partition s pid (current s)
      shows the (get-the-msgs-of-queuingport r pid) = {}
     proof(induct (ports (comm s)) pid)
       case None show ?thesis using None.hyps is-a-queuingport-def p2 by auto
     next
      case (Some pt)
      have b1:(ports (comm s)) pid = Some pt by (simp add: Some.hyps)
      show ?thesis
      proof(induct the ((ports (comm s)) pid))
        case (Sampling x1 x2 x3 x4)
        have c1:pt = Sampling x1 x2 x3 x4 by (simp add: Sampling.hyps b1)
        then show ?thesis using b1 is-a-queuingport-def p2 by auto
      next
        case (Queuing x1 x2 x3 x4 x5)
        have c1:pt = Queuing x1 x2 x3 x4 x5 by (simp add: Queuing.hyps b1)
        with p1 p2 b1 c1 have (ports (comm r)) pid = Some (Queuing x1 x2 x3 x4 {})
          unfolding clear-queuing-port-def clear-msg-queuingport-def
          proof −
            assume r = (if ¬ is-a-queuingport s pid ∨ ¬ is-a-port-of-partition s pid (current s)
```

```
            ∨ ¬ is-dest-port s pid then s
            else let cs = comm s; pts = ports cs; pt = ports cs pid
              in s(|comm := cs (|ports := pts(pid ↦ case the pt of
                  Queuing spid name maxs d x ⇒ Queuing spid name maxs d {} |
                  Sampling spid list port-direction option ⇒ the pt)|)|))
      hence r = s (|comm := comm s (|ports := ports (comm s)(pid ↦ case the (ports (comm s) pid) of
            Queuing n cs na p M ⇒ Queuing n cs na p {} |
            Sampling n cs p z ⇒ the (ports (comm s) pid))|)|)
        by (metis (full-types) p2 p3 p4)
      hence ports (comm r) pid = Some (case the (ports (comm s) pid) of
          Queuing n cs na p M ⇒ Queuing n cs na p {} |
          Sampling n cs p z ⇒ the (ports (comm s) pid))
        by simp
      thus ?thesis
        by (metis (no-types) Port-Type.simps(5) Queuing.hyps)
    qed
  then show ?thesis by (simp add: get-port-byid-def get-the-msgs-of-queuingport-def)
  qed
qed
```

2.3.2 Invariants: port consistent

```
definition get-ports-util :: (port-id ⇀ 'a) ⇒ port-id set
  where get-ports-util f ≡ {x. f x ≠ None}

definition port-consistent :: State ⇒ bool
  where port-consistent s ≡ ∀ p. ((part-ports s) p ≠ None) = ((ports (comm s)) p ≠ None) ∧
                    ((ports (comm s)) p ≠ None ⟶ p = get-portid-in-type (the ((ports (comm s)) p))) ∧
                    ((ports (comm s)) p ≠ None ⟶ get-portname-by-type (the ((ports (comm s)) p))
                      ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s) p)) )

lemma pt-cons-s0 : port-consistent s0t
  by(clarsimp simp:port-consistent-def s0t-init system-init-def)

lemma crt-smpl-port-presrv-pt-cons:
assumes p1:port-consistent s
  and   p2:s' = fst (create-sampling-port sysconf s pname)
shows   port-consistent s'
proof –
{
  fix p
  have (part-ports s' p ≠ None) = (ports (comm s') p ≠ None)
      ∧ ((ports (comm s')) p ≠ None ⟶ p = get-portid-in-type (the ((ports (comm s')) p)))
      ∧ ((ports (comm s')) p ≠ None ⟶ get-portname-by-type (the ((ports (comm s')) p))
```

```
                         ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s') p)) )
proof(cases get-samplingport-conf sysconf pname = None
          ∨ get-portid-by-name s pname ≠ None
          ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current s))
  assume b0:get-samplingport-conf sysconf pname = None
          ∨ get-portid-by-name s pname ≠ None
          ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current s)
  have s' = s using b0 create-sampling-port-def p2 by auto
  then show ?thesis using p1 port-consistent-def by blast
next
  assume b1:¬ (get-samplingport-conf sysconf pname = None
          ∨ get-portid-by-name s pname ≠ None
          ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current s))
  then show ?thesis
    proof(cases p=get-portid-in-type (the (get-samplingport-conf sysconf pname)))
      assume c0:p=get-portid-in-type (the (get-samplingport-conf sysconf pname))
      show ?thesis using b1 port-partition by fastforce
    next
      assume c1:p ≠ get-portid-in-type (the (get-samplingport-conf sysconf pname))
      show ?thesis
      proof(cases part-ports s p ≠ None)
        assume d0:part-ports s p ≠ None
        have d1:(ports (comm s)) p ≠ None using d0 p1 port-consistent-def by auto
        have d7:p = get-portid-in-type (the ((ports (comm s)) p)) using d1 p1 port-consistent-def by auto
        have d3:part-ports s' p ≠ None using b1 port-partition by fastforce
        have d4:(ports (comm s')) p ≠ None using b1 port-partition by fastforce
        have d6:p = get-portid-in-type (the ((ports (comm s')) p))
        proof -
          obtain pp :: Sys-Config ⇒ char list ⇒ Port-Type where
            pp sysconf pname ∈ ports-conf (commconf sysconf)
            by (meson b1 get-samplingport-conf-def)
          then show ?thesis
            by (metis port-name-diff)
        qed
        have d8:(ports (comm s')) p = (ports (comm s)) p
          using b1 port-partition by auto
        have d9:(part-ports s') p = (part-ports s) p
          using b1 port-partition by fastforce
        have d10:get-portname-by-type (the ((ports (comm s')) p))
                   ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s') p))
              using d1 d8 d9 p1 port-consistent-def by auto
        then show ?thesis using d3 d4 d6 p1 port-consistent-def by auto
      next
```

```
          assume e0:¬ (part-ports s p ≠ None)
          have e1:(ports (comm s)) p = None using e0 p1 port-consistent-def by blast
          have e3:part-ports s′ p = None
            using b1 port-partition by auto
          have e4:(ports (comm s′)) p = None using b1 port-partition by auto
          then show ?thesis by (simp add: e1 e3 e4)
        qed
      qed
  qed
}
then show ?thesis unfolding port-consistent-def by blast
qed

lemma write-smpl-msg-presrv-pt-cons:
⟦port-consistent s; s′ = fst (write-sampling-message s pid msg)⟧ ⟹ port-consistent s′
  apply(clarsimp simp:exec-event-def)
  apply(clarsimp simp:write-sampling-message-def
                      update-sampling-port-msg-def)
  apply(case-tac ports (comm s) pid)
  apply simp
  apply(case-tac a)
  apply simp
  by (metis Int-absorb empty-iff option.distinct(1) port-consistent-def port-partition)

lemma crt-que-port-presrv-pt-cons:
  assumes p1:port-consistent s
    and   p2:s′ = fst (create-queuing-port sysconf s pname)
  shows   port-consistent s′
  proof −
  {
    fix p
    have (part-ports s′ p ≠ None) = (ports (comm s′) p ≠ None)
          ∧ ((ports (comm s′)) p ≠ None ⟶ p = get-portid-in-type (the ((ports (comm s′)) p)))
          ∧ ((ports (comm s′)) p ≠ None ⟶ get-portname-by-type (the ((ports (comm s′)) p))
                                   ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s′) p)) )
    proof(cases get-queuingport-conf sysconf pname = None
                ∨ get-portid-by-name s pname ≠ None
                ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current s))
      assume b0:get-queuingport-conf sysconf pname = None
                ∨ get-portid-by-name s pname ≠ None
                ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current s)
      then show ?thesis using create-queuing-port-def eq-fst-iff
          p1 p2 port-consistent-def by auto
```

```
  next
    assume b1:¬ (get-queuingport-conf sysconf pname = None
                ∨ get-portid-by-name s pname ≠ None
                ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current s))
    then show ?thesis
    proof(cases p=get-portid-in-type (the (get-queuingport-conf sysconf pname)))
      assume c0:p=get-portid-in-type (the (get-queuingport-conf sysconf pname))
      show ?thesis using b1 port-partition by fastforce
    next
      assume c1:p ≠ get-portid-in-type (the (get-queuingport-conf sysconf pname))
      show ?thesis
      proof(cases part-ports s p ≠ None)
        assume d0:part-ports s p ≠ None
        have d1:(ports (comm s)) p ≠ None using d0 p1 port-consistent-def by auto
        have d3:part-ports s' p ≠ None  using b1 port-partition by fastforce
        have d7:p = get-portid-in-type (the ((ports (comm s)) p))
          using d1 p1 port-consistent-def by auto
        have d4:(ports (comm s')) p ≠ None using b1 port-partition by auto
        have d6:p = get-portid-in-type (the ((ports (comm s')) p))
          using b1 port-partition by auto
        have d8:(ports (comm s')) p = (ports (comm s)) p
          using b1 port-partition by auto
        have d9:(part-ports s') p = (part-ports s) p
          using b1 port-partition by auto
        have d10:get-portname-by-type (the ((ports (comm s')) p))
                   ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s') p))
          using d1 d8 d9 p1 port-consistent-def by auto
        then show ?thesis using d3 d4 d6 p1 port-consistent-def by auto
      next
        assume e0:¬ (part-ports s p ≠ None)
        have e1:(ports (comm s)) p = None using e0 p1 port-consistent-def by auto
        have e3:part-ports s' p = None using b1 port-partition by auto
        have e4:(ports (comm s')) p = None using b1 port-partition by auto
        then show ?thesis by (simp add: e1 e3 e4)
      qed
    qed
  qed
}
then show ?thesis unfolding port-consistent-def by blast
qed

lemma st-msg-destspl-ports-nchg-ports :
assumes   p1:nports = st-msg-destspl-ports oports pids m
```

```
 shows  ∀ x. (oports x ≠ None) = (nports x ≠ None)
 proof −
 {
   fix x
   have (oports x ≠ None) = (nports x ≠ None)
   proof(cases oports x = None)
     assume a0:oports x = None
     with p1 have nports x = None unfolding st-msg-destspl-ports-def by simp
     with a0 show ?thesis by simp
   next
     assume b0:oports x ≠ None
     show ?thesis
       proof(induct the (oports x))
         case (Queuing x1 x2 x3 x4 x5)
         have c1:oports x = Some (Queuing x1 x2 x3 x4 x5) by (simp add: Queuing.hyps b0)
         with p1 have nports x = Some (Queuing x1 x2 x3 x4 x5)
           unfolding st-msg-destspl-ports-def by simp
         with c1 show ?case by simp
       next
         case (Sampling x1 x2 x3 x4)
         have c1:oports x = Some (Sampling x1 x2 x3 x4) by (simp add: Sampling.hyps b0)
         with p1 have nports x = Some (Sampling x1 x2 x3 (Some m))
           unfolding st-msg-destspl-ports-def by simp
         with c1 show ?case by simp
       qed
   qed
 }
 then show ?thesis by auto
 qed

lemma update-sampling-ports-msg-nchg-ports:
 assumes  p1:s′ = update-sampling-ports-msg s pts m
 shows  ∀ x. ((ports (comm s)) x ≠ None) = ((ports (comm s′)) x ≠ None) ∧
          ((part-ports s) x ≠ None) = ((part-ports s′) x ≠ None)
proof −
  {fix x
  have ((part-ports s) x ≠ None) = ((part-ports s′) x ≠ None)
    by (metis State.ext-inject State.surjective State.update-convs(3) p1 update-sampling-ports-msg-def)
  also have ((ports (comm s)) x ≠ None) = ((ports (comm s′)) x ≠ None)
  proof −
    fix x :: nat
    have (|ports = ports (comm s′), ... = Communication-State.more (comm s′)|)
     = comm (update-sampling-ports-msg s pts m)
```

```
      by (metis Communication-State.surjective p1)
    then have (|ports = st-msg-destspl-ports (ports (comm
      (|current = current s, partitions = partitions s, comm = comm s, part-ports = part-ports s, ... = State.more s|)))
      pts m, ... = Communication-State.more (comm s)|) = (|ports = ports (comm s'), ... = Communication-State.more (comm s')|)
      by (metis (no-types) Communication-State.surjective Communication-State.update-convs(1)
        State.ext-inject State.surjective State.update-convs(3) update-sampling-ports-msg-def)
    then have (ports (comm s) x ≠ None) ≠ (ports (comm s') x = None)
      by (metis (no-types) Communication-State.ext-inject State.surjective st-msg-destspl-ports-nchg-ports)
    then show (ports (comm s) x ≠ None) = (ports (comm s') x ≠ None)
      by satx
  qed
  ultimately have ((ports (comm s)) x ≠ None) = ((ports (comm s')) x ≠ None) ∧
              ((part-ports s) x ≠ None) = ((part-ports s') x ≠ None) by auto
  } thus ?thesis by blast
qed


lemma trans-spl-msg-presrv-pt-cons:port-consistent s ⟹ port-consistent (transf-sampling-msg s ch)
proof –
{
  assume a0:port-consistent s
  show ?thesis
  proof(induct ch)
    case (Channel-Sampling cn sn dns) show ?case
    proof(clarsimp simp:transf-sampling-msg-def Let-def,rule conjI,rule impI)
      show (∃ y. get-portid-by-name s sn = Some y) ∧ card (get-portids-by-names s dns) = card dns ⟹
            port-consistent (update-sampling-ports-msg s (get-portids-by-names s dns)
              (the (get-the-msg-of-samplingport s (the (get-portid-by-name s sn)))))
      proof –
      {
        let ?s' = update-sampling-ports-msg s (get-portids-by-names s dns)
                  (the (get-the-msg-of-samplingport s (the (get-portid-by-name s sn))))
        from a0 have b0:∀ p. ((part-ports s p ≠ None) = (ports (comm s) p ≠ None))
                  ∧ ((ports (comm s)) p ≠ None ⟶ p = get-portid-in-type (the ((ports (comm s)) p)))
                  ∧ ((ports (comm s)) p ≠ None ⟶ get-portname-by-type (the ((ports (comm s)) p))
                        ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s) p)) )
                  using port-consistent-def by auto
        have b1:∀ x. ((ports (comm s)) x ≠ None) = ((ports (comm ?s')) x ≠ None) ∧
          ((part-ports s) x ≠ None) = ((part-ports ?s') x ≠ None)
          using update-sampling-ports-msg-nchg-ports by presburger
        have b2:∀ x. ((ports (comm ?s')) x ≠ None ⟶ x = get-portid-in-type (the ((ports (comm ?s')) x)))
          using b0 b1 port-partition by fastforce
        have b3:∀ x. ((ports (comm ?s')) x ≠ None ⟶ get-portname-by-type (the ((ports (comm ?s')) x))
```

```
                    ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports ?s') x)) )
              by (metis Int-absorb b0 emptyE port-partition update-sampling-ports-msg-nchg-ports)
        with b0 b1 b2 have port-consistent ?s' using port-consistent-def by metis
      }
      then show ?thesis by auto
      qed
      show (get-portid-by-name s sn = None ⟶ port-consistent s) ∧
           (card (get-portids-by-names s dns) ≠ card dns ⟶ port-consistent s)
        by (simp add: a0)
    qed
    case (Channel-Queuing cn sn dn) show ?case by (simp add: a0)
  qed
}
qed

lemma remove-msg-from-queuingport-presv-port-cons:
assumes  p1:s' = fst (remove-msg-from-queuingport s pt)
  and    p2:port-consistent s
shows  port-consistent s'
proof(induct (ports (comm s)) pt)
  case None  show ?thesis by (simp add: None.hyps option.case-eq-if p1 p2 remove-msg-from-queuingport-def)
next
  case (Some t)
  have a0:(ports (comm s)) pt = Some t by (simp add: Some.hyps)
  show ?thesis
  proof(induct the ((ports (comm s)) pt))
    case (Queuing x1 x2 x3 x4 x5)
    from p2 have ∀ p. ((part-ports s) p ≠ None) = ((ports (comm s)) p ≠ None) ∧
                    ((ports (comm s)) p ≠ None ⟶ p = get-portid-in-type (the ((ports (comm s)) p))) ∧
                       ((ports (comm s)) p ≠ None ⟶ get-portname-by-type (the ((ports (comm s)) p))
                         ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s) p)) )
      using port-consistent-def by auto
    then show ?thesis by (metis a0 disjoint-iff-not-equal option.distinct(1) port-partition)
  next
    case (Sampling x1 x2 x3 x4)
    have Some (Sampling x1 x2 x3 x4) = ports (comm s) pt by (simp add: Sampling.hyps a0)
    then have s' = s unfolding remove-msg-from-queuingport-def
      by (metis (no-types, lifting) Port-Type.simps(6) eq-fst-iff option.simps(5)
        p1 remove-msg-from-queuingport-def)
    with p2 show ?thesis by simp
  qed
qed
```

```
lemma recv-que-msg-presv-port-cons:
  ⟦s' = fst (receive-queuing-message s pt); port-consistent s⟧ ⟹ port-consistent s'
    apply(clarsimp simp:exec-event-def)
    apply(clarsimp simp:receive-queuing-message-def remove-msg-from-queuingport-def)
    apply(case-tac ports (comm s) pt)
    apply simp
    apply(case-tac a)
    apply (metis (full-types) remove-msg-from-queuingport-def
      remove-msg-from-queuingport-presv-port-cons)
    by simp

lemma insert-msg2queuing-port-presv-port-cons:
assumes   p1:s' = insert-msg2queuing-port s pt m
  and     p2:port-consistent s
shows   port-consistent s'
proof(induct (ports (comm s)) pt)
  case None  show ?thesis by (simp add: None.hyps insert-msg2queuing-port-def option.case-eq-if p1 p2)
next
  case (Some t)
  have a0:(ports (comm s)) pt = Some t by (simp add: Some.hyps)
  show ?thesis
  proof(induct the ((ports (comm s)) pt))
    case (Queuing x1 x2 x3 x4 x5)
    from p2 have b1:∀ p. ((part-ports s) p ≠ None) = ((ports (comm s)) p ≠ None) ∧
                        ((ports (comm s)) p ≠ None ⟶ p = get-portid-in-type (the ((ports (comm s)) p))) ∧
                           ((ports (comm s)) p ≠ None ⟶ get-portname-by-type (the ((ports (comm s)) p))
                            ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s) p)) )
      using port-consistent-def by auto
    show ?thesis by (metis a0 b1 disjoint-iff-not-equal option.distinct(1) port-partition)
  next
    case (Sampling x1 x2 x3 x4)
    have Some (Sampling x1 x2 x3 x4) = ports (comm s) pt by (simp add: Sampling.hyps a0)
    then have s' = s unfolding insert-msg2queuing-port-def
      by (metis Port-Type.simps(6) insert-msg2queuing-port-def option.simps(5) p1)
    with p2 show ?thesis by simp
  qed
qed

lemma send-que-msg-presv-port-cons:
  ⟦s' = fst (send-queuing-message-maylost sysconf s pt m); port-consistent s⟧ ⟹ port-consistent s'
    apply(clarsimp simp:exec-event-def)
    apply(clarsimp simp:send-queuing-message-maylost-def
        replace-msg2queuing-port-def insert-msg2queuing-port-def)
```

```
    apply(case-tac ports (comm s) pt)
    apply simp
    apply(case-tac a)
    apply (metis (full-types) insert-msg2queuing-port-def
        insert-msg2queuing-port-presv-port-cons)
    by simp

lemma trans-que-msg-presrv-pt-cons:port-consistent s ⟹ port-consistent (transf-queuing-msg-maylost sysconf s ch)
proof –
{
  assume a0:port-consistent s
  show ?thesis
  proof(induct ch)
    case (Channel-Queuing cn sn dn) show ?case
    proof –
    {
      let ?sm = remove-msg-from-queuingport s (the (get-portid-by-name s sn))
      let ?s0 = fst ?sm
      let ?s1 = replace-msg2queuing-port ?s0 (the (get-portid-by-name s dn)) (the (snd ?sm))
      let ?s2 = insert-msg2queuing-port ?s0 (the (get-portid-by-name s dn)) (the (snd ?sm))
      let ?s3 = transf-queuing-msg-maylost sysconf s ch
      have b1:port-consistent ?s0 by (simp add: a0 remove-msg-from-queuingport-presv-port-cons)
      then have b2:port-consistent ?s1 by (simp add: replace-msg2queuing-port-def)
      with b2 have b5:port-consistent ?s2 by (simp add: b1 insert-msg2queuing-port-presv-port-cons)
      then show ?thesis
        by (clarsimp simp:transf-queuing-msg-maylost-def a0 b2)
    }
    qed
    case (Channel-Sampling cn sn dns) show ?case by (simp add: a0)
  qed
}
qed

lemma clr-que-port-presrv-pt-cons:
assumes   p1:s' = clear-queuing-port s pid
  and     p2:port-consistent s
shows   port-consistent s'
proof –
{
  fix p
  have ((part-ports s') p ≠ None) = ((ports (comm s')) p ≠ None) ∧
          ((ports (comm s')) p ≠ None ⟶ p = get-portid-in-type (the ((ports (comm s')) p))) ∧
          ((ports (comm s')) p ≠ None ⟶ get-portname-by-type (the ((ports (comm s')) p))
```

```
          ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s′) p)) )
  proof(cases ¬ is-a-queuingport s pid
              ∨ ¬ is-a-port-of-partition s pid (current s)
              ∨ ¬ is-dest-port s pid)
    assume a0:¬ is-a-queuingport s pid
              ∨ ¬ is-a-port-of-partition s pid (current s)
              ∨ ¬ is-dest-port s pid
    with p1 have s′ = s unfolding clear-queuing-port-def by auto
    with p2 show ?thesis using port-consistent-def by auto
  next
    assume a1:¬(¬ is-a-queuingport s pid
              ∨ ¬ is-a-port-of-partition s pid (current s)
              ∨ ¬ is-dest-port s pid)
    then show ?thesis by (metis a1 emptyE inf.idem is-a-port-of-partition-def
          option.distinct(1) p2 port-consistent-def port-partition)
  qed
}
then show ?thesis using port-consistent-def by blast
qed

lemma set-part-mode-presrv-pt-cons:
assumes   p1:s′ = set-partition-mode sysconf s m
  and     p2:port-consistent s
shows   port-consistent s′
proof −
{
  fix p
  have ((part-ports s′) p ≠ None) = ((ports (comm s′)) p ≠ None) ∧
        ((ports (comm s′)) p ≠ None ⟶ p = get-portid-in-type (the ((ports (comm s′)) p))) ∧
        ((ports (comm s′)) p ≠ None ⟶ get-portname-by-type (the ((ports (comm s′)) p))
          ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s′) p)) )
  proof(cases (partconf sysconf) (current s) ≠ None ∧ (partitions s) (current s) ≠ None ∧
            ¬ (part-mode (the ((partitions s) (current s))) = COLD-START ∧ m = WARM-START))
    assume a0:(partconf sysconf) (current s) ≠ None ∧ (partitions s) (current s) ≠ None ∧
            ¬ (part-mode (the ((partitions s) (current s))) = COLD-START ∧ m = WARM-START)
    show ?thesis using port-consistent-def
      using a0 p1 p2 set-partition-mode-def by force
  next
    assume a1:¬((partconf sysconf) (current s) ≠ None ∧ (partitions s) (current s) ≠ None ∧
            ¬ (part-mode (the ((partitions s) (current s))) = COLD-START ∧ m = WARM-START))
    then have s′ = s using p1 set-partition-mode-def by auto
    then show ?thesis using p2 port-consistent-def by auto
   qed
```

```
}
then show ?thesis using port-consistent-def by blast
qed

lemma pt-cons-execevt : port-consistent s ⟹ ∀ s'. (s,s')∈exec-event a ⟶ port-consistent s'
proof –
{
  assume a1:port-consistent s
  {
    fix s'
    assume p0: (s,s')∈exec-event a
    then have port-consistent s'
    proof(cases event-enabled s a = True)
      assume b0:event-enabled s a ≠ True
      with a1 p0 show ?thesis using exec-event-def by simp
    next
      assume b1:event-enabled s a = True
      with p0 show ?thesis
      proof(induct a)
        case (hyperc x) then show ?case
          apply (induct x)
          using a1 crt-smpl-port-presrv-pt-cons exec-event-def apply auto[1]
          using a1 write-smpl-msg-presrv-pt-cons exec-event-def apply auto[1]
          using a1 read-sampling-message-def exec-event-def apply auto[1]
          using a1 get-sampling-port-id-def exec-event-def apply auto[1]
          using a1 get-sampling-port-status-def exec-event-def apply auto[1]
          using a1 crt-que-port-presrv-pt-cons exec-event-def apply auto[1]
          using a1 send-que-msg-presv-port-cons exec-event-def apply auto[1]
          using a1 recv-que-msg-presv-port-cons exec-event-def apply auto[1]
          using a1 get-queuing-port-id-def exec-event-def apply auto[1]
          using a1 get-queuing-port-status-def exec-event-def apply auto[1]
          using a1 clr-que-port-presrv-pt-cons exec-event-def apply auto[1]
          using a1 set-part-mode-presrv-pt-cons exec-event-def apply auto[1]
          using a1 get-partition-status-def exec-event-def apply auto[1]
          done
        next
        case (sys x) then show ?case
          using a1 port-consistent-def exec-event-def schedule-def
                trans-spl-msg-presrv-pt-cons trans-que-msg-presrv-pt-cons
                by (induct x, auto)
      qed
    qed
  }
```

```
  then show ?thesis by auto
 }
 qed

lemma pt-cons-exec : ∀ s s′ as. port-consistent s ∧ s′ ∈ execute as s ⟶ port-consistent s′
  proof −
  {
    fix as
    have ∀ s s′. port-consistent s ∧ s′ ∈ execute as s ⟶ port-consistent s′
    proof(induct as)
      case Nil show ?case by (auto simp add: execute-def)
      case (Cons b bs)
        assume a0:∀ s s′. port-consistent s ∧ s′ ∈ execute bs s ⟶ port-consistent s′
        show ?case
          proof −
          {
            fix s s′
            assume b0:port-consistent s
            assume b1:s′ ∈ execute (b # bs) s
            then have port-consistent s′
              proof −
              {
                from b1 have ∃ s1. (s,s1)∈exec-event b ∧ (s1,s′)∈run bs
                  using execute-def run-Cons Image-singleton mem-Collect-eq relcompEpair by auto
                then obtain s1 where c0: (s,s1)∈exec-event b ∧ (s1,s′)∈run bs by auto
                with a0 b0 have port-consistent s1 using exec-event-def pt-cons-execevt by blast
                then show ?thesis using a0 c0 execute-def by blast
              }
              qed
          }
          then show ?thesis by auto
          qed
     qed
  }
  then show ?thesis by auto
  qed

lemma port-cons-reach-state : reachable0 s ⟹ port-consistent s
  using pt-cons-exec pt-cons-s0 reachable-def reachable0-def
    by (simp add: Image-singleton execute-def)
```

2.3.3 Deadlock free

```
lemma deadlock-free : reachable0 s ⟹ (∃ e. event-enabled s e)
```

by (*metis System-Event.case*(*1*) *event-enabled.simps*(*2*))

2.4 Some lemmas of security proofs

lemma *que-port-not-samp* : *is-a-queuingport s p* ⟹ ¬ *is-a-samplingport s p*
apply(*clarsimp simp*:*is-a-queuingport-def is-a-samplingport-def*)
by (*smt Port-Type.case*(*1*) *Port-Type.case*(*2*) *Port-Type.exhaust option.case-eq-if*)

lemma *samp-port-not-que* : *is-a-samplingport s p* ⟹ ¬ *is-a-queuingport s p*
using *que-port-not-samp* **by** *auto*

lemma *src-port-not-dest* : *is-source-port s p* ⟹ ¬ *is-dest-port s p*
apply(*clarsimp simp*:*is-source-port-def is-dest-port-def*)
by (*smt Port-Type.exhaust Port-Type.simps*(*5*) *Port-Type.simps*(*6*)
option.case-eq-if port-direction.exhaust port-direction.simps(*3*) *port-direction.simps*(*4*))

lemma *dest-port-not-src* : *is-dest-port s p* ⟹ ¬ *is-source-port s p*
using *src-port-not-dest* **by** *auto*

lemma *sche-imp-not-part*:*is-a-scheduler sysconf d* ⟹ ¬ (*is-a-partition sysconf d*)
using *sch-not-part* **by** *auto*

lemma *part-imp-not-sch* : *is-a-partition sysconf d* ⟹ ¬ (*is-a-scheduler sysconf d*)
by (*auto simp add*: *sch-not-part*)

lemma *part-imp-not-tras* : *is-a-partition sysconf d* ⟹ ¬ (*is-a-transmitter sysconf d*)
by (*auto simp add*: *trans-not-part*)

lemma *trans-imp-not-part* : *is-a-transmitter sysconf d* ⟹ ¬ (*is-a-partition sysconf d*)
by (*simp add*: *trans-not-part*)

lemma *sche-imp-not-trans*:*is-a-scheduler sysconf d* ⟹ ¬ (*is-a-transmitter sysconf d*)
using *sch-not-trans* **by** *auto*

lemma *trans-imp-not-sche*:*is-a-transmitter sysconf d* ⟹ ¬ (*is-a-scheduler sysconf d*)
using *sch-not-trans* **by** *auto*

lemma *crt-imp-sche*:⟦∀ *v* .*v*∈ *the* ((*domv sysconf*) (*scheduler sysconf*)) ⟶ (*val* (*vars s*)) *v* = (*val* (*vars t*)) *v*;
current s = *current t*⟧ ⟹ (*s* ∼ (*scheduler sysconf*)∼ *t*)
by *auto*

lemma *trans-hasnoports* : *get-partition-cfg-ports-byid sysconf* (*transmitter sysconf*) = {}
by (*meson get-partition-cfg-ports-byid-def is-a-partition-def is-a-transmitter-def part-imp-not-tras*)

lemma *sched-hasnoports* : *get-partition-cfg-ports-byid sysconf* (*scheduler sysconf*) = {}
  by (*meson get-partition-cfg-ports-byid-def is-a-partition-def is-a-scheduler-def part-imp-not-sch*)

lemma *part-ports-imp-portofpart*:*part-ports s* = *part-ports s'* ⟶
  get-ports-of-partition s d = *get-ports-of-partition s' d*
proof −
{
  assume *a0*:*part-ports s* = *part-ports s'*
  have *get-ports-of-partition s d* = *get-ports-of-partition s' d*
  proof −
    have ∀ *p*. *p*∈*get-ports-of-partition s d* ⟶ *p*∈ *get-ports-of-partition s' d*
    proof −
    {
      fix *p*
      assume *p*∈*get-ports-of-partition s d*
      then have (*part-ports s*) *p* = *Some d* **by** (*simp add*: *get-ports-of-partition-def*)
      with *a0* **have** (*part-ports s'*) *p* = *Some d* **by** *simp*
      then have *p*∈*get-ports-of-partition s' d* **by** (*simp add*: *get-ports-of-partition-def*)
    }
    then show *?thesis* **by** *auto*
    qed
    moreover
    have ∀ *p*. *p*∈*get-ports-of-partition s' d* ⟶ *p*∈ *get-ports-of-partition s d*
    proof −
    {
      fix *p*
      assume *p*∈*get-ports-of-partition s' d*
      then have (*part-ports s'*) *p* = *Some d* **by** (*simp add*: *get-ports-of-partition-def*)
      with *a0* **have** (*part-ports s*) *p* = *Some d* **by** *simp*
      then have *p*∈*get-ports-of-partition s d* **by** (*simp add*: *get-ports-of-partition-def*)
    }
    then show *?thesis* **by** *auto*
    qed
    then show *?thesis* **using** *calculation* **by** *blast*
  qed
}
then show *?thesis* **by** *auto*
qed

lemma *no-cfgport-impl-noports* : ⟦*reachable0 s*; *get-partition-cfg-ports-byid sysconf d* = {}⟧
  ⟹ *get-ports-of-partition s d* = {}
proof−

```
  assume p0:reachable0 s
  assume p1:get-partition-cfg-ports-byid sysconf d = {}
  show get-ports-of-partition s d = {}
  proof −
    from p0 have b0:port-consistent s by (simp add: port-cons-reach-state)
    then have b2:∀ x. (part-ports s) x ≠ Some d
    proof −
    {
      fix x
      have (part-ports s) x = Some d ⟶ False
        proof −
        {
          assume c0:(part-ports s) x = Some d
          have c1:(ports (comm s)) x ≠ None using b0 c0 port-consistent-def by auto
          have get-portname-by-type (the ((ports (comm s)) x))
                          ∈ get-partition-cfg-ports-byid sysconf (the ((part-ports s) x))
            using b0 c1 port-consistent-def by auto
          then have False by (simp add: c0 p1)
        }
        then show ?thesis by auto
        qed
    }
    then show ?thesis by auto
    qed
    then have ¬(∃ x. (part-ports s) x = Some d) by auto
    then show ?thesis unfolding get-ports-of-partition-def by auto
  qed
qed

lemma rm-msg-queport-dec-size1:is-a-queuingport s p ∧ ¬ is-empty-port s p
⟶ get-port-buf-size s p = get-port-buf-size (fst (remove-msg-from-queuingport s p)) p + 1
proof −
{
  assume a0:is-a-queuingport s p
  assume a1:¬ is-empty-port s p
  have get-port-buf-size s p = get-port-buf-size (fst (remove-msg-from-queuingport s p)) p + 1
  proof(induct (ports (comm s)) p)
    case None show ?case using None.hyps a0 is-a-queuingport-def by auto
  next
    case (Some x)
    have b0:x = the ((ports (comm s)) p) by (metis Some.hyps option.sel)
    show ?case
    proof(induct the ((ports (comm s)) p))
```

```
      case (Queuing x1 x2 x3 x4 x5)
        have c0:(ports (comm s)) p = Some (Queuing x1 x2 x3 x4 x5) using Queuing.hyps Some.hyps b0 by auto
        let ?s' = fst (remove-msg-from-queuingport s p)
        let ?msgs = the (get-msgs-from-queuingport (the ((ports (comm s)) p)))
        let ?msgs' = the (get-msgs-from-queuingport (the ((ports (comm ?s')) p)))
        let ?m = SOME x. x∈?msgs
        let ?m1 = SOME x. x∈x5
        from c0 have c1:(ports (comm ?s')) p = Some (Queuing x1 x2 x3 x4 (x5−{?m1}))
          unfolding remove-msg-from-queuingport-def by simp
        then have c2:?msgs' = x5−{?m1} unfolding get-msgs-from-queuingport-def by simp
        have c3:x5 = ?msgs by (metis Queuing.hyps get-msgs-from-queuingport.simps(2) option.sel)
        then have c4:?m1 = ?m by simp
        from a1 have c5:card x5 ≠ 0 unfolding is-empty-port-def get-current-bufsize-port-def
          by (metis Queuing.hyps a1 get-current-bufsize-port.simps(1) get-port-byid-def is-empty-port-def)
        then have c6:card x5 > 0 by blast
        then have c7:?m ∈ x5 using c0 some-in-eq by fastforce
        with c2 c3 c4 c5 c6 have card ?msgs = card ?msgs' + 1
          by (metis One-nat-def Suc-pred add.right-neutral add-Suc-right card-Diff-singleton card-infinite)

        then show ?case unfolding get-port-buf-size-def get-current-bufsize-port-def
          by (metis Port-Type.simps(7) Queuing.hyps c1 c2 c3 get-port-byid-def option.sel)
    next
      case (Sampling x1 x2 x3 x4)
        show ?case by (smt Port-Type.simps(6) Sampling.hyps a0 case-optionE is-a-queuingport-def option.sel)
    qed
  qed
}
then show ?thesis by simp
qed

lemma rm-msg-queport-dec-size0:is-a-queuingport s p ∧ is-empty-port s p
    ⟶ get-port-buf-size s p = get-port-buf-size (fst (remove-msg-from-queuingport s p)) p
proof −
{
  assume a0:is-a-queuingport s p
  assume a1:is-empty-port s p
  have get-port-buf-size s p = get-port-buf-size (fst (remove-msg-from-queuingport s p)) p
  proof(induct (ports (comm s)) p)
    case None show ?case using None.hyps a0 is-a-queuingport-def by auto
  next
    case (Some x)
    have b0:x = the ((ports (comm s)) p) by (metis Some.hyps option.sel)
    show ?case
```

```
    proof(induct the ((ports (comm s)) p))
      case (Queuing x1 x2 x3 x4 x5)
      have c0:(ports (comm s)) p = Some (Queuing x1 x2 x3 x4 x5) using Queuing.hyps Some.hyps b0 by auto
      let ?s' = fst (remove-msg-from-queuingport s p)
      let ?msgs = the (get-msgs-from-queuingport (the ((ports (comm s)) p)))
      let ?msgs' = the (get-msgs-from-queuingport (the ((ports (comm ?s')) p)))
      let ?m = SOME x. x∈?msgs
      let ?m1 = SOME x. x∈x5
      from c0 have c1:(ports (comm ?s')) p = Some (Queuing x1 x2 x3 x4 (x5−{?m1}))
        unfolding remove-msg-from-queuingport-def Let-def by simp
      then have c2:?msgs' = x5−{?m1} unfolding get-msgs-from-queuingport-def by simp
      have c3:x5 = ?msgs by (metis Queuing.hyps get-msgs-from-queuingport.simps(2) option.sel)
      then have c4:?m1 = ?m by simp
      from a1 have c5:card x5 = 0 unfolding is-empty-port-def get-current-bufsize-port-def
        by (metis Queuing.hyps a1 get-current-bufsize-port.simps(1) get-port-byid-def is-empty-port-def)
      with c2 have c7:card ?msgs' = 0 using card-eq-0-iff by fastforce
      then show ?case unfolding get-port-buf-size-def get-current-bufsize-port-def
        by (metis Port-Type.simps(7) Queuing.hyps c1 c2 c5 get-port-byid-def option.sel)
    next
      case (Sampling x1 x2 x3 x4)
      show ?case by (smt Port-Type.simps(6) Sampling.hyps a0 case-optionE is-a-queuingport-def option.sel)
    qed
  qed
}
then show ?thesis by simp
qed

lemma clr-queport-size0:is-a-queuingport s p ∧ is-a-port-of-partition s p (current s) ∧ is-dest-port s p
                ⟶ get-port-buf-size (clear-queuing-port s p) p = 0
proof −
{
  let ?s' = clear-queuing-port s p
  assume a0:is-a-queuingport s p
  assume a1:is-a-port-of-partition s p (current s)
  assume a2:is-dest-port s p
  have get-port-buf-size ?s' p = 0
  proof(cases ¬ is-a-queuingport s p
            ∨ ¬ is-a-port-of-partition s p (current s)
            ∨ ¬ is-dest-port s p)
    assume b0:¬ is-a-queuingport s p
            ∨ ¬ is-a-port-of-partition s p (current s)
            ∨ ¬ is-dest-port s p
    with a0 a1 a2 show ?thesis by simp
```

```
  next
    assume b0:¬(¬ is-a-queuingport s p
              ∨ ¬ is-a-port-of-partition s p (current s)
              ∨ ¬ is-dest-port s p)
    then have b1:(ports (comm ?s')) p = Some (clear-msg-queuingport (the ((ports (comm s)) p)))
      unfolding clear-queuing-port-def by (smt Communication-State.select-convs(1)
        Communication-State.surjective Communication-State.update-convs(1)
        State.select-convs(3) State.surjective State.update-convs(3) fun-upd-same)
    show ?thesis
    proof(induct the ((ports (comm s)) p))
      case (Queuing x1 x2 x3 x4 x5)
        have the ((ports (comm ?s')) p) = Queuing x1 x2 x3 x4 {}
          by (metis Port-Type.simps(5) Queuing.hyps b1 clear-msg-queuingport-def option.sel)
        then show ?case unfolding get-current-bufsize-port-def get-port-buf-size-def
        Let-def get-port-byid-def by simp
    next
      case (Sampling x1 x2 x3 x4)
      with a0 show ?case by (metis Port-Type.simps(6) is-a-queuingport-def option.case-eq-if)
    qed
  qed
}
then show ?thesis by auto
qed


lemma same-part-mode:
  assumes p1: is-a-partition sysconf (current s)
      and   p2: s ∼ scheduler sysconf ∼ t
      and   p3: s ∼ current s ∼ t
    shows part-mode (the (partitions s (current s))) = part-mode (the (partitions t (current t)))
proof −
  from p1 p3 have vpeq-part s (current s) t
    using part-imp-not-sch part-imp-not-tras by fastforce
  moreover
  from p2 have current s = current t by auto
  ultimately show ?thesis by auto
qed
```

2.5 Concrete unwinding condition of "local respect"

2.5.1 proving "create sampling port" satisfying the "local respect" property

```
lemma crt-smpl-port-notchg-current:
  ⟦is-a-partition sysconf (current s); s' = fst (create-sampling-port sysconf s pname)⟧
```

```
  ⟹ current s = current s'
 by (clarsimp simp:create-sampling-port-def)
```

the state before and after executing the action "create sampling port" is observ equal to scheduler

```
lemma crt-smpl-port-sm-sche:⟦is-a-partition sysconf (current s);
                         s' = fst (create-sampling-port sysconf s pname)⟧
                            ⟹ (s∼(scheduler sysconf)∼s')
 using crt-smpl-port-notchg-current
    vpeq1-def vpeq-sched-def is-a-scheduler-def part-imp-not-sch by metis

lemma crt-sampl-port-notchg-partstate:
⟦is-a-partition sysconf (current s); is-a-partition sysconf d;
 s' = fst (create-sampling-port sysconf s pname)⟧
                       ⟹ (partitions s) d = (partitions s') d
by (clarsimp simp:create-sampling-port-def)

lemma crt-sampl-port-notchg-partportsinotherdom:
assumes p1:is-a-partition sysconf (current s)
  and  p3:(current s) ≠ d
  and  p4:s' = fst (create-sampling-port sysconf s pname)
shows  get-ports-of-partition s d = get-ports-of-partition s' d
proof –
{
  have ∀ p. p∈get-ports-of-partition s d ⟶ p∈get-ports-of-partition s' d
  proof–
  {
    fix p
    assume a0:p∈get-ports-of-partition s d
    have a1:p∈get-ports-of-partition s' d
    proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
      assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
      have b1:p ≠ get-portid-in-type (the (get-samplingport-conf sysconf pname))
        using b0 port-partition by auto
      then show ?thesis using b0 port-partition by auto
    next
      assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
      then have c1:s' = s by (simp add: create-sampling-port-def p4)
      then show ?thesis by (simp add: a0)
    qed
  }
  then show ?thesis by auto
  qed
  moreover
```

```
    have ∀ p. p∈get-ports-of-partition s′ d ⟶ p∈get-ports-of-partition s d
    proof−
    {
      fix p
      assume a0:p∈get-ports-of-partition s′ d
      have p∈get-ports-of-partition s d
      proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
        assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
        have b1:p ≠ get-portid-in-type (the (get-samplingport-conf sysconf pname))
          using b0 port-partition by auto
        then show ?thesis using b0 port-partition by auto
      next
        assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
        then have c1:s′ = s by (simp add: create-sampling-port-def p4)
        then show ?thesis using a0 by auto
      qed
    }
    then show ?thesis by auto
    qed
    then show ?thesis using calculation by blast
  }
  qed

  lemma crt-sampl-port-notchg-portsinotherdom:
  assumes p1:is-a-partition sysconf (current s)
    and   p3:(current s) ≠ d
    and   p4:s′ = fst (create-sampling-port sysconf s pname)
  shows  ∀ p. p∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
  proof −
  {
    fix p
    have p ∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
    proof −
    {
      assume a0:p ∈ get-ports-of-partition s d
      have ports (comm s) p = ports (comm (fst (create-sampling-port sysconf s pname))) p
      proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
        assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
        have b1:p ≠ get-portid-in-type (the (get-samplingport-conf sysconf pname))
          using b0 port-partition by auto
        then show ?thesis using b0 port-partition by auto
      next
        assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
```

then have *c1*:$s' = s$ **by** (*simp add*: *create-sampling-port-def p4*)
then show *?thesis* **using** *p4* **by** *auto*
qed
}
then show *?thesis* **by** (*simp add*: *p4*)
qed
}
then show *?thesis* **by** *auto*

qed

lemma *crt-sampl-port-notchg-comminotherdom*:
assumes *p0*:*reachable0 s*
and *p1*:*is-a-partition sysconf* (*current s*)
and *p3*:(*current s*) $\neq$ *d*
and *p4*:$s' =$ *fst* (*create-sampling-port sysconf s pname*)
shows *vpeq-part-comm s d* s'
proof−
have *get-ports-of-partition s d* = *get-ports-of-partition* s' *d*
using *crt-sampl-port-notchg-partportsinotherdom p0 p1 p3 p4* **by** *auto*
also have $\forall$ *p*. *p* $\in$ *get-ports-of-partition s d* $\longrightarrow$
is-a-queuingport s p = *is-a-queuingport* s' *p* $\wedge$
is-dest-port s p = *is-dest-port* s' *p* $\wedge$
(*if is-dest-port s p then get-port-buf-size s p* = *get-port-buf-size* s' *p else True*)
unfolding *is-a-queuingport-def is-dest-port-def*
using *crt-sampl-port-notchg-portsinotherdom*
p1 p3 p4 get-port-byid-def p1 p3 p4 get-port-buf-size-def **by** *auto*

ultimately show *?thesis* **by** *auto*
qed

declare *is-a-partition-def* [*cong del*]
lemma *crt-smpl-port-sm-nitfpart*:⟦*reachable0 s*; *is-a-partition sysconf* (*current s*); *is-a-partition sysconf d*;
((*current s*) \⇝ *d*); $s' =$ *fst* (*create-sampling-port sysconf s pname*)⟧
$\Longrightarrow (s \sim d \sim s')$
apply(*clarsimp*)
using *trans-imp-not-part sche-imp-not-part*
apply (*simp add*: *crt-sampl-port-notchg-partstate*)
by (*metis create-sampling-port-def fst-conv get-samplingport-conf-def port-name-diff*)

declare *is-a-partition-def* [*cong*]

lemma *crt-smpl-port-presrv-lcrsp*:

```
assumes p0:reachable0 s
  and   p1:is-a-partition sysconf (current s)
  and   p2:(current s) \⇝ d
  and   p3:s′ = fst (create-sampling-port sysconf s pname)
shows   s ∼ d ∼ s′
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  show ?thesis using crt-smpl-port-sm-sche[OF p1 p3] a0 by auto
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
  proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    show ?thesis using b0 crt-smpl-port-sm-nitfpart p0 p1 p2 p3 by blast
  next
    assume b1:¬ is-a-partition sysconf d
    show ?thesis
    proof(cases is-a-transmitter sysconf d)
      assume c0:is-a-transmitter sysconf d
      show ?thesis
      proof −
      {
        have vpeq-transmitter s d s′ unfolding vpeq-transmitter-def
        proof−
          show comm s = comm s′ ∧  part-ports s = part-ports s′
          proof(rule conjI)
          {
            show comm s = comm s′
            proof −
            {
              from p1 p2 have ¬ part-intf-transmitter sysconf (current s)
                using interference1-def by (meson a1 c0 non-interference1-def)
              then have get-partition-cfg-ports (the ((partconf sysconf) (current s))) = {}
                using get-partition-cfg-ports-byid-def  p1 port-partition by fastforce
              then have pname ∉ get-partition-cfg-ports-byid sysconf (current s)
                by (simp add: get-partition-cfg-ports-byid-def)
              then have s = s′by (simp add: create-sampling-port-def p3)
            }
            then show ?thesis by auto
            qed
            show part-ports s = part-ports s′
            proof −
            {
```

```
              from p1 p2 c0 have d0:¬ part-intf-transmitter sysconf (current s)
                using interference1-def non-interference1-def by (meson a1)
              then have d1:get-partition-cfg-ports-byid sysconf (current s) = {} using port-partition by blast
              then have d2:s=s′ by (smt create-sampling-port-def empty-iff fst-conv p3)
            }
            then show ?thesis by auto
            qed
          }
          qed
        qed
      }
      then show ?thesis using a1 b1 is-a-scheduler-def vpeq1-def by auto
      qed
    next
      assume c1:¬ is-a-transmitter sysconf d
      show ?thesis using a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def by auto
    qed
  qed
qed

lemma crt-smpl-port-presrv-lcrsp-e: local-respect-e (hyperc (Create-Sampling-Port pn))
  using crt-smpl-port-presrv-lcrsp  prod.simps(2) exec-event-def
    mem-Collect-eq  singletonD vpeq-reflexive-lemma
 by (auto cong del: vpeq1-def)
```

2.5.2 proving "write sampling message" satisfying the "local respect" property

```
lemma wrt-smpl-msg-notchg-current:
⟦is-a-partition sysconf (current s); s′ = fst (write-sampling-message s pid m)⟧
  ⟹ current s = current s′
  apply(clarsimp simp:write-sampling-message-def update-sampling-port-msg-def)
  apply(case-tac ports (comm s) pid)
  apply simp
  apply(case-tac a)
  by auto
```

the state before and after executing the action "write sampling message" is observ equal to scheduler

```
lemma wrt-smpl-msg-sm-sche:⟦is-a-partition sysconf (current s);
                              s′ = fst (write-sampling-message s pid m)⟧
                                   ⟹ (s∼(scheduler sysconf)∼s′)
 using wrt-smpl-msg-notchg-current part-imp-not-sch by (meson vpeq1-def vpeq-sched-def)

lemma wrt-smpl-msg-notchg-partstate:
             ⟦is-a-partition sysconf (current s); is-a-partition sysconf d;
```

```
            s' = fst (write-sampling-message s pid m)⟧
                 ⟹ (partitions s) d = (partitions s') d
  apply(clarsimp simp:write-sampling-message-def update-sampling-port-msg-def)
  apply(case-tac ports (comm s) pid)
  apply simp
  apply(case-tac a)
  by auto

lemma wrt-smpl-msg-notchg-partports:
    ⟦is-a-partition sysconf (current s); s' = fst (write-sampling-message s pid m)⟧⟹
        part-ports s = part-ports s'
   apply(clarsimp simp:write-sampling-message-def update-sampling-port-msg-def)
    apply(case-tac ports (comm s) pid)
    apply simp
    apply(case-tac a)
    by  auto

lemma wrt-smpl-msg-notchg-portinotherdom:
assumes p1:is-a-partition sysconf (current s)
  and   p3:(current s) ≠ d
  and   p4:s' = fst (write-sampling-message s pid m)
shows  ∀p. p∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s') p
proof −
{

  fix p
  have p ∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s') p
  proof −
  {
    assume a0:p ∈ get-ports-of-partition s d
    have a1:(part-ports s) p = Some d using a0 get-ports-of-partition-def by auto
    have ports (comm s) p = ports (comm s') p
    proof(cases p = pid)
      assume b0:p = pid
      have b1:(part-ports s) pid = Some d using a1 b0 by auto
      have b2:¬ is-a-port-of-partition s pid (current s) using b1 is-a-port-of-partition-def p3 by auto
      have b3:s' = s by (simp add: b2 p4 write-sampling-message-def)
      then show ?thesis by auto
    next
      assume c0:p ≠ pid
      show ?thesis
        using p4 apply(clarsimp simp:write-sampling-message-def update-sampling-port-msg-def)
        apply(case-tac ports (comm s) pid)
```

```
      apply simp
      apply(case-tac a)
      using c0 by auto
    qed
  }
  then show ?thesis by (simp add: p4)
  qed
 } then show ?thesis by auto
 qed


lemma wrt-smpl-msg-notchg-comminotherdom:
 assumes  p0:reachable0 s
    and  p1:is-a-partition sysconf (current s)
  and  p3:(current s) ≠ d
  and  p4:s′ = fst (write-sampling-message s pid m)
 shows  vpeq-part-comm s d s′
 proof−
  from p4 have r0: part-ports s = part-ports s′
   apply(clarsimp simp:write-sampling-message-def update-sampling-port-msg-def)
    apply(case-tac ports (comm s) pid)
    apply simp
    apply(case-tac a)
    by auto
  then have get-ports-of-partition s d = get-ports-of-partition s′ d
      using part-ports-imp-portofpart by blast
  moreover have ∀ p. p ∈ get-ports-of-partition s d ⟶
          is-a-queuingport s p = is-a-queuingport s′ p ∧
          is-dest-port s p = is-dest-port s′ p ∧
          (if is-dest-port s p then get-port-buf-size s p = get-port-buf-size s′ p else True)
   using is-a-queuingport-def get-port-buf-size-def
          is-dest-port-def get-port-byid-def p1 p3 p4 wrt-smpl-msg-notchg-portinotherdom by auto
   ultimately show ?thesis by auto
 qed

 lemma wrt-smpl-msg-sm-nitfpart:⟦reachable0 s; is-a-partition sysconf (current s); is-a-partition sysconf d;
                              ((current s) \⇝ d); s′ = fst (write-sampling-message s pid m)⟧
                                 ⟹ (s ∼ d ∼ s′)
 using  trans-imp-not-part sche-imp-not-part
 apply(clarsimp cong del: is-a-partition-def interference1-def non-interference1-def vpeq-part-comm-def)
 by (metis nintf-neq  wrt-smpl-msg-notchg-comminotherdom wrt-smpl-msg-notchg-partstate)
```

```
lemma write-smpl-msg-presrv-lcrsp:
assumes p0:reachable0 s
  and   p1:is-a-partition sysconf (current s)
  and   p2:(current s) \⇝ d
  and   p3:s' = fst (write-sampling-message s pid m)
shows   s ∼ d ∼ s'
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  then show ?thesis using is-a-scheduler-def wrt-smpl-msg-sm-sche[OF p1 p3] by auto
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
  proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    show ?thesis using b0 wrt-smpl-msg-sm-nitfpart p0 p1 p2 p3 by blast
  next
    assume b1:¬ is-a-partition sysconf d
    show ?thesis
    proof(cases is-a-transmitter sysconf d)
      assume c0:is-a-transmitter sysconf d
      have comm s = comm s' ∧  part-ports s = part-ports s'
      proof(rule conjI)
      {
        from p1 p2 have ¬ part-intf-transmitter sysconf (current s)
          using interference1-def by (smt a1 c0 non-interference1-def)
        then have d1:get-partition-cfg-ports (the ((partconf sysconf) (current s))) = {}
          using get-partition-cfg-ports-byid-def  p1 port-partition by fastforce
        then have d2:get-partition-cfg-ports-byid sysconf (current s) = {}
          by (simp add: get-partition-cfg-ports-byid-def)
        then have ¬ is-a-port-of-partition s pid (current s)
        proof(cases (ports (comm s)) pid = None)
          assume e0:(ports (comm s)) pid = None
          from p0 have e1:port-consistent s by (simp add: port-cons-reach-state)
          with e0 have e1:(part-ports s) pid = None unfolding port-consistent-def by auto
          show ?thesis by (simp add: e1 is-a-port-of-partition-def)
        next
          assume e0:¬((ports (comm s)) pid = None)
          from p0 have e1:port-consistent s by (simp add: port-cons-reach-state)
          then have get-portname-by-type (the ((ports (comm s)) pid)) ∈
              get-partition-cfg-ports-byid sysconf (the ((part-ports s) pid))
              using e0 port-consistent-def by blast
          with d2 have current s ≠ the ((part-ports s) pid) by auto
          then show ?thesis using is-a-port-of-partition-def by auto
```

```
        qed
        then have d0:s = s′ by (smt write-sampling-message-def fst-conv p3)
        then show comm s = comm s′ by simp
        with d0 show part-ports s = part-ports s′ by simp
      }
      qed
      then show ?thesis using a1 b1  by auto
    next
      assume c1:¬ is-a-transmitter sysconf d
      show ?thesis using a1 b1 c1  by auto
    qed
  qed
qed

lemma write-smpl-msg-presrv-lcrsp-e: local-respect-e (hyperc (Write-Sampling-Message pid m))
  using write-smpl-msg-presrv-lcrsp  prod.simps(2)  exec-event-def
    mem-Collect-eq singletonD vpeq-reflexive-lemma
   by (auto cong del: vpeq1-def)
```

2.5.3 proving "read sampling message" satisfying the "local respect" property

```
lemma read-smpl-msg-presrv-lcrsp:
    assumes p0:reachable0 s
      and   p1:is-a-partition sysconf (current s)
      and   p2:(current s) \⇝ d
      and   p3:s′ = fst (read-sampling-message s pid)
    shows   s ∼ d ∼ s′
 using vpeq-reflexive-lemma  p3 read-sampling-message-def by auto


lemma read-smpl-msg-presrv-lcrsp-e:
  local-respect-e (hyperc (Read-Sampling-Message pid))
  using read-smpl-msg-presrv-lcrsp  exec-event-def
     prod.simps(2) vpeq-reflexive-lemma
  by (auto cong del:  vpeq1-def)
```

2.5.4 proving "get sampling portid" satisfying the "local respect" property

```
lemma get-smpl-pid-presrv-lcrsp:
    assumes p0:reachable0 s
      and   p1:is-a-partition sysconf (current s)
      and   p2:(current s) \⇝ d
      and   p3:s′ = fst (get-sampling-port-id sysconf s pname)
    shows   s ∼ d ∼ s′
```

using *p3 get-sampling-port-id-def vpeq-reflexive-lemma* **by** *auto*

lemma *get-smpl-pid-presrv-lcrsp-e*: *local-respect-e* (*hyperc* (*Get-Sampling-Portid pid*))
using *get-smpl-pid-presrv-lcrsp exec-event-def*
prod.simps(2) *vpeq-reflexive-lemma*
by (*auto cong del*: *vpeq1-def*)

2.5.5 proving "get sampling port status" satisfying the "local respect" property

lemma *get-smpl-psts-presrv-lcrsp*:
assumes *p0*:*reachable0 s*
and *p1*:*is-a-partition sysconf* (*current s*)
and *p2*:(*current s*) $\backslash\rightsquigarrow$ *d*
and *p3*:$s' =$ *fst* (*get-sampling-port-status sysconf s pid*)
shows $s \sim d \sim s'$
using *p3 get-sampling-port-status-def vpeq-reflexive-lemma* **by** *auto*

lemma *get-smpl-psts-presrv-lcrsp-e*: *local-respect-e* (*hyperc* (*Get-Sampling-Portstatus pid*))
using *get-smpl-psts-presrv-lcrsp exec-event-def*
prod.simps(2) *vpeq-reflexive-lemma* **by** (*auto cong del*: *vpeq1-def*)

2.5.6 proving "create queuing port" satisfying the "local respect" property

lemma *crt-que-port-notchg-current*:
⟦*is-a-partition sysconf* (*current s*); $s' =$ *fst* (*create-queuing-port sysconf s pname*)⟧
$\Longrightarrow$ *current s* = *current* s'
by (*clarsimp simp*:*create-queuing-port-def*)

the state before and after executing the action "create queuing port" is observ equal to scheduler

lemma *crt-que-port-sm-sche*:⟦*is-a-partition sysconf* (*current s*);
$s' =$ *fst* (*create-queuing-port sysconf s pname*)⟧
$\Longrightarrow$ ($s\sim$(*scheduler sysconf*)$\sim s'$)
using *crt-que-port-notchg-current part-imp-not-sch* **by** *fastforce*

lemma *crt-que-port-notchg-partstate*:
⟦*is-a-partition sysconf* (*current s*); *is-a-partition sysconf d*;
$s' =$ *fst* (*create-queuing-port sysconf s pname*)⟧
$\Longrightarrow$ (*partitions s*) *d* = (*partitions* s') *d*
by (*clarsimp simp*:*create-queuing-port-def*)

```
lemma crt-que-port-notchg-partportsinotherdom:
assumes p0:reachable0 s
  and   p1:is-a-partition sysconf (current s)
  and   p3:(current s) ≠ d
  and   p4:s' = fst (create-queuing-port sysconf s pname)
shows   get-ports-of-partition s d = get-ports-of-partition s' d
proof −
{
  have ∀ p. p∈get-ports-of-partition s d ⟶ p∈get-ports-of-partition s' d
  proof−
  {
    fix p
    assume a0:p∈get-ports-of-partition s d
    have a1:p∈get-ports-of-partition s' d
    proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
      assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
      have b1:p ≠ get-portid-in-type (the (get-queuingport-conf sysconf pname))
        using b0 port-partition by auto
      then show ?thesis using b0 port-partition by auto
    next
      assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
      then have c1:s' = s by (simp add: create-queuing-port-def p4)
      then show ?thesis by (simp add: a0)
    qed
  }
  then show ?thesis by auto
  qed
  moreover
  have ∀ p. p∈get-ports-of-partition s' d ⟶ p∈get-ports-of-partition s d
  proof−
  {
    fix p
    assume a0:p∈get-ports-of-partition s' d
    have p∈get-ports-of-partition s d
    proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
      assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
      have b1:p ≠ get-portid-in-type (the (get-queuingport-conf sysconf pname))
        using b0 port-partition by auto
      then show ?thesis using b0 port-partition by auto
    next
      assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
      then have c1:s' = s by (simp add: create-queuing-port-def p4)
      then show ?thesis using a0 by auto
```

```
    qed
   }
   then show ?thesis by auto
   qed
   then show ?thesis using calculation by blast
  }
  qed

lemma crt-que-port-notchg-portsinotherdom:
assumes p1:is-a-partition sysconf (current s)
  and  p3:(current s) ≠ d
  and  p4:s′ = fst (create-queuing-port sysconf s pname)
shows ∀ p. p∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
proof −
{
  fix p
  assume a0:p ∈ get-ports-of-partition s d
  have ports (comm s) p = ports (comm s′) p
  proof −
  {
    have ports (comm s) p = ports (comm (fst (create-queuing-port sysconf s pname))) p
    proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
      assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
      have b1:p ≠ get-portid-in-type (the (get-queuingport-conf sysconf pname))
        using b0 port-partition by auto
      then show ?thesis using b0 port-partition by auto
    next
      assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
      then have c1:s′ = s by (simp add: create-queuing-port-def p4)
      then show ?thesis using p4 by auto
    qed
  }
  then show ?thesis by (simp add: p4)
  qed
} then show ?thesis by auto
qed


lemma crt-que-port-notchg-comminotherdom:
assumes  p0:reachable0 s
    and  p1:is-a-partition sysconf (current s)
  and  p3:(current s) ≠ d
  and  p4:s′ = fst (create-queuing-port sysconf s pname)
```

shows *vpeq-part-comm s d s'*
proof−
have *get-ports-of-partition s d* = *get-ports-of-partition s' d*
using *crt-que-port-notchg-partportsinotherdom p0 p1 p3 p4* **by** *auto*
also have $\forall p.\ p \in$ *get-ports-of-partition s d* $\longrightarrow$
is-a-queuingport s p = *is-a-queuingport s' p* $\wedge$
is-dest-port s p = *is-dest-port s' p* $\wedge$
(*if is-dest-port s p then get-port-buf-size s p* = *get-port-buf-size s' p else True*)
proof −
{
fix *p*
have $p \in$ *get-ports-of-partition s d* $\longrightarrow$
is-a-queuingport s p = *is-a-queuingport s' p* $\wedge$
is-dest-port s p = *is-dest-port s' p* $\wedge$
(*if is-dest-port s p then get-port-buf-size s p* = *get-port-buf-size s' p else True*)
using *get-port-buf-size-def is-a-queuingport-def*
is-dest-port-def
crt-que-port-notchg-portsinotherdom get-port-byid-def p1 p3 p4 **by** *auto*
}
then show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *crt-que-port-sm-nitfpart*:⟦*reachable0 s*; *is-a-partition sysconf* (*current s*); *is-a-partition sysconf d*;
((*current s*) $\setminus\!\leadsto$ *d*); *s'* = *fst* (*create-queuing-port sysconf s pname*)⟧
$\Longrightarrow$ ($s \sim d \sim s'$)
apply(*clarsimp simp*:*vpeq1-def cong del*: *is-a-partition-def vpeq-part-comm-def*)
using *trans-imp-not-part sche-imp-not-part*
apply (*simp add*: *crt-que-port-notchg-partstate*)
by (*metis create-queuing-port-def fst-conv get-queuingport-conf-def port-name-diff*)

lemma *crt-que-port-presrv-lcrsp*:
assumes *p0*:*reachable0 s*
and *p1*:*is-a-partition sysconf* (*current s*)
and *p2*:(*current s*) $\setminus\!\leadsto$ *d*
and *p3*:*s'* = *fst* (*create-queuing-port sysconf s pname*)
shows $s \sim d \sim s'$
proof(*cases is-a-scheduler sysconf d*)
assume *a0*:*is-a-scheduler sysconf d*
show *?thesis* **using** *a0 crt-que-port-sm-sche*[*OF p1 p3*] **by** *auto*
next

```
assume a1:¬ is-a-scheduler sysconf d
show ?thesis
proof(cases is-a-partition sysconf d)
  assume b0:is-a-partition sysconf d
  show ?thesis using b0 crt-que-port-sm-nitfpart p0 p1 p2 p3 by blast
next
  assume b1:¬ is-a-partition sysconf d
  show ?thesis
  proof(cases is-a-transmitter sysconf d)
    assume c0:is-a-transmitter sysconf d
    have vpeq-transmitter s d s′ unfolding vpeq-transmitter-def
    proof−
      show comm s = comm s′ ∧  part-ports s = part-ports s′
      proof(rule conjI)
      {
        show comm s = comm s′
        proof −
        {
        from p1 p2 have ¬ part-intf-transmitter sysconf (current s)
          using interference1-def by (smt a1 c0 non-interference1-def)
        then have get-partition-cfg-ports (the ((partconf sysconf) (current s))) = {}
          using get-partition-cfg-ports-byid-def is-a-partition-def p1 port-partition by fastforce
        then have pname ∉ get-partition-cfg-ports-byid sysconf (current s)
          by (simp add: get-partition-cfg-ports-byid-def)
        then have s = s′ by (simp add: create-queuing-port-def p3)
        }
        then show ?thesis by auto
        qed
        show part-ports s = part-ports s′
        proof −
        {
          from p1 p2 c0 have d0:¬ part-intf-transmitter sysconf (current s)
            using interference1-def non-interference1-def by (meson a1)
          then have d1:get-partition-cfg-ports-byid sysconf (current s) = {} using port-partition by blast
          then have d2:s=s′ by (smt create-queuing-port-def empty-iff fst-conv p3)
        }
        then show ?thesis by auto
        qed
      }
      qed
    qed

    then show ?thesis using a1 b1 is-a-scheduler-def vpeq1-def by auto
```

```
      next
        assume c1:¬ is-a-transmitter sysconf d
        show ?thesis using a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def by auto
      qed
    qed
  qed

lemma crt-que-port-presrv-lcrsp-e: local-respect-e (hyperc (Create-Queuing-Port p))
  using crt-que-port-presrv-lcrsp  exec-event-def mem-Collect-eq
     prod.simps(2) singletonD vpeq-reflexive-lemma
  by (auto cong del: is-a-partition-def  vpeq1-def)
```

2.5.7 proving "send queuing message(may lost)" satisfying the "local respect" property

```
lemma snd-que-msg-lst-notchg-current:
  ⟦is-a-partition sysconf (current s); s′ = fst (send-queuing-message-maylost sysconf s pid m)⟧
    ⟹ current s = current s′
    apply (simp add: insert-msg2queuing-port-def
              send-queuing-message-maylost-def replace-msg2queuing-port-def)
    apply(case-tac ports (comm s) pid)
    apply simp
    apply(case-tac a)
    by auto


lemma snd-que-msg-lst-sm-sche:⟦is-a-partition sysconf (current s);
                              s′ = fst (send-queuing-message-maylost sysconf s pid m)⟧
                                  ⟹ (s∼(scheduler sysconf)∼s′)
  apply (auto simp add: insert-msg2queuing-port-def vpeq-reflexive-lemma
              replace-msg2queuing-port-def send-queuing-message-maylost-def)
  apply(case-tac ports (comm s) pid)
  apply (simp add: vpeq-reflexive-lemma)
  apply(case-tac a)
  by (auto simp add: vpeq-reflexive-lemma)

lemma snd-que-msg-lst-notchg-partstate:
              ⟦is-a-partition sysconf (current s); is-a-partition sysconf d;
              s′ = fst (send-queuing-message-maylost sysconf s pid m)⟧
                                    ⟹ (partitions s) d = (partitions s′) d

apply(clarsimp simp:insert-msg2queuing-port-def
       replace-msg2queuing-port-def send-queuing-message-maylost-def)
apply(case-tac ports (comm s) pid)
apply simp
```

```
  apply(case-tac a)
  by auto

lemma snd-que-msg-lst-notchg-partports:
assumes p1:is-a-partition sysconf (current s)
  and   p2:s' = fst (send-queuing-message-maylost sysconf s pid m)
shows   part-ports s = part-ports s'
proof(cases ¬ is-a-queuingport s pid
                      ∨ ¬ is-source-port s pid
                      ∨ ¬ is-a-port-of-partition s pid (current s))
  assume b0:¬ is-a-queuingport s pid
              ∨ ¬ is-source-port s pid
              ∨ ¬ is-a-port-of-partition s pid (current s)
  with p2 show ?thesis using send-queuing-message-maylost-def by auto
next
  assume b1:¬(¬ is-a-queuingport s pid
              ∨ ¬ is-source-port s pid
              ∨ ¬ is-a-port-of-partition s pid (current s))
  show ?thesis
  proof(cases is-full-portqueuing sysconf s pid)
    assume c0:is-full-portqueuing sysconf s pid
    with b1 have c1:s' = s by (simp add: p2 replace-msg2queuing-port-def
                                  send-queuing-message-maylost-def)
    then show ?thesis by auto
  next
    assume d0:¬ is-full-portqueuing sysconf s pid
    have d1:s' = insert-msg2queuing-port s pid m
      using b1 d0 p2 send-queuing-message-maylost-def by auto
    with b1 show ?thesis
      proof(induct (ports (comm s)) pid)
        case None show ?case using None.hyps d1 insert-msg2queuing-port-def by auto
      next
        case (Some x)
        have e0:(ports (comm s)) pid = Some x by (simp add: Some.hyps)
        show ?case
        proof(induct the ((ports (comm s)) pid))
          case (Queuing x1 x2 x3 x4 x5)
          show ?case by (smt Communication-State.select-convs(1) Communication-State.surjective
            Communication-State.update-convs(2) Port-Type.simps(5) Queuing.hyps
            State.select-convs(3) State.select-convs(4) State.surjective State.update-convs(3)
            d1 insert-msg2queuing-port-def option.case-eq-if)
        next
          case (Sampling x1 x2 x3 x4)
```

```
        show ?case using Sampling.hyps d1 e0 insert-msg2queuing-port-def by auto
      qed
    qed
  qed
qed

lemma snd-que-msg-lst-notchg-portsinotherdom:
assumes p1:is-a-partition sysconf (current s)
  and   p3:(current s) ≠ d
  and   p4:s′ = fst (send-queuing-message-maylost sysconf s pid m)
shows ∀ p. p∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
proof –
{
  fix p
  have p ∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
  proof –
  {
    assume a0:p ∈ get-ports-of-partition s d
    have a1:(part-ports s) p = Some d using a0 get-ports-of-partition-def by auto
    have ports (comm s) p = ports (comm s′) p
    proof(cases p = pid)
      assume b0:p = pid
      have b1:(part-ports s) pid = Some d using a1 b0 by auto
      have b2:¬ is-a-port-of-partition s pid (current s) using b1 is-a-port-of-partition-def p3 by auto
      have b3:s′ = s by (simp add: b2 p4 send-queuing-message-maylost-def)
      then show ?thesis by auto
    next
      assume c0:p ≠ pid
      show ?thesis
      proof(cases ¬ is-a-queuingport s pid
                  ∨ ¬ is-source-port s pid
                  ∨ ¬ is-a-port-of-partition s pid (current s))
        assume b0:¬ is-a-queuingport s pid
                  ∨ ¬ is-source-port s pid
                  ∨ ¬ is-a-port-of-partition s pid (current s)
        show ?thesis using a1 b0 p4 send-queuing-message-maylost-def by auto
      next
        assume b1:¬(¬ is-a-queuingport s pid
                  ∨ ¬ is-source-port s pid
                  ∨ ¬ is-a-port-of-partition s pid (current s))
        show ?thesis
        proof(cases is-full-portqueuing sysconf s pid)
          assume c0:is-full-portqueuing sysconf s pid
```

```
          with b1 have c1:s' = s by (simp add: p4 replace-msg2queuing-port-def
                                      send-queuing-message-maylost-def)
          then show ?thesis using a1 by auto
        next
          assume d0:¬ is-full-portqueuing sysconf s pid
          have d1:s' = insert-msg2queuing-port s pid m
            using b1 d0 p4 send-queuing-message-maylost-def by auto
          with b1 show ?thesis
            proof(induct (ports (comm s)) pid)
              case None show ?case
                by (simp add: None.hyps d1 insert-msg2queuing-port-def option.case-eq-if)
            next
              case (Some x)
              have e0:(ports (comm s)) pid = Some x by (simp add: Some.hyps)
              show ?case
              proof(induct the ((ports (comm s)) pid))
                case (Queuing x1 x2 x3 x4 x5)
                have f0:the ((ports (comm s)) pid) = Queuing x1 x2 x3 x4 x5
                  by (simp add: Queuing.hyps)
                show ?case by (smt Communication-State.ext-inject Communication-State.surjective
                  Communication-State.update-convs(1) Port-Type.simps(5) State.select-convs(3)
                  State.surjective State.update-convs(3) c0 d1 f0 fun-upd-other
                  insert-msg2queuing-port-def option.case-eq-if)
              next
                case (Sampling x1 x2 x3 x4)
                 have f0:the ((ports (comm s)) pid) = Sampling x1 x2 x3 x4
                  by (simp add: Sampling)
                show ?case using d1 e0 f0 insert-msg2queuing-port-def by auto
              qed
            qed
         qed
        qed
      qed
    }
    then show ?thesis by (simp add: p4)
    qed
  }
  then show ?thesis by auto
  qed

  lemma get-port-size-eq:
  assumes a0: p ≠ pid
  shows get-port-buf-size s p = get-port-buf-size (fst (send-queuing-message-maylost sysconf s pid m)) p
```

```
 apply (simp add: insert-msg2queuing-port-def replace-msg2queuing-port-def send-queuing-message-maylost-def)
 apply(case-tac ports (comm s) pid)
 apply simp
 apply(case-tac a)
 using a0 get-port-byid-def get-port-buf-size-def by auto

lemma snd-que-msg-lst-notchg-comminotherdom:
assumes p0:reachable0 s
  and p1:is-a-partition sysconf (current s)
  and p3:(current s) ≠ d
  and p4:s' = fst (send-queuing-message-maylost sysconf s pid m)
shows vpeq-part-comm s d s'
proof−
  from p4 have r0:part-ports s = part-ports s' using p1 snd-que-msg-lst-notchg-partports by blast
  then have get-ports-of-partition s d = get-ports-of-partition s' d
   using part-ports-imp-portofpart by blast
  also have ∀ p. p ∈ get-ports-of-partition s d ⟶
          is-a-queuingport s p = is-a-queuingport s' p ∧
          is-dest-port s p = is-dest-port s' p ∧
          (if is-dest-port s p then get-port-buf-size s p = get-port-buf-size s' p else True)
  proof −
  {
   fix p
   have p ∈ get-ports-of-partition s d ⟶
      is-a-queuingport s p = is-a-queuingport s' p ∧
      is-dest-port s p = is-dest-port s' p ∧
      (if is-dest-port s p then get-port-buf-size s p = get-port-buf-size s' p else True)
   proof(rule impI)
   {
     assume a0:p ∈ get-ports-of-partition s d
     have is-a-queuingport s p = is-a-queuingport s' p
       unfolding is-a-queuingport-def using snd-que-msg-lst-notchg-portsinotherdom
          a0 p1 p3 p4 interference1-def non-interference1-def by auto
     moreover have is-dest-port s p = is-dest-port s' p ∧
          (if is-dest-port s p then get-port-buf-size s p = get-port-buf-size s' p else True)
     proof(rule conjI)
     {
       show is-dest-port s p = is-dest-port s' p
        unfolding is-dest-port-def using snd-que-msg-lst-notchg-portsinotherdom
          a0 p1 p3 p4 interference1-def non-interference1-def by smt
       show if is-dest-port s p then get-port-buf-size s p = get-port-buf-size s' p else True
       proof −
       {
```

```
      assume c0:is-dest-port s p
      have get-port-buf-size s p = get-port-buf-size s' p
      proof(cases p = pid)
        assume d0:p = pid
        with c0 have is-dest-port s pid by simp
        then have d1:¬ is-source-port s pid by (simp add: dest-port-not-src)
        with p4 have s' = s unfolding send-queuing-message-maylost-def by simp
        then show ?thesis by simp
      next
        assume d0: p ≠ pid
        with p4 get-port-size-eq show ?thesis by simp
      qed
    } then show ?thesis by auto
      qed
    }
    qed

    ultimately  show is-a-queuingport s p = is-a-queuingport s' p ∧
          is-dest-port s p = is-dest-port s' p ∧
          (if is-dest-port s p then get-port-buf-size s p = get-port-buf-size s' p else True)
    by auto
  } qed
  }
  then show ?thesis by auto qed
  ultimately show ?thesis by auto
qed

lemma snd-que-msg-lst-sm-nitfpart:⟦reachable0 s; is-a-partition sysconf (current s); is-a-partition sysconf d;
                        ((current s) \↝ d); s' = fst (send-queuing-message-maylost sysconf s pid m)⟧
                            ⟹ (s ∼ d ∼ s')
  apply(clarsimp cong del: is-a-partition-def)
  apply(rule conjI)
  using trans-imp-not-part  trans-imp-not-part apply fastforce
  apply(rule impI)
  apply(rule conjI)
  using  sche-imp-not-part apply fastforce
  apply(rule impI)
  apply(rule conjI)
  apply (simp add: snd-que-msg-lst-notchg-partstate)
  by (meson snd-que-msg-lst-notchg-comminotherdom vpeq-part-comm-def)
```

```
lemma snd-que-msg-lst-presrv-lcrsp:
assumes p0:reachable0 s
  and   p1:is-a-partition sysconf (current s)
  and   p2:(current s) \⇝ d
  and   p3:s' = fst (send-queuing-message-maylost sysconf s pid m)
shows   s ∼ d ∼ s'
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  show ?thesis using a0 is-a-scheduler-def snd-que-msg-lst-sm-sche[OF p1 p3] by auto
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
  proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    show ?thesis using b0 snd-que-msg-lst-sm-nitfpart p0 p1 p2 p3 by blast
  next
    assume b1:¬ is-a-partition sysconf d
    show ?thesis
    proof(cases is-a-transmitter sysconf d)
      assume c0:is-a-transmitter sysconf d
      show ?thesis
      proof −
      {
        have comm s = comm s' ∧ part-ports s = part-ports s'
        proof(rule conjI)
        {
          from p1 p2 have ¬ part-intf-transmitter sysconf (current s)
            using interference1-def by (smt a1 c0 non-interference1-def)
          then have d1:get-partition-cfg-ports (the ((partconf sysconf) (current s))) = {}
            using get-partition-cfg-ports-byid-def is-a-partition-def p1 port-partition by fastforce
          then have d2:get-partition-cfg-ports-byid sysconf (current s) = {}
            by (simp add: get-partition-cfg-ports-byid-def)
          then have ¬ is-a-port-of-partition s pid (current s)
          proof(cases (ports (comm s)) pid = None)
            assume e0:(ports (comm s)) pid = None
            then show ?thesis using port-cons-reach-state[OF p0]
              port-consistent-def is-a-port-of-partition-def by auto
          next
            assume e0:¬((ports (comm s)) pid = None)
            then show ?thesis using port-cons-reach-state[OF p0]
              port-consistent-def d2 is-a-port-of-partition-def by auto
          qed
          then have d0:s = s' by (auto simp add: send-queuing-message-maylost-def p3)
```

```
          then show comm s = comm s' by simp
          with d0 show part-ports s = part-ports s' by simp
        }
        qed
      }
      then show ?thesis using a1 b1 by auto
      qed
    next
      assume c1:¬ is-a-transmitter sysconf d
      show ?thesis using a1 b1 c1 by auto
    qed
  qed
qed

lemma snd-que-msg-lst-presrv-lcrsp-e: local-respect-e (hyperc (Send-Queuing-Message p m))
  using snd-que-msg-lst-presrv-lcrsp  exec-event-def mem-Collect-eq
      prod.simps(2) singletonD vpeq-reflexive-lemma
  by (auto cong del: is-a-partition-def   vpeq1-def)
```

2.5.8 proving "receive queuing message" satisfying the "local respect" property

```
lemma rec-que-msg-notchg-current:
    ⟦is-a-partition sysconf (current s); s' = fst (receive-queuing-message s pid)⟧
      ⟹ current s = current s'
      apply(clarsimp simp:receive-queuing-message-def remove-msg-from-queuingport-def)
      apply(case-tac ports (comm s) pid)
      apply simp
      apply(case-tac a)
      by auto


  lemma rec-que-msg-sm-sche:⟦is-a-partition sysconf (current s);
                              s' = fst (receive-queuing-message s pid)⟧
                                 ⟹ (s∼(scheduler sysconf)∼s')
     apply(clarsimp simp:receive-queuing-message-def remove-msg-from-queuingport-def cong del:  vpeq1-def)
      apply(case-tac ports (comm s) pid)
      apply (simp add: vpeq-reflexive-lemma cong del: vpeq1-def)
      apply(case-tac a)
      using vpeq-reflexive-lemma by auto

  lemma rec-que-msg-notchg-partstate:
              ⟦is-a-partition sysconf (current s); is-a-partition sysconf d;
              s' = fst (receive-queuing-message s pid)⟧
                                  ⟹ (partitions s) d = (partitions s') d
```

```
    apply(clarsimp simp:receive-queuing-message-def remove-msg-from-queuingport-def)
    apply(case-tac ports (comm s) pid)
    apply simp
    apply(case-tac a)
    by (auto simp add: vpeq-reflexive-lemma)

lemma rec-que-msg-notchg-partports:
assumes p1:is-a-partition sysconf (current s)
  and   p2:s′ = fst (receive-queuing-message s pid)
shows   part-ports s = part-ports s′

proof(cases (¬ is-a-queuingport s pid
                ∨ ¬ is-a-port-of-partition s pid (current s)
                ∨ ¬ is-dest-port s pid
                ∨ is-empty-portqueuing s pid))
  assume b0:(¬ is-a-queuingport s pid
            ∨ ¬ is-a-port-of-partition s pid (current s)
            ∨ ¬ is-dest-port s pid
            ∨ is-empty-portqueuing s pid)
  show ?thesis using b0 p2 receive-queuing-message-def by auto
next
  assume b1:¬(¬ is-a-queuingport s pid
            ∨ ¬ is-a-port-of-partition s pid (current s)
            ∨ ¬ is-dest-port s pid
            ∨ is-empty-portqueuing s pid)
  have b2:s′ = fst (remove-msg-from-queuingport s pid)
    using b1 p2 receive-queuing-message-def by auto
  then show ?thesis
  proof(induct (ports (comm s)) pid)
    case None show ?case using None.hyps b2 remove-msg-from-queuingport-def by auto
  next
    case (Some x)
    have e0:(ports (comm s)) pid = Some x by (simp add: Some.hyps)
    show ?case
      proof(induct the ((ports (comm s)) pid))
        case (Queuing x1 x2 x3 x4 x5)
        show ?case by (smt Port-Type.simps(5) Queuing.hyps State.select-convs(4)
          State.surjective State.update-convs(3) b2 eq-fst-iff option.case-eq-if
          remove-msg-from-queuingport-def)
      next
        case (Sampling x1 x2 x3 x4)
        show ?case using Sampling.hyps b2 e0 remove-msg-from-queuingport-def by auto
      qed
```

```
  qed
qed

lemma rec-que-msg-notchg-portsinotherdom:
assumes p1:is-a-partition sysconf (current s)
  and   p3:(current s) ≠ d
  and   p4:s′ = fst (receive-queuing-message s pid)
shows ∀ p. p∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
proof –
{
  show ?thesis
  proof –
  {
    fix p
    have p ∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
    proof –
    {
      assume a0:p ∈ get-ports-of-partition s d
      have a1:(part-ports s) p = Some d using a0 get-ports-of-partition-def by auto
      have ports (comm s) p = ports (comm s′) p
      proof(cases p = pid)
        assume b0:p = pid
        have b1:(part-ports s) pid = Some d using a1 b0 by auto
        have b2:¬ is-a-port-of-partition s pid (current s) using b1 is-a-port-of-partition-def p3 by auto
        have b3:s′ = s by (simp add: b2 p4 receive-queuing-message-def)
        then show ?thesis by auto
      next
        assume c0:p ≠ pid
        show ?thesis
        proof(induct (ports (comm s)) pid)
          case None show ?case
            by (simp add: None.hyps is-dest-port-def option.case-eq-if p4 receive-queuing-message-def)
        next
          case (Some x)
          have e0:(ports (comm s)) pid = Some x by (simp add: Some.hyps)
          show ?case
          proof(induct the ((ports (comm s)) pid))
            case (Queuing x1 x2 x3 x4 x5)
            have f0:the ((ports (comm s)) pid) = Queuing x1 x2 x3 x4 x5
              by (simp add: Queuing.hyps)
            show ?case  by (smt Communication-State.ext-inject Communication-State.surjective
              Communication-State.update-convs(1) Port-Type.simps(5) State.select-convs(3)
              State.surjective State.update-convs(3) c0 f0 fst-conv fun-upd-other
```

option.case-eq-if p4 receive-queuing-message-def remove-msg-from-queuingport-def)
next
case (*Sampling x1 x2 x3 x4*)
have *f0*:*the* ((*ports* (*comm s*)) *pid*) = *Sampling x1 x2 x3 x4*
by (*simp add*: *Sampling*)
show *?case* **using** *e0 f0 p4 receive-queuing-message-def*
remove-msg-from-queuingport-def **by** *auto*
qed
qed
qed
}
then show *?thesis* **by** (*simp add*: *p4*)
qed
}
then show *?thesis* **by** *auto*
qed
}
qed

lemma *rec-que-msg-notchg-comminotherdom*:
assumes *p0*:*reachable0 s*
and *p1*:*is-a-partition sysconf* (*current s*)
and *p3*:(*current s*) $\neq$ *d*
and *p4*:*s'* = *fst* (*receive-queuing-message s pid*)
shows *vpeq-part-comm s d s'*
proof –
from *p4* **have** *r0*:*part-ports s* = *part-ports s'* **using** *p1 rec-que-msg-notchg-partports* **by** *simp*
then have *get-ports-of-partition s d* = *get-ports-of-partition s' d*
using *part-ports-imp-portofpart* **by** *blast*
also have $\forall$ *p*. *p* $\in$ *get-ports-of-partition s d* $\longrightarrow$
is-a-queuingport s p = *is-a-queuingport s' p* $\wedge$
is-dest-port s p = *is-dest-port s' p* $\wedge$
(*if is-dest-port s p then get-port-buf-size s p* = *get-port-buf-size s' p else True*)
using *is-a-queuingport-def is-dest-port-def get-port-buf-size-def*
rec-que-msg-notchg-portsinotherdom get-port-byid-def p1 p3 p4 **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *rec-que-msg-sm-nitfpart*:⟦*reachable0 s*; *is-a-partition sysconf* (*current s*); *is-a-partition sysconf d*;
((*current s*) \\$\leadsto$ *d*); *s'* = *fst* (*receive-queuing-message s pid*)⟧
$\Longrightarrow$ (*s* $\sim$ *d* $\sim$ *s'*)
apply(*clarsimp cong del*: *is-a-partition-def vpeq-part-comm-def*)
apply(*rule conjI*)

```
    using  trans-imp-not-part apply fastforce
    apply(rule impI)
    apply(rule conjI)
    using sche-imp-not-part apply fastforce
    apply(clarsimp simp:vpeq-part-def cong del: is-a-partition-def vpeq-part-comm-def)
    apply(rule conjI)
    apply (simp add: rec-que-msg-notchg-partstate  cong del: is-a-partition-def)
    using rec-que-msg-notchg-comminotherdom by metis

lemma rec-que-msg-presrv-lcrsp:
    assumes p0:reachable0 s
      and   p1:is-a-partition sysconf (current s)
      and   p2:(current s) \⇝ d
      and   p3:s' = fst (receive-queuing-message s pid)
    shows   s ∼ d ∼ s'
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  show ?thesis using a0 is-a-scheduler-def rec-que-msg-sm-sche[OF p1 p3] by auto
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
  proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    show ?thesis using  b0 rec-que-msg-sm-nitfpart p0 p1 p2 p3 by blast
  next
    assume b1:¬ is-a-partition sysconf d
    show ?thesis
    proof(cases is-a-transmitter sysconf d)
      assume c0:is-a-transmitter sysconf d
      show ?thesis
      proof −
      {
        have comm s = comm s' ∧  part-ports s = part-ports s'
        proof−
        {
          from p1 p2 have ¬ part-intf-transmitter sysconf (current s)
            using interference1-def by (smt a1 c0 non-interference1-def)
          then have d1:get-partition-cfg-ports (the ((partconf sysconf) (current s))) = {}
            using get-partition-cfg-ports-byid-def is-a-partition-def p1 port-partition by fastforce
          then have d2:get-partition-cfg-ports-byid sysconf (current s) = {}
            by (simp add: get-partition-cfg-ports-byid-def)
          then have ¬ is-a-port-of-partition s pid (current s)
          proof(cases (ports (comm s)) pid = None)
```

```
          assume e0:(ports (comm s)) pid = None
          thus ?thesis using port-cons-reach-state[OF p0]
            port-consistent-def is-a-port-of-partition-def by auto
        next
          assume e0:¬((ports (comm s)) pid = None)
          thus ?thesis using port-cons-reach-state[OF p0]
            d2 port-consistent-def is-a-port-of-partition-def by auto
        qed
        then show ?thesis by (auto simp add: receive-queuing-message-def p3)
      }
      qed
    }
    then show ?thesis using a1 b1 is-a-scheduler-def vpeq1-def by auto
    qed
  next
    assume c1:¬ is-a-transmitter sysconf d
    show ?thesis using a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def by auto
  qed
 qed
qed

lemma rec-que-msg-presrv-lcrsp-e: local-respect-e (hyperc (Receive-Queuing-Message p))
 using rec-que-msg-presrv-lcrsp exec-event-def prod.simps(2) vpeq-reflexive-lemma
 by (auto cong del: is-a-partition-def vpeq1-def)
```

2.5.9 proving "get queuing portid" satisfying the "local respect" property

```
lemma get-que-pid-presrv-lcrsp:
   assumes p0:reachable0 s
     and p1:is-a-partition sysconf (current s)
     and p2:(current s) \⇝ d
     and p3:s′ = fst (get-queuing-port-id sysconf s pname)
   shows s ∼ d ∼ s′
 proof −
   have a0:s′ = s by (simp add: p3 get-queuing-port-id-def)
   then show ?thesis using vpeq-reflexive-lemma by auto
 qed

lemma get-que-pid-presrv-lcrsp-e: local-respect-e (hyperc (Get-Queuing-Portid p))
 using get-que-pid-presrv-lcrsp exec-event-def prod.simps(2) vpeq-reflexive-lemma
by (auto cong del: vpeq1-def)
```

2.5.10 proving "get queuing port status" satisfying the "local respect" property

lemma *get-que-psts-presrv-lcrsp*:
  assumes *p0*:*reachable0 s*
    and *p1*:*is-a-partition sysconf* (*current s*)
    and *p2*:(*current s*) $\backslash\leadsto$ *d*
    and *p3*:*s'* = *fst* (*get-queuing-port-status sysconf s pid*)
  shows $s \sim d \sim s'$
 proof −
  have *a0*:*s'* = *s* **by** (*simp add*: *p3 get-queuing-port-status-def*)
  then show *?thesis* **using** *vpeq-reflexive-lemma* **by** *auto*
 qed

lemma *get-que-psts-presrv-lcrsp-e*: *local-respect-e* (*hyperc* (*Get-Queuing-Portstatus p*))
 using *get-que-psts-presrv-lcrsp exec-event-def prod.simps*(*2*) *vpeq-reflexive-lemma*
by (*auto cong del*: *vpeq1-def*)

2.5.11 proving "clear queuing port" satisfying the "local respect" property

lemma *clr-que-port-notchg-current*:
  ⟦*is-a-partition sysconf* (*current s*); *s'* = *clear-queuing-port s pid*⟧
   $\Longrightarrow$ *current s* = *current s'*
   by (*clarsimp simp*:*clear-queuing-port-def Let-def*)

lemma *clr-que-port-sm-sche*:⟦*is-a-partition sysconf* (*current s*);
      s' = *clear-queuing-port s pid*⟧
       $\Longrightarrow$ ($s\sim$(*scheduler sysconf*)$\sim s'$)
  by (*clarsimp simp*:*clear-queuing-port-def*)

lemma *clr-que-port-notchg-partstate*:
    ⟦*is-a-partition sysconf* (*current s*); *is-a-partition sysconf d*;
     s' = *clear-queuing-port s pid*⟧ $\Longrightarrow$ (*partitions s*) *d* = (*partitions s'*) *d*
   by (*clarsimp simp*:*clear-queuing-port-def*)

lemma *clr-que-port-notchg-partports*:
  assumes *p1*:*s'* = *clear-queuing-port s pid*
  shows *part-ports s* = *part-ports s'*
  proof(*cases* ¬ *is-a-queuingport s pid*
     ∨ ¬ *is-a-port-of-partition s pid* (*current s*)
     ∨ ¬ *is-dest-port s pid*)
   assume *b0*:¬ *is-a-queuingport s pid*
     ∨ ¬ *is-a-port-of-partition s pid* (*current s*)
     ∨ ¬ *is-dest-port s pid*

```
    then show ?thesis using p1 clear-queuing-port-def by auto
  next
    assume b1:¬(¬ is-a-queuingport s pid
          ∨ ¬ is-a-port-of-partition s pid (current s)
          ∨ ¬ is-dest-port s pid)
    with p1 show ?thesis unfolding clear-queuing-port-def Let-def by simp

  qed

lemma clr-que-port-notchg-portsinotherdom:
  assumes p1:is-a-partition sysconf (current s)
    and  p3:(current s) ≠ d
    and  p4:s′ = clear-queuing-port s pid
  shows ∀ p. p∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
proof –
 {
  fix p
  assume a0:p ∈ get-ports-of-partition s d
  have p ∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s′) p
  proof –
  {
    have a1:(part-ports s) p = Some d using a0 get-ports-of-partition-def by auto
    have ports (comm s) p = ports (comm s′) p
    proof(cases ¬ is-a-queuingport s pid
          ∨ ¬ is-a-port-of-partition s pid (current s)
          ∨ ¬ is-dest-port s pid)
      assume b0:¬ is-a-queuingport s pid
            ∨ ¬ is-a-port-of-partition s pid (current s)
            ∨ ¬ is-dest-port s pid
      with p4 have b1:s′ = s unfolding clear-queuing-port-def by auto
      then show ?thesis using a1 by auto
    next
      assume b1:¬(¬ is-a-queuingport s pid
            ∨ ¬ is-a-port-of-partition s pid (current s)
            ∨ ¬ is-dest-port s pid)
      with p4 show ?thesis unfolding clear-queuing-port-def Let-def
        using a1 is-a-port-of-partition-def p3 by auto
    qed
  }
  then show ?thesis by (simp add: p4)
  qed
 }
 then show ?thesis by auto
```

qed

lemma *clr-que-port-notchg-comminotherdom*:
assumes *p0*:*reachable0 s*
and *p1*:*is-a-partition sysconf* (*current s*)
and *p3*:(*current s*) $\neq$ *d*
and *p4*:*s'* = *clear-queuing-port s pid*
shows *vpeq-part-comm s d s'*

proof −
from *p4* **have** *r0*: *part-ports s* = *part-ports s'* **using** *clr-que-port-notchg-partports* **by** *blast*

then have *get-ports-of-partition s d* = *get-ports-of-partition s' d*
using *part-ports-imp-portofpart* **by** *blast*

moreover have $\forall$ *p*. *p* $\in$ *get-ports-of-partition s d* $\longrightarrow$
is-a-queuingport s p = *is-a-queuingport s' p* $\wedge$
is-dest-port s p = *is-dest-port s' p* $\wedge$
(*if is-dest-port s p then get-port-buf-size s p* = *get-port-buf-size s' p else True*)
using *is-dest-port-def get-port-buf-size-def is-a-queuingport-def*
clr-que-port-notchg-portsinotherdom get-port-byid-def p1 p3 p4 **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *clr-que-port-sm-nitfpart*:⟦*reachable0 s*; *is-a-partition sysconf* (*current s*); *is-a-partition sysconf d*;
((*current s*) \\$\rightsquigarrow$ *d*); *s'* = *clear-queuing-port s pid*⟧
$\Longrightarrow$ (*s* $\sim$ *d* $\sim$ *s'*)
apply(*clarsimp cong del*: *is-a-partition-def interference1-def non-interference1-def vpeq-part-comm-def*)
apply(*rule conjI*)
using *trans-imp-not-part* **apply** *fastforce*
apply(*rule impI*)
apply(*rule conjI*)
using *sche-imp-not-part* **apply** *fastforce*
apply(*clarsimp cong del*: *is-a-partition-def interference1-def non-interference1-def vpeq-part-comm-def*)
apply(*rule conjI*)
apply (*simp add*: *clr-que-port-notchg-partstate*
cong del: *vpeq-part-comm-def is-a-partition-def interference1-def non-interference1-def*)
using *clr-que-port-notchg-comminotherdom nintf-neq* **by** *blast*

lemma *clr-que-port-presrv-lcrsp*:

```
  assumes p0:reachable0 s
    and   p1:is-a-partition sysconf (current s)
    and   p2:(current s) \↝ d
    and   p3:s' = clear-queuing-port s pid
  shows   s ∼ d ∼ s'
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  show ?thesis using a0 is-a-scheduler-def clr-que-port-sm-sche[OF p1 p3] by auto
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
    proof(cases is-a-partition sysconf d)
      assume b0:is-a-partition sysconf d
      show ?thesis using clr-que-port-sm-nitfpart [OF p0 p1 b0 p2 p3] by blast
    next
      assume b1:¬ is-a-partition sysconf d
      show ?thesis
      proof(cases is-a-transmitter sysconf d)
        assume c0:is-a-transmitter sysconf d
        show ?thesis
          proof −
          {
            have vpeq-transmitter s d s' unfolding vpeq-transmitter-def
            proof−
              show comm s = comm s' ∧ part-ports s = part-ports s'
              proof(rule conjI)
              {
                from p1 p2 have ¬ part-intf-transmitter sysconf (current s)
                  using interference1-def by (smt a1 c0 non-interference1-def)
                then have d1:get-partition-cfg-ports (the ((partconf sysconf) (current s))) = {}
                  using get-partition-cfg-ports-byid-def p1 port-partition by fastforce
                then have d2:get-partition-cfg-ports-byid sysconf (current s) = {}
                  by (simp add: get-partition-cfg-ports-byid-def)
                then have ¬ is-a-port-of-partition s pid (current s)
                  proof(cases (ports (comm s)) pid = None)
                    assume e0:(ports (comm s)) pid = None
                    from p0 have e1:port-consistent s by (simp add: port-cons-reach-state)
                    with e0 have e1:(part-ports s) pid = None unfolding port-consistent-def by auto
                    show ?thesis by (simp add: e1 is-a-port-of-partition-def)
                  next
                    assume e0:¬((ports (comm s)) pid = None)
                    from p0 have e1:port-consistent s by (simp add: port-cons-reach-state)
                    then have get-portname-by-type (the ((ports (comm s)) pid)) ∈
```

```
              get-partition-cfg-ports-byid sysconf (the ((part-ports s) pid))
              using e0 port-consistent-def by blast
            with d2 have current s ≠ the ((part-ports s) pid) by auto
            then show ?thesis using is-a-port-of-partition-def by auto
          qed
        then have d0:s = s' by (smt clear-queuing-port-def fst-conv p3)
        then show comm s = comm s' by simp
        with d0 show part-ports s = part-ports s' by simp
      }
      qed
    qed
  }
  then show ?thesis using a1 b1 is-a-scheduler-def vpeq1-def by auto
  qed
next
  assume c1:¬ is-a-transmitter sysconf d
  show ?thesis using a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def by auto
qed
qed
qed

lemma clr-que-port-presrv-lcrsp-e: local-respect-e (hyperc (Clear-Queuing-Port p))
  using clr-que-port-presrv-lcrsp exec-event-def prod.simps(2) vpeq-reflexive-lemma
  by (auto cong del: vpeq1-def)
```

2.5.12 proving "get partition statue" satisfying the "local respect" property

```
lemma get-part-status-presrv-lcrsp:
    assumes p0:reachable0 s
      and p1:is-a-partition sysconf (current s)
      and p2:(current s) \⇝ d
      and p3:s' = fst (get-partition-status sysconf s)
    shows s ∼ d ∼ s'
  proof −
    have a0:s' = s by (simp add: p3 get-partition-status-def)
    then show ?thesis using vpeq-reflexive-lemma by auto
  qed

lemma get-part-status-presrv-lcrsp-e: local-respect-e (hyperc (Get-Partition-Status))
  using get-part-status-presrv-lcrsp exec-event-def prod.simps(2) vpeq-reflexive-lemma
  by (auto cong del: is-a-partition-def vpeq1-def)
```

2.5.13 proving "set partition mode" satisfying the "local respect" property

lemma *set-part-mode-notchg-current*:
$\llbracket$*is-a-partition sysconf* (*current s*); *s'* = *set-partition-mode sysconf s m*$\rrbracket$
$\Longrightarrow$ *current s* = *current s'*
apply(*clarsimp simp*:*set-partition-mode-def*)
done

lemma *set-part-mode-sm-sche*:$\llbracket$*is-a-partition sysconf* (*current s*);
s' = *set-partition-mode sysconf s m*$\rrbracket$
$\Longrightarrow$ (*s*$\sim$(*scheduler sysconf*)$\sim$*s'*)
using *set-part-mode-notchg-current* *part-imp-not-sch* **by** *fastforce*

lemma *set-part-mode-notchg-partstate-inotherdom*:
$\llbracket$*is-a-partition sysconf* (*current s*); *is-a-partition sysconf d*; *current s* $\neq$ *d*;
s' = *set-partition-mode sysconf s m*$\rrbracket$
$\Longrightarrow$ (*partitions s*) *d* = (*partitions s'*) *d*
apply(*clarsimp simp*:*set-partition-mode-def*)
done

lemma *set-part-mode-notchg-port*:
$\llbracket$*is-a-partition sysconf* (*current s*); *s'* = *set-partition-mode sysconf s m*$\rrbracket$
$\Longrightarrow$ $\forall$ *p*. *p*$\in$ *get-ports-of-partition s d* $\longrightarrow$ *ports* (*comm s*) *p* = *ports* (*comm s'*) *p*
apply(*clarsimp simp*:*set-partition-mode-def*)
done

lemma *set-part-mode-notchg-partports*:
$\llbracket$*is-a-partition sysconf* (*current s*); *s'* = *set-partition-mode sysconf s m*$\rrbracket$$\Longrightarrow$
part-ports s = *part-ports s'*
apply(*clarsimp simp*:*set-partition-mode-def*)
done

lemma *set-part-mode-notchg-comm*:
assumes *p0*:*reachable0 s*
and *p1*:*is-a-partition sysconf* (*current s*)
and *p3*:(*current s*) $\neq$ *d*
and *p4*:*s'* = *set-partition-mode sysconf s m*
shows *vpeq-part-comm s d s'*
using *get-ports-of-partition-def no-cfgport-impl-noports p0 p1 p4*
port-partition set-part-mode-notchg-partports **by** *fastforce*

lemma *set-part-mode-notchg-comm2*:
$\llbracket$*reachable0 s*; *is-a-partition sysconf* (*current s*); (*current s*) $\neq$ *d*; *s'* = *set-partition-mode sysconf s m*$\rrbracket$
$\Longrightarrow$ *comm s* = *comm s'*

apply(*clarsimp simp*:*set-partition-mode-def*)
done

lemma *set-part-mode-sm-nitfpart*:$[\![$*reachable0 s*; *is-a-partition sysconf* (*current s*); *is-a-partition sysconf d*;
((*current s*) $\backslash\rightsquigarrow$ *d*); $s' =$ *set-partition-mode sysconf s m*$]\!]$
$\Longrightarrow (s \sim d \sim s')$
apply(*clarsimp cong del*: *is-a-partition-def non-interference1-def vpeq-part-comm-def*)
apply(*rule conjI*)
using *is-a-transmitter-def trans-imp-not-part* **apply** *blast*
apply(*rule impI*)
apply(*rule conjI*)
using *is-a-scheduler-def sche-imp-not-part* **apply** *blast*
apply(*clarsimp simp*:*vpeq-part-def cong del*: *is-a-partition-def non-interference1-def vpeq-part-comm-def*)
apply(*rule conjI*)
using *set-part-mode-notchg-partstate-inotherdom* **apply** *fastforce*
using *set-part-mode-notchg-comm nintf-neq* **by** *blast*

lemma *set-part-mode-presrv-lcrsp*:
assumes *p0*:*reachable0 s*
and *p1*:*is-a-partition sysconf* (*current s*)
and *p2*:(*current s*) $\backslash\rightsquigarrow$ *d*
and *p3*:$s' =$ *set-partition-mode sysconf s m*
shows $s \sim d \sim s'$
proof(*cases is-a-scheduler sysconf d*)
assume *a0*:*is-a-scheduler sysconf d*
show *?thesis* **using** *a0 set-part-mode-sm-sche*[*OF p1*] *is-a-scheduler-def p3* **by** *auto*
next
assume *a1*:$\neg$ *is-a-scheduler sysconf d*
show *?thesis*
proof(*cases is-a-partition sysconf d*)
assume *b0*:*is-a-partition sysconf d*
show *?thesis* **using** *b0 set-part-mode-sm-nitfpart p0 p1 p2 p3* **by** *blast*
next
assume *b1*:$\neg$ *is-a-partition sysconf d*
show *?thesis*
proof(*cases is-a-transmitter sysconf d*)
assume *c0*:*is-a-transmitter sysconf d*
show *?thesis*
proof −
{
have *vpeq-transmitter s d s'* **unfolding** *vpeq-transmitter-def*
proof−
show *comm s* = *comm s'* $\wedge$ *part-ports s* = *part-ports s'*

```
                using set-part-mode-notchg-partports set-part-mode-notchg-comm2
                  by (metis b1 p0 p1 p3)
              qed
            }
            then show ?thesis using a1 b1 is-a-scheduler-def vpeq1-def by auto
            qed
        next
          assume c1:¬ is-a-transmitter sysconf d
          show ?thesis using a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def by auto
        qed
      qed
  qed
```

```
lemma set-part-mode-presrv-lcrsp-e: local-respect-e (hyperc (Set-Partition-Mode p))
  using set-part-mode-presrv-lcrsp  exec-event-def prod.simps(2)  vpeq-reflexive-lemma
   by (auto cong del:   vpeq1-def)
```

2.5.14 proving "schedule" satisfying the "local respect" property

```
lemma schedule-presrv-lcrsp:
    assumes p0:(scheduler sysconf) \⇝ d
    shows   s ∼ d ∼ s′
    using p0 by auto
```

```
lemma schedule-presrv-lcrsp-e: local-respect-e (sys Schedule)
  using schedule-presrv-lcrsp exec-event-def prod.simps(2) vpeq-reflexive-lemma by auto
```

2.5.15 proving "Transfer Sampling Message" satisfying the "local respect" property

```
lemma trans-smpl-msg-notchg-current:
    ⟦is-a-transmitter sysconf (current s); s′ = transf-sampling-msg s c⟧
      ⟹ current s = current s′
      apply(induct c)
      apply (clarsimp simp:update-sampling-ports-msg-def Let-def)
      by simp
```

```
lemma trans-smpl-msg-sm-sche:⟦is-a-transmitter sysconf (current s);
                            s′ = transf-sampling-msg s c⟧
                                 ⟹ (s∼(scheduler sysconf)∼s′)
   using trans-smpl-msg-notchg-current sch-not-trans vpeq1-def vpeq-sched-def by presburger
```

```
lemma trans-smpl-msg-notchg-partstate:
             ⟦is-a-transmitter sysconf (current s); is-a-partition sysconf d;
              s′ = transf-sampling-msg s c⟧ ⟹ (partitions s) d = (partitions s′) d
```

```
  apply(induct c)
  apply (clarsimp simp:transf-sampling-msg-def Let-def)
  apply (clarsimp simp:update-sampling-ports-msg-def Let-def)
  by (simp add: vpeq-reflexive-lemma)

lemma trans-smpl-msg-notchg-partports:
  s' = transf-sampling-msg s c ⟶ part-ports s = part-ports s'
  proof(induct c)
    case (Channel-Sampling name sn dns) show ?case
      proof(cases get-portid-by-name s sn≠None ∧ card (get-portids-by-names s dns) = card dns)
      {
        assume a0:get-portid-by-name s sn≠None ∧ card (get-portids-by-names s dns) = card dns
        show ?thesis unfolding transf-sampling-msg-def update-sampling-ports-msg-def Let-def by simp
      }
      then show ?thesis by fastforce
      qed
  next
    case (Channel-Queuing nm sn dn)
    show ?case by simp
  qed

lemma trans-smpl-msg-notchg-portsinotherdom:
  assumes p1:is-a-transmitter sysconf (current s)
    and   p2:reachable0 s
    and   p3:(current s) \⇝ d
    and   p4:s' = transf-sampling-msg s c
  shows ∀ p. p∈ get-ports-of-partition s d ⟶ ports (comm s) p = ports (comm s') p
  proof(cases is-a-scheduler sysconf d)
    assume a0:is-a-scheduler sysconf d
    with p2 have a3:get-ports-of-partition s d = {}
      using no-cfgport-impl-noports is-a-scheduler-def sched-hasnoports by auto
    then show ?thesis by simp
  next
    assume a1:¬ is-a-scheduler sysconf d
    show ?thesis
      proof(cases is-a-partition sysconf d)
        assume b0:is-a-partition sysconf d
        with p1 p3 have b1:¬ transmitter-intf-part sysconf d
          by (metis a1 interference1-def non-interference1-def trans-imp-not-part)
        then have b2:get-partition-cfg-ports (the ((partconf sysconf) d)) = {}
          using b0 get-partition-cfg-ports-byid-def is-a-partition-def port-partition by fastforce
        then have b3:get-partition-cfg-ports-byid sysconf d = {}
          by (simp add: get-partition-cfg-ports-byid-def)
```

```
      with p2 have b4:get-ports-of-partition s d = {} using no-cfgport-impl-noports by auto
      then show ?thesis by simp
    next
      assume b1:¬is-a-partition sysconf d
      show ?thesis
        proof(cases is-a-transmitter sysconf d)
          assume c0:is-a-transmitter sysconf d
          with p1 p3 have current s = d by (simp add: is-a-transmitter-def)
          with p3 show ?thesis using interference1-def non-interference1-def by auto
        next
          assume c1:¬ is-a-transmitter sysconf d
          with a1 b1 have c2:get-partition-cfg-ports-byid sysconf d = {}
            by (simp add: get-partition-cfg-ports-byid-def is-a-partition-def)
          with p2 have c2:get-ports-of-partition s d = {} using no-cfgport-impl-noports by auto
          then show ?thesis  by simp
        qed
    qed
  qed

lemma trans-smpl-msg-notchg-comminotherdom:
  assumes   p0:reachable0 s
      and   p1:is-a-transmitter sysconf  (current s)
    and   p3:(current s) \⇝ d
    and   p4:s′ = transf-sampling-msg s c
  shows   vpeq-part-comm s d s′
   proof−
     from p3 have p5:(current s) ≠ d using non-interference1-def interference1-def by auto
     from p4 have part-ports s = part-ports s′ using trans-smpl-msg-notchg-partports by blast
     then have get-ports-of-partition s d = get-ports-of-partition s′ d
        using part-ports-imp-portofpart by blast

     moreover have∀ p. p ∈ get-ports-of-partition s d ⟶
              is-a-queuingport s p = is-a-queuingport s′ p ∧
              is-dest-port s p = is-dest-port s′ p ∧
              (if is-dest-port s p then get-port-buf-size s p = get-port-buf-size s′ p else True)
     using  get-port-buf-size-def is-dest-port-def is-a-queuingport-def
      trans-smpl-msg-notchg-portsinotherdom get-port-byid-def p0 p1 p3 p4 by auto
     ultimately show ?thesis by auto
   qed

lemma trans-smpl-msg-sm-nitfpart:⟦reachable0 s; is-a-transmitter sysconf (current s); is-a-partition sysconf d;
                              ((current s) \⇝ d); s′ = transf-sampling-msg s c⟧
                                   ⟹ (s ∼ d ∼ s′)
```

```
  apply(clarsimp simp:vpeq1-def cong del: non-interference1-def is-a-transmitter-def
      is-a-partition-def vpeq-part-comm-def)
  apply(rule conjI)
  using trans-imp-not-part apply fastforce
  apply(rule impI)
  apply(rule conjI)
  using  sche-imp-not-part apply fastforce
  apply(clarsimp simp:vpeq-part-def cong del: non-interference1-def
      is-a-transmitter-def is-a-partition-def vpeq-part-comm-def)
  apply(rule conjI)
  apply (simp add: trans-smpl-msg-notchg-partstate vpeq-part-comm-def)
  using  trans-smpl-msg-notchg-comminotherdom nintf-neq by metis

lemma trans-smpl-msg-presrv-lcrsp:
 assumes p0:reachable0 s
  and   p1:current s = transmitter sysconf
  and   p2:(current s) \⇝ d
  and   p3:s' = transf-sampling-msg s c
 shows   s ∼ d ∼ s'
proof(cases is-a-scheduler sysconf d)
 assume a0:is-a-scheduler sysconf d
 show ?thesis using a0  p1 p3 trans-smpl-msg-sm-sche by auto
next
 assume a1:¬ is-a-scheduler sysconf d
 show ?thesis
  proof(cases is-a-partition sysconf d)
   assume b0:is-a-partition sysconf d
   show ?thesis using p1  trans-smpl-msg-sm-nitfpart[OF p0 - b0 p2 p3] by auto
  next
   assume b1:¬ is-a-partition sysconf d
   show ?thesis
   proof(cases is-a-transmitter sysconf d)
    assume c0:is-a-transmitter sysconf d
    with p1 p3 have current s = d by (simp add: is-a-transmitter-def)
    with p3 show ?thesis using interference1-def non-interference1-def p2 by auto
   next
    assume c1:¬ is-a-transmitter sysconf d
    show ?thesis using a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def by auto
   qed
  qed
qed

lemma trans-smpl-msg-presrv-lcrsp-e: local-respect-e (sys (Transfer-Sampling-Message c))
```

```
  using trans-smpl-msg-presrv-lcrsp  exec-event-def
   prod.simps(2) vpeq-reflexive-lemma by (auto cong del: vpeq1-def)
```

2.5.16 proving "Transfer Queuing Message" satisfying the "local respect" property

```
lemma trans-que-msg-mlost-notchg-current:
  is-a-transmitter sysconf (current s) ⟹ current s = current (transf-queuing-msg-maylost sysconf s c)
  proof(induct c)
    case (Channel-Queuing name sn dn) show ?case
      proof −
        let ?sp = get-portid-by-name s sn
        let ?dp = get-portid-by-name s dn
        let ?sm = remove-msg-from-queuingport s (the ?sp)
        let ?s1 = fst ?sm
        let ?s2 = replace-msg2queuing-port ?s1 (the ?dp) (the (snd ?sm))
        let ?s3 = insert-msg2queuing-port ?s1 (the ?dp) (the (snd ?sm))
        have a0:current ?s1 = current s
          proof(induct (ports (comm s)) (the ?sp))
            case None show ?thesis using None.hyps remove-msg-from-queuingport-def by auto
          next
            case (Some x) show ?thesis
              proof(induct the ((ports (comm s)) (the ?sp)))
                case (Queuing x1 x2 x3 x4 x5) show ?thesis
                  by (smt Port-Type.simps(5) Queuing.hyps State.ext-inject
                    State.surjective State.update-convs(3) fstI option.case-eq-if
                    remove-msg-from-queuingport-def)

              next
                case (Sampling x1 x2 x3 x4) show ?thesis
                    by (metis (no-types, lifting) Port-Type.simps(6) Sampling.hyps
                      fst-conv option.case-eq-if remove-msg-from-queuingport-def)
              qed
           qed
        have a1:current ?s2 = current ?s1 by (simp add: replace-msg2queuing-port-def)
        have a2:current ?s3 = current ?s1
          proof(induct (ports (comm ?s1)) (the ?dp))
            case None show ?thesis by (simp add: None.hyps insert-msg2queuing-port-def option.case-eq-if)
          next
            case (Some x) show ?thesis
              proof(induct the ((ports (comm ?s1)) (the ?dp)))
                case (Queuing x1 x2 x3 x4 x5) show ?thesis
                    by (smt Port-Type.simps(5) Queuing.hyps State.select-convs(1)
                        State.surjective State.update-convs(3) insert-msg2queuing-port-def option.case-eq-if)
              next
```

```
          case (Sampling x1 x2 x3 x4) show ?thesis
            by (smt Port-Type.simps(6) Sampling.hyps
                insert-msg2queuing-port-def option.case-eq-if)
        qed
      qed
    show ?thesis
      proof(cases ?sp ≠ None ∧ ?dp ≠ None ∧ has-msg-inportqueuing s (the ?sp))
        assume b0:?sp ≠ None ∧ ?dp ≠ None ∧ has-msg-inportqueuing s (the ?sp)
        show ?thesis
          proof(cases is-full-portqueuing sysconf (fst ?sm) (the ?dp))
            assume c0:is-full-portqueuing sysconf (fst ?sm) (the ?dp)
            then have transf-queuing-msg-maylost sysconf s (Channel-Queuing name sn dn) = ?s2
              by (smt b0 transf-queuing-msg-maylost.simps(1))
            with a0 a1 show ?thesis by simp
          next
            assume c1:¬ is-full-portqueuing sysconf (fst ?sm) (the ?dp)
            then have transf-queuing-msg-maylost sysconf s (Channel-Queuing name sn dn) = ?s3
              by (smt b0 transf-queuing-msg-maylost.simps(1))
            with a0 a2 show ?thesis by simp
          qed
      next
        assume c0:¬(?sp ≠ None ∧ ?dp ≠ None ∧ has-msg-inportqueuing s (the ?sp))
        then have transf-queuing-msg-maylost sysconf s (Channel-Queuing name sn dn) = s
          by (smt transf-queuing-msg-maylost.simps(1))
        then show ?thesis by auto
      qed
   qed
  next
   case (Channel-Sampling name sn dns) show ?case by auto
  qed

lemma trans-que-msg-mlost-sm-sche:⟦is-a-transmitter sysconf (current s);
                         s′ = transf-queuing-msg-maylost sysconf s c⟧
                            ⟹ (s∼(scheduler sysconf)∼s′)
  using trans-que-msg-mlost-notchg-current sch-not-trans by auto

lemma trans-que-msg-mlost-notchg-partstate:
      ⟦is-a-transmitter sysconf (current s); is-a-partition sysconf d;
      s′ = transf-queuing-msg-maylost sysconf s c⟧ ⟹ (partitions s) d = (partitions s′) d
  proof(induct c)
   case (Channel-Queuing name sn dn) show ?case
     proof −
       let ?sp = get-portid-by-name s sn
```

```
let ?dp = get-portid-by-name s dn
let ?sm = remove-msg-from-queuingport s (the ?sp)
let ?s1 = fst ?sm
let ?s2 = replace-msg2queuing-port ?s1 (the ?dp) (the (snd ?sm))
let ?s3 = insert-msg2queuing-port ?s1 (the ?dp) (the (snd ?sm))
have a0:(partitions s) d = (partitions ?s1) d
  proof(induct (ports (comm s)) (the ?sp))
    case None show ?thesis using None.hyps remove-msg-from-queuingport-def by auto
  next
    case (Some x) show ?thesis
      proof(induct the ((ports (comm s)) (the ?sp)))
        case (Queuing x1 x2 x3 x4 x5) show ?thesis
          by (smt Port-Type.simps(5) Queuing.hyps State.ext-inject
            State.surjective State.update-convs(3) fstI option.case-eq-if
            remove-msg-from-queuingport-def)

      next
        case (Sampling x1 x2 x3 x4) show ?thesis
            by (metis (no-types, lifting) Port-Type.simps(6) Sampling.hyps
              fst-conv option.case-eq-if remove-msg-from-queuingport-def)
      qed
   qed
have a1:(partitions ?s2) d = (partitions ?s1) d by (simp add: replace-msg2queuing-port-def)
have a2:(partitions ?s3) d = (partitions ?s1) d
  proof(induct (ports (comm ?s1)) (the ?dp))
    case None show ?thesis by (simp add: None.hyps insert-msg2queuing-port-def option.case-eq-if)
  next
    case (Some x) show ?thesis
      proof(induct the ((ports (comm ?s1)) (the ?dp)))
        case (Queuing x1 x2 x3 x4 x5) show ?thesis
            by (smt Port-Type.simps(5) Queuing.hyps State.select-convs(2)
              State.surjective State.update-convs(3) insert-msg2queuing-port-def option.case-eq-if)
      next
        case (Sampling x1 x2 x3 x4) show ?thesis
            by (smt Port-Type.simps(6) Sampling.hyps
                insert-msg2queuing-port-def option.case-eq-if)
      qed
  qed
show ?thesis
  proof(cases ?sp ≠ None ∧ ?dp ≠ None ∧ has-msg-inportqueuing s (the ?sp))
    assume b0:?sp ≠ None ∧ ?dp ≠ None ∧ has-msg-inportqueuing s (the ?sp)
    show ?thesis
      proof(cases is-full-portqueuing sysconf (fst ?sm) (the ?dp))
```

assume *c0:is-full-portqueuing sysconf* (*fst ?sm*) (*the ?dp*)
then have *transf-queuing-msg-maylost sysconf s* (*Channel-Queuing name sn dn*) = *?s2*
by (*smt b0 transf-queuing-msg-maylost.simps*(*1*))
with *a0 a1* **show** *?thesis* **by** (*simp add*: *Channel-Queuing.prems*(*3*))
next
assume *c1*:¬ *is-full-portqueuing sysconf* (*fst ?sm*) (*the ?dp*)
then have *transf-queuing-msg-maylost sysconf s* (*Channel-Queuing name sn dn*) = *?s3*
by (*smt b0 transf-queuing-msg-maylost.simps*(*1*))
with *a0 a2* **show** *?thesis* **by** (*simp add*: *Channel-Queuing.prems*(*3*))
qed
next
assume *c0*:¬(*?sp* ≠ *None* ∧ *?dp* ≠ *None* ∧ *has-msg-inportqueuing s* (*the ?sp*))
then have *transf-queuing-msg-maylost sysconf s* (*Channel-Queuing name sn dn*) = *s*
by (*smt transf-queuing-msg-maylost.simps*(*1*))
then show *?thesis* **by** (*simp add*: *Channel-Queuing.prems*(*3*))
qed
qed
next
case (*Channel-Sampling name sn dns*) **show** *?case* **by** (*simp add*: *Channel-Sampling.prems*(*3*))
qed

lemma *trans-que-msg-mlost-notchg-partports*:
s′ = *transf-queuing-msg-maylost sysconf s c* ⟹
part-ports s = *part-ports s′*
proof(*induct c*)
case (*Channel-Queuing name sn dn*) **show** *?case*
proof −
let *?sp* = *get-portid-by-name s sn*
let *?dp* = *get-portid-by-name s dn*
let *?sm* = *remove-msg-from-queuingport s* (*the ?sp*)
let *?s1* = *fst ?sm*
let *?s2* = *replace-msg2queuing-port ?s1* (*the ?dp*) (*the* (*snd ?sm*))
let *?s3* = *insert-msg2queuing-port ?s1* (*the ?dp*) (*the* (*snd ?sm*))
have *b0:part-ports s* = *part-ports ?s1*
proof(*induct* (*ports* (*comm s*)) (*the ?sp*))
case *None* **show** *?thesis* **using** *None.hyps remove-msg-from-queuingport-def* **by** *auto*
next
case (*Some x*) **show** *?thesis*
proof(*induct the* ((*ports* (*comm s*)) (*the ?sp*)))
case (*Queuing x1 x2 x3 x4 x5*) **show** *?thesis*
by (*smt Port-Type.simps*(*5*) *Queuing.hyps State.select-convs*(*4*) *State.surjective*
State.update-convs(*3*) *fstI option.case-eq-if remove-msg-from-queuingport-def*)
next

```
      case (Sampling x1 x2 x3 x4) show ?thesis
        by (metis (no-types, lifting) Port-Type.simps(6) Sampling.hyps
          fst-conv option.case-eq-if remove-msg-from-queuingport-def)
    qed
  qed
have b1:part-ports ?s2 = part-ports ?s1
  by (simp add: replace-msg2queuing-port-def)
have b2:part-ports ?s3 = part-ports ?s1
  proof(induct (ports (comm ?s1)) (the ?dp))
    case None show ?thesis by (simp add: None.hyps insert-msg2queuing-port-def option.case-eq-if)
  next
    case (Some x) show ?thesis
      proof(induct the ((ports (comm ?s1)) (the ?dp)))
        case (Queuing x1 x2 x3 x4 x5) show ?thesis
          by (smt Communication-State.select-convs(1) Communication-State.surjective
            Communication-State.update-convs(2) Port-Type.simps(5) Queuing.hyps
            State.select-convs(3) State.select-convs(4) State.surjective State.update-convs(3)
            insert-msg2queuing-port-def option.case-eq-if)
      next
        case (Sampling x1 x2 x3 x4) show ?thesis
            by (smt Port-Type.simps(6) Sampling.hyps
              insert-msg2queuing-port-def option.case-eq-if)
      qed
  qed
show ?thesis
  proof(cases ?sp ≠ None ∧ ?dp ≠ None ∧ has-msg-inportqueuing s (the ?sp))
    assume c0:?sp ≠ None ∧ ?dp ≠ None ∧ has-msg-inportqueuing s (the ?sp)
    show ?thesis
      proof(cases is-full-portqueuing sysconf (fst ?sm) (the ?dp))
        assume d0:is-full-portqueuing sysconf (fst ?sm) (the ?dp)
        then have transf-queuing-msg-maylost sysconf s (Channel-Queuing name sn dn) = ?s2
          by (smt c0 transf-queuing-msg-maylost.simps(1))
        with b0 b1 show ?thesis by (simp add: Channel-Queuing.prems(1))
      next
        assume c1:¬ is-full-portqueuing sysconf (fst ?sm) (the ?dp)
        then have transf-queuing-msg-maylost sysconf s (Channel-Queuing name sn dn) = ?s3
          by (smt c0 transf-queuing-msg-maylost.simps(1))
        with b0 b2 show ?thesis by (simp add: Channel-Queuing.prems(1))
      qed
  next
    assume c0:¬(?sp ≠ None ∧ ?dp ≠ None ∧ has-msg-inportqueuing s (the ?sp))
    then have transf-queuing-msg-maylost sysconf s (Channel-Queuing name sn dn) = s
      by (smt transf-queuing-msg-maylost.simps(1))
```

then show *?thesis* **by** (*simp add*: *Channel-Queuing.prems*(*1*))
qed
qed
next
case (*Channel-Sampling name sn dns*) **show** *?case* **by** (*simp add*: *Channel-Sampling.prems*(*1*))
qed

lemma *trans-que-msg-mlost-notchg-portsinotherdom*:
assumes *p1*:*is-a-transmitter sysconf* (*current s*)
and *p2*:*reachable0 s*
and *p3*:(*current s*) $\setminus\leadsto$ *d*
and *p4*:*s'* = *transf-queuing-msg-maylost sysconf s c*
shows $\forall$ *p*. *p* $\in$ *get-ports-of-partition s d* $\longrightarrow$ *ports* (*comm s*) *p* = *ports* (*comm s'*) *p*
proof(*cases is-a-scheduler sysconf d*)
assume *a0*:*is-a-scheduler sysconf d*
with *p2* **have** *a3*:*get-ports-of-partition s d* = {}
using *no-cfgport-impl-noports sched-hasnoports* **by** *auto*
then show *?thesis* **by** *simp*
next
assume *a1*:$\neg$ *is-a-scheduler sysconf d*
show *?thesis*
proof(*cases is-a-partition sysconf d*)
assume *b0*:*is-a-partition sysconf d*
with *p1 p3* **have** *b1*:$\neg$ *transmitter-intf-part sysconf d*
by (*metis a1 interference1-def non-interference1-def trans-imp-not-part*)
then have *b2*:*get-partition-cfg-ports* (*the* ((*partconf sysconf*) *d*)) = {}
using *b0 get-partition-cfg-ports-byid-def is-a-partition-def port-partition* **by** *fastforce*
then have *b3*:*get-partition-cfg-ports-byid sysconf d* = {}
by (*simp add*: *get-partition-cfg-ports-byid-def*)
with *p2* **have** *b4*:*get-ports-of-partition s d* = {} **using** *no-cfgport-impl-noports* **by** *auto*
then show *?thesis* **by** *simp*
next
assume *b1*:$\neg$*is-a-partition sysconf d*
show *?thesis*
proof(*cases is-a-transmitter sysconf d*)
assume *c0*:*is-a-transmitter sysconf d*
with *p1 p3* **have** *current s* = *d* **by** (*simp add*: *is-a-transmitter-def*)
with *p3* **show** *?thesis* **using** *interference1-def non-interference1-def* **by** *auto*
next
assume *c1*:$\neg$ *is-a-transmitter sysconf d*
with *a1 b1* **have** *c2*:*get-partition-cfg-ports-byid sysconf d* = {}
by (*simp add*: *get-partition-cfg-ports-byid-def is-a-partition-def*)
with *p2* **have** *c2*:*get-ports-of-partition s d* = {} **using** *no-cfgport-impl-noports* **by** *auto*

```
        then show ?thesis  by simp
      qed
    qed
  qed

lemma trans-que-msg-mlost-notchg-comminotherdom:
 assumes   p0:reachable0 s
     and   p1:is-a-transmitter sysconf (current s)
   and   p3:(current s) \↝ d
   and   p4:s' = transf-queuing-msg-maylost sysconf s c
 shows   vpeq-part-comm s d s'
proof−
  from p3 have p5:(current s) ≠ d using non-interference1-def interference1-def by auto
  from p4 have part-ports s = part-ports s' using trans-que-msg-mlost-notchg-partports by blast
  then have get-ports-of-partition s d = get-ports-of-partition s' d
     using part-ports-imp-portofpart by blast

  moreover have ∀ p. p ∈ get-ports-of-partition s d ⟶
          is-a-queuingport s p = is-a-queuingport s' p ∧
          is-dest-port s p = is-dest-port s' p ∧
          (if is-dest-port s p then get-port-buf-size s p = get-port-buf-size s' p else True)
  using  is-dest-port-def is-a-queuingport-def trans-que-msg-mlost-notchg-portsinotherdom get-port-byid-def
        p0 p1 p3 p4 get-port-buf-size-def by auto

  ultimately show ?thesis by auto
qed


lemma trans-que-msg-mlost-sm-nitfpart:
 ⟦reachable0 s; is-a-transmitter sysconf (current s); is-a-partition sysconf d;
   ((current s) \↝ d); s' = transf-queuing-msg-maylost sysconf s c⟧ ⟹ (s ∼ d ∼ s')
   apply(clarsimp simp:vpeq1-def cong del: is-a-transmitter-def
       is-a-partition-def non-interference1-def vpeq-part-comm-def)
   apply(rule conjI)
   using  trans-imp-not-part apply fastforce
   apply(rule impI)
   apply(rule conjI)
   using  sche-imp-not-part apply fastforce
   apply(clarsimp cong del: is-a-transmitter-def is-a-partition-def non-interference1-def vpeq-part-comm-def)
   apply(rule conjI )
   apply (simp add: trans-que-msg-mlost-notchg-partstate)
   using trans-que-msg-mlost-notchg-comminotherdom by blast
```

```
lemma trans-que-msg-mlost-presrv-lcrsp:
  assumes p0:reachable0 s
    and   p1:current s = transmitter sysconf
    and   p2:(current s) \⇝ d
    and   p3:s′ = transf-queuing-msg-maylost sysconf s c
  shows   s ∼ d ∼ s′
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  show ?thesis using a0 is-a-scheduler-def p1 p3 trans-que-msg-mlost-sm-sche using is-a-transmitter-def by auto
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
    proof(cases is-a-partition sysconf d)
      assume b0:is-a-partition sysconf d
      show ?thesis using p1 trans-que-msg-mlost-sm-nitfpart[OF p0 - b0 p2 p3] by auto
    next
      assume b1:¬ is-a-partition sysconf d
      show ?thesis
      proof(cases is-a-transmitter sysconf d)
        assume c0:is-a-transmitter sysconf d
        with p1 p3 have current s = d by (simp add: is-a-transmitter-def)
        with p3 show ?thesis using p2 by auto
      next
        assume c1:¬ is-a-transmitter sysconf d
        show ?thesis using a1 b1 c1  by auto
      qed
    qed
qed
```

```
lemma trans-que-msg-mlost-presrv-lcrsp-e: local-respect-e (sys (Transfer-Queuing-Message c))
  using trans-que-msg-mlost-presrv-lcrsp  exec-event-def
    prod.simps(2)vpeq-reflexive-lemma by (auto cong del: vpeq1-def)
```

2.5.17 proving the "local respect" property

```
theorem local-respect:local-respect
  proof –
    {
      fix e
      have local-respect-e e
        apply(induct e)
        using crt-smpl-port-presrv-lcrsp-e write-smpl-msg-presrv-lcrsp-e
              read-smpl-msg-presrv-lcrsp-e  get-smpl-pid-presrv-lcrsp-e
              get-smpl-psts-presrv-lcrsp-e crt-que-port-presrv-lcrsp-e
```

```
            snd-que-msg-lst-presrv-lcrsp-e rec-que-msg-presrv-lcrsp-e
            get-que-pid-presrv-lcrsp-e  get-que-psts-presrv-lcrsp-e
            clr-que-port-presrv-lcrsp-e  set-part-mode-presrv-lcrsp-e
            get-part-status-presrv-lcrsp-e
        apply (rule Hypercall.induct)
        using schedule-presrv-lcrsp-e trans-smpl-msg-presrv-lcrsp-e
           trans-que-msg-mlost-presrv-lcrsp-e
        by (rule System-Event.induct)
      }
      then show ?thesis using local-respect-all-evt by blast
    qed
```

2.6 Concrete unwinding condition of "weakly step consistent"

2.6.1 proving "create sampling port" satisfying the "step consistent" property

```
lemma crt-smpl-port-presrv-comm-part-ports:
  assumes  p1:reachable0 s ∧ reachable0 t
    and  p2:s ∼ (transmitter sysconf) ∼ t
    and  p5:s ∼ (scheduler sysconf) ∼ t
    and  p3:s' = fst (create-sampling-port sysconf s pname)
    and  p4:t' = fst (create-sampling-port sysconf t pname)
   shows  comm s' = comm t' ∧ part-ports s' = part-ports t'
   proof –
   {
    from p2 have a0:vpeq-transmitter s (transmitter sysconf) t using sch-not-trans  by auto
    then have a1:comm s = comm t ∧ part-ports s = part-ports t  by auto
    from p1 have a2:port-consistent s ∧ port-consistent t by (simp add: port-cons-reach-state)
    show ?thesis
      proof(cases get-samplingport-conf sysconf pname = None
                 ∨ get-portid-by-name s pname ≠ None
                 ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current s))
        assume d0:get-samplingport-conf sysconf pname = None
                 ∨ get-portid-by-name s pname ≠ None
                 ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current s)
        with p3 have d1:s' = s by (simp add: create-sampling-port-def)
        have d2:get-samplingport-conf sysconf pname = None
                 ∨ get-portid-by-name t pname ≠ None
                 ∨ pname ∉ get-partition-cfg-ports-byid sysconf (current t)
                 by (meson disjoint-iff-not-equal port-partition)
        with p4 have d3:t' = t using create-sampling-port-def by auto
        with a1 d1 show ?thesis by simp
      next
        let ?nid = get-portid-in-type (the (get-samplingport-conf sysconf pname))
```

assume *e0*:¬(*get-samplingport-conf sysconf pname* = *None*
∨ *get-portid-by-name s pname* ≠ *None*
∨ *pname* ∉ *get-partition-cfg-ports-byid sysconf* (*current s*))
with *p3* **have** *e1*:*part-ports s′* = (*part-ports s*)(*?nid*:= (*Some* (*current s*)))
using *port-partition* **by** *auto*
with *p4 e0* **have** *e2*:*part-ports t′* = (*part-ports t*)(*?nid* := (*Some* (*current t*)))
using *port-partition* **by** *auto*
with *p3 e0* **have** *e3*:*ports* (*comm s′*) = (*ports* (*comm s*))(*?nid* := *get-samplingport-conf sysconf pname*)
using *port-partition* **by** *auto*
with *p4 e0* **have** *e4*:*ports* (*comm t′*) = (*ports* (*comm t*))(*?nid* := *get-samplingport-conf sysconf pname*)
using *port-partition* **by** *auto*
with *a1 e1 e2 e3 e4* **show** *?thesis* **using** *p5 sched-current-lemma e0 port-partition* **by** *fastforce*
qed
}
qed

lemma *crt-smpl-port-presrv-comm-of-current-part*:
assumes *p1*:*reachable0 s* ∧ *reachable0 t*
and *p2*:*s* ∼ (*scheduler sysconf*) ∼ *t*
and *p3*:*s′* = *fst* (*create-sampling-port sysconf s pname*)
and *p4*:*t′* = *fst* (*create-sampling-port sysconf t pname*)
and *p5*:(*current s*) = *d*
and *p6*:*vpeq-part-comm s d t*
and *p7*:*is-a-partition sysconf d*
shows *vpeq-part-comm s′ d t′*
apply(*clarsimp*, *rule conjI*)
proof −
{
from *p6* **have** *a1*:*get-ports-of-partition s d* = *get-ports-of-partition t d*
by *auto*
from *p2* **have** *a2*:*current s* = *current t* **by** *auto*
show *g0*:*get-ports-of-partition s′ d* = *get-ports-of-partition t′ d*
proof −
{
have ∀ *p*. *p*∈*get-ports-of-partition s′ d* ⟶ *p*∈*get-ports-of-partition t′ d*
proof−
{
fix *p*
assume *a0*:*p*∈*get-ports-of-partition s′ d*
have *a3*:*p*∈*get-ports-of-partition t′ d*
proof(*cases pname* ∈ *get-partition-cfg-ports-byid sysconf* (*current s*))
assume *b0*:*pname* ∈ *get-partition-cfg-ports-byid sysconf* (*current s*)
with *a2* **have** *b1*:*pname* ∈ *get-partition-cfg-ports-byid sysconf* (*current t*) **by** *simp*

```
          have b2:p ≠ get-portid-in-type (the (get-samplingport-conf sysconf pname))
            using b0 port-partition by auto
          then show ?thesis using b0 port-partition by auto
        next
          assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
          with a2 have c1:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current t)) by simp
          have c2:s′ = s by (simp add: c0 create-sampling-port-def p3)
          have c3:t′ = t by (simp add: c1 create-sampling-port-def p4)
          then show ?thesis using a0 a1 c2 by auto
        qed
    }
    then show ?thesis by auto
    qed
  moreover
  have ∀ p. p∈get-ports-of-partition t′ d ⟶ p∈get-ports-of-partition s′ d
    proof−
    {
      fix p
      assume a0:p∈get-ports-of-partition t′ d
      have a3:p∈get-ports-of-partition s′ d
         proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
          assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
          with a2 have b1:pname ∈ get-partition-cfg-ports-byid sysconf (current t) by simp
          have b2:p ≠ get-portid-in-type (the (get-samplingport-conf sysconf pname))
            using b0 port-partition by auto
          then show ?thesis using b0 port-partition by auto
        next
          assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
          with a2 have c1:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current t)) by simp
          have c2:s′ = s by (simp add: c0 create-sampling-port-def p3)
          have c3:t′ = t by (simp add: c1 create-sampling-port-def p4)
          then show ?thesis using a0 a1 c2 by auto
        qed
    }
    then show ?thesis by auto
    qed
  then show ?thesis using calculation by blast
 }
 qed
next
 from p6 have a1:get-ports-of-partition s d = get-ports-of-partition t d
    unfolding vpeq-part-comm-def Let-def by auto
 from p2 have a2:current s = current t by auto
```

```
    show ∀ p. (is-dest-port s′ p ⟶ p ∈ get-ports-of-partition s′ d ⟶
     is-a-queuingport s′ p = is-a-queuingport t′ p ∧ is-dest-port t′ p
     ∧ get-port-buf-size s′ p = get-port-buf-size t′ p) ∧ (¬ is-dest-port s′ p ⟶
     p ∈ get-ports-of-partition s′ d ⟶ is-a-queuingport s′ p = is-a-queuingport t′ p ∧ ¬ is-dest-port t′ p)
      proof −
      {
        fix p
        have (is-dest-port s′ p ⟶
           p ∈ get-ports-of-partition s′ d ⟶
           is-a-queuingport s′ p = is-a-queuingport t′ p ∧ is-dest-port t′ p
           ∧ get-port-buf-size s′ p = get-port-buf-size t′ p) ∧ (¬ is-dest-port s′ p ⟶
           p ∈ get-ports-of-partition s′ d ⟶ is-a-queuingport s′ p = is-a-queuingport t′ p ∧ ¬ is-dest-port t′ p)
          proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
          assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
            with a2 have b1:pname ∈ get-partition-cfg-ports-byid sysconf (current t) by simp
            have b2:p ≠ get-portid-in-type (the (get-samplingport-conf sysconf pname))
              using b0 port-partition by auto
            then show ?thesis using b0 port-partition by auto
          next
            assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
            with a2 have c1:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current t)) by simp
            have c2:s′ = s by (simp add: c0 create-sampling-port-def p3)
            have c3:t′ = t by (simp add: c1 create-sampling-port-def p4)
            then show ?thesis using c0 a1 c2 using p6  by auto
          qed
      }
      then show ?thesis by auto
      qed
  }
  qed

lemma crt-smpl-port-presrv-wk-stp-cons:
  assumes p1:is-a-partition sysconf (current s)
    and   p2:reachable0 s ∧ reachable0 t
    and   p3:s ∼ d ∼ t
    and   p4:s ∼ (scheduler sysconf) ∼ t
    and   p5:(current s) ⇝ d
    and   p6:s ∼ (current s) ∼ t
    and   p7:s′ = fst (create-sampling-port sysconf s pname)
    and   p8:t′ = fst (create-sampling-port sysconf t pname)
  shows   s′ ∼ d ∼ t′
 proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
```

```
  show ?thesis using a0  p1 p5 sche-imp-not-part by (metis is-a-scheduler-def no-intf-sched-help)
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
    proof(cases is-a-partition sysconf d)
      assume b0:is-a-partition sysconf d
      show ?thesis
        proof(cases current s = d)
          assume c0:current s = d
          have d0:vpeq-part s' d t'
            proof -
            {

              have e1:partitions s' d = partitions t' d
                proof -
                {
                  from p3 b0 have f1:partitions s d = partitions t d
                    using a1 part-imp-not-tras  by fastforce
                  from p7 have f2:partitions s d = partitions s' d
                    using  b0 crt-sampl-port-notchg-partstate p1 by blast
                  from p8 have f3:partitions t d = partitions t' d
                    using b0 c0 crt-sampl-port-notchg-partstate p4 sched-current-lemma
                      by simp
                  with f1 f2 have partitions s' d = partitions t' d by auto
                }
                then show ?thesis by auto
                qed
              have e2:vpeq-part-comm s' d t'
                proof -
                {
                  from p3 a1 b0 have f1:vpeq-part-comm s d t
                    using part-imp-not-tras by fastforce
                  then have vpeq-part-comm s' d t'
                    using c0 crt-smpl-port-presrv-comm-of-current-part p1 p2 p4 p7 p8 by auto
                }
                then show ?thesis by auto
                qed
              with e1 have vpeq-part s' d t' by auto
            }
            then show ?thesis by auto
            qed
          then show ?thesis using a1 b0 c0
            using trans-imp-not-part by fastforce
```

```
        next
          assume c1:current s ≠ d
          have d1:vpeq-part s' d t'
          proof -
          {
            have e1:partitions s' d = partitions t' d
              using a1  crt-sampl-port-notchg-partstate[OF p1 b0 p7]
                 p3 p4 p7 p8 part-imp-not-tras sched-current-lemma
                 b0 c1  p1 p5 part-imp-not-sch by auto
            have e2:vpeq-part-comm s' d t'
              by (metis a1 b0 create-sampling-port-def fst-conv get-samplingport-conf-def
                  is-a-scheduler-def is-a-transmitter-def p3 p7 p8 part-imp-not-tras
                  port-name-diff vpeq1-def vpeq-part-def)
           with e1 have vpeq-part s' d t'  by auto
          }
          then show ?thesis by auto
          qed
          show ?thesis  using a1 b0 c1
          using  trans-imp-not-part d1 by fastforce
        qed
      next
        assume b1:¬ is-a-partition sysconf d
        show ?thesis
        proof(cases is-a-transmitter sysconf d)
          assume c0:is-a-transmitter sysconf d
          show ?thesis
             using a1 b1 p3 c0 crt-smpl-port-presrv-comm-part-ports[OF p2 - p4 p7 p8] by auto
        next
          assume c1:¬ is-a-transmitter sysconf d
          show ?thesis using a1 b1 c1 by auto
        qed
      qed
  qed

lemma crt-smpl-port-presrv-wk-stp-cons-e: weak-step-consistent-e (hyperc (Create-Sampling-Port p))
  using crt-smpl-port-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
    non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
      by (smt Event.case(1) Hypercall.case(1) domain-of-event.simps(1)
          event-enabled.simps(1) option.sel prod.simps(2))
```

2.6.2 proving "write sampling message" satisfying the "step consistent" property

```
lemma wrt-smpl-msg-presrv-comm-part-ports:
  assumes   p1:reachable0 s ∧ reachable0 t
```

```
    and   p2:s ∼ (transmitter sysconf) ∼ t
    and   p5:s ∼ (scheduler sysconf) ∼ t
    and   p3:s′ = fst (write-sampling-message s pid m)
    and   p4:t′ = fst (write-sampling-message t pid m)
  shows   comm s′ = comm t′ ∧ part-ports s′ = part-ports t′
  proof −
  {
    from p2 have a0:vpeq-transmitter s (transmitter sysconf) t using sch-not-trans  by auto
    then have a1:comm s = comm t ∧ part-ports s = part-ports t  by auto
    from p1 have a2:port-consistent s ∧ port-consistent t by (simp add: port-cons-reach-state)
    show ?thesis
      proof(cases ¬ is-a-samplingport s pid
                  ∨ ¬ is-source-port s pid
                  ∨ ¬ is-a-port-of-partition s pid (current s))
        assume d0:¬ is-a-samplingport s pid
                  ∨ ¬ is-source-port s pid
                  ∨ ¬ is-a-port-of-partition s pid (current s)
        with p3 have d1:s′ = s by (simp add: write-sampling-message-def)
        have d2:¬ is-a-samplingport t pid
                  ∨ ¬ is-source-port t pid
                  ∨ ¬ is-a-port-of-partition t pid (current t)
          using a1 d0 is-a-port-of-partition-def is-a-samplingport-def
            is-source-port-def p5 sched-current-lemma by simp
        with p4 have d3:t′ = t using write-sampling-message-def by auto
        with a1 d1 show ?thesis by simp
      next
        assume e0:¬(¬ is-a-samplingport s pid
                  ∨ ¬ is-source-port s pid
                  ∨ ¬ is-a-port-of-partition s pid (current s))
        with p3 have e1:part-ports s′ = part-ports s
          by (metis Int-absorb a2 empty-iff is-a-port-of-partition-def
            option.distinct(1) port-consistent-def port-partition)

        with p4 e0 have e2:part-ports t′ = part-ports t
          by (metis Int-absorb a2 empty-iff is-a-port-of-partition-def
            option.distinct(1) port-consistent-def port-partition)
        have f1:s′ = update-sampling-port-msg s pid m using e0 p3 write-sampling-message-def by auto
        have f2:t′ = update-sampling-port-msg t pid m
          using a1 e0 is-a-port-of-partition-def is-a-samplingport-def
            is-source-port-def p4 p5 sched-current-lemma write-sampling-message-def
            by simp
        with p3 p4 e0 have e5:ports (comm s′) = ports (comm t′)
          proof(induct (ports (comm s)) pid)
```

```
        case None show ?case using None.hyps a1 f1 f2 update-sampling-port-msg-def by auto
      next
        case (Some x) show ?case
          proof(induct the (ports (comm s) pid))
            case (Queuing x1 x2 x3 x4 x5) show ?case
                by (smt Port-Type.simps(5) Queuing.hyps Some.hyps a1 f1 f2
                    option.sel option.simps(5) update-sampling-port-msg-def)
          next
            case (Sampling x1 x2 x3 x4) show ?case
                by (smt Communication-State.surjective Communication-State.update-convs(1)
                  Port-Type.simps(6) Sampling.hyps State.select-convs(3) State.surjective
                  State.update-convs(3) a1 f1 f2 option.case-eq-if update-sampling-port-msg-def)
          qed
      qed
    with a1 e1 e2 e5 show ?thesis using p5 sched-current-lemma by auto

  qed
}
qed

lemma wrt-smpl-msg-presrv-comm-of-current-part:
assumes   p1:reachable0 s ∧ reachable0 t
  and   p2:s ∼ (scheduler sysconf) ∼ t
  and   p3:s′ = fst (write-sampling-message s pid m)
  and   p4:t′ = fst (write-sampling-message t pid m)
  and   p5:(current s) = d
  and   p6:vpeq-part-comm s d t
  and   p7:is-a-partition sysconf d
 shows   vpeq-part-comm s′ d t′
 proof−
  from p6 have a1:get-ports-of-partition s d = get-ports-of-partition t d
      by auto
  from p2 have a2:current s = current t by auto
  from p3 p5 p7 have a3:part-ports s = part-ports s′ using wrt-smpl-msg-notchg-partports by simp
  then have a4:get-ports-of-partition s d = get-ports-of-partition s′ d
    using part-ports-imp-portofpart by blast
  from p4 p5 p7 a2 have a5:part-ports t = part-ports t′ using wrt-smpl-msg-notchg-partports by simp
  then have a6:get-ports-of-partition t d = get-ports-of-partition t′ d
    using part-ports-imp-portofpart by blast
  have g0:get-ports-of-partition s′ d = get-ports-of-partition t′ d
    using a1 a4 a6 by simp
  moreover have ∀ p. p ∈ get-ports-of-partition s′ d ⟶
          is-a-queuingport s′ p = is-a-queuingport t′ p ∧
```

```
          is-dest-port s′ p = is-dest-port t′ p ∧
          (if is-dest-port s′ p then get-port-buf-size s′ p = get-port-buf-size t′ p else True)
     using a4 by (metis  empty-iff inf.idem no-cfgport-impl-noports p1 port-partition)
     ultimately show ?thesis by auto
  qed


lemma wrt-smpl-msg-presrv-wk-stp-cons:
  assumes p1:is-a-partition sysconf (current s)
    and   p2:reachable0 s ∧ reachable0 t
    and   p3:s ∼ d ∼ t
    and   p4:s ∼ (scheduler sysconf) ∼ t
    and   p5:(current s) ⇝ d
    and   p6:s ∼ (current s) ∼ t
    and   p7:s′ = fst (write-sampling-message s pid m)
    and   p8:t′ = fst (write-sampling-message t pid m)
  shows   s′ ∼ d ∼ t′
 proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  show ?thesis by (smt a0 interference1-def p1 p5 sche-imp-not-part)
 next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
   proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    show ?thesis
     proof(cases current s = d)
      assume c0:current s = d
      have d0:vpeq-part s′ d t′
       proof −
       {
        have e1:partitions s′ d = partitions t′ d
         proof −
         {
          from p3 b0 have f1:partitions s d = partitions t d
            using a1 part-imp-not-tras by fastforce
          from p7 have f2:partitions s d = partitions s′ d
            using b0 wrt-smpl-msg-notchg-partstate p1 by auto
          from p8 have f3:partitions t d = partitions t′ d
            using b0 c0 wrt-smpl-msg-notchg-partstate p4 sched-current-lemma  by simp
          with f1 f2 have partitions s′ d = partitions t′ d by auto
         }
         then show ?thesis by auto
```

```
        qed
      have e2:vpeq-part-comm s' d t'
        proof -
        {
          from p3 a1 b0 have f1:vpeq-part-comm s d t
            using  part-imp-not-tras by fastforce
          then have vpeq-part-comm s' d t'
            using c0 wrt-smpl-msg-presrv-comm-of-current-part p1 p2 p4 p7 p8 by auto
        }
        then show ?thesis by auto
        qed
      with e1 have vpeq-part s' d t' using vpeq-part-def by auto
    }
    then show ?thesis by auto
    qed
  then show ?thesis  using a1 b0 c0
    using trans-imp-not-part by fastforce
next
  assume c1:current s ≠ d
  have d1:vpeq-part s' d t'
    proof -
    {
      have e1:partitions s' d = partitions t' d
        using a1 b0  p1 p3 p4 p7 p8
        part-not-trans  wrt-smpl-msg-notchg-partstate by auto
      have e2:vpeq-part-comm s' d t'
        proof -
          from p3 a1 b0 have f1:vpeq-part-comm s d t
            using part-imp-not-tras by fastforce
          have f2:vpeq-part-comm s d s' using c1 p1 p2 p7 wrt-smpl-msg-notchg-comminotherdom by blast
          have f3:vpeq-part-comm t d t'
            using p1 p2 p4 p8 c1 sched-current-lemma wrt-smpl-msg-notchg-comminotherdom
              by fastforce
          then show ?thesis
            using f1 f2 vpeq-part-comm-symmetric-lemma vpeq-part-comm-transitive-lemma by blast
        qed
     with e1 have vpeq-part s' d t' using vpeq-part-def by auto
    }
    then show ?thesis by auto
    qed
  show ?thesis using a1 b0 c1
  using d1 trans-imp-not-part  by auto
qed
```

```
      next
        assume b1:¬ is-a-partition sysconf d
        show ?thesis
        proof(cases is-a-transmitter sysconf d)
          assume c0:is-a-transmitter sysconf d
          show ?thesis
            proof −
            {
              have vpeq-transmitter s′ d t′ unfolding vpeq-transmitter-def
              proof−
                from p3 p7 p8
                show comm s′ = comm t′ ∧ part-ports s′ = part-ports t′
                  using c0 wrt-smpl-msg-presrv-comm-part-ports[OF p2 - p4]  by auto
              qed
            }
            then show ?thesis using a1 b1 by auto
            qed
        next
          assume c1:¬ is-a-transmitter sysconf d
          show ?thesis using a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def by auto
        qed
      qed
  qed

lemma wrt-smpl-msg-presrv-wk-stp-cons-e: weak-step-consistent-e (hyperc (Write-Sampling-Message p m))
  using wrt-smpl-msg-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
     non-interference-def singletonD sched-vpeq same-part-mode
      by (smt Event.case(1) Hypercall.case(2) domain-of-event.simps(1) event-enabled.simps(1) option.sel prod.simps(2))
```

2.6.3 proving "read sampling message" satisfying the "step consistent" property

```
lemma read-smpl-msg-presrv-wk-stp-cons:
    assumes p1:s ∼ d ∼ t
      and   p2:s′ = fst (read-sampling-message s pid)
      and   p3:t′ = fst (read-sampling-message t pid)
    shows   s′ ∼ d ∼ t′
  proof −
    have a0:s′ = s by (simp add: p2 read-sampling-message-def)
    have a1:t′ = t by (simp add: p3 read-sampling-message-def)
    then show ?thesis using a0 p1 by blast
  qed

lemma read-smpl-msg-presrv-wk-stp-cons-e: weak-step-consistent-e (hyperc (Read-Sampling-Message p))
  using read-smpl-msg-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
```

```
    non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
     by (smt Event.case(1) Hypercall.case(3) domain-of-event.simps(1)
        event-enabled.simps(1) option.sel prod.simps(2))
```

2.6.4 proving "get sampling portid" satisfying the "step consistent" property

```
lemma get-smpl-pid-presrv-wk-stp-cons:
    assumes p1:s ∼ d ∼ t
      and   p2:s' = fst (get-sampling-port-id sysconf s pname)
      and   p3:t' = fst (get-sampling-port-id sysconf t pname)
    shows   s' ∼ d ∼ t'
  proof –
    have a0:s' = s by (simp add: p2 get-sampling-port-id-def)
    have a1:t' = t by (simp add: p3 get-sampling-port-id-def)
    then show ?thesis using a0 p1 by blast
  qed

lemma get-smpl-pid-presrv-wk-stp-cons-e: weak-step-consistent-e (hyperc (Get-Sampling-Portid p))
  using get-smpl-pid-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
    non-interference-def singletonD sched-vpeq same-part-mode
     by (smt Event.case(1) Hypercall.case(4) domain-of-event.simps(1)
        event-enabled.simps(1) option.sel prod.simps(2))
```


2.6.5 proving "get sampling port status" satisfying the "step consistent" property

```
lemma get-smpl-psts-presrv-wk-stp-cons:
    assumes p1:s ∼ d ∼ t
      and   p2:s' = fst (get-sampling-port-status sysconf s pid)
      and   p3:t' = fst (get-sampling-port-status sysconf t pid)
    shows   s' ∼ d ∼ t'
  proof –
    have a0:s' = s by (simp add: p2 get-sampling-port-status-def)
    have a1:t' = t by (simp add: p3 get-sampling-port-status-def)
    then show ?thesis using a0 p1 by blast
  qed

lemma get-smpl-psts-presrv-wk-stp-cons-e: weak-step-consistent-e (hyperc (Get-Sampling-Portstatus p))
  using get-smpl-psts-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
    non-interference-def singletonD sched-vpeq same-part-mode
     by (smt Event.case(1) Hypercall.case(5) domain-of-event.simps(1) event-enabled.simps(1)
       option.sel prod.simps(2) vpeq1-def vpeq-sched-def)
```

2.6.6 proving "create queuing port" satisfying the "step consistent" property

lemma *crt-que-port-presrv-comm-part-ports*:
assumes *p1*:*reachable0 s* $\wedge$ *reachable0 t*
and *p2*:*s* $\sim$ (*transmitter sysconf*) $\sim$ *t*
and *p5*:*s* $\sim$ (*scheduler sysconf*) $\sim$ *t*
and *p3*:*s'* = *fst* (*create-queuing-port sysconf s pname*)
and *p4*:*t'* = *fst* (*create-queuing-port sysconf t pname*)
shows *comm s'* = *comm t'* $\wedge$ *part-ports s'* = *part-ports t'*
proof –
{
from *p2* **have** *a0*:*vpeq-transmitter s* (*transmitter sysconf*) *t* **using** *sch-not-trans* **by** *auto*
then have *a1*:*comm s* = *comm t* $\wedge$ *part-ports s* = *part-ports t* **by** *auto*
from *p1* **have** *a2*:*port-consistent s* $\wedge$ *port-consistent t* **by** (*simp add*: *port-cons-reach-state*)
show *?thesis*
proof(*cases get-queuingport-conf sysconf pname* = *None*
$\vee$ *get-portid-by-name s pname* $\neq$ *None*
$\vee$ *pname* $\notin$ *get-partition-cfg-ports-byid sysconf* (*current s*))
assume *d0*:*get-queuingport-conf sysconf pname* = *None*
$\vee$ *get-portid-by-name s pname* $\neq$ *None*
$\vee$ *pname* $\notin$ *get-partition-cfg-ports-byid sysconf* (*current s*)
with *p3* **have** *d1*:*s'* = *s* **by** (*simp add*: *create-queuing-port-def*)
have *d2*:*get-queuingport-conf sysconf pname* = *None*
$\vee$ *get-portid-by-name t pname* $\neq$ *None*
$\vee$ *pname* $\notin$ *get-partition-cfg-ports-byid sysconf* (*current t*)
by (*meson disjoint-iff-not-equal port-partition*)
with *p4* **have** *d3*:*t'* = *t* **using** *create-queuing-port-def* **by** *auto*
with *a1 d1* **show** *?thesis* **by** *simp*
next
let *?nid* = *get-portid-in-type* (*the* (*get-queuingport-conf sysconf pname*))
assume *e0*:$\neg$(*get-queuingport-conf sysconf pname* = *None*
$\vee$ *get-portid-by-name s pname* $\neq$ *None*
$\vee$ *pname* $\notin$ *get-partition-cfg-ports-byid sysconf* (*current s*))
with *p3* **have** *e1*:*part-ports s'* = (*part-ports s*)(*?nid*:= (*Some* (*current s*)))
using *port-partition* **by** *auto*
with *p4 e0* **have** *e2*:*part-ports t'* = (*part-ports t*)(*?nid* := (*Some* (*current t*)))
using *port-partition* **by** *auto*
with *p3 e0* **have** *e3*:*ports* (*comm s'*) = (*ports* (*comm s*))(*?nid* := *get-queuingport-conf sysconf pname*)
using *port-partition* **by** *auto*
with *p4 e0* **have** *e4*:*ports* (*comm t'*) = (*ports* (*comm t*))(*?nid* := *get-queuingport-conf sysconf pname*)
using *port-partition* **by** *auto*
with *a1 e1 e2 e3 e4* **show** *?thesis* **using** *p5 sched-current-lemma* **by** *auto*
qed
}

```
  qed

lemma crt-que-port-presrv-comm-of-current-part:
 assumes  p1:reachable0 s ∧ reachable0 t
   and  p2:s ∼ (scheduler sysconf) ∼ t
   and  p3:s' = fst (create-queuing-port sysconf s pname)
   and  p4:t' = fst (create-queuing-port sysconf t pname)
   and  p5:(current s) = d
   and  p6:vpeq-part-comm s d t
   and  p7:is-a-partition sysconf d
  shows  vpeq-part-comm s' d t'
  apply(clarsimp,rule conjI)
  proof −
  {
   from p6 have a1:get-ports-of-partition s d = get-ports-of-partition t d
     by auto
   from p2 have a2:current s = current t using sched-current-lemma by auto
   show g0:get-ports-of-partition s' d = get-ports-of-partition t' d
   proof −
   {
    have ∀ p. p∈get-ports-of-partition s' d ⟶ p∈get-ports-of-partition t' d
     proof−
     {
      fix p
      assume a0:p∈get-ports-of-partition s' d
      have a3:p∈get-ports-of-partition t' d
         proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
          assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
          with a2 have b1:pname ∈ get-partition-cfg-ports-byid sysconf (current t) by simp
          have b2:p ≠ get-portid-in-type (the (get-queuingport-conf sysconf pname))
            using b0 port-partition by auto
          then show ?thesis using b0 port-partition by auto
        next
          assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
          with a2 have c1:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current t)) by simp
          have c2:s' = s by (simp add: c0 create-queuing-port-def p3)
          have c3:t' = t by (simp add: c1 create-queuing-port-def p4)
          then show ?thesis using a0 a1 c2 by auto
        qed
     }
     then show ?thesis by auto
     qed
    moreover
```

```
        have ∀ p. p∈get-ports-of-partition t' d ⟶ p∈get-ports-of-partition s' d
          proof−
          {
            fix p
            assume a0:p∈get-ports-of-partition t' d
            have a3:p∈get-ports-of-partition s' d
               proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
                assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
                with a2 have b1:pname ∈ get-partition-cfg-ports-byid sysconf (current t) by simp
                have b2:p ≠ get-portid-in-type (the (get-queuingport-conf sysconf pname))
                  using b0 port-partition by auto
                then show ?thesis using b0 port-partition by auto
              next
                assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
                with a2 have c1:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current t)) by simp
                have c2:s' = s by (simp add: c0 create-queuing-port-def p3)
                have c3:t' = t by (simp add: c1 create-queuing-port-def p4)
                then show ?thesis using a0 a1 c2 by auto
              qed
          }
          then show ?thesis by auto
          qed
        then show ?thesis using calculation by blast
      }
      qed
    next
      from p6 have a1:get-ports-of-partition s d = get-ports-of-partition t d
          by auto
      from p2 have a2:current s = current t using sched-current-lemma by auto
      show ∀ p. (is-dest-port s' p ⟶
       p ∈ get-ports-of-partition s' d ⟶
       is-a-queuingport s' p = is-a-queuingport t' p ∧ is-dest-port t' p ∧ get-port-buf-size s' p = get-port-buf-size t' p) ∧
      (¬ is-dest-port s' p ⟶
       p ∈ get-ports-of-partition s' d ⟶ is-a-queuingport s' p = is-a-queuingport t' p ∧ ¬ is-dest-port t' p)
        proof −
        {
          fix p
          have (is-dest-port s' p ⟶
             p ∈ get-ports-of-partition s' d ⟶
             is-a-queuingport s' p = is-a-queuingport t' p ∧ is-dest-port t' p ∧ get-port-buf-size s' p = get-port-buf-size t' p) ∧
            (¬ is-dest-port s' p ⟶
             p ∈ get-ports-of-partition s' d ⟶ is-a-queuingport s' p = is-a-queuingport t' p ∧ ¬ is-dest-port t' p)
            proof(cases pname ∈ get-partition-cfg-ports-byid sysconf (current s))
```

```
      assume b0:pname ∈ get-partition-cfg-ports-byid sysconf (current s)
        with a2 have b1:pname ∈ get-partition-cfg-ports-byid sysconf (current t) by simp
        have b2:p ≠ get-portid-in-type (the (get-queuingport-conf sysconf pname))
          using b0 port-partition by auto
        then show ?thesis using b0 port-partition by auto
      next
        assume c0:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current s))
        with a2 have c1:¬(pname ∈ get-partition-cfg-ports-byid sysconf (current t)) by simp
        have c2:s′ = s by (simp add: c0 create-queuing-port-def p3)
        have c3:t′ = t by (simp add: c1 create-queuing-port-def p4)
        then show ?thesis using c0 a1 c2 using p6  by auto
      qed
    }
    then show ?thesis by auto
    qed
  }
  qed

lemma crt-que-port-presrv-wk-stp-cons:
  assumes p1:is-a-partition sysconf (current s)
    and   p2:reachable0 s ∧ reachable0 t
    and   p3:s ∼ d ∼ t
    and   p4:s ∼ (scheduler sysconf) ∼ t
    and   p5:(current s) ⇝ d
    and   p6:s ∼ (current s) ∼ t
    and   p7:s′ = fst (create-queuing-port sysconf s pname)
    and   p8:t′ = fst (create-queuing-port sysconf t pname)
  shows   s′ ∼ d ∼ t′
proof(cases is-a-scheduler sysconf d)
 assume a0:is-a-scheduler sysconf d
 show ?thesis by (smt a0 interference1-def p1 p5 sche-imp-not-part)
next
 assume a1:¬ is-a-scheduler sysconf d
 show ?thesis
 proof(cases is-a-partition sysconf d)
  assume b0:is-a-partition sysconf d
  show ?thesis
  proof(cases current s = d)
   assume c0:current s = d
   have d0:vpeq-part s′ d t′
   proof −
   {
    have e1:partitions s′ d = partitions t′ d
```

```
    proof -
    {
      from p3 b0 have f1:partitions s d = partitions t d
        using a1 is-a-scheduler-def is-a-transmitter-def
        part-imp-not-tras vpeq1-def vpeq-part-def by fastforce
      from p7 have f2:partitions s d = partitions s' d
        using b0 c0 crt-que-port-notchg-partstate by auto
      from p8 have f3:partitions t d = partitions t' d
        using b0 c0 crt-que-port-notchg-partstate p4 sched-current-lemma
        by auto
      with f1 f2 have partitions s' d = partitions t' d by auto
    } then show ?thesis by auto
    qed
    have e2:vpeq-part-comm s' d t'
    proof -
    {
      from p3 a1 b0 have f1:vpeq-part-comm s d t
        using part-imp-not-tras by fastforce
      then have vpeq-part-comm s' d t'
        using c0 crt-que-port-presrv-comm-of-current-part p1 p2 p4 p7 p8 by auto
    } then show ?thesis by auto qed
    with e1 have vpeq-part s' d t' by auto
  } then show ?thesis by auto qed
  then show ?thesis using a1 b0
    using trans-imp-not-part by fastforce
next
  assume c1:current s ≠ d
  have d1:vpeq-part s' d t'
  proof -
  {
    have e1:partitions s' d = partitions t' d
      using a1 b0 crt-que-port-notchg-partstate
       p1 p3 p4 p7 p8 part-not-trans by auto
    have e2:vpeq-part-comm s' d t'
      using b0 c1 p1 p5 part-imp-not-sch part-imp-not-tras by auto
   with e1 have vpeq-part s' d t' by auto
  }
  then show ?thesis by auto qed
  show ?thesis using a1 b0 c1 trans-imp-not-part d1 by fastforce
qed
next
  assume b1:¬ is-a-partition sysconf d
  show ?thesis
```

proof(*cases is-a-transmitter sysconf d*)
assume *c0*:*is-a-transmitter sysconf d*
have *vpeq-transmitter* s' *d* t'
using *p3 p4 p7 p8 c0 crt-que-port-presrv-comm-part-ports*[*OF p2*] **by** *auto*
then show *?thesis* **using** *a1 b1 is-a-scheduler-def vpeq1-def* **by** *auto*
next
assume *c1*:$\neg$ *is-a-transmitter sysconf d*
show *?thesis* **using** *a1 b1 c1 is-a-transmitter-def vpeq1-def* **by** *auto*
qed
qed
qed

lemma *crt-que-port-presrv-wk-stp-cons-e*: *weak-step-consistent-e* (*hyperc* (*Create-Queuing-Port p*))
using *crt-que-port-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode*
by (*smt Event.case*(*1*) *Hypercall.case*(*6*) *domain-of-event.simps*(*1*) *event-enabled.simps*(*1*) *option.sel prod.simps*(*2*))

2.6.7 proving "send queuing message" satisfying the "step consistent" property

lemma *snd-que-msg-lst-presrv-comm-part-ports*:
assumes *p1*:*reachable0 s* $\wedge$ *reachable0 t*
and *p2*:*s* $\sim$ (*transmitter sysconf*) $\sim$ *t*
and *p5*:*s* $\sim$ (*scheduler sysconf*) $\sim$ *t*
and *p3*:s' = *fst* (*send-queuing-message-maylost sysconf s pid m*)
and *p4*:t' = *fst* (*send-queuing-message-maylost sysconf t pid m*)
shows *comm* s' = *comm* t' $\wedge$ *part-ports* s' = *part-ports* t'
proof –
{
from *p2* **have** *a0*:*vpeq-transmitter s* (*transmitter sysconf*) *t* **using** *sch-not-trans vpeq1-def* **by** *auto*
then have *a1*:*comm s* = *comm t* $\wedge$ *part-ports s* = *part-ports t* **unfolding** *vpeq-transmitter-def* **by** *auto*
from *p1* **have** *a2*:*port-consistent s* $\wedge$ *port-consistent t* **by** (*simp add*: *port-cons-reach-state*)
show *?thesis*
proof(*cases* $\neg$ *is-a-queuingport s pid*
$\vee$ $\neg$ *is-source-port s pid*
$\vee$ $\neg$ *is-a-port-of-partition s pid* (*current s*))
assume *b0*:$\neg$ *is-a-queuingport s pid*
$\vee$ $\neg$ *is-source-port s pid*
$\vee$ $\neg$ *is-a-port-of-partition s pid* (*current s*)
with *p3* **have** *b1*:s' = *s* **by** (*simp add*: *send-queuing-message-maylost-def*)
have *b2*:$\neg$ *is-a-queuingport t pid*
$\vee$ $\neg$ *is-source-port t pid*
$\vee$ $\neg$ *is-a-port-of-partition t pid* (*current t*)
using *a1 b0 is-a-port-of-partition-def is-a-queuingport-def is-source-port-def p5 sched-current-lemma*

```
        by (simp add: vpeq1-def vpeq-sched-def)
    with p4 have b3:t′ = t using send-queuing-message-maylost-def by auto
    with a1 b1 show ?thesis by simp
  next
    assume b1:¬(¬ is-a-queuingport s pid
             ∨ ¬ is-source-port s pid
             ∨ ¬ is-a-port-of-partition s pid (current s))
    show ?thesis
    proof(cases is-full-portqueuing sysconf s pid)
      assume c0:is-full-portqueuing sysconf s pid
      with b1 have c1:s′ = s by (simp add: p3 replace-msg2queuing-port-def
                              send-queuing-message-maylost-def)
      with a1 c0 have c2:is-full-portqueuing sysconf t pid
        by (simp add: get-port-conf-byid-def get-port-byid-def is-full-portqueuing-def)
      then have c3:t′ = t by (simp add: p4 replace-msg2queuing-port-def
                              send-queuing-message-maylost-def)
      with a1 c1 show ?thesis by auto
    next
      assume c0:¬ is-full-portqueuing sysconf s pid
      have c1:s′ = insert-msg2queuing-port s pid m
        using b1 c0 p3 send-queuing-message-maylost-def by auto
      with a1 c0 have c2:¬ is-full-portqueuing sysconf t pid
        by (simp add: get-port-conf-byid-def get-port-byid-def is-full-portqueuing-def)
      then have c3:t′ = insert-msg2queuing-port t pid m
        using b1 c2 p4 send-queuing-message-maylost-def a1 is-a-port-of-partition-def
          is-a-queuingport-def is-source-port-def old.prod.inject p5
          prod.collapse vpeq1-def vpeq-sched-def by auto
      with b1 show ?thesis
      proof(induct (ports (comm s)) pid)
        case None show ?case by (simp add: None.hyps a1 c1 c3 insert-msg2queuing-port-def option.case-eq-if)
      next
        case (Some x)
        have e0:(ports (comm s)) pid = Some x by (simp add: Some.hyps)
        show ?case
        proof(induct the ((ports (comm s)) pid))
          case (Queuing x1 x2 x3 x4 x5)
          show ?case
            by (smt Communication-State.surjective Communication-State.update-convs(1)
              Port-Type.simps(5) Queuing.hyps State.select-convs(3) State.select-convs(4)
              State.surjective State.update-convs(3) a1 c1 c3 insert-msg2queuing-port-def
              option.case-eq-if)
        next
          case (Sampling x1 x2 x3 x4)
```

```
          show ?case using Sampling.hyps a1 c1 c3 e0 insert-msg2queuing-port-def by auto
        qed
      qed
    qed
  qed
}
qed


lemma is-dest-queuing-send:
  t' = fst (send-queuing-message-maylost sysconf t pid m) ⟹
  (is-dest-port t p = is-dest-port t' p) ∧ (is-a-queuingport t p = is-a-queuingport t' p)
  apply(clarsimp simp:send-queuing-message-maylost-def replace-msg2queuing-port-def insert-msg2queuing-port-def)
  apply(case-tac ports (comm t) pid)
  apply simp
  apply(case-tac a)
  using is-a-queuingport-def is-dest-port-def
  by auto

lemma snd-que-msg-lst-presrv-comm-of-current-part:
assumes  p1:reachable0 s ∧ reachable0 t
    and  p2:s ∼ (scheduler sysconf) ∼ t
    and  p3:s' = fst (send-queuing-message-maylost sysconf s pid m)
    and  p4:t' = fst (send-queuing-message-maylost sysconf t pid m)
    and  p5:(current s) = d
    and  p6:vpeq-part-comm s d t
    and  p7:is-a-partition sysconf d
shows  vpeq-part-comm s' d t'
proof−
  from p6 have a1:get-ports-of-partition s d = get-ports-of-partition t d
    by auto
  from p2 have a2:current s = current t using sched-current-lemma by simp
  from p3 p5 p7 have a3:part-ports s = part-ports s' using snd-que-msg-lst-notchg-partports by simp
  then have a4:get-ports-of-partition s d = get-ports-of-partition s' d
    using part-ports-imp-portofpart by blast
  from p4 p5 p7 a2 have a5:part-ports t = part-ports t' using snd-que-msg-lst-notchg-partports by simp
  then have a6:get-ports-of-partition t d = get-ports-of-partition t' d
    using part-ports-imp-portofpart by blast
  have g0:get-ports-of-partition s' d = get-ports-of-partition t' d
    using a1 a4 a6 by simp
  moreover have ∀ p. p ∈ get-ports-of-partition s' d ⟶
          is-a-queuingport s' p = is-a-queuingport t' p ∧
```

```
        is-dest-port s′ p = is-dest-port t′ p ∧
        (if is-dest-port s′ p then get-port-buf-size s′ p = get-port-buf-size t′ p else True)
   using a4 by (metis empty-iff inf.idem no-cfgport-impl-noports p1 port-partition)
   ultimately show ?thesis by auto
 qed

lemma snd-que-msg-lst-presrv-wk-stp-cons:
   assumes p1:is-a-partition sysconf (current s)
     and p2:reachable0 s ∧ reachable0 t
     and p3:s ∼ d ∼ t
     and p4:s ∼ (scheduler sysconf) ∼ t
     and p5:(current s) ⇝ d
     and p6:s ∼ (current s) ∼ t
     and p7:s′ = fst (send-queuing-message-maylost sysconf s pid m)
     and p8:t′ = fst (send-queuing-message-maylost sysconf t pid m)
   shows s′ ∼ d ∼ t′
 proof(cases is-a-scheduler sysconf d)
   assume a0:is-a-scheduler sysconf d
   show ?thesis by (metis a0 interference1-def p1 p5 sche-imp-not-part)
 next
   assume a1:¬ is-a-scheduler sysconf d
   show ?thesis
   proof(cases is-a-partition sysconf d)
     assume b0:is-a-partition sysconf d
     show ?thesis
     proof(cases current s = d)
       assume c0:current s = d
       have d0:vpeq-part s′ d t′
       proof −
       {
         have e1:partitions s′ d = partitions t′ d
         proof −
         {
           from p3 b0 have f1:partitions s d = partitions t d
             using a1 part-imp-not-tras by fastforce
           from p7 have f2:partitions s d = partitions s′ d
             using b0 c0 snd-que-msg-lst-notchg-partstate by auto
           from p8 have f3:partitions t d = partitions t′ d
             using b0 c0 p4 sched-current-lemma snd-que-msg-lst-notchg-partstate
             by auto
           with f1 f2 have partitions s′ d = partitions t′ d by auto
         }
         then show ?thesis by auto
```

```
      qed
      have e2:vpeq-part-comm s' d t'
        proof -
          from p3 a1 b0 have f1:vpeq-part-comm s d t
            using part-imp-not-tras by fastforce
          then show ?thesis
            using c0 p1 p2 p4 p7 p8 snd-que-msg-lst-presrv-comm-of-current-part by auto
        qed
      with e1 have vpeq-part s' d t' by auto
    }
    then show ?thesis by auto qed
    then show ?thesis
      using  a1 b0 trans-imp-not-part by fastforce
  next
    assume c1:current s ≠ d
    have d1:vpeq-part s' d t'
    proof -
    {
      have e1:partitions s' d = partitions t' d
        using a1 b0 is-a-partition-def  p1 p3 p4 p7 p8
        part-not-trans snd-que-msg-lst-notchg-partstate
        by auto
      have e2:vpeq-part-comm s' d t'
        proof -
          from p3 a1 b0 have f1:vpeq-part-comm s d t
            using  part-imp-not-tras by fastforce
          have f2:vpeq-part-comm s d s'
            using c1 p1 p2 p7 snd-que-msg-lst-notchg-comminotherdom by blast
          have f3:vpeq-part-comm t d t'
            using c1 p1 p2 p4 p8 sched-current-lemma
                 snd-que-msg-lst-notchg-comminotherdom
            by (meson b0 interference1-def p5 part-imp-not-sch trans-imp-not-part)
          then show ?thesis
            using f1 f2 vpeq-part-comm-symmetric-lemma vpeq-part-comm-transitive-lemma by blast
        qed

      with e1 have vpeq-part s' d t'  by auto
    }
        then show ?thesis by auto
    qed
    show ?thesis using a1 b0 c1
      using trans-imp-not-part d1 by fastforce
  qed
```

next
assume *b1*:$\neg$ *is-a-partition sysconf d*
show *?thesis*
proof(*cases is-a-transmitter sysconf d*)
assume *c0*:*is-a-transmitter sysconf d*
show *?thesis*
proof –
{
have *vpeq-transmitter s′ d t′* **unfolding** *vpeq-transmitter-def*
proof–
from *p2 p3 p4 p7 p8*
show *comm s′* = *comm t′* $\wedge$ *part-ports s′* = *part-ports t′*
using *c0 snd-que-msg-lst-presrv-comm-part-ports*[*OF p2 - p4 p7 p8*] **by** *auto*
qed
}
then show *?thesis* **using** *a1 b1* **by** *auto*
qed
next
assume *c1*:$\neg$ *is-a-transmitter sysconf d*
show *?thesis* **using** *a1 b1 c1* **by** *auto*
qed
qed
qed

lemma *snd-que-msg-lst-presrv-wk-stp-cons-e*: *weak-step-consistent-e* (*hyperc* (*Send-Queuing-Message p m*))
using *snd-que-msg-lst-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq*
non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode vpeq1-def vpeq-sched-def
by (*smt Event.case*(*1*) *Hypercall.case*(*7*) *domain-of-event.simps*(*1*)
event-enabled.simps(*1*) *option.sel prod.simps*(*2*))

2.6.8 proving "receive queuing message" satisfying the "step consistent" property

lemma *rec-que-msg-presrv-comm-part-ports*:
assumes *p1*:*reachable0 s* $\wedge$ *reachable0 t*
and *p2*:*s* $\sim$ (*transmitter sysconf*) $\sim$ *t*
and *p5*:*s* $\sim$ (*scheduler sysconf*) $\sim$ *t*
and *p3*:*s′* = *fst* (*receive-queuing-message s pid*)
and *p4*:*t′* = *fst* (*receive-queuing-message t pid*)
shows *comm s′* = *comm t′* $\wedge$ *part-ports s′* = *part-ports t′*
proof –
{
from *p2* **have** *a0*:*vpeq-transmitter s* (*transmitter sysconf*) *t* **using** *sch-not-trans* **by** *auto*
then have *a1*:*comm s* = *comm t* $\wedge$ *part-ports s* = *part-ports t* **by** *auto*
from *p1* **have** *a2*:*port-consistent s* $\wedge$ *port-consistent t* **by** (*simp add*: *port-cons-reach-state*)

```
show ?thesis
proof(cases ¬ is-a-queuingport s pid
          ∨ ¬ is-a-port-of-partition s pid (current s)
          ∨ ¬ is-dest-port s pid
          ∨ is-empty-portqueuing s pid)
  assume b0:¬ is-a-queuingport s pid
          ∨ ¬ is-a-port-of-partition s pid (current s)
          ∨ ¬ is-dest-port s pid
          ∨ is-empty-portqueuing s pid
  with p3 have b1:s' = s by (simp add: receive-queuing-message-def)
  have b2:¬ is-a-queuingport t pid
          ∨ ¬ is-a-port-of-partition t pid (current t)
          ∨ ¬ is-dest-port t pid
          ∨ is-empty-portqueuing t pid
    using a1 b0 get-port-byid-def is-a-port-of-partition-def is-a-queuingport-def
    is-dest-port-def is-empty-portqueuing-def p5 sched-current-lemma by auto
  with p4 have b3:t' = t using receive-queuing-message-def by auto
  with a1 b1 show ?thesis by simp
next
  assume b1:¬(¬ is-a-queuingport s pid
          ∨ ¬ is-a-port-of-partition s pid (current s)
          ∨ ¬ is-dest-port s pid
          ∨ is-empty-portqueuing s pid)
  then have b2:s' = fst (remove-msg-from-queuingport s pid) by (simp add: p3 receive-queuing-message-def)
  with b1 have b3:¬(¬ is-a-queuingport t pid
                  ∨ ¬ is-a-port-of-partition t pid (current t)
                  ∨ ¬ is-dest-port t pid
                  ∨ is-empty-portqueuing t pid)
      using a1 get-port-byid-def is-a-port-of-partition-def is-a-queuingport-def
        is-dest-port-def is-empty-portqueuing-def p5 sched-current-lemma  by auto
  then have b4:t' = fst (remove-msg-from-queuingport t pid) by (simp add: p4 receive-queuing-message-def)
  then show ?thesis
  proof(induct (ports (comm s)) pid)
    case None show ?case using None.hyps a1 b2 b4 remove-msg-from-queuingport-def by auto
  next
    case (Some x)
    have e0:(ports (comm s)) pid = Some x by (simp add: Some.hyps)
    show ?case
      proof(induct the ((ports (comm s)) pid))
        case (Queuing x1 x2 x3 x4 x5)
        show ?case
          by (smt Communication-State.surjective Communication-State.update-convs(1)
            Port-Type.simps(5) Queuing.hyps State.select-convs(3) State.select-convs(4)
```

```
          State.surjective State.update-convs(3) a1 b2 b4 eq-fst-iff option.case-eq-if
          remove-msg-from-queuingport-def)
      next
        case (Sampling x1 x2 x3 x4)
        show ?case using Sampling.hyps a1 b2 b4 e0 remove-msg-from-queuingport-def by auto
      qed
    qed
  qed
}
qed

lemma is-dest-queuing-receive:
  t′ = fst (receive-queuing-message  t pid) ⟹
  (is-dest-port t p = is-dest-port t′ p) ∧ (is-a-queuingport t p = is-a-queuingport t′ p)
   apply(clarsimp simp:receive-queuing-message-def remove-msg-from-queuingport-def)
   apply(case-tac ports (comm t) pid)
   apply simp
   apply(case-tac a)
   using  is-a-queuingport-def is-dest-port-def
   by auto

lemma rec-que-msg-presrv-comm-of-current-part:
  assumes p1:reachable0 s ∧ reachable0 t
    and   p2:s ∼ (scheduler sysconf) ∼ t
    and   p3:s′ = fst (receive-queuing-message s pid)
    and   p4:t′ = fst (receive-queuing-message t pid)
    and   p5:(current s) = d
    and   p6:vpeq-part-comm s d t
    and   p7:is-a-partition sysconf d
  shows   vpeq-part-comm s′ d t′
  proof−
    from p6 have a1:get-ports-of-partition s d = get-ports-of-partition t d
        by auto
    from p2 have a2:current s = current t using  sched-current-lemma by simp
    from p3 p5 p7 have a3:part-ports s = part-ports s′ using rec-que-msg-notchg-partports by simp
    then have a4:get-ports-of-partition s d = get-ports-of-partition s′ d
      using part-ports-imp-portofpart by blast
    from p4 p5 p7 a2 have a5:part-ports t = part-ports t′ using rec-que-msg-notchg-partports by simp
    then have a6:get-ports-of-partition t d = get-ports-of-partition t′ d
      using part-ports-imp-portofpart by blast
    have g0:get-ports-of-partition s′ d = get-ports-of-partition t′ d
      using a1 a4 a6 by simp
    moreover have ∀ p. p ∈ get-ports-of-partition s′ d ⟶
```

```
        is-a-queuingport s' p = is-a-queuingport t' p ∧
        is-dest-port s' p = is-dest-port t' p ∧
        (if is-dest-port s' p then get-port-buf-size s' p = get-port-buf-size t' p else True)
  using p3 p4 is-dest-queuing-receive p6 a1 a4
    by (metis  empty-iff inf.idem no-cfgport-impl-noports p1 port-partition)
  ultimately show  ?thesis by auto
qed

lemma rec-que-msg-presrv-wk-stp-cons:
  assumes p1:is-a-partition sysconf (current s)
    and   p2:reachable0 s ∧ reachable0 t
    and   p3:s ∼ d ∼ t
    and   p4:s ∼ (scheduler sysconf) ∼ t
    and   p5:(current s) ⇝ d
    and   p6:s ∼ (current s) ∼ t
    and   p7:s' = fst (receive-queuing-message s pid)
    and   p8:t' = fst (receive-queuing-message t pid)
  shows   s' ∼ d ∼ t'
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  show ?thesis by (metis a0 interference1-def p1 p5 sche-imp-not-part)
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
  proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    show ?thesis
    proof(cases current s = d)
      assume c0:current s = d
      have d0:vpeq-part s' d t'
      proof -
      {
        have e1:partitions s' d = partitions t' d
        proof -
        {
          from p3 b0 have f1:partitions s d = partitions t d
            using a1 part-imp-not-tras by fastforce
          from p7 have f2:partitions s d = partitions s' d
            using b0 c0 rec-que-msg-notchg-partstate by auto
          from p8 have f3:partitions t d = partitions t' d
            using b0 c0 p4 sched-current-lemma rec-que-msg-notchg-partstate
            by auto
          with f1 f2 have partitions s' d = partitions t' d by auto
```

```
      }
      then show ?thesis by auto qed
      have e2:vpeq-part-comm s′ d t′ using rec-que-msg-presrv-comm-of-current-part
        by (metis (no-types, lifting) a1 b0 c0 is-a-scheduler-def is-a-transmitter-def
          p2 p3 p4 p7 p8 trans-imp-not-part vpeq1-def vpeq-part-def)

      with e1 have vpeq-part s′ d t′ by auto
    } then show ?thesis by auto qed
    then show ?thesis  using a1 b0
      using  trans-imp-not-part by fastforce
  next
    assume c1:current s ≠ d
    have d1:vpeq-part s′ d t′
    proof −
    {
      have e1:partitions s′ d = partitions t′ d
        using a1 b0  p1 p3 p4 p7 p8
        part-not-trans rec-que-msg-notchg-partstate  by auto

      have e2:vpeq-part-comm s′ d t′
      proof −
        from p3 a1 b0 have f1:vpeq-part-comm s d t
          using part-imp-not-tras by fastforce
        have f2:vpeq-part-comm s d s′
          using c1 p1 p2 p7 rec-que-msg-notchg-comminotherdom by blast
        have f3:vpeq-part-comm t d t′
          using c1 p1 p2 p4 p8 rec-que-msg-notchg-comminotherdom sched-current-lemma
          by (meson b0 interference1-def p5 part-imp-not-sch trans-imp-not-part)
        then show ?thesis
          using f1 f2 vpeq-part-comm-symmetric-lemma vpeq-part-comm-transitive-lemma by blast
      qed
      with e1 have vpeq-part s′ d t′ by auto
    } then show ?thesis by auto qed
    then show ?thesis using a1 b0 c1 trans-imp-not-part  by fastforce
  qed
next
  assume b1:¬ is-a-partition sysconf d
  show ?thesis
  proof(cases is-a-transmitter sysconf d)
    assume c0:is-a-transmitter sysconf d
    show ?thesis
    proof −
    {
```

have *vpeq-transmitter s′ d t′* **unfolding** *vpeq-transmitter-def*
proof−
from *p2 p3 p4 p7 p8*
show *comm* $s' =$ *comm* $t' \wedge$ *part-ports* $s' =$ *part-ports* t'
using *c0 rec-que-msg-presrv-comm-part-ports*[*OF p2*] **by** *auto*
qed
}
then show *?thesis* **using** *a1 b1 is-a-scheduler-def vpeq1-def* **by** *auto*
qed
next
assume *c1*:$\neg$ *is-a-transmitter sysconf d*
show *?thesis* **using** *a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def* **by** *auto*
qed
qed
qed

lemma *rec-que-msg-presrv-wk-stp-cons-e*: *weak-step-consistent-e* (*hyperc* (*Receive-Queuing-Message p*))
using *rec-que-msg-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode*
by (*smt Event.case*(*1*) *Hypercall.case*(*8*) *domain-of-event.simps*(*1*)
event-enabled.simps(*1*) *option.sel prod.simps*(*2*))

2.6.9 proving "get queuing portid" satisfying the "step consistent" property

lemma *get-que-pid-presrv-wk-stp-cons*:
assumes *p1*:$s \sim d \sim t$
and *p2*:$s' =$ *fst* (*get-queuing-port-id sysconf s pname*)
and *p3*:$t' =$ *fst* (*get-queuing-port-id sysconf t pname*)
shows $s' \sim d \sim t'$
proof −
have *a0*:$s' = s$ **by** (*simp add*: *p2 get-queuing-port-id-def*)
have *a1*:$t' = t$ **by** (*simp add*: *p3 get-queuing-port-id-def*)
then show *?thesis* **using** *a0 p1* **by** *blast*
qed

lemma *get-que-pid-presrv-wk-stp-cons-e*: *weak-step-consistent-e* (*hyperc* (*Get-Queuing-Portid p*))
using *get-que-pid-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode*
by (*smt Event.case*(*1*) *Hypercall.case*(*9*) *domain-of-event.simps*(*1*)
event-enabled.simps(*1*) *option.sel prod.simps*(*2*))

2.6.10 proving "get queuing port status" satisfying the "step consistent" property

lemma *get-que-psts-presrv-wk-stp-cons*:

assumes *p1*:*s* ∼ *d* ∼ *t*
and *p2*:*s′* = *fst* (*get-queuing-port-status sysconf s pid*)
and *p3*:*t′* = *fst* (*get-queuing-port-status sysconf t pid*)
shows *s′* ∼ *d* ∼ *t′*
proof −
have *a0*:*s′* = *s* **by** (*simp add*: *p2 get-queuing-port-status-def*)
have *a1*:*t′* = *t* **by** (*simp add*: *p3 get-queuing-port-status-def*)
then show *?thesis* **using** *a0 p1* **by** *blast*
qed

lemma *get-que-psts-presrv-wk-stp-cons-e*: *weak-step-consistent-e* (*hyperc* (*Get-Queuing-Portstatus p*))
using *get-que-psts-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq*
non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
by (*smt Event.case*(*1*) *Hypercall.case*(*10*) *domain-of-event.simps*(*1*) *event-enabled.simps*(*1*)
option.sel prod.simps(*2*) *vpeq1-def vpeq-sched-def*)

2.6.11 proving "clear queuing port" satisfying the "step consistent" property

lemma *clr-que-port-presrv-comm-part-ports*:
assumes *p1*:*reachable0 s* ∧ *reachable0 t*
and *p2*:*s* ∼ (*transmitter sysconf*) ∼ *t*
and *p5*:*s* ∼ (*scheduler sysconf*) ∼ *t*
and *p3*:*s′* = *clear-queuing-port s pid*
and *p4*:*t′* = *clear-queuing-port t pid*
shows *comm s′* = *comm t′* ∧ *part-ports s′* = *part-ports t′*
proof −
{
from *p2* **have** *a0*:*vpeq-transmitter s* (*transmitter sysconf*) *t* **using** *sch-not-trans* **by** *auto*
then have *a1*:*comm s* = *comm t* ∧ *part-ports s* = *part-ports t* **by** *auto*
from *p1* **have** *a2*:*port-consistent s* ∧ *port-consistent t* **by** (*simp add*: *port-cons-reach-state*)
show *?thesis*
proof(*cases* ¬ *is-a-queuingport s pid*
∨ ¬ *is-a-port-of-partition s pid* (*current s*)
∨ ¬ *is-dest-port s pid*)
assume *b0*:¬ *is-a-queuingport s pid*
∨ ¬ *is-a-port-of-partition s pid* (*current s*)
∨ ¬ *is-dest-port s pid*
with *p3* **have** *b1*:*s′* = *s* **by** (*simp add*: *clear-queuing-port-def*)
have *b2*:¬ *is-a-queuingport t pid*
∨ ¬ *is-a-port-of-partition t pid* (*current t*)
∨ ¬ *is-dest-port t pid*
using *a1 b0 get-port-byid-def is-a-port-of-partition-def is-a-queuingport-def*
is-dest-port-def is-empty-portqueuing-def p5 sched-current-lemma
by *auto*

```
    with p4 have b3:t' = t using clear-queuing-port-def by auto
    with a1 b1 show ?thesis by simp
  next
    assume b0:¬(¬ is-a-queuingport s pid
           ∨ ¬ is-a-port-of-partition s pid (current s)
           ∨ ¬ is-dest-port s pid)
    then have b00:¬(¬ is-a-queuingport t pid
           ∨ ¬ is-a-port-of-partition t pid (current t)
           ∨ ¬ is-dest-port t pid)
      using a1 is-a-port-of-partition-def is-a-queuingport-def
        is-dest-port-def p5 sched-current-lemma by auto
    with p3 have b1:part-ports s' = part-ports s
      by (metis Int-absorb a2 empty-iff is-a-port-of-partition-def
        option.distinct(1) port-consistent-def port-partition)

    with p4 b0 have b2:part-ports t' = part-ports t
      by (metis Int-absorb a2 empty-iff is-a-port-of-partition-def
        option.distinct(1) port-consistent-def port-partition)

    with p3 b0 have b3:ports (comm s') = (ports (comm s))
      (pid := Some (clear-msg-queuingport (the ((ports (comm s)) pid))))
      by (metis (no-types, lifting) Communication-State.select-convs(1)
        Communication-State.surjective Communication-State.update-convs(1)
        State.select-convs(3) State.surjective State.update-convs(3) clear-queuing-port-def)

    with p4 b00 have b4:ports (comm t') = (ports (comm t))
      (pid := Some (clear-msg-queuingport (the ((ports (comm t)) pid))))
      by (metis (no-types, lifting) Communication-State.ext-inject Communication-State.surjective
        Communication-State.update-convs(1) State.select-convs(3) State.surjective State.update-convs(3)
        clear-queuing-port-def)
    show ?thesis by (simp add: a1 b1 b2 b3 b4)
  qed
 }
 qed

lemma is-dest-queuing-clear:
  t' = clear-queuing-port t pid ⟹
  (is-dest-port t p = is-dest-port t' p) ∧ (is-a-queuingport t p = is-a-queuingport t' p)
  apply(clarsimp simp:clear-queuing-port-def Let-def)
  apply(clarsimp simp:clear-msg-queuingport-def)
  apply(case-tac ports (comm t) pid)
  apply (simp add: is-a-queuingport-def)
```

```
  apply(case-tac a)
  by (auto simp add: is-a-queuingport-def is-dest-port-def)

lemma clr-que-port-presrv-comm-of-current-part:
assumes p1:reachable0 s ∧ reachable0 t
  and   p2:s ∼ (scheduler sysconf) ∼ t
  and   p3:s′ = clear-queuing-port s pid
  and   p4:t′ = clear-queuing-port t pid
  and   p5:(current s) = d
  and   p6:vpeq-part-comm s d t
  and   p7:is-a-partition sysconf d
shows   vpeq-part-comm s′ d t′
proof−
  from p6 have a1:get-ports-of-partition s d = get-ports-of-partition t d
    by auto
  from p3 p5 p7 have a3:part-ports s = part-ports s′ using clr-que-port-notchg-partports by blast
  then have a4:get-ports-of-partition s d = get-ports-of-partition s′ d
    using part-ports-imp-portofpart by blast
  from p4 p5 p7 have a5:part-ports t = part-ports t′ using clr-que-port-notchg-partports by blast
  then have a6:get-ports-of-partition t d = get-ports-of-partition t′ d
    using part-ports-imp-portofpart by blast
  have g0:get-ports-of-partition s′ d = get-ports-of-partition t′ d
    using a1 a4 a6 by simp
  also have ∀ p. p ∈ get-ports-of-partition s′ d ⟶
          is-a-queuingport s′ p = is-a-queuingport t′ p ∧
          is-dest-port s′ p = is-dest-port t′ p ∧
          (if is-dest-port s′ p then get-port-buf-size s′ p = get-port-buf-size t′ p else True)
  using  a4 by (metis  empty-iff inf.idem no-cfgport-impl-noports p1 port-partition)
  ultimately show  ?thesis by auto
qed

lemma clr-que-port-presrv-wk-stp-cons:
  assumes p1:is-a-partition sysconf (current s)
    and   p2:reachable0 s ∧ reachable0 t
    and   p3:s ∼ d ∼ t
    and   p4:s ∼ (scheduler sysconf) ∼ t
    and   p5:(current s) ⇝ d
    and   p6:s ∼ (current s) ∼ t
    and   p7:s′ = clear-queuing-port s pid
    and   p8:t′ = clear-queuing-port t pid
  shows   s′ ∼ d ∼ t′
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
```

```
  show ?thesis by (metis a0 interference1-def p1 p5 sche-imp-not-part)
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
  proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    show ?thesis
    proof(cases current s = d)
      assume c0:current s = d
      have d0:vpeq-part s′ d t′
      proof −
      {
        have e1:partitions s′ d = partitions t′ d
          proof −
          {
            from p3 b0 have f1:partitions s d = partitions t d
              using a1 part-imp-not-tras by fastforce
            from p7 have f2:partitions s d = partitions s′ d
              using b0 c0 clr-que-port-notchg-partstate by auto
            from p8 have f3:partitions t d = partitions t′ d
              using b0 c0 p4 sched-current-lemma clr-que-port-notchg-partstate
              by auto
            with f1 f2 have partitions s′ d = partitions t′ d by auto
          }
          then show ?thesis by auto
          qed
        have e2:vpeq-part-comm s′ d t′ using clr-que-port-presrv-comm-of-current-part
          by (metis (no-types, lifting) a1 b0 c0 is-a-scheduler-def is-a-transmitter-def
            p2 p3 p4 p7 p8 trans-imp-not-part vpeq1-def vpeq-part-def)

        with e1 have vpeq-part s′ d t′  by auto
      }
      then show ?thesis by auto
      qed
      then show ?thesis using a1 b0
        using trans-imp-not-part by fastforce
      next
        assume c1:current s ≠ d
        have d1:vpeq-part s′ d t′
        proof −
        {
          have e1:partitions s′ d = partitions t′ d
            using a1 b0 clr-que-port-notchg-partstate
```

```
          p1 p3 p4 p7 p8 part-not-trans by auto

        have e2:vpeq-part-comm s' d t'
        proof –
          from p3 a1 b0 have f1:vpeq-part-comm s d t
            using  part-imp-not-tras  by fastforce
          have f2:vpeq-part-comm s d s' using c1 p1 p2 p7 clr-que-port-notchg-comminotherdom by blast
          have f3:vpeq-part-comm t d t'
            using c1 p1 p2 p4 p8 clr-que-port-notchg-comminotherdom sched-current-lemma
            by (meson b0 interference1-def p5 part-imp-not-sch trans-imp-not-part)
          then show ?thesis
            using f1 f2 vpeq-part-comm-symmetric-lemma vpeq-part-comm-transitive-lemma by blast
        qed
        with e1 have vpeq-part s' d t'  by auto
      }
      then show ?thesis by auto
      qed
      show ?thesis using a1 b0
        using  trans-imp-not-part d1 by fastforce
    qed
  next
    assume b1:¬ is-a-partition sysconf d
    show ?thesis
    proof(cases is-a-transmitter sysconf d)
      assume c0:is-a-transmitter sysconf d
      show ?thesis
      proof –
      {
        have vpeq-transmitter s' d t' unfolding vpeq-transmitter-def
        proof–
          from p2 p3 p4 p7 p8
          show comm s' = comm t' ∧ part-ports s' = part-ports t'
            using c0 clr-que-port-presrv-comm-part-ports[OF p2 - p4 p7 p8] by auto
        qed
      }
      then show ?thesis using a1 b1 is-a-scheduler-def vpeq1-def by auto
      qed
    next
      assume c1:¬ is-a-transmitter sysconf d
      show ?thesis using a1 b1 c1 by auto
    qed
  qed
qed
```

lemma *clr-que-port-presrv-wk-stp-cons-e*: *weak-step-consistent-e* (*hyperc* (*Clear-Queuing-Port p*))
using *clr-que-port-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq*
non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
by (*smt Event.case*(*1*) *Hypercall.case*(*11*) *domain-of-event.simps*(*1*) *event-enabled.simps*(*1*)
option.sel prod.simps(*2*) *vpeq1-def vpeq-sched-def*)

2.6.12 proving "set partition mode" satisfying the "step consistent" property

lemma *set-part-mode-presrv-wk-stp-cons*:
assumes *p1*:*is-a-partition sysconf* (*current s*)
and *p2*:*reachable0 s* $\wedge$ *reachable0 t*
and *p3*:$s \sim d \sim t$
and *p4*:$s \sim$ (*scheduler sysconf*) $\sim t$
and *p5*:(*current s*) $\rightsquigarrow d$
and *p6*:$s \sim$ (*current s*) $\sim t$
and *p7*:s' = *set-partition-mode sysconf s m*
and *p8*:t' = *set-partition-mode sysconf t m*
shows $s' \sim d \sim t'$
proof(*cases is-a-scheduler sysconf d*)
assume *a0*:*is-a-scheduler sysconf d*
show *?thesis* **by** (*metis a0 interference1-def p1 p5 sche-imp-not-part*)
next
assume *a1*:$\neg$ *is-a-scheduler sysconf d*
show *?thesis*
proof(*cases is-a-partition sysconf d*)
assume *b0*:*is-a-partition sysconf d*
show *?thesis*
proof(*cases current s = d*)
assume *c0*:*current s = d*
have *d0*:*vpeq-part* s' *d* t'
proof −
{
have *e1*:*partitions* s' *d* = *partitions* t' *d*
proof −
{
from *p3 b0* **have** *f1*:*partitions s d* = *partitions t d*
using *a1 part-imp-not-tras* **by** *fastforce*
from *p4 c0* **have** *f2*: *current t = d*
using *sched-current-lemma* **by** *auto*
then have *partitions* s' *d* = *partitions* t' *d*
using *set-partition-mode-def p7 p8 c0 f1 f2* **by** *auto*
}
then show *?thesis* **by** *auto*

```
      qed
      have e2:vpeq-part-comm s' d t'
      proof -
      {
        from p3 a1 b0 have f1:vpeq-part-comm s d t
          using part-imp-not-tras by fastforce
        then have vpeq-part-comm s' d t'
          by (metis (mono-tags, lifting) emptyE inf.idem no-cfgport-impl-noports
              p1 p2 p4 p7 p8 part-ports-imp-portofpart port-partition
              set-part-mode-notchg-partports vpeq1-def vpeq-part-comm-def vpeq-sched-def)
      }
      then show ?thesis by auto qed
      with e1 have vpeq-part s' d t'  by auto
    } then show ?thesis by auto qed
    then show ?thesis using a1 b0 trans-imp-not-part by fastforce
  next
    assume c1:current s ≠ d
    have d1:vpeq-part s' d t'
    proof -
    {
      from p4 c1 have f2: current t ≠ d
            using sched-current-lemma vpeq1-def vpeq-sched-def by auto
      have e1:partitions s' d = partitions t' d
        using a1 b0 f2  p1 p3 p4 p7 p8
          part-not-trans set-part-mode-notchg-partstate-inotherdom
        by auto
      have e2:vpeq-part-comm s' d t'
        by (metis (mono-tags, hide-lams) a1 b0 c1 is-a-partition-def is-a-scheduler-def
          p1 p2 p3 p4 p7 p8 part-not-trans set-part-mode-notchg-comm vpeq1-def
          vpeq-part-comm-symmetric-lemma vpeq-part-comm-transitive-lemma vpeq-part-def vpeq-sched-def)
     with e1 have vpeq-part s' d t' using vpeq-part-def by auto
    } then show ?thesis by auto qed
    show ?thesis  using a1 b0 trans-imp-not-part d1 by fastforce
  qed
next
  assume b1:¬ is-a-partition sysconf d
  show ?thesis
  proof(cases is-a-transmitter sysconf d)
    assume c0:is-a-transmitter sysconf d
    with p3 have c1: vpeq-transmitter s d t using vpeq1-def is-a-transmitter-def sch-not-trans by auto
    show ?thesis
    proof -
    {
```

```
      have vpeq-transmitter s′ d t′ unfolding vpeq-transmitter-def
      proof(rule conjI)
        show comm s′ = comm t′
          by (metis (mono-tags, lifting) c1 is-a-partition-def p1 p2 p4
            p7 p8 sch-not-part set-part-mode-notchg-comm2 vpeq1-def
            vpeq-sched-def vpeq-transmitter-def)
        show part-ports s′ = part-ports t′
          using c1 p1 p4 p7 p8 sched-current-lemma
          set-part-mode-notchg-partports vpeq-transmitter-def vpeq1-def vpeq-sched-def by auto
      qed
      } then show ?thesis using a1 b1 by auto qed
    next
      assume c1:¬ is-a-transmitter sysconf d
      show ?thesis using a1 b1 c1 is-a-scheduler-def is-a-transmitter-def vpeq1-def by auto
    qed
  qed
qed

lemma set-part-mode-presrv-wk-stp-cons-e: weak-step-consistent-e (hyperc (Set-Partition-Mode p))
  using set-part-mode-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
    non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
      by (smt Event.case(1) Hypercall.case(12) domain-of-event.simps(1) event-enabled.simps(1)
        option.sel prod.simps(2) vpeq1-def vpeq-sched-def)
```

2.6.13 proving "get partition status" satisfying the "step consistent" property

```
lemma get-part-status-presrv-wk-stp-cons:
    assumes p1:s ∼ d ∼ t
      and   p2:s′ = fst (get-partition-status sysconf s)
      and   p3:t′ = fst (get-partition-status sysconf t)
    shows   s′ ∼ d ∼ t′
  proof −
    have a0:s′ = s by (simp add: p2 get-partition-status-def)
    have a1:t′ = t by (simp add: p3 get-partition-status-def)
    then show ?thesis using a0 p1 by blast
  qed

lemma get-part-status-presrv-wk-stp-cons-e: weak-step-consistent-e (hyperc Get-Partition-Status)
  using get-part-status-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
    non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
      by (smt Event.case(1) Hypercall.case(13) domain-of-event.simps(1) event-enabled.simps(1)
        option.sel prod.simps(2) vpeq1-def vpeq-sched-def)
```

2.6.14 proving "schedule" satisfying the "step consistent" property

```
lemma schedule-presrv-wk-stp-cons:
  assumes p1:reachable0 s ∧ reachable0 t
    and   p3:s ∼ d ∼ t
    and   p5:(scheduler sysconf) ⇝ d
    and   p6:s ∼ (scheduler sysconf) ∼ t
    and   p7:s' ∈ schedule sysconf s
    and   p8:t' ∈ schedule sysconf t
  shows   s' ∼ d ∼ t'
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  from p7 p8 have current s' = current t' unfolding schedule-def by simp
  with a0 show ?thesis using a0 p3 p7 p8 schedule-def by auto
next
  assume a1:¬ is-a-scheduler sysconf d
  show ?thesis
  proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    with p3 have b00:vpeq-part s d t unfolding vpeq1-def
      by (metis a1 is-a-scheduler-def is-a-transmitter-def trans-imp-not-part)
    have b1:vpeq-part s' d t'
    proof −
    {
      have c1:partitions s' d = partitions t' d
      proof −
      {
        from p3 b0 have f1:partitions s d = partitions t d
          using a1 part-imp-not-tras by fastforce
        from p7 have f2:partitions s d = partitions s' d
          by (simp add: schedule-def)
        from p8 have f3:partitions t d = partitions t' d
          by (simp add: schedule-def)
        with f1 f2 have partitions s' d = partitions t' d by auto
      }
      then show ?thesis by auto
      qed
      have c2:vpeq-part-comm s' d t'
      proof −
        from p7 have d1:part-ports s = part-ports s' by (simp add: schedule-def)
        from p7 have d2:comm s = comm s' by (simp add: schedule-def)
        with p7 d1 have d3:vpeq-part-comm s d s' unfolding vpeq-part-comm-def get-ports-of-partition-def
          is-a-queuingport-def is-dest-port-def get-port-buf-size-def get-port-byid-def
          get-current-bufsize-port-def by simp
```

```
        from p8 have d4:part-ports t = part-ports t′ by (simp add: schedule-def)
        from p8 have d5:comm t = comm t′ by (simp add: schedule-def)
        with p8 d4 have vpeq-part-comm t d t′ unfolding vpeq-part-comm-def get-ports-of-partition-def
          is-a-queuingport-def is-dest-port-def get-port-buf-size-def get-port-byid-def
          get-current-bufsize-port-def by simp
        with b00 d1 d2 d3 d4 d5 show ?thesis
          by (meson vpeq-part-comm-symmetric-lemma vpeq-part-comm-transitive-lemma vpeq-part-def)
      qed
      with c1 have vpeq-part s′ d t′ using vpeq-part-def by simp
    }
    then show ?thesis by auto
    qed
    then show ?thesis
    using a1 b0  trans-imp-not-part  by auto
  next
    assume b1:¬ is-a-partition sysconf d
    show ?thesis using p3  p7 p8 sch-not-trans schedule-def a1 b1  by auto
  qed
qed

lemma schedule-presrv-wk-stp-cons-e: weak-step-consistent-e (sys Schedule)
  using schedule-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
    non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
      by (smt Event.case(2) System-Event.case(1) domain-of-event.simps(2) event-enabled.simps(2)
        option.sel prod.simps(2) vpeq1-def vpeq-sched-def)
```

2.6.15 proving "Transfer Sampling Message" satisfying the "step consistent" property

```
lemma trans-smpl-msg-presrv-comm-part-ports:
assumes   p1:reachable0 s ∧ reachable0 t
    and   p2:s ∼ (transmitter sysconf) ∼ t
    and   p5:s ∼ (scheduler sysconf) ∼ t
    and   p3:s′ = transf-sampling-msg s c
    and   p4:t′ = transf-sampling-msg t c
shows   comm s′ = comm t′ ∧ part-ports s′ = part-ports t′
proof −
{
  from p2 have a0:vpeq-transmitter s (transmitter sysconf) t using sch-not-trans vpeq1-def by auto
  then have a1:comm s = comm t ∧ part-ports s = part-ports t by auto
  from p1 have a2:port-consistent s ∧ port-consistent t by (simp add: port-cons-reach-state)
  from p3 p4 show ?thesis
  proof(induct c)
    case (Channel-Sampling name sn dns)
    show ?case
```

```
    proof(cases get-portid-by-name s sn≠None ∧ card (get-portids-by-names s dns) = card dns)
      let ?pids = the (get-portid-by-name s sn)
      let ?pidt = the (get-portid-by-name t sn)
      let ?pidss = get-portids-by-names s dns
      let ?pidst = get-portids-by-names t dns
      let ?m = the (get-the-msg-of-samplingport s ?pids)
      let ?m′ = the (get-the-msg-of-samplingport t ?pidt)
      let ?s′ = update-sampling-ports-msg s ?pidss ?m
      let ?t′ = update-sampling-ports-msg t ?pidst ?m′
      assume b0:get-portid-by-name s sn≠None ∧ card (get-portids-by-names s dns) = card dns
      with a1 have b1:get-portid-by-name t sn≠None ∧ card (get-portids-by-names t dns) = card dns
        unfolding get-portid-by-name-def is-port-name-def get-portids-by-names-def by presburger
      from b0 have b2:s′ = ?s′ using Channel-Sampling.prems(1) by auto
      from b1 have b3:t′ = ?t′ using Channel-Sampling.prems(2) by auto
      from a1 have b4:?m = ?m′ unfolding get-the-msg-of-samplingport-def get-port-byid-def get-portid-by-name-def
        is-port-name-def get-msg-from-samplingport-def by auto
      from a1 have b5:?pids = ?pidt unfolding get-portid-by-name-def is-port-name-def by simp
      from a1 have b6:?pidss = ?pidst unfolding get-portids-by-names-def get-portid-by-name-def
        is-port-name-def by simp
      with a1 b4 b5 have a7:comm ?s′ = comm ?t′ ∧ part-ports ?s′ = part-ports ?t′
        unfolding update-sampling-ports-msg-def by simp
      with p3 p4 a1 b2 b3 show ?thesis by simp
    next
      assume b0:¬(get-portid-by-name s sn≠None ∧ card (get-portids-by-names s dns) = card dns)
      with a1 have b1:¬(get-portid-by-name t sn≠None ∧ card (get-portids-by-names t dns) = card dns)
        unfolding get-portid-by-name-def is-port-name-def get-portids-by-names-def by presburger
      with a1 b0 b1 Channel-Sampling show ?thesis by auto
    qed
  next
    case (Channel-Queuing nm sn dn)
    show ?case by (simp add: Channel-Queuing.prems(1) Channel-Queuing.prems(2) a1)
  qed
}
qed

lemma trans-smpl-msg-presrv-wk-stp-cons:
    assumes p1:is-a-transmitter sysconf (current s)
      and   p2:reachable0 s ∧ reachable0 t
      and   p3:s ∼ d ∼ t
      and   p4:s ∼ (scheduler sysconf) ∼ t
      and   p5:(current s) ⇝ d
      and   p6:s ∼ (current s) ∼ t
      and   p7:s′ = transf-sampling-msg s c
```

```
      and   p8:t′ = transf-sampling-msg t c
     shows   s′ ∼ d ∼ t′
proof(cases is-a-scheduler sysconf d)
  assume a0:is-a-scheduler sysconf d
  show ?thesis using a0 no-intf-sched-help p1 p5 sch-not-trans by auto
next
  assume a1:¬ is-a-scheduler sysconf d
  have a2:comm s′ = comm t′ ∧ part-ports s′ = part-ports t′
    using p1 p6 is-a-transmitter-def trans-smpl-msg-presrv-comm-part-ports[OF p2 - p4 p7 p8]
    by metis
  show ?thesis
  proof(cases is-a-partition sysconf d)
    assume b0:is-a-partition sysconf d
    show ?thesis
    proof −
      have d0:vpeq-part s′ d t′
      proof −
        have e1:partitions s′ d = partitions t′ d
          using a1 b0   p1 p3 p4 p7 p8
          part-imp-not-tras sched-current-lemma trans-smpl-msg-notchg-partstate
          by force
        from a2 have e2:vpeq-part-comm s′ d t′
          unfolding vpeq-part-comm-def  get-ports-of-partition-def is-a-queuingport-def
          is-dest-port-def get-port-buf-size-def get-current-bufsize-port-def get-port-byid-def
           by simp
        with e1 show ?thesis by auto
      qed
      then show ?thesis  using a1 b0
        using trans-imp-not-part by fastforce
    qed
  next
    assume b1:¬ is-a-partition sysconf d
    show ?thesis
    proof(cases is-a-transmitter sysconf d)
      assume c0:is-a-transmitter sysconf d
      show ?thesis
      proof −
      {
        have vpeq-transmitter s′ d t′ unfolding vpeq-transmitter-def
        proof−
          from p3  p7 p8
          show comm s′ = comm t′ ∧ part-ports s′ = part-ports t′
            using c0 trans-smpl-msg-presrv-comm-part-ports[OF p2 - p4]  by auto
```

```
      qed
    }
    then show ?thesis using a1 b1 is-a-scheduler-def vpeq1-def by auto
    qed
  next
    assume c1:¬ is-a-transmitter sysconf d
    show ?thesis using a1 b1 c1 by auto
  qed
 qed
qed
```

```
lemma trans-smpl-msg-presrv-wk-stp-cons-e: weak-step-consistent-e (sys (Transfer-Sampling-Message c))
 using trans-smpl-msg-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
  non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
   by (smt Event.case(2) System-Event.case(2) domain-of-event.simps(2) event-enabled.simps(2)
     option.sel prod.simps(2) is-a-transmitter-def vpeq1-def vpeq-sched-def)
```

2.6.16 proving "Transfer Queuing Message" satisfying the "step consistent" property


```
lemma trans-que-msg-mlost-presrv-comm-part-ports:
 assumes  p1:reachable0 s ∧ reachable0 t
   and  p2:s ∼ (transmitter sysconf) ∼ t
   and  p5:s ∼ (scheduler sysconf) ∼ t
   and  p3:s′ = transf-queuing-msg-maylost sysconf s c
   and  p4:t′ = transf-queuing-msg-maylost sysconf t c
  shows  comm s′ = comm t′ ∧ part-ports s′ = part-ports t′
  proof −
  {
   from p2 have a0:vpeq-transmitter s (transmitter sysconf) t using sch-not-trans vpeq1-def by auto
   then have a1:comm s = comm t ∧ part-ports s = part-ports t unfolding vpeq-transmitter-def by auto
   from p1 have a2:port-consistent s ∧ port-consistent t by (simp add: port-cons-reach-state)
   from p3 p4 show ?thesis
    proof(induct c)
     case (Channel-Queuing nm sn dn)
      show ?case
      proof(cases get-portid-by-name s sn ≠ None ∧ get-portid-by-name s dn ≠ None
            ∧ has-msg-inportqueuing s (the (get-portid-by-name s sn)))
       let ?sps = the (get-portid-by-name s sn)
       let ?spt = the (get-portid-by-name t sn)
       let ?dps = the (get-portid-by-name s dn)
       let ?dpt = the (get-portid-by-name t dn)
       let ?s1 = fst (remove-msg-from-queuingport s ?sps)
       let ?t1 = fst (remove-msg-from-queuingport t ?spt)
       let ?ms = snd (remove-msg-from-queuingport s ?sps)
```

let *?mt* = *snd* (*remove-msg-from-queuingport t ?spt*)
let *?s2* = *replace-msg2queuing-port ?s1 ?dps* (*the ?ms*)
let *?t2* = *replace-msg2queuing-port ?t1 ?dpt* (*the ?mt*)
let *?s3* = *insert-msg2queuing-port ?s1 ?dps* (*the ?ms*)
let *?t3* = *insert-msg2queuing-port ?t1 ?dpt* (*the ?mt*)
assume *b0*:*get-portid-by-name s sn* ≠ *None* ∧ *get-portid-by-name s dn* ≠ *None*
∧ *has-msg-inportqueuing s* (*the* (*get-portid-by-name s sn*))
with *a1* **have** *b1*:*get-portid-by-name t sn* ≠ *None* ∧ *get-portid-by-name t dn* ≠ *None*
∧ *has-msg-inportqueuing t* (*the* (*get-portid-by-name t sn*))
by (*metis get-portid-by-name-def has-msg-inportqueuing-def*)

from *a1* **have** *b2*:*?sps* = *?spt* **unfolding** *get-portid-by-name-def is-port-name-def* **by** *simp*
from *a1* **have** *b3*:*?dps* = *?dpt* **unfolding** *get-portid-by-name-def is-port-name-def* **by** *simp*
with *b2 a1* **have** *b4*:*comm ?s1* = *comm ?t1*
apply(*clarsimp simp*:*remove-msg-from-queuingport-def*)
apply(*case-tac ports* (*comm t*) *?spt*)
apply(*simp*)
apply(*case-tac a*)
apply (*smt Communication-State.surjective Communication-State.update-convs*(*1*)
Port-Type.simps(*5*) *State.select-convs*(*3*) *State.surjective State.update-convs*(*3*)
fstI option.simps(*5*))
by *simp*

with *b2 a1* **have** *b5*:*part-ports ?s1* = *part-ports ?t1*
apply(*clarsimp simp*:*remove-msg-from-queuingport-def*)
apply(*case-tac ports* (*comm t*) *?spt*)
apply(*simp*)
apply(*case-tac a*)
apply (*smt Port-Type.simps*(*5*) *State.select-convs*(*4*) *State.surjective*
State.update-convs(*3*) *fstI option.simps*(*5*))
by *simp*

from *a1 b2 b3* **have** *b6*:*?ms* = *?mt*
apply(*clarsimp simp*:*remove-msg-from-queuingport-def*)
apply(*case-tac ports* (*comm t*) *?spt*)
apply(*simp*)
apply(*case-tac a*)
apply (*metis* (*no-types, lifting*) *Port-Type.simps*(*5*) *option.simps*(*5*) *prod.collapse prod.inject*)
by *simp*

from *b4 b5 a1* **have** *b7*:*comm ?s2* = *comm ?t2* ∧ *part-ports ?s2* = *part-ports ?t2*
unfolding *replace-msg2queuing-port-def* **by** *simp*

```
from b3 b4 b5 b6 a1 have b8:comm ?s3 = comm ?t3
  apply(clarsimp simp:insert-msg2queuing-port-def)
  apply(case-tac ports (comm ?t1) ?dpt)
  apply(simp)
  apply(case-tac a)
  apply (metis Int-absorb a2 empty-iff option.distinct(1) port-consistent-def
    port-partition remove-msg-from-queuingport-presv-port-cons)
  by simp

from b3 b4 b5 b6 a1 have b9:part-ports ?s3 = part-ports ?t3
  apply(clarsimp simp:insert-msg2queuing-port-def)
  apply(case-tac ports (comm ?t1) ?dpt)
  apply(simp)
  apply(case-tac a)
  apply (metis Int-absorb a2 empty-iff option.distinct(1) port-consistent-def
    port-partition remove-msg-from-queuingport-presv-port-cons)
  by simp

show ?thesis
  proof(cases is-full-portqueuing sysconf ?s1 ?dps)
    assume c0:is-full-portqueuing sysconf ?s1 ?dps
    with b3 b4 have c1:is-full-portqueuing sysconf ?t1 ?dpt
      unfolding is-full-portqueuing-def Let-def get-port-conf-byid-def
      get-max-bufsize-of-port-def get-current-bufsize-port-def get-port-byid-def by auto
    from p3 b0 c0 have c2:s′ = ?s2
      by (smt Channel-Queuing.prems(1) transf-queuing-msg-maylost.simps(1))
    from p4 b1 c1 have c3:t′ = ?t2
      by (smt Channel-Queuing.prems(2) transf-queuing-msg-maylost.simps(1))

    with b7 c2 show ?thesis by simp
  next
    assume c0:¬is-full-portqueuing sysconf ?s1 ?dps
    with b3 b4 have c1:¬is-full-portqueuing sysconf ?t1 ?dpt
      unfolding is-full-portqueuing-def Let-def get-port-conf-byid-def
      get-max-bufsize-of-port-def get-current-bufsize-port-def get-port-byid-def by auto
    from p3 b0 c0 have c2:s′ = ?s3
      by (smt Channel-Queuing.prems(1) transf-queuing-msg-maylost.simps(1))
    from p4 b1 c1 have c3:t′ = ?t3
      by (smt Channel-Queuing.prems(2) transf-queuing-msg-maylost.simps(1))

    with b8 b9 c2 show ?thesis by simp
  qed
next
```

assume *b0*:$\neg$(*get-portid-by-name s sn* $\neq$ *None* $\wedge$ *get-portid-by-name s dn* $\neq$ *None*
$\wedge$ *has-msg-inportqueuing s* (*the* (*get-portid-by-name s sn*)))
with *a1* **have** *b1*:$\neg$(*get-portid-by-name t sn* $\neq$ *None* $\wedge$ *get-portid-by-name t dn* $\neq$ *None*
$\wedge$ *has-msg-inportqueuing t* (*the* (*get-portid-by-name t sn*)))
by (*metis get-portid-by-name-def has-msg-inportqueuing-def*)
with *p3 b0* **have** *b2*:$s' = s$ **unfolding** *transf-queuing-msg-maylost-def*
Let-def Channel-Queuing.prems(*1*) **by** *auto*
with *p4 b1* **have** *b3*:$t' = t$ **unfolding** *transf-queuing-msg-maylost-def*
Let-def Channel-Queuing.prems(*2*) **by** *auto*
with *a1 b2* **show** *?thesis* **by** *simp*
qed
next
case (*Channel-Sampling x1 x2 x3*)
show *?case* **by** (*simp add*: *Channel-Sampling.prems*(*1*) *Channel-Sampling.prems*(*2*) *a1*)
qed
}
qed

lemma *trans-que-msg-mlost-presrv-wk-stp-cons*:
assumes *p1*:*is-a-transmitter sysconf* (*current s*)
and *p2*:*reachable0 s* $\wedge$ *reachable0 t*
and *p3*:$s \sim d \sim t$
and *p4*:$s \sim$ (*scheduler sysconf*) $\sim t$
and *p5*:(*current s*) $\rightsquigarrow d$
and *p6*:$s \sim$ (*current s*) $\sim t$
and *p7*:s' = *transf-queuing-msg-maylost sysconf s c*
and *p8*:t' = *transf-queuing-msg-maylost sysconf t c*
shows $s' \sim d \sim t'$
proof(*cases is-a-scheduler sysconf d*)
assume *a0*:*is-a-scheduler sysconf d*
show *?thesis* **using** *a0 no-intf-sched-help p1 p5 sch-not-trans* **by** *auto*
next
assume *a1*:$\neg$ *is-a-scheduler sysconf d*
have *a2*:*comm* s' = *comm* t' $\wedge$ *part-ports* s' = *part-ports* t'
using *p1 p6 trans-que-msg-mlost-presrv-comm-part-ports*[*OF p2 - p4 p7 p8*]
by (*metis is-a-transmitter-def*)
show *?thesis*
proof(*cases is-a-partition sysconf d*)
assume *b0*:*is-a-partition sysconf d*
show *?thesis*
proof $-$
have *d0*:*vpeq-part* s' *d* t'
proof $-$

```
        have e1:partitions s′ d = partitions t′ d
          using a1 b0  p1 p3 p4 p7 p8
          part-imp-not-tras sched-current-lemma trans-que-msg-mlost-notchg-partstate
          by force
        from a2 have e2:vpeq-part-comm s′ d t′
          unfolding vpeq-part-comm-def Let-def get-ports-of-partition-def is-a-queuingport-def
          is-dest-port-def get-port-buf-size-def get-current-bufsize-port-def get-port-byid-def by simp
        with e1 show ?thesis by auto
      qed
      then show ?thesis  using a1 b0
        using trans-imp-not-part by fastforce
    qed
  next
    assume b1:¬ is-a-partition sysconf d
    show ?thesis
    proof(cases is-a-transmitter sysconf d)
      assume c0:is-a-transmitter sysconf d
      show ?thesis
      proof −
      {
        have vpeq-transmitter s′ d t′ unfolding vpeq-transmitter-def
        proof−
          from p2 p3 p4 p7 p8
          show comm s′ = comm t′ ∧ part-ports s′ = part-ports t′
            using c0 trans-que-msg-mlost-presrv-comm-part-ports by force
        qed
      }
      then show ?thesis using a1 b1 by auto
      qed
    next
      assume c1:¬ is-a-transmitter sysconf d
      show ?thesis using a1 b1 c1 by auto
    qed
  qed
qed

lemma trans-que-msg-mlost-presrv-wk-stp-cons-e: weak-step-consistent-e (sys (Transfer-Queuing-Message c))
  using trans-que-msg-mlost-presrv-wk-stp-cons weak-step-consistent-e-def exec-event-def mem-Collect-eq
    non-interference1-def non-interference-def singletonD sched-vpeq same-part-mode
      by (smt Event.case(2) System-Event.case(3) domain-of-event.simps(2) event-enabled.simps(2)
          option.sel prod.simps(2) is-a-transmitter-def vpeq1-def vpeq-sched-def)
```

2.6.17 proving the "weakly step consistent" property

theorem *weak-step-consistent*:*weak-step-consistent*
proof –
{
fix *e*
have *weak-step-consistent-e e*
apply(*induct e*)
using *crt-smpl-port-presrv-wk-stp-cons-e wrt-smpl-msg-presrv-wk-stp-cons-e read-smpl-msg-presrv-wk-stp-cons-e get-smpl-pid-presrv-wk-stp-cons-e get-smpl-psts-presrv-wk-stp-cons-e crt-que-port-presrv-wk-stp-cons-e snd-que-msg-lst-presrv-wk-stp-cons-e rec-que-msg-presrv-wk-stp-cons-e get-que-pid-presrv-wk-stp-cons-e get-que-psts-presrv-wk-stp-cons-e clr-que-port-presrv-wk-stp-cons-e set-part-mode-presrv-wk-stp-cons-e get-part-status-presrv-wk-stp-cons-e*
apply (*rule Hypercall.induct*)
using *schedule-presrv-wk-stp-cons-e trans-smpl-msg-presrv-wk-stp-cons-e trans-que-msg-mlost-presrv-wk-stp-cons-e*
by (*rule System-Event.induct*)
}
then show *?thesis* **using** *weak-step-consistent-all-evt* **by** *blast*
qed

2.7 Information flow security of top-level specification

theorem *noninfluence-sat*: *noninfluence*
using *local-respect uc-eq-noninf weak-step-consistent weak-with-step-cons* **by** *blast*

theorem *noninfluence-gen-sat*: *noninfluence-gen*
using *noninf-eq-noninf-gen noninfluence-sat* **by** *blast*

theorem *weak-noninfluence-sat*: *weak-noninfluence* **using** *noninf-impl-weak noninfluence-sat* **by** *blast*

theorem *nonleakage-sat*: *nonleakage*
using *noninf-impl-nonlk noninfluence-sat* **by** *blast*

theorem *noninterference-r-sat*: *noninterference-r*
using *noninf-impl-nonintf-r noninfluence-sat* **by** *blast*

theorem *noninterference-sat*: *noninterference*
using *noninterference-r-sat nonintf-r-impl-noninterf* **by** *blast*

theorem *weak-noninterference-r-sat*: *weak-noninterference-r*
using *noninterference-r-sat nonintf-r-impl-wk-nonintf-r* **by** *blast*

theorem *weak-noninterference-sat*: *weak-noninterference*
using *noninterference-sat nonintf-impl-weak* **by** *blast*

end

3 Second-level specification and security proofs

theory *SK-L2Spec*
imports *SK-SecurityModel SK-TopSpec*

begin

declare [[*smt-timeout* = *90*]]

3.1 Definitions

3.1.1 Data type, basic components, and state

type-synonym *process-id* = *nat*
datatype *process-state* = *DORMANT* | *READY* | *WAITING* | *SUSPEND* | *RUNNING*

type-synonym *proc-priority-type* = *nat*

record *Proc-State* = *state* :: *process-state*
priority :: *proc-priority-type*

record *StateR* = *State* +
procs :: *partition-id* ⇀ (*process-id set*)
cur-proc-part :: *partition-id* ⇀ *process-id*
proc-state :: *partition-id* × *process-id* ⇀ *Proc-State*

definition *abstract-state* :: *StateR* ⇒ *State* (⇑- [*50*])
where *abstract-state* r = ⦇*current* = *current* r,
partitions = *partitions* r,
comm = *comm* r,
part-ports = *part-ports* r
⦈

definition *abstract-state-rev* :: *StateR* ⇒ *State* ⇒ *StateR* (-⇓- [*50*])
where *abstract-state-rev* r' r = r'⦇*current* := *current* r,
partitions := *partitions* r,
comm := *comm* r,

part-ports := *part-ports r*⦈

3.1.2 Events

datatype *Hypercall′* = *Create-Sampling-Port port-name*
| *Write-Sampling-Message port-id Message*
| *Read-Sampling-Message port-id*
| *Get-Sampling-Portid port-name*
| *Get-Sampling-Portstatus port-id*
| *Create-Queuing-Port port-name*
| *Send-Queuing-Message port-id Message*
| *Receive-Queuing-Message port-id*
| *Get-Queuing-Portid port-name*
| *Get-Queuing-Portstatus port-id*
| *Clear-Queuing-Port port-id*
| *Set-Partition-Mode partition-mode-type*
| *Get-Partition-Status*
| *Create-Process proc-priority-type*
| *Start-Process process-id*
| *Stop-Process process-id*
| *Resume-Process process-id*
| *Suspend-Process process-id*
| *Set-Priority process-id proc-priority-type*
| *Get-Process-Status process-id*

datatype *System-EventR* = *Schedule*
| *Transfer-Sampling-Message Channel-Type*
| *Transfer-Queuing-Message Channel-Type*
| *Schedule-Process*

datatype *EventR* = *hyperc′ Hypercall′* | *sys′ System-EventR*

3.1.3 Event specification

definition *create-sampling-portR* :: $\mathit{Sys\text{-}Config} \Rightarrow \mathit{StateR} \Rightarrow \mathit{port\text{-}name} \Rightarrow (\mathit{StateR} \times \mathit{port\text{-}id\ option})$ **where**
$\mathit{create\text{-}sampling\text{-}portR}\ sc\ s\ p \equiv \mathit{let}\ s'= (\mathit{create\text{-}sampling\text{-}port}\ sc\ (\Uparrow s)\ p)\ \mathit{in}\ (s\Downarrow(\mathit{fst}\ s'),\mathit{snd}\ s')$

definition *write-sampling-messageR* :: $\mathit{StateR} \Rightarrow \mathit{port\text{-}id} \Rightarrow \mathit{Message} \Rightarrow (\mathit{StateR} \times \mathit{bool})$ **where**
$\mathit{write\text{-}sampling\text{-}messageR}\ s\ p\ m \equiv \mathit{let}\ s'= (\mathit{write\text{-}sampling\text{-}message}\ (\Uparrow s)\ p\ m)\ \mathit{in}\ (s\Downarrow(\mathit{fst}\ s'),\mathit{snd}\ s')$

definition *read-sampling-messageR* :: $\mathit{StateR} \Rightarrow \mathit{port\text{-}id} \Rightarrow (\mathit{StateR} \times \mathit{Message\ option})$ **where**
$\mathit{read\text{-}sampling\text{-}messageR}\ s\ \mathit{pid} \equiv \mathit{let}\ s'= (\mathit{read\text{-}sampling\text{-}message}\ (\Uparrow s)\ \mathit{pid})\ \mathit{in}\ (s\Downarrow(\mathit{fst}\ s'),\mathit{snd}\ s')$

definition *get-sampling-port-idR* :: $\mathit{Sys\text{-}Config} \Rightarrow \mathit{StateR} \Rightarrow \mathit{port\text{-}name} \Rightarrow (\mathit{StateR} \times \mathit{port\text{-}id\ option})$ **where**

get-sampling-port-idR sc s pname ≡ let s′= (get-sampling-port-id sc (⇑s) pname) in (s⇓(fst s′),snd s′)

definition *get-sampling-port-statusR* :: *Sys-Config ⇒ StateR ⇒ port-id ⇒ (StateR × Port-Type option)* **where**
get-sampling-port-statusR sc s pid ≡ let s′= (get-sampling-port-status sc (⇑s) pid) in (s⇓(fst s′),snd s′)

definition *create-queuing-portR* :: *Sys-Config ⇒ StateR ⇒ port-name ⇒ (StateR × port-id option)* **where**
create-queuing-portR sc s p ≡ let s′= (create-queuing-port sc (⇑s) p) in (s⇓(fst s′),snd s′)

definition *send-queuing-message-maylostR* :: *Sys-Config ⇒ StateR ⇒ port-id ⇒ Message ⇒ (StateR × bool)* **where**
send-queuing-message-maylostR sc s p m ≡
    let s′= (send-queuing-message-maylost sc (⇑s) p m) in (s⇓(fst s′),snd s′)

definition *receive-queuing-messageR* :: *StateR ⇒ port-id ⇒ (StateR × Message option)* **where**
receive-queuing-messageR s pid ≡ let s′= (receive-queuing-message (⇑s) pid) in (s⇓(fst s′),snd s′)

definition *get-queuing-port-idR* :: *Sys-Config ⇒ StateR ⇒ port-name ⇒ (StateR × port-id option)* **where**
get-queuing-port-idR sc s pname ≡ let s′= (get-queuing-port-id sc (⇑s) pname) in (s⇓(fst s′),snd s′)

definition *get-queuing-port-statusR* :: *Sys-Config ⇒ StateR ⇒ port-id ⇒ (StateR × Port-Type option)* **where**
get-queuing-port-statusR sc s pid ≡ let s′= (get-queuing-port-status sc (⇑s) pid) in (s⇓(fst s′),snd s′)

definition *clear-queuing-portR* :: *StateR ⇒ port-id ⇒ StateR* **where**
clear-queuing-portR s pid ≡ let s′= (clear-queuing-port (⇑s) pid) in (s⇓s′)

definition *setRun2Ready* :: *StateR ⇒ StateR* **where**
setRun2Ready s ≡ if is-a-partition sysconf (current s) ∧ cur-proc-part s (current s) ≠ None then
        let prs = proc-state s;
            cur = the ((cur-proc-part s) (current s));
            stt = the (prs (current s, cur)) in
        s⦇cur-proc-part := ((cur-proc-part s)) (current s := None),
            proc-state := prs((current s, cur) := Some (stt⦇state:=READY⦈))⦈
        else s

definition *schedule-process* :: *StateR ⇒ StateR set* **where**
schedule-process s ≡ if is-a-partition sysconf (current s)
            ∧ part-mode (the ((partitions s) (current s))) = NORMAL then
        (let s′ = setRun2Ready s;
            readyprs = {p. p∈the (procs s′ (current s′)) ∧
                state (the (proc-state s′ (current s′,p))) = READY};
            selp = SOME p. p∈{x. state (the (proc-state s′ (current s′,x))) = READY ∧
                    (∀ y∈readyprs. priority (the (proc-state s′ (current s′,x))) ≥
                        priority (the (proc-state s′ (current s′,y))))};
            st = the ((proc-state s′) (current s′, selp));

proc-st = *proc-state s′*;
cur-pr = *cur-proc-part s′ in*
{*s′*⦇*proc-state* := *proc-st* ((*current s′*, *selp*) := *Some* (*st*⦇*state* := *RUNNING*⦈)),
cur-proc-part := *cur-pr*(*current s′* := *Some selp*)⦈})
else
{*s*}

definition *scheduleR* :: *Sys-Config* ⇒ *StateR* ⇒ *StateR set* **where**
scheduleR sc s ≡ ⋃ *s′*∈*schedule sc* (⇑*s*). {*s*⇓*s′*}

definition *get-partition-statusR* ::
Sys-Config ⇒ *StateR* ⇒ (*StateR* × (*Partition-Conf option*) × (*Partition-State-Type option*)) **where**
get-partition-statusR sc s ≡ *let s′*= (*get-partition-status sc* (⇑*s*)) *in* (*s*⇓(*fst s′*),*snd s′*)

definition *remove-partition-resources* :: *StateR* ⇒ *partition-id* ⇒ *StateR* **where**
remove-partition-resources s part ≡
let proc-state′ = (λ(*pt*, *p*). *if pt* = *part then None else* (*proc-state s*) (*pt*,*p*));
procs′ = (*procs s*)(*part* := *None*) *in*
s⦇*procs* := *procs′*, *proc-state*:=*proc-state′*⦈

definition *set-procs-to-normal* :: *StateR* ⇒ *partition-id* ⇒ *StateR* **where**
set-procs-to-normal s part ≡ *if is-a-partition sysconf part then*
let prs = *proc-state s*;
proc-state′ = (λ(*pt*, *p*).
(*let pst* = *prs* (*pt*,*p*) *in*
if pt = *part* ∧ *state* (*the pst*) = *WAITING*
then Some ((*the pst*)⦇*state*:=*READY*⦈)
else prs (*pt*,*p*))) *in*
s⦇*proc-state* := *proc-state′*⦈
else s

definition *set-partition-modeR* :: *Sys-Config* ⇒ *StateR* ⇒ *partition-mode-type* ⇒ *StateR* **where**
set-partition-modeR sc s m ≡
(*if* (*partconf sc*) (*current s*) ≠ *None* ∧ (*partitions s*) (*current s*) ≠ *None* ∧
¬ (*part-mode* (*the* ((*partitions s*) (*current s*))) = *COLD-START* ∧ *m* = *WARM-START*) *then*
let pts = *partitions s*;
pstate = *the* (*pts* (*current s*));
s′ = (*if m* = *NORMAL then*
set-procs-to-normal s (*current s*)
else if part-mode (*the* ((*partitions s*) (*current s*))) = *NORMAL then*
remove-partition-resources s (*current s*)
else s)
in s′⦇*partitions* := *pts*(*current s′* := *Some* (*pstate*⦇*part-mode* := *m*⦈))⦈

```
    else
      s)


definition transf-sampling-msgR :: StateR ⇒ Channel-Type ⇒ StateR where
  transf-sampling-msgR s c ≡
         let s'= (transf-sampling-msg (⇑s) c) in (s⇓s')

definition transf-queuing-msg-maylostR :: Sys-Config ⇒ StateR ⇒ Channel-Type ⇒ StateR where
  transf-queuing-msg-maylostR sc s c ≡
         let s'= (transf-queuing-msg-maylost sc (⇑s) c) in (s⇓s')

definition create-process :: StateR ⇒ proc-priority-type ⇒ (StateR × process-id option) where
  create-process s pri ≡ if part-mode (the ((partitions s) (current s))) = WARM-START
                          ∨ part-mode (the ((partitions s) (current s))) = COLD-START
                         then
                          let pid = (SOME p. p ∉ the ((procs s) (current s)));
                              procs' = (procs s) ((current s) := Some ((the ((procs s) (current s))) ∪ {pid}));
                              proc-state' = (proc-state s) ((current s,pid) := Some (|state = DORMANT, priority = pri|)) in
                          (s(|procs:=procs', proc-state:=proc-state'|), Some pid)
                         else (s, None)

definition set-process-priority :: StateR ⇒ process-id ⇒ proc-priority-type ⇒ StateR where
  set-process-priority s p pri ≡
      if (proc-state s) (current s, p) ≠ None ∧ (state (the ((proc-state s) (current s, p)))) ≠ DORMANT then
          let st = state (the ((proc-state s) (current s, p)));
              proc-state' = (proc-state s) ((current s,p) := Some (|state = st, priority = pri|)) in
              s(|proc-state:=proc-state'|)
       else s

definition start-process :: StateR ⇒ process-id ⇒ StateR where
  start-process s p ≡ if p ∈ the ((procs s) (current s)) ∧ (proc-state s) (current s, p) ≠ None
                      ∧ (state (the ((proc-state s) (current s, p)))) = DORMANT then
                        let st = (if part-mode (the ((partitions s) (current s))) = NORMAL
                                  then READY
                                  else WAITING);
                            pst = (the ((proc-state s) (current s, p)));
                            proc-state' = (proc-state s) ((current s, p) := Some (pst (|state := st|))) in
                             s(|proc-state:=proc-state'|)
                      else s

definition stop-process :: StateR ⇒ process-id ⇒ StateR where
  stop-process s p ≡ if p ∈ the ((procs s) (current s)) ∧ (proc-state s) (current s, p) ≠ None
```

∧ (*state* (*the* ((*proc-state s*) (*current s*, *p*)))) ≠ *DORMANT then*
let pri = *priority* (*the* ((*proc-state s*) (*current s*, *p*)));
proc-state′ = (*proc-state s*) ((*current s*, *p*) := *Some* ⦇*state* = *DORMANT*,*priority* = *pri*⦈) *in*
s⦇*proc-state*:=*proc-state′*⦈
else s

definition *suspend-process* :: *StateR* ⇒ *process-id* ⇒ *StateR* **where**
suspend-process s p ≡ *if p* ∈ *the* ((*procs s*) (*current s*)) ∧ (*proc-state s*) (*current s*, *p*) ≠ *None*
∧ (*state* (*the* ((*proc-state s*) (*current s*, *p*)))) ≠ *DORMANT*
∧ (*state* (*the* ((*proc-state s*) (*current s*, *p*)))) ≠ *SUSPEND then*
let pri = *priority* (*the* ((*proc-state s*) (*current s*, *p*)));
proc-state′ = (*proc-state s*) ((*current s*, *p*) := *Some* ⦇*state* = *SUSPEND*,*priority* = *pri*⦈) *in*
s⦇*proc-state*:=*proc-state′*⦈
else s

definition *resume-process* :: *StateR* ⇒ *process-id* ⇒ *StateR* **where**
resume-process s p ≡ *if p* ∈ *the* ((*procs s*) (*current s*)) ∧ (*proc-state s*) (*current s*, *p*) ≠ *None*
∧ (*state* (*the* ((*proc-state s*) (*current s*, *p*)))) = *SUSPEND then*
let pri = *priority* (*the* ((*proc-state s*) (*current s*, *p*)));
proc-state′ = (*proc-state s*) ((*current s*, *p*) := *Some* ⦇*state* = *READY*,*priority* = *pri*⦈) *in*
s⦇*proc-state*:=*proc-state′*⦈
else s

definition *get-process-status* :: *StateR* ⇒ *process-id* ⇒ (*StateR* × (*Proc-State option*)) **where**
get-process-status s p ≡ (*s*,(*proc-state s*) (*current s*, *p*))

definition *system-initR* :: *Sys-Config* ⇒ *StateR*
where *system-initR sc* ≡ *let s0* = *system-init sc in*
⦇*current* = *current s0*,
partitions = *partitions s0*,
comm = *comm s0*,
part-ports = *part-ports s0*,
procs = (λ *x*. *None*),
cur-proc-part = (λ *x*. *None*),
proc-state = (λ *x*. *None*)
⦈
declare *abstract-state-def*[*cong del*] **and** *abstract-state-rev-def*[*cong del*] **and**
create-sampling-portR-def [*cong del*] **and** *write-sampling-messageR-def*[*cong del*] **and**
read-sampling-messageR-def[*cong del*] **and** *get-sampling-port-idR-def*[*cong del*] **and**
get-sampling-port-statusR-def[*cong del*] **and** *create-queuing-portR-def*[*cong del*] **and**
send-queuing-message-maylostR-def[*cong del*] **and** *receive-queuing-messageR-def*[*cong del*] **and**

get-queuing-port-idR-def [*cong del*] **and** *get-queuing-port-statusR-def* [*cong del*] **and**
clear-queuing-portR-def [*cong del*] **and** *setRun2Ready-def* [*cong del*] **and** *schedule-process-def* [*cong del*] **and**
scheduleR-def [*cong del*] **and** *get-partition-statusR-def* [*cong del*] **and** *remove-partition-resources-def* [*cong del*] **and**
set-procs-to-normal-def [*cong del*] **and** *set-partition-modeR-def* [*cong*] **and** *transf-sampling-msgR-def* [*cong del*] **and**
transf-queuing-msg-maylostR-def [*cong del*] **and** *create-process-def* [*cong*] **and** *set-process-priority-def* [*cong del*] **and**
start-process-def [*cong del*] **and** *stop-process-def* [*cong*] **and** *suspend-process-def* [*cong del*] **and**
resume-process-def [*cong del*] **and** *get-process-status-def* [*cong del*] **and** *set-partition-mode-def* [*cong del*]

declare *abstract-state-def* [*cong*] **and** *abstract-state-rev-def* [*cong*] **and**
create-sampling-portR-def [*cong*] **and** *write-sampling-messageR-def* [*cong*] **and**
read-sampling-messageR-def [*cong*] **and** *get-sampling-port-idR-def* [*cong*] **and**
get-sampling-port-statusR-def [*cong*] **and** *create-queuing-portR-def* [*cong*] **and**
send-queuing-message-maylostR-def [*cong*] **and** *receive-queuing-messageR-def* [*cong*] **and**
get-queuing-port-idR-def [*cong*] **and** *get-queuing-port-statusR-def* [*cong*] **and**
clear-queuing-portR-def [*cong*] **and** *setRun2Ready-def* [*cong*] **and** *schedule-process-def* [*cong*] **and**
scheduleR-def [*cong*] **and** *get-partition-statusR-def* [*cong*] **and** *remove-partition-resources-def* [*cong*] **and**
set-procs-to-normal-def [*cong*] **and** *set-partition-modeR-def* [*cong*] **and** *transf-sampling-msgR-def* [*cong*] **and**
transf-queuing-msg-maylostR-def [*cong*] **and** *create-process-def* [*cong*] **and** *set-process-priority-def* [*cong*] **and**
start-process-def [*cong*] **and** *stop-process-def* [*cong*] **and** *suspend-process-def* [*cong*] **and**
resume-process-def [*cong*] **and** *get-process-status-def* [*cong*] **and** *set-partition-mode-def* [*cong*] **and** *schedule-def* [*cong*]

3.2 Instantiation and Its Proofs of Security Model

consts *s0r* :: *StateR*

specification (*s0r*)
s0r-init: *s0r* = *system-initR sysconf*
by *simp*

primrec *event-enabledR* :: *StateR* ⇒ *EventR* ⇒ *bool*
where *event-enabledR-hc*: *event-enabledR s* (*hyperc′ h*) = (*is-a-partition sysconf* (*current s*)
∧ *part-mode* (*the* (*partitions s* (*current s*))) ≠ *IDLE*) |
event-enabledR-sys: *event-enabledR s* (*sys′ h*) = (*case h of Schedule* ⇒ *True* |
Transfer-Sampling-Message c ⇒ (*current s* = *transmitter sysconf*) |
Transfer-Queuing-Message c ⇒ (*current s* = *transmitter sysconf*) |
Schedule-Process ⇒ (*is-a-partition sysconf* (*current s*)
∧ *part-mode* (*the* (*partitions s* (*current s*))) = *NORMAL*))

definition *exec-eventR* :: *EventR* ⇒ (*StateR* × *StateR*) *set* **where**
exec-eventR e = {(*s*,*s′*). *s′*∈ (*if event-enabledR s e then* (
case e of hyperc′ (*Create-Sampling-Port pname*) ⇒ {*fst* (*create-sampling-portR sysconf s pname*)} |
hyperc′ (*Write-Sampling-Message p m*) ⇒ {*fst* (*write-sampling-messageR s p m*)} |
hyperc′ (*Read-Sampling-Message p*) ⇒ {*fst* (*read-sampling-messageR s p*)} |
hyperc′ (*Get-Sampling-Portid pname*) ⇒ {*fst* (*get-sampling-port-idR sysconf s pname*)} |

```
            hyperc′ (Get-Sampling-Portstatus p) ⇒ {fst (get-sampling-port-statusR sysconf s p)} |
            hyperc′ (Create-Queuing-Port pname) ⇒ {fst (create-queuing-portR sysconf s pname)} |
            hyperc′ (Send-Queuing-Message p m) ⇒ {fst (send-queuing-message-maylostR sysconf s p m)} |
            hyperc′ (Receive-Queuing-Message p) ⇒ {fst (receive-queuing-messageR s p)} |
            hyperc′ (Get-Queuing-Portid pname) ⇒ {fst (get-queuing-port-idR sysconf s pname)} |
            hyperc′ (Get-Queuing-Portstatus p) ⇒ {fst (get-queuing-port-statusR sysconf s p)} |
            hyperc′ (Clear-Queuing-Port p) ⇒ {clear-queuing-portR s p} |
            hyperc′ (Set-Partition-Mode m) ⇒ {set-partition-modeR sysconf s m} |
            hyperc′ (Get-Partition-Status) ⇒ {fst (get-partition-statusR sysconf s)} |
            hyperc′ (Create-Process pri) ⇒ {fst (create-process s pri)} |
            hyperc′ (Start-Process p) ⇒ {start-process s p} |
            hyperc′ (Stop-Process p) ⇒ {stop-process s p} |
            hyperc′ (Resume-Process p) ⇒ {resume-process s p} |
            hyperc′ (Suspend-Process p) ⇒ {suspend-process s p} |
            hyperc′ (Set-Priority p pri) ⇒ {set-process-priority s p pri} |
            hyperc′ (Get-Process-Status p) ⇒ {fst (get-process-status s p)} |
            sys′ Schedule ⇒ scheduleR sysconf s |
            sys′ (Transfer-Sampling-Message c) ⇒ {transf-sampling-msgR s c} |
            sys′ (Transfer-Queuing-Message c) ⇒ {transf-queuing-msg-maylostR sysconf s c} |
            sys′ (Schedule-Process) ⇒ schedule-process s )
              else {s})}

definition eR :: EventR ⇒ Event option where
  eR e ≡
    case e of hyperc′ (Create-Sampling-Port pname) ⇒ Some (hyperc (Hypercall.Create-Sampling-Port pname)) |
            hyperc′ (Write-Sampling-Message p m) ⇒ Some (hyperc (Hypercall.Write-Sampling-Message p m)) |
            hyperc′ (Read-Sampling-Message p) ⇒ Some (hyperc (Hypercall.Read-Sampling-Message p)) |
            hyperc′ (Get-Sampling-Portid pname) ⇒ Some (hyperc (Hypercall.Get-Sampling-Portid pname)) |
            hyperc′ (Get-Sampling-Portstatus p) ⇒ Some (hyperc (Hypercall.Get-Sampling-Portstatus p)) |
            hyperc′ (Create-Queuing-Port pname) ⇒ Some (hyperc (Hypercall.Create-Queuing-Port pname)) |
            hyperc′ (Send-Queuing-Message p m) ⇒ Some (hyperc (Hypercall.Send-Queuing-Message p m)) |
            hyperc′ (Receive-Queuing-Message p) ⇒ Some (hyperc (Hypercall.Receive-Queuing-Message p)) |
            hyperc′ (Get-Queuing-Portid pname) ⇒ Some (hyperc (Hypercall.Get-Queuing-Portid pname)) |
            hyperc′ (Get-Queuing-Portstatus p) ⇒ Some (hyperc (Hypercall.Get-Queuing-Portstatus p)) |
            hyperc′ (Clear-Queuing-Port p) ⇒ Some (hyperc (Hypercall.Clear-Queuing-Port p)) |
            hyperc′ (Set-Partition-Mode m) ⇒ Some (hyperc (Hypercall.Set-Partition-Mode m)) |
            hyperc′ (Get-Partition-Status) ⇒ Some (hyperc (Hypercall.Get-Partition-Status)) |
            hyperc′ (Create-Process pri) ⇒ None |
            hyperc′ (Start-Process p) ⇒ None |
            hyperc′ (Stop-Process p) ⇒ None |
            hyperc′ (Resume-Process p) ⇒ None |
            hyperc′ (Suspend-Process p) ⇒ None |
            hyperc′ (Set-Priority p pri) ⇒ None |
```

```
            hyperc' (Get-Process-Status p) ⇒ None |
            sys' Schedule ⇒  Some (sys (System-Event.Schedule)) |
            sys' (Transfer-Sampling-Message c) ⇒ Some (sys (System-Event.Transfer-Sampling-Message c)) |
            sys' (Transfer-Queuing-Message c) ⇒ Some (sys (System-Event.Transfer-Queuing-Message c)) |
            sys' (Schedule-Process) ⇒ None

primrec domain-of-eventR :: StateR ⇒ EventR ⇒ domain-id option
  where domain-of-eventR-hc: domain-of-eventR s (hyperc' h) = Some (current s) |
        domain-of-eventR-sys: domain-of-eventR s (sys' h) =  (case h of Schedule ⇒ Some (scheduler sysconf) |
                                        Transfer-Sampling-Message c ⇒ Some (transmitter sysconf) |
                                        Transfer-Queuing-Message c ⇒ Some (transmitter sysconf) |
                                        Schedule-Process ⇒ Some (current s) )

lemma domain-domainR : eR e ≠ None ⟹ domain-of-eventR s e = domain-of-event (⇑s) (the (eR e))
  proof(induct e)
    case (hyperc' x) then show ?case
      proof(induct x) qed(simp add:eR-def)+
  next
    case (sys' x) then show ?case
      proof(induct x) qed(simp add:eR-def)+
  qed

definition vpeq-part-procs :: StateR ⇒ domain-id ⇒ StateR ⇒ bool ((- ∼. - .∼△ -))
  where vpeq-part-procs s d t ≡ if is-a-partition sysconf d then
                                ((procs s) d = (procs t) d) ∧
                                (∀ p. (proc-state s) (d,p) = (proc-state t) (d,p)) ∧
                                (cur-proc-part s) d = (cur-proc-part t) d
                              else True

lemma vpeq-part-procs-transitive-lemma :
  ∀ s t r d. (vpeq-part-procs s d t) ∧ (vpeq-part-procs t d r) ⟶ (vpeq-part-procs s d r)
  using vpeq-part-procs-def by auto

lemma vpeq-part-procs-symmetric-lemma : ∀ s t d. (vpeq-part-procs s d t) ⟶ (vpeq-part-procs t d s)
  using vpeq-part-procs-def by auto

lemma vpeq-part-procs-reflexive-lemma : ∀ s d. (vpeq-part-procs s d s)
  using vpeq-part-procs-def by auto

definition vpeqR :: StateR ⇒ domain-id ⇒ StateR ⇒ bool ((- ∼. - .∼ -))
  where vpeqR s d t ≡  ((⇑s) ∼d∼ (⇑t)) ∧ (s∼.d.∼△t)

declare vpeqR-def[cong] and vpeq-part-procs-def[cong]
```

lemma *vpeqR-transitive-lemma* : $\forall\ s\ t\ r\ d.\ (vpeqR\ s\ d\ t) \wedge (vpeqR\ t\ d\ r) \longrightarrow (vpeqR\ s\ d\ r)$
  apply(*clarsimp cong del*: *vpeq1-def*)
  using *vpeq1-transitive-lemma vpeq-part-procs-transitive-lemma* **by** *blast*

lemma *vpeqR-symmetric-lemma* : $\forall\ s\ t\ d.\ (vpeqR\ s\ d\ t) \longrightarrow (vpeqR\ t\ d\ s)$
 apply(*clarsimp cong del*: *vpeq1-def*)
 using *vpeq1-symmetric-lemma vpeq-part-procs-symmetric-lemma* **by** *blast*

lemma *vpeqR-reflexive-lemma* : $\forall\ s\ d.\ (vpeqR\ s\ d\ s)$
 using *vpeq1-reflexive-lemma vpeq-part-procs-reflexive-lemma* **by** *auto*

lemma *vpeqR-vpeq1* : $vpeqR\ s\ d\ t \Longrightarrow vpeq1\ (\Uparrow s)\ d\ (\Uparrow t)$
 by *fastforce*

lemma *sched-currentR-lemma* :
 $\forall s\ t\ a.\ vpeqR\ s\ (scheduler\ sysconf)\ t \longrightarrow (domain\text{-}of\text{-}eventR\ s\ a) = (domain\text{-}of\text{-}eventR\ t\ a)$
  using *vpeqR-def vpeq1-def abstract-state-def vpeq-sched-def*
  by (*metis* (*no-types*, *lifting*) *EventR.exhaust State.select-convs*(*1*) *domain-of-eventR.simps*)

lemma *scheproc-hasnexts*: $schedule\text{-}process\ s \neq \{\}$
 apply(*case-tac is-a-partition sysconf* (*current s*) $\wedge$ *part-mode* (*the* ((*partitions s*) (*current s*))) = *NORMAL*)
 by *auto*

lemma *reachable-l2*: $\forall s\ a.\ (SM.reachable0\ s0r\ exec\text{-}eventR)\ s \longrightarrow (\exists s'.\ (s,\ s') \in exec\text{-}eventR\ a)$
 proof –
 {
  fix *s a*
  assume *p0*: (*SM.reachable0 s0r exec-eventR*) *s*
  have $\exists s'.\ (s,\ s') \in exec\text{-}eventR\ a$
   proof(*induct a*)
    case (*hyperc′ x*) **show** *?case*
     proof(*induct x*) **qed**(*auto simp add*:*exec-eventR-def*)
    next
    case (*sys′ x*) **then show** *?case*
     proof(*induct x*)
      case (*Schedule*) **show** *?case* **using** *exec-eventR-def* **by** *fastforce*
      case (*Transfer-Sampling-Message c*) **show** *?case* **using** *exec-eventR-def* **by** *fastforce*
      case (*Transfer-Queuing-Message c*) **show** *?case* **using** *exec-eventR-def* **by** *fastforce*
      case (*Schedule-Process*) **show** *?case* **using** *exec-eventR-def scheproc-hasnexts* **by** *fastforce*
     qed
   qed
 }

then show *?thesis* **by** *auto*
qed

interpretation *SM-enabled*
s0r exec-eventR domain-of-eventR scheduler sysconf vpeqR interference1
using *vpeqR-transitive-lemma vpeqR-symmetric-lemma vpeqR-reflexive-lemma sched-currentR-lemma*
schedeler-intf-all-help no-intf-sched-help reachable-l2 nintf-reflx
SM.intro[*of vpeqR scheduler sysconf domain-of-eventR interference1*]
SM-enabled-axioms.intro [*of s0r exec-eventR*]
SM-enabled.intro[*of domain-of-eventR scheduler sysconf vpeqR interference1 s0r exec-eventR*] **by** *blast*

3.3 Unwinding conditions on new state variables

definition *weak-step-consistent-new* :: *bool* **where**
weak-step-consistent-new ≡ ∀ *a d s t*. *reachable0 s* ∧ *reachable0 t* ∧ (*s* ∼. *d* .∼ *t*) ∧ (*s* ∼. (*scheduler sysconf*) .∼ *t*) ∧
((*the* (*domain-of-eventR s a*)) ⇝ *d*) ∧ (*s* ∼. (*the* (*domain-of-eventR s a*)) .∼ *t*) ⟶
(∀ *s′ t′*. (*s*,*s′*) ∈ *exec-eventR a* ∧ (*t*,*t′*) ∈ *exec-eventR a* ⟶ (*s′* ∼.*d*.∼$_{\Delta}$ *t′*))

definition *step-consistent-new* :: *bool* **where**
step-consistent-new ≡ ∀ *a d s t*. *reachable0 s* ∧ *reachable0 t* ∧ (*s* ∼. *d* .∼ *t*) ∧ (*s* ∼. (*scheduler sysconf*) .∼ *t*) ∧
((*the* (*domain-of-eventR s a*)) ⇝ *d*) ⟶ (*s* ∼. (*the* (*domain-of-eventR s a*)) .∼ *t*) ⟶
(∀ *s′ t′*. (*s*,*s′*) ∈ *exec-eventR a* ∧ (*t*,*t′*) ∈ *exec-eventR a* ⟶ (*s′* ∼.*d*.∼$_{\Delta}$ *t′*))

definition *local-respect-new* :: *bool* **where**
local-respect-new ≡ ∀ *a d s s′*. *reachable0 s* ∧ ((*the* (*domain-of-eventR s a*)) \⇝ *d*) ∧ (*s*,*s′*)∈*exec-eventR a*
⟶ (*s* ∼. *d* .∼$_{\Delta}$ *s′*)

definition *weak-step-consistent-new-e* :: *EventR* ⇒ *bool* **where**
weak-step-consistent-new-e a ≡ ∀ *d s t*. *reachable0 s* ∧ *reachable0 t* ∧ (*s* ∼. *d* .∼ *t*) ∧ (*s* ∼. (*scheduler sysconf*) .∼ *t*) ∧
((*the* (*domain-of-eventR s a*)) ⇝ *d*) ∧ (*s* ∼. (*the* (*domain-of-eventR s a*)) .∼ *t*) ⟶
(∀ *s′ t′*. (*s*,*s′*) ∈ *exec-eventR a* ∧ (*t*,*t′*) ∈ *exec-eventR a* ⟶ (*s′* ∼.*d*.∼$_{\Delta}$ *t′*))

definition *step-consistent-new-e* :: *EventR* ⇒ *bool* **where**
step-consistent-new-e a ≡ ∀ *d s t*. *reachable0 s* ∧ *reachable0 t* ∧ (*s* ∼. *d* .∼ *t*) ∧ (*s* ∼. (*scheduler sysconf*) .∼ *t*) ∧
((*the* (*domain-of-eventR s a*)) ⇝ *d*) ⟶ (*s* ∼. (*the* (*domain-of-eventR s a*)) .∼ *t*) ⟶
(∀ *s′ t′*. (*s*,*s′*) ∈ *exec-eventR a* ∧ (*t*,*t′*) ∈ *exec-eventR a* ⟶ (*s′* ∼.*d*.∼$_{\Delta}$ *t′*))

definition *local-respect-new-e* :: *EventR* ⇒ *bool* **where**
local-respect-new-e a ≡ ∀ *d s s′*. *reachable0 s* ∧ ((*the* (*domain-of-eventR s a*)) \⇝ *d*) ∧ (*s*,*s′*)∈*exec-eventR a*
⟶ (*s* ∼. *d* .∼$_{\Delta}$ *s′*)

declare *weak-step-consistent-new-def*[*cong del*] **and** *step-consistent-new-def*[*cong del*] **and** *local-respect-new-def*[*cong del*] **and**
weak-step-consistent-new-e-def[*cong del*] **and** *step-consistent-new-e-def*[*cong del*] **and** *local-respect-new-e-def*[*cong del*]

declare *weak-step-consistent-new-def*[*cong*] **and** *step-consistent-new-def*[*cong*] **and** *local-respect-new-def*[*cong*] **and** *weak-step-consistent-new-e-def*[*cong*] **and** *step-consistent-new-e-def*[*cong*] **and** *local-respect-new-e-def*[*cong*]

declare *weak-step-consistent-new-def*[*cong del*] **and** *step-consistent-new-def*[*cong del*] **and** *local-respect-new-def*[*cong del*] **and** *weak-step-consistent-new-e-def*[*cong del*] **and** *step-consistent-new-e-def*[*cong del*] **and** *local-respect-new-e-def*[*cong del*]

lemma *weak-step-consistent-new-all-evt* : *weak-step-consistent-new* = ($\forall$ *a*. *weak-step-consistent-new-e a*)
by (*simp add*:*weak-step-consistent-new-def weak-step-consistent-new-e-def*)

lemma *step-consistent-new-all-evt* : *step-consistent-new* = ($\forall$ *a*. *step-consistent-new-e a*)
by (*simp add*:*step-consistent-new-def step-consistent-new-e-def*)

lemma *local-respect-new-all-evt* : *local-respect-new* = ($\forall$ *a*. *local-respect-new-e a*)
by (*simp add*:*local-respect-new-def local-respect-new-e-def*)

3.4 Proofs of refinement

3.4.1 Refinement of existing events at upper level

lemma *create-sampling-port-ref-lemma*:
$\forall$ *s*. *fst* (*create-sampling-port sc* ($\Uparrow$*s*) *p*) = $\Uparrow$(*fst* (*create-sampling-portR sc s p*))
by *auto*

lemma *write-sampling-message-ref-lemma*:
$\forall$ *s*. *fst* (*write-sampling-message* ($\Uparrow$*s*) *p m*) = $\Uparrow$(*fst* (*write-sampling-messageR s p m*))
by *simp*

lemma *read-sampling-message-ref-lemma*:
$\forall$ *s*. *fst* (*read-sampling-message* ($\Uparrow$*s*) *p*) = $\Uparrow$(*fst* (*read-sampling-messageR s p*))
by *simp*

lemma *get-sampling-port-id-ref-lemma*:
$\forall$ *s*. *fst* (*get-sampling-port-id sc* ($\Uparrow$*s*) *p*) = $\Uparrow$(*fst* (*get-sampling-port-idR sc s p*))
by *simp*

lemma *get-sampling-port-status-ref-lemma*:
$\forall$ *s*. *fst* (*get-sampling-port-status sc* ($\Uparrow$*s*) *p*) = $\Uparrow$(*fst* (*get-sampling-port-statusR sc s p*))
by *simp*

lemma *create-queuing-port-ref-lemma*:
$\forall$ *s*. *fst* (*create-queuing-port sc* ($\Uparrow$*s*) *p*) = $\Uparrow$(*fst* (*create-queuing-portR sc s p*))
by *auto*

lemma *send-queuing-message-maylost-ref-lemma*:
$\forall s.\ \mathit{fst}\ (\mathit{send\text{-}queuing\text{-}message\text{-}maylost}\ sc\ (\Uparrow s)\ p\ m) = \Uparrow(\mathit{fst}\ (\mathit{send\text{-}queuing\text{-}message\text{-}maylostR}\ sc\ s\ p\ m))$
by *simp*

lemma *receive-queuing-message-ref-lemma*:
$\forall s.\ \mathit{fst}\ (\mathit{receive\text{-}queuing\text{-}message}\ (\Uparrow s)\ p) = \Uparrow(\mathit{fst}\ (\mathit{receive\text{-}queuing\text{-}messageR}\ s\ p))$
by *auto*

lemma *get-queuing-port-id-ref-lemma*:
$\forall s.\ \mathit{fst}\ (\mathit{get\text{-}queuing\text{-}port\text{-}id}\ sc\ (\Uparrow s)\ p) = \Uparrow(\mathit{fst}\ (\mathit{get\text{-}queuing\text{-}port\text{-}idR}\ sc\ s\ p))$
by *auto*

lemma *get-queuing-port-status-ref-lemma*:
$\forall s.\ \mathit{fst}\ (\mathit{get\text{-}queuing\text{-}port\text{-}status}\ sc\ (\Uparrow s)\ p) = \Uparrow(\mathit{fst}\ (\mathit{get\text{-}queuing\text{-}port\text{-}statusR}\ sc\ s\ p))$
by *auto*

lemma *clear-queuing-port-ref-lemma*:
$\forall s.\ \mathit{clear\text{-}queuing\text{-}port}\ (\Uparrow s)\ p = \Uparrow(\mathit{clear\text{-}queuing\text{-}portR}\ s\ p)$
by *auto*

lemma *schedule-ref-lemma*: $\forall s\ s'.\ (s' \in \mathit{scheduleR}\ sc\ s) \longrightarrow (\Uparrow s') \in (\mathit{schedule}\ sc\ (\Uparrow s))$
by *auto*

lemma *get-partition-status-ref-lemma*:
$\forall s.\ \mathit{fst}\ (\mathit{get\text{-}partition\text{-}status}\ sc\ (\Uparrow s)) = \Uparrow(\mathit{fst}\ (\mathit{get\text{-}partition\text{-}statusR}\ sc\ s))$
by *simp*

lemma *set-partition-mode-ref-lemma*: $\forall s.\ \mathit{set\text{-}partition\text{-}mode}\ sc\ (\Uparrow s)\ m = \Uparrow(\mathit{set\text{-}partition\text{-}modeR}\ sc\ s\ m)$
proof −
{
fix *s*
let $?s' = \mathit{set\text{-}partition\text{-}modeR}\ sc\ s\ m$
let $?us' = \Uparrow(?s')$
let $?t = \Uparrow s$
let $?t' = \mathit{set\text{-}partition\text{-}mode}\ sc\ ?t\ m$
have *a0*: $\mathit{current}\ ?t' = \mathit{current}\ ?us'$
using *set-partition-mode-def*
by *auto*
moreover
have $\mathit{partitions}\ ?t' = \mathit{partitions}\ ?us'$
proof −
have *b0*: $\mathit{partitions}\ s = \mathit{partitions}\ ?t \land \mathit{current}\ s = \mathit{current}\ ?t$
by *simp*

{
fix *p*
have *partitions ?t′ p = partitions ?us′ p*
proof(*cases* (*partconf sc*) (*current s*) ≠ *None* ∧ (*partitions s*) (*current s*) ≠ *None* ∧
¬ (*part-mode* (*the* ((*partitions s*) (*current s*))) = *COLD-START* ∧ *m* = *WARM-START*))
assume *c0*: (*partconf sc*) (*current s*) ≠ *None* ∧ (*partitions s*) (*current s*) ≠ *None* ∧
¬ (*part-mode* (*the* ((*partitions s*) (*current s*))) = *COLD-START* ∧ *m* = *WARM-START*)
with *b0* **have** *c1*: (*partconf sc*) (*current ?t*) ≠ *None* ∧ (*partitions ?t*) (*current ?t*) ≠ *None* ∧
¬ (*part-mode* (*the* ((*partitions ?t*) (*current ?t*))) = *COLD-START* ∧ *m* = *WARM-START*)
by *simp*
show *?thesis*
proof(*cases current ?t = p*)
assume *d0*: *current ?t = p*
with *c1* **have** *partitions ?t′ p* = *Some* ((*the* (*partitions ?t p*)) (|*part-mode* := *m*|))
by *auto*
moreover
from *b0 c0 d0* **have** *partitions ?s′ p* = *Some* ((*the* (*partitions s p*)) (|*part-mode* := *m*|))
by *auto*
ultimately show *?thesis* **by** (*auto cong del*: *set-partition-mode-def*)
next
assume *d0*: *current ?t ≠ p*
with *c1* **have** *partitions ?t′ p = partitions ?t p*
by *auto*
moreover
from *b0 c0 d0* **have** *partitions ?s′ p = partitions s p*
by *auto*
ultimately show *?thesis* **by** *auto*
qed
next
assume *c0*: ¬ ((*partconf sc*) (*current s*) ≠ *None* ∧ (*partitions s*) (*current s*) ≠ *None* ∧
¬ (*part-mode* (*the* ((*partitions s*) (*current s*))) = *COLD-START* ∧ *m* = *WARM-START*))
thus *?thesis* **by** *auto*
qed
}
then show *?thesis* **by** *auto*
qed
thus *?thesis* **by** *auto*
} **qed**

lemma *transf-sampling-msg-ref-lemma*: ∀ *s*. *transf-sampling-msg* (⇑*s*) *c* = ⇑(*transf-sampling-msgR s c*)
by *auto*

lemma *transf-queuing-msg-maylost-ref-lemma*:
$\forall s.$ *transf-queuing-msg-maylost sc* ($\Uparrow s$) $c = \Uparrow$(*transf-queuing-msg-maylostR sc s c*)
by *auto*

3.4.2 new events introduced at this level dont change abstract state

lemma *setrun2ready-nchastate-lemma*:
$s' =$ *setRun2Ready s* $\Longrightarrow (\Uparrow s) = (\Uparrow s')$
by *auto*

lemma *schedule-process-nchastate-lemma*:
$\forall s' \in$(*schedule-process s*). $(\Uparrow s) = (\Uparrow s')$
proof −
{
fix s'
assume *p0*: $s' \in$(*schedule-process s*)

let $?s' =$ *setRun2Ready s*
let *?readyprs* = {*p. p*∈*the* (*procs ?s'* (*current ?s'*)) ∧
state (*the* (*proc-state ?s'* (*current ?s'*,*p*))) = *READY*}
have $(\Uparrow s) = (\Uparrow s')$
proof(*cases is-a-partition sysconf* (*current s*) ∧ *part-mode* (*the* ((*partitions s*) (*current s*))) = *NORMAL*)
assume *a0*: *is-a-partition sysconf* (*current s*) ∧ *part-mode* (*the* ((*partitions s*) (*current s*))) = *NORMAL*
let $?s' =$ *setRun2Ready s*
have *b2*: $(\Uparrow s) = (\Uparrow ?s')$ **using** *setrun2ready-nchastate-lemma* **by** *blast*
have *b3*: $(\Uparrow ?s') = (\Uparrow s')$ **using** *a0 p0* **by** *auto*
with *b2* **show** *?thesis* **by** (*auto cong del*: *abstract-state-def setRun2Ready-def*)
next
assume *a0*: ¬ (*is-a-partition sysconf* (*current s*) ∧ *part-mode* (*the* ((*partitions s*) (*current s*))) = *NORMAL*)
then show *?thesis* **using** *p0* **by** *auto*
qed
}
then show *?thesis* **by** (*auto cong del*: *abstract-state-def*)
qed

lemma *create-process-nchastate-lemma*:
$\forall s.$ $(\Uparrow s) = \Uparrow$(*fst* (*create-process s pri*))
by *auto*

lemma *set-process-priority-nchastate-lemma*:
$\forall s.$ $(\Uparrow s) = \Uparrow$(*set-process-priority s p pri*)
by *auto*

lemma *start-process-nchastate-lemma*:
$\forall s.\ (\Uparrow s) = \Uparrow(\textit{start-process}\ s\ p)$
by *auto*

lemma *stop-process-nchastate-lemma*:
$\forall s.\ (\Uparrow s) = \Uparrow(\textit{stop-process}\ s\ p)$
by *auto*

lemma *suspend-process-nchastate-lemma*:
$\forall s.\ (\Uparrow s) = \Uparrow(\textit{suspend-process}\ s\ p)$
by *auto*

lemma *resume-process-nchastate-lemma*:
$\forall s.\ (\Uparrow s) = \Uparrow(\textit{resume-process}\ s\ p)$
by *auto*

lemma *get-process-status-nchastate-lemma*:
$\forall s.\ (\Uparrow s) = \Uparrow(\textit{fst}\ (\textit{get-process-status}\ s\ p))$
by *auto*

3.4.3 proof of refinement

lemma *s0-ref-lemma* : $(\Uparrow s0r) = s0t$
by (*simp add*: *s0t-init s0r-init system-initR-def*)

lemma *refine-exec-event* : $(s,t) \in \textit{exec-eventR}\ e \Longrightarrow (\textit{eR}\ e = \textit{None} \longrightarrow (\Uparrow s) = (\Uparrow t))$
$\wedge\ (\textit{eR}\ e \neq \textit{None} \longrightarrow (\Uparrow s, \Uparrow t) \in \textit{exec-event}\ (\textit{the}\ (\textit{eR}\ e)))$
proof −
assume *p0*: $(s,t) \in \textit{exec-eventR}\ e$
then show $(\textit{eR}\ e = \textit{None} \longrightarrow (\Uparrow s) = (\Uparrow t)) \wedge (\textit{eR}\ e \neq \textit{None} \longrightarrow (\Uparrow s, \Uparrow t) \in \textit{exec-event}\ (\textit{the}\ (\textit{eR}\ e)))$
proof(*induct e*)
case (*hyperc′ x*) **then show** *?case*
proof(*induct x*)
case (*Create-Sampling-Port y*)
let *?e* = *Hypercall.Create-Sampling-Port y*
let *?er* = *Create-Sampling-Port y*
have *event-enabledR s* (*hyperc′ ?er*) = *event-enabled* $(\Uparrow s)$ (*hyperc ?e*)
by *auto*
then have $((\Uparrow s),(\Uparrow t)) \in$ *exec-event* (*hyperc ?e*)
using *create-sampling-port-ref-lemma exec-eventR-def exec-event-def*
Create-Sampling-Port.prems
by (*auto cong del*: *abstract-state-def*)
then show *?case* **using** *eR-def* **by** *auto*

```
next
  case (Write-Sampling-Message x1 y)
    let ?e = Hypercall.Write-Sampling-Message x1 y
    let ?er = Write-Sampling-Message x1 y
    have event-enabledR s (hyperc′ ?er) = event-enabled (⇑s) (hyperc ?e)
      using event-enabled-def abstract-state-def by auto
    then have ((⇑s),(⇑t)) ∈ exec-event (hyperc ?e)
        using write-sampling-message-ref-lemma exec-eventR-def exec-event-def
          Write-Sampling-Message.prems
         by (auto cong del: abstract-state-def)
    then show ?case using eR-def by auto
next
  case (Read-Sampling-Message y)
    let ?e = Hypercall.Read-Sampling-Message y
    let ?er = Read-Sampling-Message y
    have event-enabledR s (hyperc′ ?er) = event-enabled (⇑s) (hyperc ?e)
      using event-enabled-def abstract-state-def by auto
    then have ((⇑s),(⇑t)) ∈ exec-event (hyperc ?e)
        using read-sampling-message-ref-lemma exec-eventR-def exec-event-def
          Read-Sampling-Message.prems by (auto cong del: abstract-state-def)
    then show ?case using eR-def by auto
next
  case (Get-Sampling-Portid y)
    let ?e = Hypercall.Get-Sampling-Portid y
    let ?er = Get-Sampling-Portid y
    have event-enabledR s (hyperc′ ?er) = event-enabled (⇑s) (hyperc ?e)
      using event-enabled-def abstract-state-def by auto
    then have ((⇑s),(⇑t)) ∈ exec-event (hyperc ?e)
        using get-sampling-port-id-ref-lemma exec-eventR-def exec-event-def
          Get-Sampling-Portid by (auto cong del: abstract-state-def)
    then show ?case using eR-def by auto
next
  case (Get-Sampling-Portstatus y)
    let ?e = Hypercall.Get-Sampling-Portstatus y
    let ?er = Get-Sampling-Portstatus y
    have event-enabledR s (hyperc′ ?er) = event-enabled (⇑s) (hyperc ?e)
      using event-enabled-def abstract-state-def by auto
    then have ((⇑s),(⇑t)) ∈ exec-event (hyperc ?e)
        using get-sampling-port-status-ref-lemma exec-eventR-def exec-event-def
          Get-Sampling-Portstatus.prems by (auto cong del: abstract-state-def)
    then show ?case using eR-def by auto
next
  case (Create-Queuing-Port y)
```

let *?e* = *Hypercall.Create-Queuing-Port y*
let *?er* = *Create-Queuing-Port y*
have *event-enabledR s* (*hyperc′ ?er*) = *event-enabled* ($\Uparrow s$) (*hyperc ?e*)
using *event-enabled-def abstract-state-def* **by** *auto*
then have $((\Uparrow s),(\Uparrow t)) \in$ *exec-event* (*hyperc ?e*)
using *create-queuing-port-ref-lemma exec-eventR-def exec-event-def*
Create-Queuing-Port.prems **by** (*auto cong del*: *abstract-state-def*)
then show *?case* **using** *eR-def* **by** *auto*
next
case (*Send-Queuing-Message x1 y1*)
let *?e* = *Hypercall.Send-Queuing-Message x1 y1*
let *?er* = *Send-Queuing-Message x1 y1*
have *event-enabledR s* (*hyperc′ ?er*) = *event-enabled* ($\Uparrow s$) (*hyperc ?e*)
using *event-enabled-def abstract-state-def* **by** *auto*
then have $((\Uparrow s),(\Uparrow t)) \in$ *exec-event* (*hyperc ?e*)
using *send-queuing-message-maylost-ref-lemma exec-eventR-def exec-event-def*
Send-Queuing-Message.prems **by** (*auto cong del*: *abstract-state-def*)
then show *?case* **using** *eR-def* **by** *auto*
next
case (*Receive-Queuing-Message x1*)
let *?e* = *Hypercall.Receive-Queuing-Message x1*
let *?er* = *Receive-Queuing-Message x1*
have *event-enabledR s* (*hyperc′ ?er*) = *event-enabled* ($\Uparrow s$) (*hyperc ?e*)
using *event-enabled-def abstract-state-def* **by** *auto*
then have $((\Uparrow s),(\Uparrow t)) \in$ *exec-event* (*hyperc ?e*)
using *receive-queuing-message-ref-lemma exec-eventR-def exec-event-def*
Receive-Queuing-Message.prems **by** (*auto cong del*: *abstract-state-def*)
then show *?case* **using** *eR-def* **by** *auto*
next
case (*Get-Queuing-Portid x1*)
let *?e* = *Hypercall.Get-Queuing-Portid x1*
let *?er* = *Get-Queuing-Portid x1*
have *event-enabledR s* (*hyperc′ ?er*) = *event-enabled* ($\Uparrow s$) (*hyperc ?e*)
using *event-enabled-def abstract-state-def* **by** *auto*
then have $((\Uparrow s),(\Uparrow t)) \in$ *exec-event* (*hyperc ?e*)
using *get-queuing-port-id-ref-lemma exec-eventR-def exec-event-def*
Get-Queuing-Portid.prems **by** (*auto cong del*: *abstract-state-def*)
then show *?case* **using** *eR-def* **by** *auto*
next
case (*Get-Queuing-Portstatus x1*)
let *?e* = *Hypercall.Get-Queuing-Portstatus x1*
let *?er* = *Get-Queuing-Portstatus x1*
have *event-enabledR s* (*hyperc′ ?er*) = *event-enabled* ($\Uparrow s$) (*hyperc ?e*)

```
      using event-enabled-def abstract-state-def by auto
    then have ((⇑s),(⇑t)) ∈ exec-event (hyperc ?e)
        using get-queuing-port-status-ref-lemma exec-eventR-def exec-event-def
          Get-Queuing-Portstatus.prems by (auto cong del: abstract-state-def)
    then show ?case using eR-def by auto
next
  case (Clear-Queuing-Port x1)
    let ?e = Hypercall.Clear-Queuing-Port x1
    let ?er = Clear-Queuing-Port x1
    have event-enabledR s (hyperc′ ?er) = event-enabled (⇑s) (hyperc ?e)
      using event-enabled-def abstract-state-def by auto
    then have ((⇑s),(⇑t)) ∈ exec-event (hyperc ?e)
        using clear-queuing-port-ref-lemma exec-eventR-def exec-event-def
          Clear-Queuing-Port.prems by (auto cong del: abstract-state-def)
    then show ?case using eR-def by auto
next
  case (Set-Partition-Mode x1)
    let ?e = Hypercall.Set-Partition-Mode x1
    let ?er = Set-Partition-Mode x1
    have event-enabledR s (hyperc′ ?er) = event-enabled (⇑s) (hyperc ?e)
      using event-enabled-def abstract-state-def by auto
    then have ((⇑s),(⇑t)) ∈ exec-event (hyperc ?e)
        using set-partition-mode-ref-lemma exec-eventR-def exec-event-def
          Set-Partition-Mode.prems
          by (auto cong del: set-partition-modeR-def
                  abstract-state-def event-enabledR-def
                   event-enabled-def set-partition-mode-def)
    then show ?case using eR-def by auto
next
  case (Get-Partition-Status)
    let ?e = Hypercall.Get-Partition-Status
    let ?er = Get-Partition-Status
    have event-enabledR s (hyperc′ ?er) = event-enabled (⇑s) (hyperc ?e)
      using event-enabled-def abstract-state-def by auto
    then have ((⇑s),(⇑t)) ∈ exec-event (hyperc ?e)
        using get-partition-status-ref-lemma exec-eventR-def exec-event-def
          Get-Partition-Status.prems by (auto cong del: abstract-state-def)
    then show ?case using eR-def by auto
next
  case (Create-Process x1)
    let ?er = Create-Process x1
    show ?case using eR-def exec-eventR-def Create-Process.prems create-process-nchastate-lemma
      by (metis (no-types, lifting) EventR.simps(5) Hypercall′.simps(413))
```

mem-Collect-eq old.prod.case singletonD)
next
case (*Start-Process x1*)
let *?er* = *Start-Process x1*
show *?case* **using** *eR-def exec-eventR-def Start-Process.prems start-process-nchastate-lemma*
by (*metis* (*no-types*, *lifting*) *EventR.simps*(*5*) *Hypercall'.simps*(*414*)
mem-Collect-eq old.prod.case singletonD)
next
case (*Stop-Process x1*)
let *?er* = *Stop-Process x1*
show *?case* **using** *eR-def exec-eventR-def Stop-Process.prems stop-process-nchastate-lemma*
by (*metis* (*no-types*, *lifting*) *EventR.simps*(*5*) *Hypercall'.simps*(*415*)
mem-Collect-eq old.prod.case singletonD)
next
case (*Resume-Process x1*)
let *?er* = *Resume-Process x1*
show *?case* **using** *eR-def exec-eventR-def Resume-Process.prems resume-process-nchastate-lemma*
by (*metis* (*no-types*, *lifting*) *EventR.simps*(*5*) *Hypercall'.simps*(*416*)
mem-Collect-eq old.prod.case singletonD)
next
case (*Suspend-Process x1*)
let *?er* = *Suspend-Process x1*
show *?case* **using** *eR-def exec-eventR-def Suspend-Process.prems suspend-process-nchastate-lemma*
by (*metis* (*no-types*, *lifting*) *EventR.simps*(*5*) *Hypercall'.simps*(*417*)
mem-Collect-eq old.prod.case singletonD)
next
case (*Set-Priority x1 y1*)
let *?er* = *Set-Priority x1 y1*
show *?case* **using** *eR-def exec-eventR-def Set-Priority.prems set-process-priority-nchastate-lemma*
by (*metis* (*no-types*, *lifting*) *EventR.simps*(*5*) *Hypercall'.simps*(*418*)
mem-Collect-eq old.prod.case singletonD)
next
case (*Get-Process-Status x1*)
let *?er* = *Get-Process-Status x1*
show *?case* **using** *eR-def exec-eventR-def Get-Process-Status.prems get-process-status-nchastate-lemma*
by (*metis* (*no-types*, *lifting*) *EventR.simps*(*5*) *Hypercall'.simps*(*419*)
mem-Collect-eq old.prod.case singletonD)
qed
next
case (*sys' x*) **then show** *?case*
proof(*induct x*)
case (*Schedule*)
let *?e* = *System-Event.Schedule*

```
        let ?er = Schedule
        have event-enabledR s (sys' ?er) = event-enabled (⇑s) (sys ?e)
          by auto
        then have ((⇑s),(⇑t)) ∈ exec-event (sys ?e)
            using schedule-ref-lemma exec-eventR-def exec-event-def
              Schedule.prems by (auto cong del: scheduleR-def)
        then show ?case using eR-def by auto
    next
      case (Transfer-Sampling-Message x1)
        let ?e = System-Event.Transfer-Sampling-Message x1
        let ?er = Transfer-Sampling-Message x1
        have event-enabledR s (sys' ?er) = event-enabled (⇑s) (sys ?e)
           by auto
        then have ((⇑s),(⇑t)) ∈ exec-event (sys ?e)
            using transf-sampling-msg-ref-lemma exec-eventR-def exec-event-def
              Transfer-Sampling-Message.prems by auto
        then show ?case using eR-def by auto
    next
      case (Transfer-Queuing-Message x1)
        let ?e = System-Event.Transfer-Queuing-Message x1
        let ?er = Transfer-Queuing-Message x1
        have event-enabledR s (sys' ?er) = event-enabled (⇑s) (sys ?e)
           by auto
        then have ((⇑s),(⇑t)) ∈ exec-event (sys ?e)
            using transf-queuing-msg-maylost-ref-lemma exec-eventR-def exec-event-def
              Transfer-Queuing-Message.prems  by auto
        then show ?case using eR-def by auto
    next
      case (Schedule-Process)
        let ?er = Schedule-Process
        show ?case using eR-def exec-eventR-def Schedule-Process.prems schedule-process-nchastate-lemma
          by (metis (no-types, lifting) EventR.simps(6) System-EventR.simps(18)
             prod.simps(2) mem-Collect-eq singletonD)
   qed
  qed
qed

lemma reachR-reach1:
  ∀ s as s'. SK-TopSpec.reachable0 (⇑s) ∧
              reachable0 s ∧ s'∈execute as s  ⟶
             SK-TopSpec.reachable0 (⇑s')
proof –
{
```

```
    fix as
    have ∀ s s'. SK-TopSpec.reachable0 (⇑s) ∧ reachable0 s ∧ s'∈execute as s
                  ⟶ SK-TopSpec.reachable0 (⇑s')
    proof(induct as)
      case Nil show ?case using execute-def by fastforce
    next
      case (Cons b bs)
      assume a0: ∀ s s'. SK-TopSpec.reachable0 (⇑s) ∧ reachable0 s ∧ s'∈execute bs s
                  ⟶ SK-TopSpec.reachable0 (⇑s')
      show ?case
      proof -
      {
        fix s s'
        assume b0: SK-TopSpec.reachable0 (⇑s) ∧ reachable0 s ∧ s'∈execute (b # bs) s
        have b2: current s = current (⇑s) ∧ partitions s = partitions (⇑s) by (simp add:abstract-state-def)
        have b3: ∀ s1. (s,s1)∈exec-eventR b ⟶ SK-TopSpec.reachable0 (⇑s1)
        proof -
        {
          fix s1
          assume c0: (s,s1)∈exec-eventR b
          then have SK-TopSpec.reachable0 (⇑s1)
            using SK-TopSpec.reachableStep b0 refine-exec-event by metis
        }
        then show ?thesis by auto
        qed
        from b0 have ∃ s1. (s,s1)∈exec-eventR b ∧ (s1,s')∈run bs using execute-def
          by (simp add: relcomp.simps)
        then obtain s1 where b4: (s,s1)∈exec-eventR b ∧ (s1,s')∈run bs by auto
        with b3 have b5: SK-TopSpec.reachable0 (⇑s1) by simp
        have b6: SK-L2Spec.reachable0 s1 using SK-L2Spec.reachableStep b0 b4 by blast
        with b4 b5 a0 have SK-TopSpec.reachable0 (⇑s') using execute-def by auto
      } then show ?thesis by auto
      qed
    qed
  } then show ?thesis by auto
  qed


lemma reachR-reach: reachable0 s ⟹ SK-TopSpec.reachable0 (⇑s)
  using reachR-reach1 SK-L2Spec.reachable0-def reachable-s0 SK-TopSpec.reachable-s0 s0-ref-lemma
    by (metis Image-singleton-iff execute-def reachable-def)

primrec rmtau :: 'a option list => 'a list
```

```
  where rmtau [] = [] |
      rmtau (a#as) = (if a ≠ None then
                          the a # rmtau as
                         else rmtau as)

lemma refine-sound-helper: ∀ es st sr. st = ⇑sr ⟶
        (image abstract-state (execute es sr)) ⊆ (SK-TopSpec.execute (rmtau (map eR es)) st)
  proof –
  {
    fix es
    have ∀ st sr. st = ⇑sr ⟶
        (image abstract-state (execute es sr)) ⊆ (SK-TopSpec.execute (rmtau (map eR es)) st)
    proof(induct es)
      case Nil show ?case
      proof –
      {
        fix st sr
        assume a0: st = ⇑sr
        then have abstract-state ‘ SK-L2Spec.execute [] sr = {st}
          using SK-L2Spec.execute-def by auto
        moreover
        from a0 have SK-TopSpec.execute (rmtau (map eR [])) st = {st}
          using SK-TopSpec.execute-def SK-L2Spec.run.run-Nil by simp
        ultimately have abstract-state ‘ SK-L2Spec.execute [] sr ⊆ SK-TopSpec.execute (rmtau (map eR [])) st
          by blast
      } then show ?thesis by auto qed
      next
        case (Cons a as)
        assume a0: ∀ st sr. st = ⇑sr ⟶
              abstract-state ‘ SK-L2Spec.execute as sr ⊆ SK-TopSpec.execute (rmtau (map eR as)) st
        show ?case
        proof –
        {
          fix st sr
          assume b0: st = ⇑sr
          have b1: SK-L2Spec.execute (a # as) sr = Image (exec-eventR a O run as) {sr}
            using SK-L2Spec.execute-def SK-L2Spec.run.run-Cons by simp

          have abstract-state ‘ SK-L2Spec.execute (a # as) sr ⊆ SK-TopSpec.execute (rmtau (map eR (a # as))) st
          proof(cases eR a = None)
            assume c0: eR a = None
            then have c1:rmtau (map eR (a # as)) = rmtau (map eR as)
              using rmtau-def by simp
```

```
    let ?nextsr = SK-L2Spec.next-states sr a
    have c2:Image (exec-eventR a O run as) {sr} = Image (run as) ?nextsr
      using SK-L2Spec.next-states-def by auto
    {
      fix s
      assume d0: s∈abstract-state ‘ SK-L2Spec.execute (a # as) sr
      with b1 c2 have ∃ s'∈?nextsr. s∈abstract-state ‘ Image (run as) {s'}
        by auto
      then obtain s' where d1:s'∈?nextsr ∧ s∈abstract-state ‘ Image (run as) {s'} by auto
      from c0 d1 have d2: st = ⇑s' using refine-exec-event SK-L2Spec.next-states-def
        b0 by auto
      with a0 have abstract-state ‘ SK-L2Spec.execute as s' ⊆ SK-TopSpec.execute (rmtau (map eR as)) st
        by simp
      with c1 d1 have s∈SK-TopSpec.execute (rmtau (map eR (a # as))) st
        using SK-L2Spec.execute-def subsetCE by auto
    }
    then show ?thesis by blast
  next
    assume c0: eR a ≠ None
    then have c1:rmtau (map eR (a # as)) = (the (eR a)) # (rmtau (map eR as))
      using rmtau-def by simp
    let ?nextsr = SK-L2Spec.next-states sr a
    let ?nextst = SK-TopSpec.next-states st (the (eR a))
    have c2:Image (exec-eventR a O run as) {sr} = Image (run as) ?nextsr
      using SK-L2Spec.next-states-def by auto
    have c3:Image (exec-event (the (eR a)) O SK-TopSpec.run (rmtau (map eR as))) {st}
          = Image (SK-TopSpec.run (rmtau (map eR as))) ?nextst
      using SK-TopSpec.next-states-def by auto
    {
      fix s
      assume d0: s∈abstract-state ‘ SK-L2Spec.execute (a # as) sr
      with b1 c2 have ∃ s'∈?nextsr. s∈abstract-state ‘ Image (run as) {s'}
        using Image-singleton-iff SK-L2Spec.next-states-def imageE
          image-eqI mem-Collect-eq relcomp.cases by auto
      then obtain s' where d1:s'∈?nextsr ∧ s∈abstract-state ‘ Image (run as) {s'} by auto
      from c0 d1 have ∃ st'∈?nextst. st' = ⇑s' using refine-exec-event SK-L2Spec.next-states-def
        b0 by (simp add: SK-TopSpec.next-states-def)
      then obtain st' where d2: st'∈?nextst ∧ st' = ⇑s' by auto
      from a0 d1 d2 have abstract-state ‘ SK-L2Spec.execute as s' ⊆ SK-TopSpec.execute (rmtau (map eR as)) st'
        by simp
      with c1 c2 c3 d1 d2 have s∈SK-TopSpec.execute (rmtau (map eR (a # as))) st
        using SK-L2Spec.execute-def ImageI Image-singleton-iff SK-TopSpec.execute-def
          SK-TopSpec.run.run-Cons subsetCE by auto
```

```
      }
      then show ?thesis by fastforce
    qed
  }
  then show ?thesis by blast
  qed
 qed
}
then show ?thesis by auto
qed
```

```
theorem refine-sound: (image abstract-state (execute es s0r)) ⊆ (SK-TopSpec.execute (rmtau (map eR es)) s0t)
  using refine-sound-helper s0-ref-lemma by fastforce
```

3.4.4 unwinding conditions of refinement

```
lemma weak-step-consistent-new-evt-ref:
  ∀ e. eR e = None ∧ weak-step-consistent-new-e e ⟶ SK-L2Spec.weak-step-consistent-e e
    by (metis SK-L2Spec.weak-step-consistent-e-def refine-exec-event vpeqR-def
        weak-step-consistent-new-e-def)

lemma local-respect-new-evt-ref:
  ∀ e. eR e = None ∧ local-respect-new-e e ⟶ SK-L2Spec.local-respect-e e
    using SK-L2Spec.local-respect-e-def SK-TopSpec.non-interference-def
      local-respect-new-e-def non-interference1-def refine-exec-event
      vpeqR-def vpeqR-reflexive-lemma by metis

lemma weak-step-consistent-evt-ref:
  ∀ e. eR e ≠ None ∧ SK-TopSpec.weak-step-consistent-e (the (eR e))
        ∧ weak-step-consistent-new-e e ⟶ SK-L2Spec.weak-step-consistent-e e
  by (smt SK-L2Spec.weak-step-consistent-e-def SK-TopSpec.step-consistent-def
      SK-TopSpec.weak-with-step-cons domain-domainR local-respect reachR-reach
      refine-exec-event vpeqR-def vpeqR-vpeq1 weak-step-consistent weak-step-consistent-new-e-def)

lemma local-respect-evt-ref:
  ∀ e. eR e ≠ None ∧ SK-TopSpec.local-respect-e (the (eR e))
        ∧ local-respect-new-e e ⟶ SK-L2Spec.local-respect-e e
  using SK-L2Spec.local-respect-e-def SK-TopSpec.local-respect-e-def
        SK-TopSpec.non-interference-def domain-domainR local-respect-new-e-def
        non-interference1-def reachR-reach refine-exec-event vpeqR-def by metis

lemma abs-sc-new-imp: ⟦SK-TopSpec.weak-step-consistent; weak-step-consistent-new⟧
      ⟹ SK-L2Spec.weak-step-consistent
  using SK-L2Spec.weak-step-consistent-all-evt SK-TopSpec.weak-step-consistent-all-evt
```

weak-step-consistent-evt-ref weak-step-consistent-new-all-evt
weak-step-consistent-new-evt-ref **by** *blast*

lemma *abs-lr-new-imp*: ⟦*SK-TopSpec.local-respect*; *local-respect-new*⟧ $\Longrightarrow$ *SK-L2Spec.local-respect*
using *SK-L2Spec.local-respect-all-evt SK-TopSpec.local-respect-all-evt*
local-respect-evt-ref local-respect-new-all-evt local-respect-new-evt-ref **by** *blast*

theorem *noninfl-refinement*: ⟦*SK-TopSpec.local-respect*; *SK-TopSpec.weak-step-consistent*;
weak-step-consistent-new; *local-respect-new*⟧ $\Longrightarrow$ *noninfluence*
using *SK-L2Spec.UnwindingTheorem1 SK-L2Spec.noninf-eq-noninf-gen*
abs-lr-new-imp abs-sc-new-imp **by** *metis*

3.5 Existing events preserve "local respect" on new state variables

3.5.1 proving "create sampling port"

lemma *crt-smpl-portR-presrv-lcrsp-new*:
assumes *p3*:$s' = fst$ (*create-sampling-portR sysconf s pname*)
shows $s \sim .\ d\ . \sim_{\triangle} s'$ **using** *p3* **by** *fastforce*

lemma *crt-smpl-portR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc′* (*Create-Sampling-Port pn*))
using *crt-smpl-portR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.2 proving "write sampling message"

lemma *write-smpl-msgR-presrv-lcrsp-new*:
assumes *p3*:$s' = fst$ (*write-sampling-messageR s pid m*)
shows $s \sim .\ d\ . \sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *write-smpl-msgR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc′* (*Write-Sampling-Message p m*))
using *write-smpl-msgR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.3 proving "read sampling message"

lemma *read-smpl-msgR-presrv-lcrsp-new*:
assumes *p3*:$s' = fst$ (*read-sampling-messageR s pid*)
shows $s \sim .\ d\ . \sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *read-smpl-msgR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc′* (*Read-Sampling-Message p*))

using *read-smpl-msgR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.4 proving "get sampling portid"

lemma *get-smpl-pidR-presrv-lcrsp-new*:
assumes $p3$:$s' = fst$ (*get-sampling-port-idR sysconf s pname*)
shows $s \sim. d .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *get-smpl-pidR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Get-Sampling-Portid p*))
using *get-smpl-pidR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.5 proving "get sampling port status"

lemma *get-smpl-pstsR-presrv-lcrsp-new*:
assumes $p3$:$s' = fst$ (*get-sampling-port-statusR sysconf s pid*)
shows $s \sim. d .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *get-smpl-pstsR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Get-Sampling-Portstatus p*))
using *get-smpl-pstsR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.6 proving "create queuing port"

lemma *crt-que-portR-presrv-lcrsp-new*:
assumes $p3$:$s' = fst$ (*create-queuing-portR sysconf s pname*)
shows $s \sim. d .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *crt-que-portR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Create-Queuing-Port p*))
using *crt-que-portR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.7 proving "send queuing message(may lost)"

lemma *snd-que-msg-lstR-presrv-lcrsp-new*:
assumes $p3$:$s' = fst$ (*send-queuing-message-maylostR sysconf s pid m*)
shows $s \sim. d .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *snd-que-msg-lstR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Send-Queuing-Message p m*))
using *snd-que-msg-lstR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.8 proving "receive queuing message"

lemma *rec-que-msgR-presrv-lcrsp-new*:
assumes *p3*: $s' = fst$ (*receive-queuing-messageR s pid*)
shows $s \sim .\ d\ .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *rec-que-msgR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Receive-Queuing-Message p*))
using *rec-que-msgR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.9 proving "get queuing portid"

lemma *get-que-pidR-presrv-lcrsp-new*:
assumes *p3*: $s' = fst$ (*get-queuing-port-idR sysconf s pname*)
shows $s \sim .\ d\ .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *get-que-pidR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Get-Queuing-Portid p*))
using *get-que-pidR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.10 proving "get queuing port status"

lemma *get-que-pstsR-presrv-lcrsp-new*:
assumes *p3*: $s' = fst$ (*get-queuing-port-statusR sysconf s pid*)
shows $s \sim .\ d\ .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *get-que-pstsR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Get-Queuing-Portstatus p*))
using *get-que-pstsR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.11 proving "clear queuing port"

lemma *clr-que-portR-presrv-lcrsp-new*:
assumes *p3*: $s' =$ *clear-queuing-portR s pid*
shows $s \sim .\ d\ .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *clr-que-portR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Clear-Queuing-Port p*))
using *clr-que-portR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
by *fastforce*

3.5.12 proving "get partition statue"

lemma *get-part-statusR-presrv-lcrsp-new*:
assumes *p3*:$s' = $ *fst* (*get-partition-statusR sysconf s*)
shows $s \sim . \ d \ .\sim_{\triangle} s'$
using *p3* **by** *fastforce*

lemma *get-part-statusR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc' Get-Partition-Status*)
using *get-part-statusR-presrv-lcrsp-new local-respect-new-e-def exec-eventR-def*
vpeq-part-procs-reflexive-lemma **by** (*fastforce cong del*: *vpeq-part-procs-def*)

3.5.13 proving "set partition mode"

lemma *set-procs-to-normal-presrv-lcrsp-new*:
assumes *p3*: *current* $s \neq d$
and *p4*: $s' = $ *set-procs-to-normal s* (*current s*)
shows $s \sim . \ d \ .\sim_{\triangle} s'$
using *p3 p4* **by** *auto*

lemma *remove-partition-resources-presrv-lcrsp-new*:
assumes *p3*: *current* $s \neq d$
and *p4*: $s' = $ *remove-partition-resources s* (*current s*)
shows $s \sim . \ d \ .\sim_{\triangle} s'$
using *p3 p4* **by** *auto*

lemma *set-part-modeR-presrv-lcrsp-new*:
assumes
p3: *current* $s \neq d$
and *p4*: $s' = $ *set-partition-modeR sysconf s m*
shows $s \sim . \ d \ .\sim_{\triangle} s'$
using *p3 p4* **by** *auto*

lemma *set-part-modeR-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Set-Partition-Mode p*))
using *set-part-modeR-presrv-lcrsp-new local-respect-new-e-def*
exec-eventR-def nintf-neq domain-of-eventR-hc event-enabledR-hc
by (*fastforce cong del*: *set-partition-modeR-def non-interference1-def*)

3.5.14 proving "schedule"

lemma *scheduleR-presrv-lcrsp-new*:
  assumes *p2*:(*scheduler sysconf*) $\setminus\rightsquigarrow$ *d*
  shows $s \sim. \ d \ .\sim_\triangle \ s'$
 using *p2 schedeler-intf-all-help* **by** *auto*

lemma *scheduleR-presrv-lcrsp-new-e*: *local-respect-new-e* (*sys′ Schedule*)
 using *scheduleR-presrv-lcrsp-new local-respect-new-e-def*
  exec-eventR-def nintf-neq domain-of-eventR-hc event-enabledR-hc
 by (*auto cong del*: *non-interference1-def*)

3.5.15 proving "Transfer Sampling Message"

lemma *trans-smpl-msgR-presrv-lcrsp-new*:
  assumes *p3*:$s' =$ *transf-sampling-msgR s c*
  shows $s \sim. \ d \ .\sim_\triangle \ s'$
 using *p3* **by** *fastforce*

lemma *trans-smpl-msgR-presrv-lcrsp-new-e*: *local-respect-new-e* (*sys′* (*Transfer-Sampling-Message c*))
 using *trans-smpl-msgR-presrv-lcrsp-new local-respect-new-e-def*
  exec-eventR-def nintf-neq
  domain-of-eventR-hc event-enabledR-hc
 by (*fastforce cong del*: *non-interference1-def*)

3.5.16 proving "Transfer Queuing Message"

lemma *trans-que-msg-mlostR-presrv-lcrsp-new*:
  assumes *p3*:$s' =$ *transf-queuing-msg-maylostR sysconf s c*
  shows $s \sim. \ d \ .\sim_\triangle \ s'$
 using *p3* **by** *fastforce*

lemma *trans-que-msg-mlostR-presrv-lcrsp-new-e*: *local-respect-new-e* (*sys′* (*Transfer-Queuing-Message c*))
 using *trans-que-msg-mlostR-presrv-lcrsp-new local-respect-new-e-def*
  exec-eventR-def nintf-neq domain-of-eventR-hc event-enabledR-hc
 by (*fastforce cong del*: *non-interference1-def*)

3.6 New events preserve "local respect" on new state variables

lemma *create-process-vpeq-part-procs*:
 assumes
   p3: *current* $s \neq d$
  and *p4*: $s' =$ *fst* (*create-process s pri*)
 shows $s \sim. \ d \ .\sim_\triangle \ s'$

using *p3 p4* **by** *auto*

lemma *create-process-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc′* (*Create-Process p*))
using *create-process-vpeq-part-procs local-respect-new-e-def*
exec-eventR-def nintf-neq
domain-of-eventR-hc event-enabledR-hc
by (*auto cong del*: *create-process-def non-interference1-def*)

lemma *set-process-priority-vpeq-part-procs*:
assumes
p3: *current* $s \neq d$
and *p4*: $s' =$ *set-process-priority s p pri*
shows $s \sim .\ d\ .\!\sim_\Delta s'$
using *p3 p4* **by** *auto*

lemma *set-process-priority-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc′* (*Set-Priority p pri*))
using *set-process-priority-vpeq-part-procs local-respect-new-e-def*
exec-eventR-def nintf-neq vpeq-part-procs-def
by (*auto cong del*: *non-interference1-def set-process-priority-def*)

lemma *start-process-vpeq-part-procs*:
assumes *p3*: *current* $s \neq d$
and *p4*: $s' =$ *start-process s p*
shows $s \sim .\ d\ .\!\sim_\Delta s'$
using *p3 p4* **by** *auto*

lemma *start-process-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc′* (*Start-Process p*))
using *start-process-vpeq-part-procs local-respect-new-e-def*
exec-eventR-def nintf-neq
by (*auto cong del*: *non-interference1-def start-process-def*)

lemma *stop-process-vpeq-part-procs*:
assumes
p3: *current* $s \neq d$
and *p4*: $s' =$ *stop-process s p*
shows $s \sim .\ d\ .\!\sim_\Delta s'$
using *p3 p4* **by** *auto*

lemma *stop-process-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc′* (*Stop-Process p*))
using *stop-process-vpeq-part-procs local-respect-new-e-def*
exec-eventR-def nintf-neq
by (*auto cong del*: *stop-process-def non-interference1-def*)

lemma *suspend-process-vpeq-part-procs*:
assumes
p3: *current* $s \neq d$
and *p4*: $s' =$ *suspend-process s p*
shows $s \sim .\ d\ .\sim_{\Delta}\ s'$
using *p3 p4* **by** *auto*

lemma *suspend-process-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Suspend-Process p*))
using *suspend-process-vpeq-part-procs local-respect-new-e-def*
exec-eventR-def nintf-neq
by (*auto cong del*: *suspend-process-def non-interference1-def*)

lemma *resume-process-vpeq-part-procs*:
assumes *p3*: *current* $s \neq d$
and *p4*: $s' =$ *resume-process s p*
shows $s \sim .\ d\ .\sim_{\Delta}\ s'$
using *p3 p4* **by** *auto*

lemma *resume-process-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Resume-Process p*))
using *resume-process-vpeq-part-procs local-respect-new-e-def*
exec-eventR-def nintf-neq vpeq-part-procs-def
by (*auto cong del*: *resume-process-def non-interference1-def*)

lemma *get-process-status-vpeq-part-procs*:
assumes
p3: *current* $s \neq d$
and *p4*: $s' =$ *fst* (*get-process-status s p*)
shows $s \sim .\ d\ .\sim_{\Delta}\ s'$
using *p3 p4* **by** *auto*

lemma *get-process-status-presrv-lcrsp-new-e*: *local-respect-new-e* (*hyperc'* (*Get-Process-Status p*))
using *get-process-status-vpeq-part-procs local-respect-new-e-def*
exec-eventR-def nintf-neq
by (*auto cong del*: *get-process-status-def non-interference1-def*)

lemma *schedule-process-vpeq-part-procs*:
assumes *p3*: *current* $s \neq d$
and *p4*: $s' \in$ *schedule-process s*
shows $s \sim .\ d\ .\sim_{\Delta}\ s'$
proof −

```
      let ?s' = setRun2Ready s
      let ?readyprs = {p. p∈the (procs ?s' (current ?s')) ∧
                          state (the (proc-state ?s' (current ?s',p))) = READY}
      show ?thesis
        proof(cases is-a-partition sysconf (current s) ∧ part-mode (the ((partitions s) (current s))) = NORMAL)
          assume a0: is-a-partition sysconf (current s) ∧ part-mode (the ((partitions s) (current s))) = NORMAL
          let ?s' = setRun2Ready s
          let ?readyprs = {p. p∈the (procs ?s' (current ?s')) ∧
                              state (the (proc-state ?s' (current ?s',p))) = READY}
          let ?selp = SOME p. p∈{x. state (the (proc-state ?s' (current ?s',x))) = READY ∧
                                    (∀ y∈?readyprs. priority (the (proc-state ?s' (current ?s',x))) ≥
                                         priority (the (proc-state ?s' (current ?s',y))))}
          let ?st = the ((proc-state ?s') (current ?s', ?selp))
          let ?proc-st = proc-state ?s'
          let ?cur-pr = cur-proc-part ?s'
          from a0 have a1: schedule-process s = {?s'(|proc-state := ?proc-st ((current ?s', ?selp) := Some (?st(|state := RUNNING|))),
                                cur-proc-part := ?cur-pr(current ?s' := Some ?selp)|)}
            by auto
          then have b2: vpeq-part-procs s d ?s' using p3  by auto
          have b4: current s = current ?s' by auto
          then have b3: vpeq-part-procs ?s' d s'
            using p3 a1 p4 by(auto cong del: schedule-process-def setRun2Ready-def)
          with b2 show ?thesis using vpeq-part-procs-transitive-lemma by blast
      next
        assume a0: ¬ (is-a-partition sysconf (current s) ∧ part-mode (the ((partitions s) (current s))) = NORMAL)
        then show ?thesis using p4 by auto
      qed
    qed

lemma schedule-process-presrv-lcrsp-new-e: local-respect-new-e (sys' Schedule-Process)
  using schedule-process-vpeq-part-procs local-respect-new-e-def
    exec-eventR-def nintf-neq domain-of-eventR-hc
   by (auto cong del: non-interference1-def schedule-process-def)
```

3.7 Proving the "local respect" property on new variables

```
  theorem local-respect-new:local-respect-new
  proof –
    {
      fix e
      have local-respect-new-e e
        apply(induct e)
        using crt-smpl-portR-presrv-lcrsp-new-e write-smpl-msgR-presrv-lcrsp-new-e
              read-smpl-msgR-presrv-lcrsp-new-e get-smpl-pidR-presrv-lcrsp-new-e
```

get-smpl-pstsR-presrv-lcrsp-new-e crt-que-portR-presrv-lcrsp-new-e
snd-que-msg-lstR-presrv-lcrsp-new-e rec-que-msgR-presrv-lcrsp-new-e
get-que-pidR-presrv-lcrsp-new-e get-que-pstsR-presrv-lcrsp-new-e
clr-que-portR-presrv-lcrsp-new-e set-part-modeR-presrv-lcrsp-new-e
get-part-statusR-presrv-lcrsp-new-e create-process-presrv-lcrsp-new-e
start-process-presrv-lcrsp-new-e stop-process-presrv-lcrsp-new-e
resume-process-presrv-lcrsp-new-e suspend-process-presrv-lcrsp-new-e
set-process-priority-presrv-lcrsp-new-e get-process-status-presrv-lcrsp-new-e
apply(*rule Hypercall*′*.induct*)
using *scheduleR-presrv-lcrsp-new-e trans-smpl-msgR-presrv-lcrsp-new-e*
trans-que-msg-mlostR-presrv-lcrsp-new-e schedule-process-presrv-lcrsp-new-e
by(*rule System-EventR.induct*)
}
then show *?thesis* **using** *local-respect-new-all-evt* **by** *simp*
qed

3.8 Existing events preserve "step consistent" on new state variables

lemma *crt-smpl-portR-presrv-wk-stp-cons-new*:
assumes
$p1$:$s \sim .\ d\ .\sim t$
and $p2$:$s' = fst$ (*create-sampling-portR sysconf s pname*)
and $p3$:$t' = fst$ (*create-sampling-portR sysconf t pname*)
shows $s' \sim .\ d\ .\sim_\Delta t'$
using *p1 p2 p3* **by** *fastforce*

lemma *crt-smpl-portR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc*′ (*Create-Sampling-Port pn*))
using *crt-smpl-portR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
non-interference1-def non-interference-def singletonD
by (*smt EventR.case*(*1*) *Hypercall*′*.case*(*1*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *wrt-smpl-msgR-presrv-wk-stp-cons-new*:
assumes $p1$:$s \sim .\ d\ .\sim t$
and $p2$:$s' = fst$ (*write-sampling-messageR s pid m*)
and $p3$:$t' = fst$ (*write-sampling-messageR t pid m*)
shows $s' \sim .\ d\ .\sim_\Delta t'$
using *p1 p2 p3* **by** *fastforce*

lemma *wrt-smpl-msgR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc*′ (*Write-Sampling-Message p m*))
using *wrt-smpl-msgR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq

non-interference1-def non-interference-def singletonD event-enabledR-hc domain-of-eventR-hc
abstract-state-def vpeqR-def
by (*smt EventR.case*(*1*) *Hypercall'.case*(*2*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def mem-Collect-eq singletonD option.sel prod.simps(*2*))

lemma *read-smpl-msgR-presrv-wk-stp-cons-new*:
assumes *p1*:$s \sim .\ d\ .\sim t$
and *p2*:$s' =$ *fst* (*read-sampling-messageR s pid*)
and *p3*:$t' =$ *fst* (*read-sampling-messageR t pid*)
shows $s' \sim .\ d\ .\sim_{\triangle} t'$
using *p1 p2 p3* **by** *fastforce*

lemma *read-smpl-msgR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Read-Sampling-Message p*))
using *read-smpl-msgR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
non-interference1-def non-interference-def singletonD
by (*smt EventR.case*(*1*) *Hypercall'.case*(*3*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *get-smpl-pidR-presrv-wk-stp-cons-new*:
assumes *p1*:$s \sim .\ d\ .\sim t$
and *p2*:$s' =$ *fst* (*get-sampling-port-idR sysconf s pname*)
and *p3*:$t' =$ *fst* (*get-sampling-port-idR sysconf t pname*)
shows $s' \sim .\ d\ .\sim_{\triangle} t'$
using *p1 p2 p3* **by** *fastforce*

lemma *get-smpl-pidR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Get-Sampling-Portid p*))
using *get-smpl-pidR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
non-interference1-def non-interference-def singletonD
by (*smt EventR.case*(*1*) *Hypercall'.case*(*4*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *get-smpl-pstsR-presrv-wk-stp-cons-new*:
assumes *p1*:$s \sim .\ d\ .\sim t$
and *p2*:$s' =$ *fst* (*get-sampling-port-statusR sysconf s pid*)
and *p3*:$t' =$ *fst* (*get-sampling-port-statusR sysconf t pid*)
shows $s' \sim .\ d\ .\sim_{\triangle} t'$
using *p1 p2 p3* **by** *fastforce*

lemma *get-smpl-pstsR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Get-Sampling-Portstatus p*))
using *get-smpl-pstsR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
non-interference1-def non-interference-def singletonD
by (*smt EventR.case*(*1*) *Hypercall'.case*(*5*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *crt-que-portR-presrv-wk-stp-cons-new*:
assumes $p1{:}s \sim .\ d\ .\sim t$
and $p2{:}s' = fst\ (create\text{-}queuing\text{-}portR\ sysconf\ s\ pname)$
and $p3{:}t' = fst\ (create\text{-}queuing\text{-}portR\ sysconf\ t\ pname)$
shows $s' \sim .\ d\ .\sim_{\Delta}\ t'$
using *p1 p2 p3* **by** *auto*

lemma *crt-que-portR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Create-Queuing-Port p*))
using *crt-que-portR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
non-interference1-def non-interference-def singletonD
by (*smt EventR.case*(*1*) *Hypercall'.case*(*6*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *snd-que-msg-lstR-presrv-wk-stp-cons-new*:
assumes
$p1{:}s \sim .\ d\ .\sim t$
and $p2{:}s' = fst\ (send\text{-}queuing\text{-}message\text{-}maylostR\ sysconf\ s\ pid\ m)$
and $p3{:}t' = fst\ (send\text{-}queuing\text{-}message\text{-}maylostR\ sysconf\ t\ pid\ m)$
shows $s' \sim .\ d\ .\sim_{\Delta}\ t'$
using *p1 p2 p3* **by** *auto*

lemma *snd-que-msg-lstR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Send-Queuing-Message p m*))
using *snd-que-msg-lstR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
non-interference1-def non-interference-def singletonD
by (*smt EventR.case*(*1*) *Hypercall'.case*(*7*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *rec-que-msgR-presrv-wk-stp-cons-new*:
assumes $p1{:}s \sim .\ d\ .\sim t$
and $p2{:}s' = fst\ (receive\text{-}queuing\text{-}messageR\ s\ pid)$
and $p3{:}t' = fst\ (receive\text{-}queuing\text{-}messageR\ t\ pid)$

shows $s' \sim . \ d \ . \sim_{\Delta} \ t'$
using *p1 p2 p3* **by** *auto*

lemma *rec-que-msgR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc′* (*Receive-Queuing-Message p*))
using *rec-que-msgR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
non-interference1-def non-interference-def singletonD
by (*smt EventR.case*(*1*) *Hypercall′.case*(*8*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *get-que-pidR-presrv-wk-stp-cons-new*:
assumes *p1*: $s \sim . \ d \ . \sim \ t$
and *p2*: $s' = fst$ (*get-queuing-port-idR sysconf s pname*)
and *p3*: $t' = fst$ (*get-queuing-port-idR sysconf t pname*)
shows $s' \sim . \ d \ . \sim_{\Delta} \ t'$ **using** *p1 p2 p3* **by** *auto*

lemma *get-que-pidR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc′* (*Get-Queuing-Portid p*))
using *get-que-pidR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
by (*smt EventR.case*(*1*) *Hypercall′.case*(*9*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *get-que-pstsR-presrv-wk-stp-cons-new*:
assumes *p1*: $s \sim . \ d \ . \sim \ t$
and *p2*: $s' = fst$ (*get-queuing-port-statusR sysconf s pid*)
and *p3*: $t' = fst$ (*get-queuing-port-statusR sysconf t pid*)
shows $s' \sim . \ d \ . \sim_{\Delta} \ t'$ **using** *p1 p2 p3* **by** *auto*

lemma *get-que-pstsR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc′* (*Get-Queuing-Portstatus p*))
using *get-que-pstsR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
non-interference1-def non-interference-def singletonD
by (*smt EventR.case*(*1*) *Hypercall′.case*(*10*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *clr-que-portR-presrv-wk-stp-cons-new*:
assumes *p1*: $s \sim . \ d \ . \sim \ t$
and *p2*: $s' =$ *clear-queuing-portR s pid*
and *p3*: $t' =$ *clear-queuing-portR t pid*
shows $s' \sim . \ d \ . \sim_{\Delta} \ t'$

using *p1 p2 p3* **by** *auto*

lemma *clr-que-portR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc′* (*Clear-Queuing-Port p*))
using *clr-que-portR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
by (*smt EventR.case*(*1*) *Hypercall′.case*(*11*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *get-part-statusR-presrv-wk-stp-cons-new*:
assumes $p1{:}s \sim .\ d\ .\!\sim t$
and $p2{:}s' = fst\ (get\text{-}partition\text{-}statusR\ sysconf\ s)$
and $p3{:}t' = fst\ (get\text{-}partition\text{-}statusR\ sysconf\ t)$
shows $s' \sim .\ d\ .\!\sim_{\triangle} t'$ **using** *p1 p2 p3* **by** *auto*

lemma *get-part-statusR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc′ Get-Partition-Status*)
using *get-part-statusR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
by (*smt EventR.case*(*1*) *Hypercall′.case*(*13*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *scheduleR-presrv-wk-stp-cons-new*:
assumes $p1{:}s \sim .\ d\ .\!\sim t$
and $p2{:}s' \in scheduleR\ sysconf\ s$
and $p3{:}t' \in scheduleR\ sysconf\ t$
shows $s' \sim .\ d\ .\!\sim_{\triangle} t'$ **using** *p1 p2 p3* **by** *auto*

lemma *scheduleR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*sys′ Schedule*)
using *scheduleR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def same-part-mode sched-vpeq
by (*smt EventR.case*(*2*) *System-EventR.case*(*1*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-sys event-enabledR-sys mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *trans-smpl-msgR-presrv-wk-stp-cons-new*:
assumes
$p1{:}s \sim .\ d\ .\!\sim t$
and $p2{:}s' = transf\text{-}sampling\text{-}msgR\ s\ c$
and $p3{:}t' = transf\text{-}sampling\text{-}msgR\ t\ c$
shows $s' \sim .\ d\ .\!\sim_{\triangle} t'$ **using** *p1 p2 p3* **by** *auto*

lemma *trans-smpl-msgR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*sys′* (*Transfer-Sampling-Message c*))
using *trans-smpl-msgR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def same-part-mode is-a-transmitter-def vpeq1-def vpeq-sched-def
by (*smt EventR.case*(*2*) *System-EventR.case*(*2*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-sys event-enabledR-sys mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *trans-que-msg-mlostR-presrv-wk-stp-cons-new*:
assumes *p1*:$s \sim.\ d\ .\sim t$
and *p2*:$s' =$ *transf-queuing-msg-maylostR sysconf s c*
and *p3*:$t' =$ *transf-queuing-msg-maylostR sysconf t c*
shows $s' \sim.\ d\ .\sim_{\triangle} t'$ **using** *p1 p2 p3* **by** *auto*

lemma *trans-que-msg-mlostR-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*sys′* (*Transfer-Queuing-Message c*))
using *trans-que-msg-mlostR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def same-part-mode is-a-transmitter-def vpeq1-def vpeq-sched-def
by (*smt EventR.case*(*2*) *System-EventR.case*(*3*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-sys event-enabledR-sys mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

lemma *set-procs-to-normal-presrv-wk-stp-cons-new*:
assumes *p1*:*is-a-partition sysconf d*
and *p2*:$s \sim.\ d\ .\sim t$
and *p3*:$s \sim.$ (*scheduler sysconf*) $.\sim t$
and *p4*:$s' =$ *set-procs-to-normal s* (*current s*)
and *p5*:$t' =$ *set-procs-to-normal t* (*current t*)
shows $s' \sim.\ d\ .\sim_{\triangle} t'$
proof −
from *p1 p2* **have** *a0*: (*partitions s*) *d* = (*partitions t*) *d*
using *part-imp-not-sch part-imp-not-tras* **by** *force*
then show *?thesis* **using** *p1 p2 p3 p4 p5* **by** *auto*
qed

lemma *remove-partition-resources-presrv-wk-stp-cons-new*:
assumes *p1*:*is-a-partition sysconf d*
and *p2*:$s \sim.\ d\ .\sim t$
and *p3*:$s \sim.$ (*scheduler sysconf*) $.\sim t$
and *p4*:$s' =$ *remove-partition-resources s* (*current s*)
and *p5*:$t' =$ *remove-partition-resources t* (*current t*)
shows $s' \sim.\ d\ .\sim_{\triangle} t'$
proof −

```
    from p1 p2 have a0: (partitions s) d = (partitions t) d
      using part-imp-not-sch part-imp-not-tras by force
     show ?thesis using p1 p2 p3  p4 p5 by auto
  qed

lemma set-part-modeR-presrv-wk-stp-cons-new:
  assumes p1:is-a-partition sysconf (current s)
    and   p2:reachable0 s ∧ reachable0 t
    and   p3:s ∼. d .∼ t
    and   p4:s ∼. (scheduler sysconf) .∼ t
    and   p5:(current s) ⇝ d
    and   p6:s ∼. (current s) .∼ t
    and   p7:s' = set-partition-modeR sysconf s m
    and   p8:t' = set-partition-modeR sysconf t m
 shows    s' ∼. d .∼△ t'
 proof(cases is-a-partition sysconf d)
    assume a0:is-a-partition sysconf d
    show ?thesis
    proof(cases current s = d)
      assume b0: current s = d
      with p3 a0 have b1: (partitions s) d = (partitions t) d
        using part-imp-not-sch part-imp-not-tras by force
        thus ?thesis using p3 p1 a0 b0 p8 p2 p4 p6 p7 by auto
    next
      assume b0: current s ≠ d
      with p1 p2 p7 a0 have b1: vpeq-part-procs s d s'
        by auto
      from p1 p2 p3 p8 a0 p4 b0 have b2: vpeq-part-procs t d t'
        by auto
      from p3 b1 a0 b2 show ?thesis by auto
    qed
next
  assume b1:¬ is-a-partition sysconf d
  show ?thesis using b1 by auto
qed

lemma set-part-modeR-presrv-wk-stp-cons-new-e: weak-step-consistent-new-e (hyperc' (Set-Partition-Mode p))
  using set-part-modeR-presrv-wk-stp-cons-new weak-step-consistent-new-e-def
    exec-eventR-def same-part-mode sched-vpeq singletonD EventR.case(1) Hypercall'.case(12)
      by (smt State.select-convs(1) State.select-convs(2)
            abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
            singletonD vpeqR-def option.sel prod.simps(2))
```

3.9 New events preserve "step consistent" on new state variables

3.9.1 proving "Create process"

lemma *create-process-presrv-wk-stp-cons-new*:
  assumes *p1*:*is-a-partition sysconf* (*current s*)
   and *p2*:$s \sim. d .\sim t$
   and *p3*:$s \sim. (scheduler\ sysconf) .\sim t$
   and *p4*:$s \sim. (current\ s) .\sim t$
   and *p5*:$s' = fst\ (create\text{-}process\ s\ pri)$
   and *p6*:$t' = fst\ (create\text{-}process\ t\ pri)$
  shows $s' \sim. d .\sim_{\Delta} t'$
  proof(*cases is-a-partition sysconf d*)
   assume *a0*:*is-a-partition sysconf d*
   from *p3* **have** *a1*: $current\ s = current\ t$ **by** *auto*
   show *?thesis*
    proof(*cases* $current\ s = d$)
     assume *b0*: $current\ s = d$
     with *p2 a0* **have** *b1*: $(partitions\ s)\ d = (partitions\ t)\ d$
      using *part-imp-not-sch part-imp-not-tras* **by** *force*
      thus *?thesis* **using** *p1 p2 p3 p4 p6 p5* **by** *fastforce*
    next
     assume *b0*: $current\ s \neq d$
     with *p5* **have** *b1*: *vpeq-part-procs s d s'*
      using *create-process-vpeq-part-procs*[*OF b0 p5*] **by** (*simp cong del*:)
     moreover from *p6 a0 a1 b0* **have** *b2*: *vpeq-part-procs t d t'*
      using *create-process-vpeq-part-procs*[*OF b0*] **by** *simp*
     ultimately show *?thesis* **using** *p2* **by** *auto*
    qed
  next
   assume *b1*:$\neg$ *is-a-partition sysconf d*
   then show *?thesis* **by** *auto*
  qed

lemma *create-process-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Create-Process p*))
 using *create-process-presrv-wk-stp-cons-new weak-step-consistent-new-e-def exec-eventR-def*
  same-part-mode sched-vpeq
   by (*smt EventR.case*(*1*) *Hypercall'.case*(*14*) *State.select-convs*(*1*) *State.select-convs*(*2*)
    abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
    singletonD vpeqR-def option.sel prod.simps(*2*))

3.9.2 proving "set process priority"

lemma *set-process-priority-presrv-wk-stp-cons-new*:

```
  assumes p1:is-a-partition sysconf (current s)
    and  p2:reachable0 s ∧ reachable0 t
    and  p3:s ∼. d .∼ t
    and  p4:s ∼. (scheduler sysconf) .∼ t
    and  p5:s ∼. (current s) .∼ t
    and  p6:s' = set-process-priority s pr pri
    and  p7:t' = set-process-priority t pr pri
  shows  s' ∼. d .∼△ t'
  proof(cases is-a-partition sysconf d)
    assume a0:is-a-partition sysconf d
    from p3 p4 have a1: current s = current t
      by auto
    show ?thesis
      proof(cases current s = d)
        assume b0: current s = d
        with p3 a0 have b1: (partitions s) d = (partitions t) d
          using part-imp-not-sch part-imp-not-tras by force
        thus ?thesis using  p1 a0 b0 p5  p4 p6 p7 by auto
      next
        assume b0: current s ≠ d
        with p1 p2 p6 a0 have b1: vpeq-part-procs s d s'
          by auto
        from p1 p2 p7 a0 a1 b0 have b2: vpeq-part-procs t d t'
          by auto
        from p3 b1 a0 b2 show ?thesis by auto
      qed
  next
    assume b1:¬ is-a-partition sysconf d
    then show ?thesis by auto
  qed

lemma set-process-priority-presrv-wk-stp-cons-new-e: weak-step-consistent-new-e (hyperc' (Set-Priority p pri))
  using set-process-priority-presrv-wk-stp-cons-new weak-step-consistent-new-e-def
    exec-eventR-def same-part-mode sched-vpeq domain-of-eventR-hc event-enabledR-hc
      by (smt EventR.case(1) Hypercall'.case(19) State.select-convs(1) State.select-convs(2)
          abstract-state-def mem-Collect-eq old.prod.case singletonD vpeqR-def option.sel prod.simps(2))
```

3.9.3 proving "start process"

```
lemma start-process-presrv-wk-stp-cons-new:
  assumes p1:is-a-partition sysconf (current s)
    and  p2:reachable0 s ∧ reachable0 t
    and  p3:s ∼. d .∼ t
    and  p4:s ∼. (scheduler sysconf) .∼ t
```

and *p5*:*s* $\sim$. (*current s*) .$\sim$ *t*
and *p6*:*s'* = *start-process s pr*
and *p7*:*t'* = *start-process t pr*
shows *s'* $\sim$. *d* .$\sim_\Delta$ *t'*
proof(*cases is-a-partition sysconf d*)
assume *a0*:*is-a-partition sysconf d*
from *p3 p4* **have** *a1*: *current s* = *current t*
by *auto*
show *?thesis*
proof(*cases current s* = *d*)
assume *b0*: *current s* = *d*
with *p3 a0* **have** *b1*: (*partitions s*) *d* = (*partitions t*) *d*
using *part-imp-not-sch part-imp-not-tras* **by** *force*
thus *?thesis* **using** *p1 a0 b0 p5 p4 p6 p7* **by** *auto*
next
assume *b0*: *current s* $\neq$ *d*
with *p1 p2 p6 a0* **have** *b1*: *vpeq-part-procs s d s'*
by *auto*
from *p1 p2 p7 a0 a1 b0* **have** *b2*: *vpeq-part-procs t d t'*
by *auto*
from *p3 b1 a0 b2* **show** *?thesis* **by** *auto*
qed
next
assume *b1*:$\neg$ *is-a-partition sysconf d*
then show *?thesis* **by** *auto*
qed

lemma *start-process-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Start-Process p*))
using *start-process-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def same-part-mode sched-vpeq
by (*smt EventR.case*(*1*) *Hypercall'.case*(*15*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

3.9.4 proving "stop process"

lemma *stop-process-presrv-wk-stp-cons-new*:
assumes *p1*:*is-a-partition sysconf* (*current s*)
and *p2*:*reachable0 s* $\wedge$ *reachable0 t*
and *p3*:*s* $\sim$. *d* .$\sim$ *t*
and *p4*:*s* $\sim$. (*scheduler sysconf*) .$\sim$ *t*
and *p5*:*s* $\sim$. (*current s*) .$\sim$ *t*
and *p6*:*s'* = *stop-process s pr*

and *p7*:t' = *stop-process t pr*
shows $s' \sim .\ d\ .\sim_\Delta\ t'$
using *p1 p2 p3 p4 p5 p6 p7* **by** *auto*

lemma *stop-process-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Stop-Process p*))
using *stop-process-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def same-part-mode sched-vpeq
by (*smt EventR.case*(*1*) *Hypercall'.case*(*16*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

3.9.5 proving "suspend process"

lemma *suspend-process-presrv-wk-stp-cons-new*:
assumes *p1*:*is-a-partition sysconf* (*current s*)
and *p2*:*reachable0 s* $\wedge$ *reachable0 t*
and *p3*:$s \sim .\ d\ .\sim t$
and *p4*:$s \sim .$ (*scheduler sysconf*) $.\sim t$
and *p5*:$s \sim .$ (*current s*) $.\sim t$
and *p6*:s' = *suspend-process s pr*
and *p7*:t' = *suspend-process t pr*
shows $s' \sim .\ d\ .\sim_\Delta\ t'$
using *p1 p2 p3 p4 p5 p6 p7* **by** *auto*

lemma *suspend-process-presrv-wk-stp-cons-new-e*: *weak-step-consistent-new-e* (*hyperc'* (*Suspend-Process p*))
using *suspend-process-presrv-wk-stp-cons-new weak-step-consistent-new-e-def*
exec-eventR-def same-part-mode sched-vpeq
by (*smt EventR.case*(*1*) *Hypercall'.case*(*18*) *State.select-convs*(*1*) *State.select-convs*(*2*)
abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
singletonD vpeqR-def option.sel prod.simps(*2*))

3.9.6 proving "resume process"

lemma *resume-process-presrv-wk-stp-cons-new*:
assumes *p1*:*is-a-partition sysconf* (*current s*)
and *p2*:*reachable0 s* $\wedge$ *reachable0 t*
and *p3*:$s \sim .\ d\ .\sim t$
and *p4*:$s \sim .$ (*scheduler sysconf*) $.\sim t$
and *p5*:$s \sim .$ (*current s*) $.\sim t$
and *p6*:s' = *resume-process s pr*
and *p7*:t' = *resume-process t pr*
shows $s' \sim .\ d\ .\sim_\Delta\ t'$
using *p1 p2 p3 p4 p5 p6 p7* **by** *auto*

```
lemma resume-process-presrv-wk-stp-cons-new-e: weak-step-consistent-new-e (hyperc' (Resume-Process p))
  using resume-process-presrv-wk-stp-cons-new weak-step-consistent-new-e-def
    exec-eventR-def same-part-mode sched-vpeq
        by (smt EventR.case(1) Hypercall'.case(17) State.select-convs(1) State.select-convs(2)
              abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
              singletonD vpeqR-def option.sel prod.simps(2))
```

3.9.7 proving "get process status"

```
lemma get-process-status-presrv-wk-stp-cons-new:
    assumes p1:s ∼. d .∼ t
      and   p2:s' = fst (get-process-status s pr)
      and   p3:t' = fst (get-process-status t pr)
    shows   s' ∼. d .∼△ t'
  proof −
    have a0:s' = s using p2 by auto
    have a1:t' = t using p3 by auto
    then show ?thesis using a0 p1 by auto
  qed
```

```
lemma get-process-status-presrv-wk-stp-cons-new-e: weak-step-consistent-new-e (hyperc' (Get-Process-Status p))
  using get-process-status-presrv-wk-stp-cons-new weak-step-consistent-new-e-def
    exec-eventR-def domain-of-eventR-def event-enabledR-def same-part-mode sched-vpeq
    non-interference1-def non-interference-def singletonD
        by (smt EventR.case(1) Hypercall'.case(20) State.select-convs(1) State.select-convs(2)
              abstract-state-def domain-of-eventR-hc event-enabledR-hc mem-Collect-eq
              singletonD vpeqR-def option.sel prod.simps(2))
```

3.9.8 proving "schedule process"

```
lemma setrun2ready-presrv-wk-stp-cons-new:
    assumes
            p1:s ∼. d .∼ t
      and   p2:s ∼. (scheduler sysconf) .∼ t
      and   p3:s' = setRun2Ready s
      and   p4:t' = setRun2Ready t
    shows   s' ∼. d .∼△ t'
   using p1 p2 p3 p4 by auto
```

```
lemma schedule-process-presrv-wk-stp-cons-new:
  assumes p1:is-a-partition sysconf (current s)
    and   p2:reachable0 s ∧ reachable0 t
    and   p3:s ∼. d .∼ t
    and   p4:s ∼. (scheduler sysconf) .∼ t
```

```
  and   p5:(current s) ⇝ d
  and   p6:s ∼. (current s) .∼ t
  and   p7:s' ∈ schedule-process s
  and   p8:t' ∈ schedule-process t
shows   s' ∼. d .∼△ t'

proof(cases is-a-partition sysconf d)
  assume a0:is-a-partition sysconf d
  from p3 p4 have a1: current s = current t
    using sched-currentR-lemma domain-of-eventR-hc
      by auto
  show ?thesis
  proof(cases current s = d)
    assume b0: current s = d
    with p3 a0 have b1: (partitions s) d = (partitions t) d
      using part-imp-not-sch part-imp-not-tras by force
    from p3 a0 have b2: vpeq-part-procs s d t by (simp add: vpeqR-def)
    with p1 b0 have b3: (procs s) d = (procs t) d ∧
                  (∀ p. (proc-state s) (d,p) = (proc-state t) (d,p))∧
                  (cur-proc-part s) d = (cur-proc-part t) d
      by auto
    with p7 p8 have r1: procs s' d = procs t' d
      using schedule-process-def setRun2Ready-def by (smt StateR.select-convs(1) StateR.surjective
        StateR.update-convs(2) StateR.update-convs(3)  singletonD)
    moreover
    have r2: (cur-proc-part s') d = (cur-proc-part t') d ∧ (∀ p. (proc-state s') (d,p) = (proc-state t') (d,p))
    proof –
      let ?cons = is-a-partition sysconf (current s)
               ∧ part-mode (the ((partitions s) (current s))) = NORMAL
      let ?cont = is-a-partition sysconf (current t)
               ∧ part-mode (the ((partitions t) (current t))) = NORMAL
      have b9: ?cons = ?cont using a1 b0 b1 by auto
      show ?thesis
      proof(cases ?cons)
        assume c0: ?cons
        then have c1: ?cont using b9 by simp
        let ?s' = setRun2Ready s
        let ?readyprs = {p. p∈the (procs ?s' (current ?s')) ∧
                            state (the (proc-state ?s' (current ?s',p))) = READY}
        let ?selp = SOME p. p∈{x. state (the (proc-state ?s' (current ?s',x))) = READY ∧
                            (∀ y∈?readyprs. priority (the (proc-state ?s' (current ?s',x))) ≥
                                   priority (the (proc-state ?s' (current ?s',y))))}
        let ?st = the ((proc-state ?s') (current ?s', ?selp))
```

```
      let ?proc-st = proc-state ?s′
      let ?cur-pr = cur-proc-part ?s′
      from c0 have c2: schedule-process s = {?s′(|proc-state := ?proc-st ((current ?s′, ?selp) := Some (?st(|state := RUNNING|))),
                                  cur-proc-part := ?cur-pr(current ?s′ := Some ?selp)|)}
        using schedule-process-def [of s] by auto

      let ?t′ = setRun2Ready t
      let ?readyprst = {p. p∈the (procs ?t′ (current ?t′)) ∧
                          state (the (proc-state ?t′ (current ?t′,p))) = READY}
      let ?selpt = SOME p. p∈{x. state (the (proc-state ?t′ (current ?t′,x))) = READY ∧
                                  (∀ y∈?readyprst. priority (the (proc-state ?t′ (current ?t′,x))) ≥
                                        priority (the (proc-state ?t′ (current ?t′,y))))}
      let ?stt = the ((proc-state ?t′) (current ?t′, ?selpt))
      let ?proc-stt = proc-state ?t′
      let ?cur-prt = cur-proc-part ?t′
      from c1 have c3: schedule-process t = {?t′(|proc-state := ?proc-stt ((current ?t′, ?selpt) := Some (?stt(|state := RUNNING|))),
                                  cur-proc-part := ?cur-prt(current ?t′ := Some ?selpt)|)}
        using schedule-process-def [of t] by auto
      have c4: ?s′ ∼. d .∼Δ ?t′
        using b0 p1 p2 p4 p6 setrun2ready-presrv-wk-stp-cons-new
        by (fastforce cong del:setRun2Ready-def vpeq-part-procs-def)
      then have c5: ((procs ?s′) d = (procs ?t′) d) ∧
                 (∀ p. (proc-state ?s′) (d,p) = (proc-state ?t′) (d,p)) ∧
                 (cur-proc-part ?s′) d = (cur-proc-part ?t′) d
        using a0 by auto
      have c7: current ?s′ = current ?t′ using a1
        by fastforce
      have c8: current s = current ?s′ using a1 setrun2ready-nchastate-lemma by fastforce
      then  show ?thesis using  p7 p8 c2 a0 c3 c5 a0 c7 c8 b0 a1
       by (fastforce cong del: setRun2Ready-def)
    next
      assume c0: ¬ ?cons
      thus ?thesis using p7 p8 p3 b3 b9 by auto
    qed
  qed
  ultimately show ?thesis
    by auto
next
  assume b0: current s ≠ d
  from  p8 p7 a0 a1 b0 p3
  show ?thesis using schedule-process-vpeq-part-procs
    by (auto cong del: setRun2Ready-def)
qed
```

```
  next
    assume b1:¬ is-a-partition sysconf d
    then show ?thesis by auto
  qed

lemma schedule-process-presrv-wk-stp-cons-new-e: weak-step-consistent-new-e (sys′ Schedule-Process)
  using schedule-process-presrv-wk-stp-cons-new weak-step-consistent-new-e-def
    exec-eventR-def same-part-mode sched-vpeq
      by (smt EventR.case(2) System-EventR.case(4) State.select-convs(1) State.select-convs(2)
            abstract-state-def domain-of-eventR-sys event-enabledR-sys mem-Collect-eq
            singletonD vpeqR-def option.sel prod.simps(2))
```

3.10 Proving the "step consistent" property on new state variables

```
theorem weak-step-consistent-new:weak-step-consistent-new
  proof –
    {
      fix e
      have weak-step-consistent-new-e e
        apply(induct e)
        using crt-smpl-portR-presrv-wk-stp-cons-new-e wrt-smpl-msgR-presrv-wk-stp-cons-new-e
              read-smpl-msgR-presrv-wk-stp-cons-new-e get-smpl-pidR-presrv-wk-stp-cons-new-e
              get-smpl-pstsR-presrv-wk-stp-cons-new-e crt-que-portR-presrv-wk-stp-cons-new-e
              snd-que-msg-lstR-presrv-wk-stp-cons-new-e rec-que-msgR-presrv-wk-stp-cons-new-e
              get-que-pidR-presrv-wk-stp-cons-new-e get-que-pstsR-presrv-wk-stp-cons-new-e
              clr-que-portR-presrv-wk-stp-cons-new-e set-part-modeR-presrv-wk-stp-cons-new-e
              get-part-statusR-presrv-wk-stp-cons-new-e create-process-presrv-wk-stp-cons-new-e
              start-process-presrv-wk-stp-cons-new-e stop-process-presrv-wk-stp-cons-new-e
              resume-process-presrv-wk-stp-cons-new-e suspend-process-presrv-wk-stp-cons-new-e
              set-process-priority-presrv-wk-stp-cons-new-e get-process-status-presrv-wk-stp-cons-new-e
        apply(rule Hypercall′.induct)
        using scheduleR-presrv-wk-stp-cons-new-e trans-smpl-msgR-presrv-wk-stp-cons-new-e
          trans-que-msg-mlostR-presrv-wk-stp-cons-new-e schedule-process-presrv-wk-stp-cons-new-e
          by (rule System-EventR.induct)
    }
    then show ?thesis using weak-step-consistent-new-all-evt by simp
  qed
```

3.11 Information flow security of second-level specification

```
theorem noninfluence-sat: noninfluence
  using noninfl-refinement local-respect-new weak-step-consistent-new
    local-respect weak-step-consistent by blast
```

theorem *noninfluence-gen-sat*: *noninfluence-gen*
using *noninf-eq-noninf-gen noninfluence-sat* **by** *blast*

theorem *weak-noninfluence-sat*: *weak-noninfluence*
using *noninf-impl-weak noninfluence-sat* **by** *blast*

theorem *nonleakage-sat*: *nonleakage*
using *noninf-impl-nonlk noninfluence-sat* **by** *blast*

theorem *noninterference-r-sat*: *noninterference-r*
using *noninf-impl-nonintf-r noninfluence-sat* **by** *blast*

theorem *noninterference-sat*: *noninterference*
using *noninterference-r-sat nonintf-r-impl-noninterf* **by** *blast*

theorem *weak-noninterference-r-sat*: *weak-noninterference-r*
using *noninterference-r-sat nonintf-r-impl-wk-nonintf-r* **by** *blast*

theorem *weak-noninterference-sat*: *weak-noninterference*
using *noninterference-sat nonintf-impl-weak* **by** *blast*

end