Conservation of CSP Noninterference Security under Concurrent Composition

Pasquale Noce
Security Certification Specialist at Arjo Systems, Italy
pasquale dot noce dot lavoro at gmail dot com
pasquale dot noce at arjosystems dot com

March 17, 2025

Abstract

In his outstanding work on Communicating Sequential Processes, Hoare has defined two fundamental binary operations allowing to compose the input processes into another, typically more complex, process: sequential composition and concurrent composition. Particularly, the output of the latter operation is a process in which any event not shared by both operands can occur whenever the operand that admits the event can engage in it, whereas any event shared by both operands can occur just in case both can engage in it.

This paper formalizes Hoare's definition of concurrent composition and proves, in the general case of a possibly intransitive policy, that CSP noninterference security is conserved under this operation. This result, along with the previous analogous one concerning sequential composition, enables the construction of more and more complex processes enforcing noninterference security by composing, sequentially or concurrently, simpler secure processes, whose security can in turn be proven using either the definition of security, or unwinding theorems.

Contents

1 Concurrent composition and noninterference security

theory *ConcurrentComposition*
imports *Noninterference-Sequential-Composition.Propaedeutics*
begin

In his outstanding work on Communicating Sequential Processes [1], Hoare has defined two fundamental binary operations allowing to compose the input processes into another, typically more complex, process: sequential composition and concurrent composition. Particularly, the output of the latter operation is a process in which any event not shared by both operands can occur whenever the operand that admits the event can engage in it, whereas any event shared by both operands can occur just in case both can engage in it. In other words, shared events are those that synchronize the concurrent processes, which on the contrary can engage asynchronously in the respective non-shared events.

This paper formalizes Hoare's definition of concurrent composition and proves, in the general case of a possibly intransitive policy, that CSP noninterference security [6] is conserved under this operation, viz. the security of both of the input processes implies that of the output process. This result, along with the analogous one concerning sequential composition attained in [10], enables the construction of more and more complex processes enforcing noninterference security by composing, sequentially or concurrently, simpler secure processes, whose security can in turn be proven using either the definition of security formulated in [6], or the unwinding theorems demonstrated in [9], [7], and [8].

Throughout this paper, the salient points of definitions and proofs are commented; for additional information, cf. Isabelle documentation, particularly [5], [4], [3], and [2].

1.1 Propaedeutic definitions and lemmas

The starting point is comprised of some definitions and lemmas propaedeutic to the proof of the target security conservation theorem.

Particularly, the definition of operator *after* given in [1] is formalized, and it is proven that for any secure process P and any trace xs of P, P after xs is still a secure process. Then, this result is used to generalize the lemma stating the closure of the failures of a secure process P under intransitive purge, proven in [10], to the futures of P associated to any one of its traces. This is a generalization of the former result since *futures* P xs = *failures* P for $xs = []$.

lemma *sinks-aux-elem* [*rule-format*]:
$u \in$ *sinks-aux I D U xs* $\longrightarrow u \in U \vee (\exists x \in$ *set xs*. $u = D\ x)$
by (*induction xs rule*: *rev-induct*, *simp-all*, *blast*)

lemma *ipurge-ref-aux-cons*:
ipurge-ref-aux I D U $(x \# xs)$ $X =$ *ipurge-ref-aux I D* (*sinks-aux I D U* $[x]$) *xs X*
by (*subgoal-tac* $x \# xs = [x] @ xs$, *simp only*: *ipurge-ref-aux-append*, *simp*)

lemma *process-rule-1-futures*:
$xs \in$ *traces P* $\Longrightarrow ([], \{\}) \in$ *futures P xs*
by (*simp add*: *futures-def*, *rule traces-failures*)

lemma *process-rule-3-futures*:
$(ys, Y) \in$ *futures P xs* $\Longrightarrow Y' \subseteq Y \Longrightarrow (ys, Y') \in$ *futures P xs*
by (*simp add*: *futures-def*, *rule process-rule-3*)

lemma *process-rule-4-futures*:
$(ys, Y) \in$ *futures P xs* $\Longrightarrow$
  $(ys @ [x], \{\}) \in$ *futures P xs* $\vee$ $(ys,$ *insert x Y*$) \in$ *futures P xs*
by (*simp add*: *futures-def*, *subst append-assoc* [*symmetric*], *rule process-rule-4*)

lemma *process-rule-5-general* [*rule-format*]:
$xs \in$ *divergences P* $\longrightarrow xs @ ys \in$ *divergences P*
proof (*induction ys rule*: *rev-induct*, *simp*, *rule impI*, *simp*)
qed (*subst append-assoc* [*symmetric*], *rule process-rule-5*)

Here below is the definition of operator *after*, for which a symbolic notation similar to the one used in [1] is introduced. Then, it is proven that for any process P and any trace *xs* of P, the failures set and the divergences set of P after *xs* indeed enjoy their respective characteristic properties as defined in [6].

definition *future-divergences* :: *'a process* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list set* **where**
future-divergences P xs $\equiv \{ys.\ xs @ ys \in$ *divergences P*$\}$

definition *after* :: *'a process* $\Rightarrow$ *'a list* $\Rightarrow$ *'a process* (**infixl** ‹\› *64*) **where**
$P \setminus xs \equiv$ *Abs-process* (*futures P xs*, *future-divergences P xs*)

lemma *process-rule-5-futures*:
$ys \in$ *future-divergences P xs* $\Longrightarrow ys @ [x] \in$ *future-divergences P xs*
by (*simp add*: *future-divergences-def*, *subst append-assoc* [*symmetric*],
rule process-rule-5)

lemma *process-rule-6-futures*:
$ys \in$ *future-divergences P xs* $\Longrightarrow (ys, Y) \in$ *futures P xs*
by (*simp add*: *futures-def future-divergences-def*, *rule process-rule-6*)

lemma *after-rep*:
  assumes *A*: *xs* ∈ *traces P*
  shows *Rep-process* (*P* \ *xs*) = (*futures P xs*, *future-divergences P xs*)
    (**is** - = *?X*)
proof (*subst after-def*, *rule Abs-process-inverse*, *simp add*: *process-set-def*,
 (*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+)
  show *process-prop-1 ?X*
  proof (*simp add*: *process-prop-1-def*)
  qed (*rule process-rule-1-futures* [*OF A*])
next
  show *process-prop-2 ?X*
  proof (*simp add*: *process-prop-2-def del*: *all-simps*, (*rule allI*)+, *rule impI*)
  qed (*rule process-rule-2-futures*)
next
  show *process-prop-3 ?X*
  proof (*simp add*: *process-prop-3-def del*: *all-simps*, (*rule allI*)+, *rule impI*,
   erule conjE)
  qed (*rule process-rule-3-futures*)
next
  show *process-prop-4 ?X*
  proof (*simp add*: *process-prop-4-def*, (*rule allI*)+, *rule impI*)
  qed (*rule process-rule-4-futures*)
next
  show *process-prop-5 ?X*
  proof (*simp add*: *process-prop-5-def*, *rule allI*, *rule impI*, *rule allI*)
  qed (*rule process-rule-5-futures*)
next
  show *process-prop-6 ?X*
  proof (*simp add*: *process-prop-6-def*, *rule allI*, *rule impI*, *rule allI*)
  qed (*rule process-rule-6-futures*)
qed

lemma *after-failures*:
  assumes *A*: *xs* ∈ *traces P*
  shows *failures* (*P* \ *xs*) = *futures P xs*
by (*simp add*: *failures-def after-rep* [*OF A*])

lemma *after-futures*:
  assumes *A*: *xs* ∈ *traces P*
  shows *futures* (*P* \ *xs*) *ys* = *futures P* (*xs* @ *ys*)
by (*simp add*: *futures-def after-failures* [*OF A*])

Finally, the closure of the futures of a secure process under intransitive purge is proven.

lemma *after-secure*:
  assumes *A*: *xs* ∈ *traces P*
  shows *secure P I D* ⟹ *secure* (*P* \ *xs*) *I D*

by (*simp add*: *secure-def after-futures* [*OF A*], *blast*)

lemma *ipurge-tr-ref-aux-futures*:
$[\![$*secure P I D*; (*ys*, *Y*) $\in$ *futures P xs*$]\!] \Longrightarrow$
  (*ipurge-tr-aux I D U ys*, *ipurge-ref-aux I D U ys Y*) $\in$ *futures P xs*
proof (*subgoal-tac xs* $\in$ *traces P*, *simp add*: *after-failures* [*symmetric*],
 rule ipurge-tr-ref-aux-failures, *rule after-secure*, *assumption*+)
qed (*simp add*: *futures-def*, *drule failures-traces*, *rule process-rule-2-traces*)

lemma *ipurge-tr-ref-aux-failures-general*:
$[\![$*secure P I D*; (*xs* @ *ys*, *Y*) $\in$ *failures P*$]\!] \Longrightarrow$
  (*xs* @ *ipurge-tr-aux I D U ys*, *ipurge-ref-aux I D U ys Y*) $\in$ *failures P*
by (*drule ipurge-tr-ref-aux-futures*, *simp-all add*: *futures-def*)

1.2 Concurrent composition

In [1], the concurrent composition of two processes P, Q, expressed using notation $P \parallel Q$, is defined as a process whose alphabet is the union of the alphabets of P and Q, so that the shared events requiring the synchronous participation of both processes are those in the intersection of their alphabets.

In the formalization of Communicating Sequential Processes developed in [6], the alphabets of P and Q are the data types $'a$ and $'b$ nested in their respective types $'a$ *process* and $'b$ *process*. Therefore, for any two maps p, q, the concurrent composition of P and Q with respect to p and q, expressed using notation $P \parallel Q$ <p, q>, is defined in what follows as a process of type $'c$ *process*, where meaningful events are those in *range p* $\cup$ *range q* and shared events are those in *range p* $\cap$ *range q*.

The case where $-$ (*range p* $\cup$ *range q*) $\neq$ {} constitutes a generalization of the definition given in [1], and the events in $-$ (*range p* $\cup$ *range q*), not being mapped to any event in the alphabets of the input processes, shall be understood as fake events lacking any meaning. Consistently with this interpretation, such events are allowed to occur in divergent traces only – necessarily, since divergences are capable by definition of giving rise to any sort of event. As a result, while in [1] the refusals associated to non-divergent traces are the union of two sets, a refusal of P and a refusal of Q, in the following definition they are the union of three sets instead, where the third set is any subset of $-$ (*range p* $\cup$ *range q*).

Since the definition given in [1] preserves the identity of the events of the input processes, a further generalization resulting from the following definition corresponds to the case where either map p, q is not injective. However, as shown below, these generalizations turn out to compromise neither the compliance of the output of concurrent composition with the characteristic properties of processes as defined in [6], nor even the validity of the target security conservation theorem.

Since divergences can contain fake events, whereas non-divergent traces cannot, it is necessary to add divergent failures to the failures set explicitly. The following definition of the divergences set restricts the definition given in [1], as it identifies a divergence with an arbitrary extension of an event sequence *xs* being a divergence of both P and Q, rather than a divergence of either process and a trace of the other one. This is a reasonable restriction, in that it requires the concurrent composition of P and Q to admit a shared event x in a divergent trace just in case both P and Q diverge and can then accept x, analogously to what is required for a non-divergent trace. Anyway, the definitions match if the input processes do not diverge, which is the case for any process of practical significance (cf. [1]).

definition *con-comp-divergences* ::
'a process ⇒ 'b process ⇒ ('a ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ 'c list set **where**
con-comp-divergences P Q p q ≡
{*xs* @ *ys* | *xs ys*.
set xs ⊆ *range p* ∪ *range q* ∧
map (*inv p*) [*x*←*xs*. *x* ∈ *range p*] ∈ *divergences P* ∧
map (*inv q*) [*x*←*xs*. *x* ∈ *range q*] ∈ *divergences Q*}

definition *con-comp-failures* ::
'a process ⇒ 'b process ⇒ ('a ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ 'c failure set **where**
con-comp-failures P Q p q ≡
{(*xs*, *X* ∪ *Y* ∪ *Z*) | *xs X Y Z*.
set xs ⊆ *range p* ∪ *range q* ∧
X ⊆ *range p* ∧ *Y* ⊆ *range q* ∧ *Z* ⊆ − (*range p* ∪ *range q*) ∧
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ‘ *X*) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *inv q* ‘ *Y*) ∈ *failures Q*} ∪
{(*xs*, *X*). *xs* ∈ *con-comp-divergences P Q p q*}

definition *con-comp* ::
'a process ⇒ 'b process ⇒ ('a ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ 'c process **where**
con-comp P Q p q ≡
Abs-process (*con-comp-failures P Q p q*, *con-comp-divergences P Q p q*)

abbreviation *con-comp-syntax* ::
'a process ⇒ 'b process ⇒ ('a ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ 'c process
(‹(- ‖ - <-, ->)› *55*)
where
P ‖ *Q* <*p*, *q*> ≡ *con-comp P Q p q*

Here below is the proof that, for any two processes P, Q and any two maps p, q, sets *con-comp-failures P Q p q* and *con-comp-divergences P Q p q* enjoy the characteristic properties of the failures and the divergences sets of a process as defined in [6].

lemma *con-comp-prop-1*:
([], {}) ∈ *con-comp-failures P Q p q*
proof (*simp add*: *con-comp-failures-def*)
qed (*rule disjI1*, *rule conjI*, (*rule process-rule-1*)+)

lemma *con-comp-prop-2*:
(*xs* @ [*x*], *X*) ∈ *con-comp-failures P Q p q* ⟹
(*xs*, {}) ∈ *con-comp-failures P Q p q*
proof (*simp add*: *con-comp-failures-def del*: *filter-append*,
erule disjE, (*erule exE*)+, (*erule conjE*)+, *rule disjI1*)
fix *X Y*
assume
A: *set xs* ⊆ *range p* ∪ *range q* **and**
B: (*map* (*inv p*) [*x*←*xs* @ [*x*]. *x* ∈ *range p*], *inv p* ‘ *X*) ∈ *failures P* **and**
C: (*map* (*inv q*) [*x*←*xs* @ [*x*]. *x* ∈ *range q*], *inv q* ‘ *Y*) ∈ *failures Q*
show *set xs* ⊆ *range p* ∪ *range q* ∧
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], {}) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], {}) ∈ *failures Q*
proof (*simp add*: *A*, *rule conjI*, *cases x* ∈ *range p*,
case-tac [*3*] *x* ∈ *range q*)
assume *x* ∈ *range p*
hence (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @ [*inv p x*], *inv p* ‘ *X*) ∈ *failures P*
using *B* **by** *simp*
thus (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], {}) ∈ *failures P*
by (*rule process-rule-2*)
next
assume *x* ∉ *range p*
hence (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ‘ *X*) ∈ *failures P*
using *B* **by** *simp*
moreover have {} ⊆ *inv p* ‘ *X* **..**
ultimately show (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], {}) ∈ *failures P*
by (*rule process-rule-3*)
next
assume *x* ∈ *range q*
hence (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] @ [*inv q x*], *inv q* ‘ *Y*) ∈ *failures Q*
using *C* **by** *simp*
thus (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], {}) ∈ *failures Q*
by (*rule process-rule-2*)
next
assume *x* ∉ *range q*
hence (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *inv q* ‘ *Y*) ∈ *failures Q*
using *C* **by** *simp*
moreover have {} ⊆ *inv q* ‘ *Y* **..**
ultimately show (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], {}) ∈ *failures Q*
by (*rule process-rule-3*)
qed
next
assume *A*: *xs* @ [*x*] ∈ *con-comp-divergences P Q p q*
show

set xs ⊆ *range p* ∪ *range q* ∧
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], {}) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], {}) ∈ *failures Q* ∨
xs ∈ *con-comp-divergences P Q p q*
(**is** *?A* ∨ -)
proof (*insert A*, *simp add*: *con-comp-divergences-def*,
((*erule exE*)*?*, *erule conjE*)+)
fix *ws ys*
assume
B: *xs* @ [*x*] = *ws* @ *ys* **and**
C: *set ws* ⊆ *range p* ∪ *range q* **and**
D: *map* (*inv p*) [*x*←*ws*. *x* ∈ *range p*] ∈ *divergences P* **and**
E: *map* (*inv q*) [*x*←*ws*. *x* ∈ *range q*] ∈ *divergences Q*
show *?A* ∨ (∃ *ws'*.
(∃ *ys'*. *xs* = *ws'* @ *ys'*) ∧
set ws' ⊆ *range p* ∪ *range q* ∧
map (*inv p*) [*x*←*ws'*. *x* ∈ *range p*] ∈ *divergences P* ∧
map (*inv q*) [*x*←*ws'*. *x* ∈ *range q*] ∈ *divergences Q*)
(**is** - ∨ (∃ *ws'*. *?B ws'*))
proof (*cases ys*, *rule disjI1*, *rule-tac* [*2*] *disjI2*)
case *Nil*
hence *set* (*xs* @ [*x*]) ⊆ *range p* ∪ *range q*
using *B* **and** *C* **by** *simp*
hence *insert x* (*set xs*) ⊆ *range p* ∪ *range q*
by *simp*
moreover have *set xs* ⊆ *insert x* (*set xs*)
by (*rule subset-insertI*)
ultimately have *set xs* ⊆ *range p* ∪ *range q*
by *simp*
moreover have *map* (*inv p*) [*x*←*xs* @ [*x*]. *x* ∈ *range p*] ∈ *divergences P*
using *Nil* **and** *B* **and** *D* **by** *simp*
hence (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], {}) ∈ *failures P*
proof (*cases x* ∈ *range p*, *simp-all*)
assume *map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @ [*inv p x*] ∈ *divergences P*
hence (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @ [*inv p x*], {}) ∈ *failures P*
by (*rule process-rule-6*)
thus *?thesis*
by (*rule process-rule-2*)
next
assume *map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] ∈ *divergences P*
thus *?thesis*
by (*rule process-rule-6*)
qed
moreover have *map* (*inv q*) [*x*←*xs* @ [*x*]. *x* ∈ *range q*] ∈ *divergences Q*
using *Nil* **and** *B* **and** *E* **by** *simp*
hence (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], {}) ∈ *failures Q*
proof (*cases x* ∈ *range q*, *simp-all*)
assume *map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] @ [*inv q x*] ∈ *divergences Q*
hence (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] @ [*inv q x*], {}) ∈ *failures Q*

by (*rule process-rule-6*)
thus *?thesis*
by (*rule process-rule-2*)
next
assume *map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] ∈ *divergences Q*
thus *?thesis*
by (*rule process-rule-6*)
qed
ultimately show *?A*
by *blast*
next
case *Cons*
moreover have *butlast* (*xs* @ [*x*]) = *butlast* (*ws* @ *ys*)
using *B* **by** *simp*
ultimately have *xs* = *ws* @ *butlast ys*
by (*simp add*: *butlast-append*)
hence ∃ *ys'*. *xs* = *ws* @ *ys'* **..**
hence *?B ws*
using *C* **and** *D* **and** *E* **by** *simp*
thus ∃ *ws'*. *?B ws'* **..**
qed
qed
qed

lemma *con-comp-prop-3*:
⟦(*xs*, *Y*) ∈ *con-comp-failures P Q p q*; *X* ⊆ *Y*⟧ ⟹
(*xs*, *X*) ∈ *con-comp-failures P Q p q*
proof (*simp add*: *con-comp-failures-def*, *erule disjE*, *simp-all*,
(*erule exE*)+, (*erule conjE*)+, *rule disjI1*, *simp*)
fix *X' Y' Z'*
assume
A: *X* ⊆ *X'* ∪ *Y'* ∪ *Z'* **and**
B: *X'* ⊆ *range p* **and**
C: *Y'* ⊆ *range q* **and**
D: *Z'* ⊆ − *range p* **and**
E: *Z'* ⊆ − *range q* **and**
F: (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ‘ *X'*) ∈ *failures P* **and**
G: (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *inv q* ‘ *Y'*) ∈ *failures Q*
show ∃ *X' Y' Z'*.
X = *X'* ∪ *Y'* ∪ *Z'* ∧
X' ⊆ *range p* ∧
Y' ⊆ *range q* ∧
Z' ⊆ − *range p* ∧
Z' ⊆ − *range q* ∧
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ‘ *X'*) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *inv q* ‘ *Y'*) ∈ *failures Q*
proof (*rule-tac x* = *X'* ∩ *X* **in** *exI*, *rule-tac x* = *Y'* ∩ *X* **in** *exI*,
rule-tac x = *Z'* ∩ *X* **in** *exI*, (*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+)
show *X* = *X'* ∩ *X* ∪ *Y'* ∩ *X* ∪ *Z'* ∩ *X*

```
    using A by blast
  next
   show X′ ∩ X ⊆ range p
    using B by blast
  next
   show Y′ ∩ X ⊆ range q
    using C by blast
  next
   show Z′ ∩ X ⊆ − range p
    using D by blast
  next
   show Z′ ∩ X ⊆ − range q
    using E by blast
  next
   have inv p ‘ (X′ ∩ X) ⊆ inv p ‘ X′
    by blast
   with F show (map (inv p) [x←xs. x ∈ range p], inv p ‘ (X′ ∩ X))
     ∈ failures P
    by (rule process-rule-3)
  next
   have inv q ‘ (Y′ ∩ X) ⊆ inv q ‘ Y′
    by blast
   with G show (map (inv q) [x←xs. x ∈ range q], inv q ‘ (Y′ ∩ X))
     ∈ failures Q
    by (rule process-rule-3)
  qed
qed

lemma con-comp-prop-4:
 (xs, X) ∈ con-comp-failures P Q p q ⟹
   (xs @ [x], {}) ∈ con-comp-failures P Q p q ∨
   (xs, insert x X) ∈ con-comp-failures P Q p q
proof (simp add: con-comp-failures-def del: filter-append,
 erule disjE, (erule exE)+, (erule conjE)+, simp-all del: filter-append)
  fix X Y Z
  assume
   A: X ⊆ range p and
   B: Y ⊆ range q and
   C: Z ⊆ − range p and
   D: Z ⊆ − range q and
   E: (map (inv p) [x←xs. x ∈ range p], inv p ‘ X) ∈ failures P and
   F: (map (inv q) [x←xs. x ∈ range q], inv q ‘ Y) ∈ failures Q
  show
   (x ∈ range p ∨ x ∈ range q) ∧
     (map (inv p) [x←xs @ [x]. x ∈ range p], {}) ∈ failures P ∧
     (map (inv q) [x←xs @ [x]. x ∈ range q], {}) ∈ failures Q ∨
   xs @ [x] ∈ con-comp-divergences P Q p q ∨
   (∃X′ Y′ Z′.
     insert x (X ∪ Y ∪ Z) = X′ ∪ Y′ ∪ Z′ ∧
```

$X' \subseteq$ *range p* $\wedge$
$Y' \subseteq$ *range q* $\wedge$
$Z' \subseteq -$ *range p* $\wedge$
$Z' \subseteq -$ *range q* $\wedge$
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ‘ *X'*) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *inv q* ‘ *Y'*) ∈ *failures Q*) ∨
xs ∈ *con-comp-divergences P Q p q*
(**is** - ∨ - ∨ *?A* ∨ -)
proof (*cases x* ∈ *range p*, *case-tac* [!] *x* ∈ *range q*, *simp-all*)
assume
G: *x* ∈ *range p* **and**
H: *x* ∈ *range q*
show
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @ [*inv p x*], {}) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] @ [*inv q x*], {}) ∈ *failures Q* ∨
xs @ [*x*] ∈ *con-comp-divergences P Q p q* ∨
?A ∨
xs ∈ *con-comp-divergences P Q p q*
(**is** *?B* ∨ -)
proof (*cases ?B*, *simp-all del*: *disj-not1*, *erule disjE*)
assume
I: (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @ [*inv p x*], {}) ∉ *failures P*
have *?A*
proof (*rule-tac x* = *insert x X* **in** *exI*, *rule-tac x* = *Y* **in** *exI*,
rule-tac x = *Z* **in** *exI*, (*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+)
show *insert x* (*X* ∪ *Y* ∪ *Z*) = *insert x X* ∪ *Y* ∪ *Z*
by *simp*
next
show *insert x X* ⊆ *range p*
using *A* **and** *G* **by** *simp*
next
show *Y* ⊆ *range q*
using *B* **.**
next
show *Z* ⊆ − *range p*
using *C* **.**
next
show *Z* ⊆ − *range q*
using *D* **.**
next
have
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @ [*inv p x*], {})
∈ *failures P* ∨
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *insert* (*inv p x*) (*inv p* ‘ *X*))
∈ *failures P*
using *E* **by** (*rule process-rule-4*)
thus (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ‘ *insert x X*) ∈ *failures P*
using *I* **by** *simp*
next

show (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *inv q* ‘ *Y*) ∈ *failures Q*
using *F* **.**
qed
thus *?thesis*
by *simp*
next
assume
I: (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] @ [*inv q x*], {}) ∉ *failures Q*
have *?A*
proof (*rule-tac x* = *X* **in** *exI*, *rule-tac x* = *insert x Y* **in** *exI*,
rule-tac x = *Z* **in** *exI*, (*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+)
show *insert x* (*X* ∪ *Y* ∪ *Z*) = *X* ∪ *insert x Y* ∪ *Z*
by *simp*
next
show *X* ⊆ *range p*
using *A* **.**
next
show *insert x Y* ⊆ *range q*
using *B* **and** *H* **by** *simp*
next
show *Z* ⊆ − *range p*
using *C* **.**
next
show *Z* ⊆ − *range q*
using *D* **.**
next
show (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ‘ *X*) ∈ *failures P*
using *E* **.**
next
have
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] @ [*inv q x*], {})
∈ *failures Q* ∨
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *insert* (*inv q x*) (*inv q* ‘ *Y*))
∈ *failures Q*
using *F* **by** (*rule process-rule-4*)
thus (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *inv q* ‘ *insert x Y*) ∈ *failures Q*
using *I* **by** *simp*
qed
thus *?thesis*
by *simp*
qed
next
assume *G*: *x* ∈ *range p*
show
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @ [*inv p x*], {}) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], {}) ∈ *failures Q* ∨
xs @ [*x*] ∈ *con-comp-divergences P Q p q* ∨
?A ∨
xs ∈ *con-comp-divergences P Q p q*

```
  proof (cases (map (inv p) [x←xs. x ∈ range p] @ [inv p x], {})
   ∈ failures P)
    case True
    moreover have {} ⊆ inv q ‘ Y ..
    with F have (map (inv q) [x←xs. x ∈ range q], {}) ∈ failures Q
     by (rule process-rule-3)
    ultimately show ?thesis
     by simp
  next
    case False
    have ?A
    proof (rule-tac x = insert x X in exI, rule-tac x = Y in exI,
     rule-tac x = Z in exI, (subst conj-assoc [symmetric])+, (rule conjI)+)
      show insert x (X ∪ Y ∪ Z) = insert x X ∪ Y ∪ Z
       by simp
    next
      show insert x X ⊆ range p
       using A and G by simp
    next
      show Y ⊆ range q
       using B .
    next
      show Z ⊆ − range p
       using C .
    next
      show Z ⊆ − range q
       using D .
    next
      have
       (map (inv p) [x←xs. x ∈ range p] @ [inv p x], {})
          ∈ failures P ∨
        (map (inv p) [x←xs. x ∈ range p], insert (inv p x) (inv p ‘ X))
          ∈ failures P
       using E by (rule process-rule-4)
      thus (map (inv p) [x←xs. x ∈ range p], inv p ‘ insert x X) ∈ failures P
       using False by simp
    next
      show (map (inv q) [x←xs. x ∈ range q], inv q ‘ Y) ∈ failures Q
       using F .
    qed
    thus ?thesis
     by simp
  qed
next
  assume G: x ∈ range q
  show
   (map (inv p) [x←xs. x ∈ range p], {}) ∈ failures P ∧
      (map (inv q) [x←xs. x ∈ range q] @ [inv q x], {}) ∈ failures Q ∨
    xs @ [x] ∈ con-comp-divergences P Q p q ∨
```

?A ∨
xs ∈ *con-comp-divergences P Q p q*
proof (*cases* (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] @ [*inv q x*], {})
∈ *failures Q*)
case *True*
moreover have {} ⊆ *inv p* ' *X* **..**
with *E* **have** (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], {}) ∈ *failures P*
by (*rule process-rule-3*)
ultimately show *?thesis*
by *simp*
next
case *False*
have *?A*
proof (*rule-tac x* = *X* **in** *exI*, *rule-tac x* = *insert x Y* **in** *exI*,
rule-tac x = *Z* **in** *exI*, (*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+)
show *insert x* (*X* ∪ *Y* ∪ *Z*) = *X* ∪ *insert x Y* ∪ *Z*
by *simp*
next
show *X* ⊆ *range p*
using *A* **.**
next
show *insert x Y* ⊆ *range q*
using *B* **and** *G* **by** *simp*
next
show *Z* ⊆ − *range p*
using *C* **.**
next
show *Z* ⊆ − *range q*
using *D* **.**
next
show (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ' *X*) ∈ *failures P*
using *E* **.**
next
have
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*] @ [*inv q x*], {})
∈ *failures Q* ∨
(*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *insert* (*inv q x*) (*inv q* ' *Y*))
∈ *failures Q*
using *F* **by** (*rule process-rule-4*)
thus (*map* (*inv q*) [*x*←*xs*. *x* ∈ *range q*], *inv q* ' *insert x Y*) ∈ *failures Q*
using *False* **by** *simp*
qed
thus *?thesis*
by *simp*
qed
next
assume
G: *x* ∉ *range p* **and**
H: *x* ∉ *range q*

```
    have ?A
    proof (rule-tac x = X in exI, rule-tac x = Y in exI,
     rule-tac x = insert x Z in exI, (subst conj-assoc [symmetric])+,
     (rule conjI)+)
      show insert x (X ∪ Y ∪ Z) = X ∪ Y ∪ insert x Z
        by simp
    next
      show X ⊆ range p
        using A .
    next
      show Y ⊆ range q
        using B .
    next
      show insert x Z ⊆ − range p
        using C and G by simp
    next
      show insert x Z ⊆ − range q
        using D and H by simp
    next
      show (map (inv p) [x←xs. x ∈ range p], inv p ‘ X) ∈ failures P
        using E .
    next
      show (map (inv q) [x←xs. x ∈ range q], inv q ‘ Y) ∈ failures Q
        using F .
    qed
    thus
     xs @ [x] ∈ con-comp-divergences P Q p q ∨
      ?A ∨
      xs ∈ con-comp-divergences P Q p q
     by simp
  qed
qed

lemma con-comp-prop-5:
 xs ∈ con-comp-divergences P Q p q ⟹
    xs @ [x] ∈ con-comp-divergences P Q p q
proof (simp add: con-comp-divergences-def, erule exE, (erule conjE)+, erule exE)
  fix xs' ys'
  assume
    A: set xs' ⊆ range p ∪ range q and
    B: map (inv p) [x←xs'. x ∈ range p] ∈ divergences P and
    C: map (inv q) [x←xs'. x ∈ range q] ∈ divergences Q and
    D: xs = xs' @ ys'
  show ∃ xs'.
    (∃ ys'. xs @ [x] = xs' @ ys') ∧
    set xs' ⊆ range p ∪ range q ∧
    map (inv p) [x←xs'. x ∈ range p] ∈ divergences P ∧
    map (inv q) [x←xs'. x ∈ range q] ∈ divergences Q
  proof (rule-tac x = xs' in exI, simp-all add: A B C)
```

qed (*rule-tac x* = *ys'* @ [*x*] **in** *exI*, *simp add*: *D*)
qed

lemma *con-comp-prop-6*:
xs ∈ *con-comp-divergences P Q p q* ⟹
(*xs*, *X*) ∈ *con-comp-failures P Q p q*
by (*simp add*: *con-comp-failures-def*)

lemma *con-comp-rep*:
Rep-process (*P* ∥ *Q* <*p*, *q*>) =
(*con-comp-failures P Q p q*, *con-comp-divergences P Q p q*)
(**is** - = *?X*)
proof (*subst con-comp-def*, *rule Abs-process-inverse*, *simp add*: *process-set-def*,
(*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+)
show *process-prop-1 ?X*
proof (*simp add*: *process-prop-1-def*)
qed (*rule con-comp-prop-1*)
next
show *process-prop-2 ?X*
proof (*simp add*: *process-prop-2-def del*: *all-simps*, (*rule allI*)+, *rule impI*)
qed (*rule con-comp-prop-2*)
next
show *process-prop-3 ?X*
proof (*simp add*: *process-prop-3-def del*: *all-simps*, (*rule allI*)+, *rule impI*,
erule conjE)
qed (*rule con-comp-prop-3*)
next
show *process-prop-4 ?X*
proof (*simp add*: *process-prop-4-def*, (*rule allI*)+, *rule impI*)
qed (*rule con-comp-prop-4*)
next
show *process-prop-5 ?X*
proof (*simp add*: *process-prop-5-def*, *rule allI*, *rule impI*, *rule allI*)
qed (*rule con-comp-prop-5*)
next
show *process-prop-6 ?X*
proof (*simp add*: *process-prop-6-def*, *rule allI*, *rule impI*, *rule allI*)
qed (*rule con-comp-prop-6*)
qed

Here below, the previous result is applied to derive useful expressions for the outputs of the functions returning the elements of a process, as defined in [6] and [9], when acting on the concurrent composition of a pair of processes.

lemma *con-comp-failures*:
failures (*P* ∥ *Q* <*p*, *q*>) = *con-comp-failures P Q p q*
by (*simp add*: *failures-def con-comp-rep*)

lemma *con-comp-divergences*:
divergences (*P* ‖ *Q* <*p*, *q*>) = *con-comp-divergences P Q p q*
by (*simp add*: *divergences-def con-comp-rep*)

lemma *con-comp-futures*:
futures (*P* ‖ *Q* <*p*, *q*>) *xs* =
  {(*ys*, *Y*). (*xs* @ *ys*, *Y*) ∈ *con-comp-failures P Q p q*}
by (*simp add*: *futures-def con-comp-failures*)

lemma *con-comp-traces*:
traces (*P* ‖ *Q* <*p*, *q*>) = *Domain* (*con-comp-failures P Q p q*)
by (*simp add*: *traces-def con-comp-failures*)

lemma *con-comp-refusals*:
refusals (*P* ‖ *Q* <*p*, *q*>) *xs* ≡ *con-comp-failures P Q p q* ‘‘ {*xs*}
by (*simp add*: *refusals-def con-comp-failures*)

lemma *con-comp-next-events*:
next-events (*P* ‖ *Q* <*p*, *q*>) *xs* =
  {*x*. *xs* @ [*x*] ∈ *Domain* (*con-comp-failures P Q p q*)}
by (*simp add*: *next-events-def con-comp-traces*)

In what follows, three lemmas are proven. The first one, whose proof makes use of the axiom of choice, establishes an additional property required for the above definition of concurrent composition to be correct, namely that for any two processes whose refusals are closed under set union, their concurrent composition still be such, which is what is expected for any process of practical significance (cf. [9]). The other two lemmas are auxiliary properties of concurrent composition used in the proof of the target security conservation theorem.

lemma *con-comp-ref-union-closed*:
 assumes
  A: *ref-union-closed P* **and**
  B: *ref-union-closed Q*
 shows *ref-union-closed* (*P* ‖ *Q* <*p*, *q*>)
proof (*simp add*: *ref-union-closed-def con-comp-failures con-comp-failures-def con-comp-divergences-def del*: *SUP-identity-eq cong*: *SUP-cong-simp*, (*rule allI*)+, (*rule impI*)+,
erule exE, *rule disjI1*)
 fix *xs A X*
 assume ∀ *X* ∈ *A*. ∃ *R S T*.
  X = *R* ∪ *S* ∪ *T* ∧
  set xs ⊆ *range p* ∪ *range q* ∧
  R ⊆ *range p* ∧
  S ⊆ *range q* ∧
  T ⊆ − *range p* ∧

$T \subseteq - \mathit{range}\ q\ \wedge$
$(\mathit{map}\ (\mathit{inv}\ p)\ [x{\leftarrow}xs.\ x \in \mathit{range}\ p],\ \mathit{inv}\ p\ `\ R) \in \mathit{failures}\ P\ \wedge$
$(\mathit{map}\ (\mathit{inv}\ q)\ [x{\leftarrow}xs.\ x \in \mathit{range}\ q],\ \mathit{inv}\ q\ `\ S) \in \mathit{failures}\ Q$
(**is** $\forall X \in A.\ \exists R\ S\ T.\ ?F\ X\ R\ S\ T$)
hence $\exists r.\ \forall X \in A.\ \exists S\ T.\ ?F\ X\ (r\ X)\ S\ T$
by (*rule bchoice*)
then obtain r **where** $\forall X \in A.\ \exists S\ T.\ ?F\ X\ (r\ X)\ S\ T$ **..**
hence $\exists s.\ \forall X \in A.\ \exists T.\ ?F\ X\ (r\ X)\ (s\ X)\ T$
by (*rule bchoice*)
then obtain s **where** $\forall X \in A.\ \exists T.\ ?F\ X\ (r\ X)\ (s\ X)\ T$ **..**
hence $\exists t.\ \forall X \in A.\ ?F\ X\ (r\ X)\ (s\ X)\ (t\ X)$
by (*rule bchoice*)
then obtain t **where** C: $\forall X \in A.\ ?F\ X\ (r\ X)\ (s\ X)\ (t\ X)$ **..**
assume D: $X \in A$
show $\exists R\ S\ T.\ ?F\ (\bigcup X \in A.\ X)\ R\ S\ T$
proof (*rule-tac* $x = \bigcup X \in A.\ r\ X$ **in** *exI*, *rule-tac* $x = \bigcup X \in A.\ s\ X$ **in** *exI*,
rule-tac $x = \bigcup X \in A.\ t\ X$ **in** *exI*, (*subst conj-assoc* [*symmetric*])+,
(*rule conjI*)+)
show $(\bigcup X \in A.\ X) = (\bigcup X \in A.\ r\ X) \cup (\bigcup X \in A.\ s\ X) \cup (\bigcup X \in A.\ t\ X)$
proof (*simp add*: *set-eq-iff*, *rule allI*, *rule iffI*, *erule-tac* [*2*] *disjE*,
erule-tac [*3*] *disjE*, *erule-tac* [!] *bexE*)
fix $x\ X$
have $\forall X \in A.\ X = r\ X \cup s\ X \cup t\ X$
using C **by** *simp*
moreover assume E: $X \in A$
ultimately have $X = r\ X \cup s\ X \cup t\ X$ **..**
moreover assume $x \in X$
ultimately have $x \in r\ X \vee x \in s\ X \vee x \in t\ X$
by *blast*
hence $\exists X \in A.\ x \in r\ X \vee x \in s\ X \vee x \in t\ X$
using E **..**
thus $(\exists X \in A.\ x \in r\ X) \vee (\exists X \in A.\ x \in s\ X) \vee (\exists X \in A.\ x \in t\ X)$
by *blast*
next
fix $x\ X$
have $\forall X \in A.\ X = r\ X \cup s\ X \cup t\ X$
using C **by** *simp*
moreover assume E: $X \in A$
ultimately have $X = r\ X \cup s\ X \cup t\ X$ **..**
moreover assume $x \in r\ X$
ultimately have $x \in X$
by *blast*
thus $\exists X \in A.\ x \in X$
using E **..**
next
fix $x\ X$
have $\forall X \in A.\ X = r\ X \cup s\ X \cup t\ X$
using C **by** *simp*
moreover assume E: $X \in A$

ultimately have $X = r\ X \cup s\ X \cup t\ X$ **..**
moreover assume $x \in s\ X$
ultimately have $x \in X$
by *blast*
thus $\exists\, X \in A.\ x \in X$
using E **..**
next
fix $x\ X$
have $\forall\, X \in A.\ X = r\ X \cup s\ X \cup t\ X$
using C **by** *simp*
moreover assume E: $X \in A$
ultimately have $X = r\ X \cup s\ X \cup t\ X$ **..**
moreover assume $x \in t\ X$
ultimately have $x \in X$
by *blast*
thus $\exists\, X \in A.\ x \in X$
using E **..**
qed
next
have $\forall\, X \in A.\ \mathit{set}\ \mathit{xs} \subseteq \mathit{range}\ p \cup \mathit{range}\ q$
using C **by** *simp*
thus $\mathit{set}\ \mathit{xs} \subseteq \mathit{range}\ p \cup \mathit{range}\ q$
using D **..**
next
show $(\bigcup X \in A.\ r\ X) \subseteq \mathit{range}\ p$
proof (*rule subsetI*, *erule UN-E*)
fix $x\ X$
have $\forall\, X \in A.\ r\ X \subseteq \mathit{range}\ p$
using C **by** *simp*
moreover assume $X \in A$
ultimately have $r\ X \subseteq \mathit{range}\ p$ **..**
moreover assume $x \in r\ X$
ultimately show $x \in \mathit{range}\ p$ **..**
qed
next
show $(\bigcup X \in A.\ s\ X) \subseteq \mathit{range}\ q$
proof (*rule subsetI*, *erule UN-E*)
fix $x\ X$
have $\forall\, X \in A.\ s\ X \subseteq \mathit{range}\ q$
using C **by** *simp*
moreover assume $X \in A$
ultimately have $s\ X \subseteq \mathit{range}\ q$ **..**
moreover assume $x \in s\ X$
ultimately show $x \in \mathit{range}\ q$ **..**
qed
next
show $(\bigcup X \in A.\ t\ X) \subseteq -\ \mathit{range}\ p$
proof (*rule subsetI*, *erule UN-E*)
fix $x\ X$

```
    have ∀ X ∈ A. t X ⊆ − range p
     using C by simp
    moreover assume X ∈ A
    ultimately have t X ⊆ − range p ..
    moreover assume x ∈ t X
    ultimately show x ∈ − range p ..
  qed
next
  show (⋃ X ∈ A. t X) ⊆ − range q
  proof (rule subsetI, erule UN-E)
    fix x X
    have ∀ X ∈ A. t X ⊆ − range q
     using C by simp
    moreover assume X ∈ A
    ultimately have t X ⊆ − range q ..
    moreover assume x ∈ t X
    ultimately show x ∈ − range q ..
  qed
next
  let ?A' = {inv p ` X | X. X ∈ r ` A}
  have
   (∃ X. X ∈ ?A') ⟶
    (∀ X ∈ ?A'. (map (inv p) [x←xs. x ∈ range p], X) ∈ failures P) ⟶
      (map (inv p) [x←xs. x ∈ range p], ⋃ X ∈ ?A'. X) ∈ failures P
   using A by (simp add: ref-union-closed-def)
  moreover have ∃ X. X ∈ ?A'
   using D by blast
  ultimately have
   (∀ X ∈ ?A'. (map (inv p) [x←xs. x ∈ range p], X) ∈ failures P) ⟶
      (map (inv p) [x←xs. x ∈ range p], ⋃ X ∈ ?A'. X) ∈ failures P ..
  moreover have
   ∀ X ∈ ?A'. (map (inv p) [x←xs. x ∈ range p], X) ∈ failures P
  proof (rule ballI, simp, erule exE, erule conjE)
    fix R R'
    assume R ∈ r ` A
    hence ∃ X ∈ A. R = r X
     by (simp add: image-iff)
    then obtain X where E: X ∈ A and F: R = r X ..
    have ∀ X ∈ A. (map (inv p) [x←xs. x ∈ range p], inv p ` r X) ∈ failures P
     using C by simp
    hence (map (inv p) [x←xs. x ∈ range p], inv p ` r X) ∈ failures P
     using E ..
    moreover assume R' = inv p ` R
    ultimately show (map (inv p) [x←xs. x ∈ range p], R') ∈ failures P
     using F by simp
  qed
  ultimately have (map (inv p) [x←xs. x ∈ range p], ⋃ X ∈ ?A'. X)
   ∈ failures P ..
  moreover have (⋃ X ∈ ?A'. X) = inv p ` (⋃ X ∈ A. r X)
```

```
  proof (subst set-eq-iff, simp, rule allI, rule iffI, (erule exE, erule conjE)+)
    fix a R R'
    assume R ∈ r ` A
    hence ∃ X ∈ A. R = r X
     by (simp add: image-iff)
    then obtain X where E: X ∈ A and F: R = r X ..
    assume a ∈ R' and R' = inv p ` R
    hence a ∈ inv p ` r X
     using F by simp
    hence ∃ x ∈ r X. a = inv p x
     by (simp add: image-iff)
    then obtain x where G: x ∈ r X and H: a = inv p x ..
    have x ∈ (⋃ X ∈ A. r X)
     using E and G by (rule UN-I)
    with H have ∃ x ∈ (⋃ X ∈ A. r X). a = inv p x ..
    thus a ∈ inv p ` (⋃ X ∈ A. r X)
     by (simp add: image-iff)
  next
    fix a
    assume a ∈ inv p ` (⋃ X ∈ A. r X)
    hence ∃ x ∈ (⋃ X ∈ A. r X). a = inv p x
     by (simp add: image-iff)
    then obtain x where E: x ∈ (⋃ X ∈ A. r X) and F: a = inv p x ..
    obtain X where G: X ∈ A and H: x ∈ r X using E ..
    show ∃ R'. (∃ R. R' = inv p ` R ∧ R ∈ r ` A) ∧ a ∈ R'
    proof (rule-tac x = inv p ` r X in exI, rule conjI,
     rule-tac x = r X in exI)
    qed (rule-tac [2] image-eqI, simp add: G, simp add: F, simp add: H)
  qed
  ultimately show (map (inv p) [x←xs. x ∈ range p], inv p ` (⋃ X ∈ A. r X))
    ∈ failures P
   by simp
next
  let ?A' = {inv q ` X | X. X ∈ s ` A}
  have
   (∃ X. X ∈ ?A') ⟶
    (∀ X ∈ ?A'. (map (inv q) [x←xs. x ∈ range q], X) ∈ failures Q) ⟶
      (map (inv q) [x←xs. x ∈ range q], ⋃ X ∈ ?A'. X) ∈ failures Q
   using B by (simp add: ref-union-closed-def)
  moreover have ∃ X. X ∈ ?A'
   using D by blast
  ultimately have
   (∀ X ∈ ?A'. (map (inv q) [x←xs. x ∈ range q], X) ∈ failures Q) ⟶
      (map (inv q) [x←xs. x ∈ range q], ⋃ X ∈ ?A'. X) ∈ failures Q ..
  moreover have
   ∀ X ∈ ?A'. (map (inv q) [x←xs. x ∈ range q], X) ∈ failures Q
  proof (rule ballI, simp, erule exE, erule conjE)
    fix S S'
    assume S ∈ s ` A
```

hence $\exists X \in A.\ S = s\ X$
by (*simp add*: *image-iff*)
then obtain X **where** E: $X \in A$ **and** F: $S = s\ X$ **..**
have $\forall X \in A.$ (*map* (*inv q*) [x←*xs*. $x \in$ *range q*], *inv q* ‘ $s\ X$) $\in$ *failures Q*
using C **by** *simp*
hence (*map* (*inv q*) [x←*xs*. $x \in$ *range q*], *inv q* ‘ $s\ X$) $\in$ *failures Q*
using E **..**
moreover assume $S' =$ *inv q* ‘ S
ultimately show (*map* (*inv q*) [x←*xs*. $x \in$ *range q*], S') $\in$ *failures Q*
using F **by** *simp*
qed
ultimately have (*map* (*inv q*) [x←*xs*. $x \in$ *range q*], $\bigcup X \in$ *?A'*. X)
$\in$ *failures Q* **..**
moreover have ($\bigcup X \in$ *?A'*. X) = *inv q* ‘ ($\bigcup X \in A.\ s\ X$)
proof (*subst set-eq-iff*, *simp*, *rule allI*, *rule iffI*, (*erule exE*, *erule conjE*)+)
fix $b\ S\ S'$
assume $S \in s$ ‘ A
hence $\exists X \in A.\ S = s\ X$
by (*simp add*: *image-iff*)
then obtain X **where** E: $X \in A$ **and** F: $S = s\ X$ **..**
assume $b \in S'$ **and** $S' =$ *inv q* ‘ S
hence $b \in$ *inv q* ‘ $s\ X$
using F **by** *simp*
hence $\exists x \in s\ X.\ b =$ *inv q x*
by (*simp add*: *image-iff*)
then obtain x **where** G: $x \in s\ X$ **and** H: $b =$ *inv q x* **..**
have $x \in (\bigcup X \in A.\ s\ X)$
using E **and** G **by** (*rule UN-I*)
with H **have** $\exists x \in (\bigcup X \in A.\ s\ X).\ b =$ *inv q x* **..**
thus $b \in$ *inv q* ‘ ($\bigcup X \in A.\ s\ X$)
by (*simp add*: *image-iff*)
next
fix b
assume $b \in$ *inv q* ‘ ($\bigcup X \in A.\ s\ X$)
hence $\exists x \in (\bigcup X \in A.\ s\ X).\ b =$ *inv q x*
by (*simp add*: *image-iff*)
then obtain x **where** E: $x \in (\bigcup X \in A.\ s\ X)$ **and** F: $b =$ *inv q x* **..**
obtain X **where** G: $X \in A$ **and** H: $x \in s\ X$ **using** E **..**
show $\exists S'.$ ($\exists S.\ S' =$ *inv q* ‘ $S \wedge S \in s$ ‘ A) $\wedge\ b \in S'$
proof (*rule-tac x* = *inv q* ‘ $s\ X$ **in** *exI*, *rule conjI*,
rule-tac x = $s\ X$ **in** *exI*)
qed (*rule-tac* [*2*] *image-eqI*, *simp add*: G, *simp add*: F, *simp add*: H)
qed
ultimately show (*map* (*inv q*) [x←*xs*. $x \in$ *range q*], *inv q* ‘ ($\bigcup X \in A.\ s\ X$))
$\in$ *failures Q*
by *simp*
qed
qed

lemma *con-comp-failures-traces*:
$(xs, X) \in$ *con-comp-failures P Q p q* $\Longrightarrow$
  map (*inv p*) $[x \leftarrow xs.\ x \in$ *range p*$] \in$ *traces P* $\wedge$
  map (*inv q*) $[x \leftarrow xs.\ x \in$ *range q*$] \in$ *traces Q*
proof (*simp add*: *con-comp-failures-def con-comp-divergences-def*, *erule disjE*,
(*erule exE*)+, (*erule conjE*)+, *erule-tac* [*2*] *exE*, (*erule-tac* [*2*] *conjE*)+,
erule-tac [*2*] *exE*)
 fix *X Y*
 assume (*map* (*inv p*) $[x \leftarrow xs.\ x \in$ *range p*], *inv p* ' $X) \in$ *failures P*
 hence *map* (*inv p*) $[x \leftarrow xs.\ x \in$ *range p*$] \in$ *traces P*
  by (*rule failures-traces*)
 moreover assume (*map* (*inv q*) $[x \leftarrow xs.\ x \in$ *range q*], *inv q* ' $Y) \in$ *failures Q*
 hence *map* (*inv q*) $[x \leftarrow xs.\ x \in$ *range q*$] \in$ *traces Q*
  by (*rule failures-traces*)
 ultimately show *?thesis* **..**
next
 fix *vs ws*
 assume *A*: $xs = vs$ @ ws
 assume *map* (*inv p*) $[x \leftarrow vs.\ x \in$ *range p*$] \in$ *divergences P*
 hence *map* (*inv p*) $[x \leftarrow vs.\ x \in$ *range p*] @ *map* (*inv p*) $[x \leftarrow ws.\ x \in$ *range p*]
  $\in$ *divergences P*
  by (*rule process-rule-5-general*)
 hence *map* (*inv p*) $[x \leftarrow xs.\ x \in$ *range p*$] \in$ *divergences P*
  using *A* **by** *simp*
 hence (*map* (*inv p*) $[x \leftarrow xs.\ x \in$ *range p*], $\{\}) \in$ *failures P*
  by (*rule process-rule-6*)
 hence *map* (*inv p*) $[x \leftarrow xs.\ x \in$ *range p*$] \in$ *traces P*
  by (*rule failures-traces*)
 moreover assume *map* (*inv q*) $[x \leftarrow vs.\ x \in$ *range q*$] \in$ *divergences Q*
 hence *map* (*inv q*) $[x \leftarrow vs.\ x \in$ *range q*] @ *map* (*inv q*) $[x \leftarrow ws.\ x \in$ *range q*]
  $\in$ *divergences Q*
  by (*rule process-rule-5-general*)
 hence *map* (*inv q*) $[x \leftarrow xs.\ x \in$ *range q*$] \in$ *divergences Q*
  using *A* **by** *simp*
 hence (*map* (*inv q*) $[x \leftarrow xs.\ x \in$ *range q*], $\{\}) \in$ *failures Q*
  by (*rule process-rule-6*)
 hence *map* (*inv q*) $[x \leftarrow xs.\ x \in$ *range q*$] \in$ *traces Q*
  by (*rule failures-traces*)
 ultimately show *?thesis* **..**
qed

lemma *con-comp-failures-divergences*:
$(xs$ @ $y \# ys, Y) \in$ *con-comp-failures P Q p q* $\Longrightarrow$
 $y \notin$ *range p* $\Longrightarrow$
 $y \notin$ *range q* $\Longrightarrow$
  $\exists xs'.$
   $(\exists ys'.\ xs$ @ $zs = xs'$ @ $ys') \wedge$
   set $xs' \subseteq$ *range p* $\cup$ *range q* $\wedge$
   map (*inv p*) $[x \leftarrow xs'.\ x \in$ *range p*$] \in$ *divergences P* $\wedge$

```
    map (inv q) [x←xs'. x ∈ range q] ∈ divergences Q
proof (simp add: con-comp-failures-def con-comp-divergences-def,
 erule exE, (erule conjE)+, erule exE)
  fix xs' ys'
  assume
    A: y ∉ range p and
    B: y ∉ range q and
    C: set xs' ⊆ range p ∪ range q and
    D: map (inv p) [x←xs'. x ∈ range p] ∈ divergences P and
    E: map (inv q) [x←xs'. x ∈ range q] ∈ divergences Q and
    F: xs @ y # ys = xs' @ ys'
  have length xs' ≤ length xs
  proof (rule ccontr)
    assume ¬ length xs' ≤ length xs
    moreover have take (length xs') (xs @ [y] @ ys) =
      take (length xs') (xs @ [y]) @ take (length xs' − Suc (length xs)) ys
      (is - = - @ ?vs)
     by (simp only: take-append, simp)
    ultimately have take (length xs') (xs @ y # ys) = xs @ y # ?vs
     by simp
    moreover have take (length xs') (xs @ y # ys) =
      take (length xs') (xs' @ ys')
     using F by simp
    ultimately have xs' = xs @ y # ?vs
     by simp
    hence set (xs @ y # ?vs) ⊆ range p ∪ range q
     using C by simp
    hence y ∈ range p ∪ range q
     by simp
    thus False
     using A and B by simp
  qed
  moreover have xs @ zs =
    take (length xs') (xs @ zs) @ drop (length xs') (xs @ zs)
    (is - = - @ ?vs)
   by (simp only: append-take-drop-id)
  ultimately have xs @ zs = take (length xs') (xs @ y # ys) @ ?vs
   by simp
  moreover have take (length xs') (xs @ y # ys) =
    take (length xs') (xs' @ ys')
   using F by simp
  ultimately have G: xs @ zs = xs' @ ?vs
   by (simp del: take-append, simp)
  show ?thesis
  proof (rule-tac x = xs' in exI, rule conjI, rule-tac x = ?vs in exI)
  qed (subst G, simp-all add: C D E)
qed
```

In order to prove that CSP noninterference security is conserved under concurrent composition, the first issue to be solved is to identify the noninterference policy I' and the event-domain map D' with respect to which the output process is secure.

If the events of the input processes corresponding to those of the output process contained in *range p* ∩ *range q* were mapped by the respective event-domain maps D, E into distinct security domains, there would be no criterion for determining the domains of the aforesaid events of the output process, due to the equivalence of the input processes ensuing from the commutative property of concurrent composition. Therefore, D and E must map the events of the input processes into security domains of the same type $'d$, and for each x in *range p* ∩ *range q*, D and E must map the events of the input processes corresponding to x into the same domain. This requirement is formalized here below by means of predicate *consistent-maps*.

Similarly, if distinct noninterference policies applied to the input processes, there would exist some ordered pair of security domains included in one of the policies, but not in the other one. Thus, again, there would be no criterion for determining the inclusion of such a pair of domains in the policy I' applying to the output process. As a result, the input processes are required to enforce the same noninterference policy I, so that for any two domains d, e of type $'d$, the ordered pair comprised of the corresponding security domains for the output process will be included in I' just in case $(d, e) \in I$.

However, in case − (*range p* ∪ *range q*) ≠ {}, the event-domain map D' for the output process must assign a security domain to the fake events in − (*range p* ∪ *range q*) as well. Since such events lack any meaning, they may all be mapped to the same security domain, distinct from the domains of the meaningful events in *range p* ∪ *range q*. A simple way to do this is to identify the type of the security domains for the output process with $'d$ *option*. Then, for any meaningful event x, D' will assign x to domain *Some d*, where d is the domain of the events of the input processes mapped to x, whereas D' y = *None* for any fake event y. Such an event-domain map, denoted using notation *con-comp-map D E p q*, is defined here below.

Therefore, for any two security domains *Some d*, *Some e* for the output process, the above considerations about policy I' entail that (*Some d*, *Some e*) ∈ I' just in case $(d, e) \in I$. Furthermore, since fake events may only occur in divergent traces, which are extensions of divergences of the input processes comprised of meaningful events, I' must allow the security domain *None* of fake events to be affected by any meaningful domain matching pattern *Some -*. Such a noninterference policy, denoted using notation *con-comp-pol I*, is defined here below. Observe that *con-comp-pol I* keeps being reflexive or transitive if I is.

definition *con-comp-pol* ::
$('d \times 'd)$ *set* $\Rightarrow$ $('d$ *option* $\times$ $'d$ *option*$)$ *set* **where**
con-comp-pol I $\equiv$
$\{(Some\ d,\ Some\ e) \mid d\ e.\ (d,\ e) \in I\} \cup \{(u,\ v).\ v = None\}$

function *con-comp-map* ::
$('a \Rightarrow 'd) \Rightarrow ('b \Rightarrow 'd) \Rightarrow ('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'c \Rightarrow 'd$ *option* **where**
$x \in range\ p \Longrightarrow$
con-comp-map $D\ E\ p\ q\ x = Some\ (D\ (inv\ p\ x))$ |
$x \notin range\ p \Longrightarrow x \in range\ q \Longrightarrow$
con-comp-map $D\ E\ p\ q\ x = Some\ (E\ (inv\ q\ x))$ |
$x \notin range\ p \Longrightarrow x \notin range\ q \Longrightarrow$
con-comp-map $D\ E\ p\ q\ x = None$
by (*atomize-elim*, *simp-all add*: *split-paired-all*, *blast*)
termination by *lexicographic-order*

definition *consistent-maps* ::
$('a \Rightarrow 'd) \Rightarrow ('b \Rightarrow 'd) \Rightarrow ('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c) \Rightarrow bool$ **where**
consistent-maps $D\ E\ p\ q \equiv$
$\forall x \in range\ p \cap range\ q.\ D\ (inv\ p\ x) = E\ (inv\ q\ x)$

1.3 Auxiliary intransitive purge functions

Let I be a noninterference policy, D an event-domain map, U a domain set, and $xs = x\ \#\ xs'$ an event list. Suppose to take event x just in case it satisfies predicate P, to append xs' to the resulting list (matching either $[x]$ or $[]$), and then to compute the intransitive purge of the resulting list with domain set U. If recursion with respect to the input list is added, replacing xs' with the list produced by the same algorithm using xs' as input list and *sinks-aux* $I\ D\ U\ [x]$ as domain set, the final result matches that obtained by applying filter P to the intransitive purge of xs with domain set U. In fact, in each recursive step, the processed item of the input list is retained in the output list just in case it passes filter P and may be affected neither by the domains in U, nor by the domains of the previous items affected by some domain in U.

Here below is the formal definition of such purge function, named *ipurge-tr-aux-foldr* as its action resembles that of function *foldr*.

primrec *ipurge-tr-aux-foldr* ::
$('d \times 'd)$ *set* $\Rightarrow ('a \Rightarrow 'd) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'd$ *set* $\Rightarrow 'a$ *list* $\Rightarrow 'a$ *list*
where
ipurge-tr-aux-foldr $I\ D\ P\ U\ [] = []$ |
ipurge-tr-aux-foldr $I\ D\ P\ U\ (x\ \#\ xs) =$ *ipurge-tr-aux* $I\ D\ U$
$((if\ P\ x\ then\ [x]\ else\ [])$ @
ipurge-tr-aux-foldr $I\ D\ P$ (*sinks-aux* $I\ D\ U\ [x]$) xs)

Likewise, given I, D, U, $xs = x \# xs'$, and an event set X, suppose to take x just in case it satisfies predicate P, to append *ipurge-tr-aux-foldr I D P* (*sinks-aux I D U* $[x]$) xs' to the resulting list (matching either $[x]$ or $[]$), and then to compute the intransitive purge of X using the resulting list as input list and U as domain set. If recursion with respect to the input list is added, replacing X with the set produced by the same algorithm using xs' as input list, X as input set, and *sinks-aux I D U* $[x]$ as domain set, the final result matches the intransitive purge of X with input list xs and domain set U. In fact, each recursive step is such as to remove from X any event that may be affected either by the domains in U, or by the domains of the items of xs preceding the processed one which are affected by some domain in U.

From the above considerations on function *ipurge-tr-aux-foldr*, it follows that the presence of list *ipurge-tr-aux-foldr I D P* (*sinks-aux I D U* $[x]$) xs' has no impact on the final result, because none of its items may be affected by the domains in U.

Here below is the formal definition of such purge function, named *ipurge-ref-aux-foldr*, which at first glance just seems a uselessly complicate and inefficient way to compute the intransitive purge of an event set.

primrec *ipurge-ref-aux-foldr* ::
$('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'd\ set \Rightarrow 'a\ list \Rightarrow 'a\ set \Rightarrow 'a\ set$
where
ipurge-ref-aux-foldr I D P U $[]$ X = *ipurge-ref-aux I D U* $[]$ X |
ipurge-ref-aux-foldr I D P U ($x \# xs$) X = *ipurge-ref-aux I D U*
  ((*if P x then* $[x]$ *else* $[]$) @
    ipurge-tr-aux-foldr I D P (*sinks-aux I D U* $[x]$) *xs*)
  (*ipurge-ref-aux-foldr I D P* (*sinks-aux I D U* $[x]$) *xs X*)

The reason for the introduction of such intransitive purge functions is that the recursive equations contained in their definitions, along with lemma *ipurge-tr-ref-aux-failures-general*, enable to prove by induction on list ys, assuming that process P be secure in addition to further, minor premises, the following implication:

(*map* (*inv p*) (*filter* ($\lambda x.\ x \in$ *range p*) (*xs* @ *ys*)), *inv p* ‘ Y) $\in$ *failures P* $\longrightarrow$
(*map* (*inv p*) (*filter* ($\lambda x.\ x \in$ *range p*) *xs*) @ *map* (*inv p*) (*ipurge-tr-aux-foldr* (*con-comp-pol I*) (*con-comp-map D E p q*) ($\lambda x.\ x \in$ *range p*) *U ys*), *inv p* ‘ *ipurge-ref-aux-foldr* (*con-comp-pol I*) (*con-comp-map D E p q*) ($\lambda x.\ x \in$ *range p*) *U ys Y*) $\in$ *failures P*

In fact, for $ys = y \# ys'$, the induction hypothesis entails that the consequent holds if xs, ys, and U are replaced with xs @ $[y]$, ys', and *sinks-aux* (*con-comp-pol I*) (*con-comp-map D E p q*) U $[y]$, respectively. The proof can then be accomplished by applying lemma *ipurge-tr-ref-aux-failures-general* to the resulting future of trace *map* (*inv p*) (*filter* (λ*x*. *x* ∈ *range p*) *xs*), moving functions *ipurge-tr-aux* and *ipurge-ref-aux* into the arguments of *map* (*inv p*) and (‘) (*inv p*), and using the recursive equations contained in the definitions of functions *ipurge-tr-aux-foldr* and *ipurge-ref-aux-foldr*.

This property, along with the match of the outputs of functions *ipurge-tr-aux-foldr* and *ipurge-ref-aux-foldr* with the filtered intransitive purge of the input event list and the intransitive purge of the input event set, respectively, permits to solve the main proof obligations arising from the demonstration of the target security conservation theorem.

Here below is the proof of the equivalence between function *ipurge-tr-aux-foldr* and the filtered intransitive purge of an event list.

lemma *ipurge-tr-aux-foldr-subset*:
 U ⊆ *V* ⟹
  ipurge-tr-aux I D U (*ipurge-tr-aux-foldr I D P V xs*) =
   ipurge-tr-aux-foldr I D P V xs
proof (*induction xs*, *simp-all add*: *ipurge-tr-aux-union* [*symmetric*])
qed (*drule Un-absorb2*, *simp*)

lemma *ipurge-tr-aux-foldr-eq*:
 [*x*←*ipurge-tr-aux I D U xs*. *P x*] = *ipurge-tr-aux-foldr I D P U xs*
proof (*induction xs arbitrary*: *U*, *simp*)
 fix *x xs U*
 assume
  A: ⋀*U*. [*x*←*ipurge-tr-aux I D U xs*. *P x*] = *ipurge-tr-aux-foldr I D P U xs*
 show [*x*←*ipurge-tr-aux I D U* (*x* # *xs*). *P x*] =
  ipurge-tr-aux-foldr I D P U (*x* # *xs*)
 proof (*cases* ∃ *u* ∈ *U*. (*u*, *D x*) ∈ *I*,
  simp-all only: *ipurge-tr-aux-foldr.simps ipurge-tr-aux-cons*
  sinks-aux-single-event if-True if-False)
  case *True*
  have *B*: [*x*←*ipurge-tr-aux I D* (*insert* (*D x*) *U*) *xs*. *P x*] =
   ipurge-tr-aux-foldr I D P (*insert* (*D x*) *U*) *xs*
   using *A* **.**
  show [*x*←*ipurge-tr-aux I D* (*insert* (*D x*) *U*) *xs*. *P x*] = *ipurge-tr-aux I D U*
   ((*if P x then* [*x*] *else* []) @ *ipurge-tr-aux-foldr I D P* (*insert* (*D x*) *U*) *xs*)
  proof (*cases P x*, *simp-all add*: *ipurge-tr-aux-cons True*
   del: *con-comp-map.simps*)
   have *insert* (*D x*) *U* ⊆ *insert* (*D x*) *U* **..**
   hence *ipurge-tr-aux I D* (*insert* (*D x*) *U*)
    (*ipurge-tr-aux-foldr I D P* (*insert* (*D x*) *U*) *xs*) =
     ipurge-tr-aux-foldr I D P (*insert* (*D x*) *U*) *xs*
    by (*rule ipurge-tr-aux-foldr-subset*)
   thus [*x*←*ipurge-tr-aux I D* (*insert* (*D x*) *U*) *xs*. *P x*] =
    ipurge-tr-aux I D (*insert* (*D x*) *U*)
     (*ipurge-tr-aux-foldr I D P* (*insert* (*D x*) *U*) *xs*)

```
      using B by simp
    next
      have U ⊆ insert (D x) U
       by (rule subset-insertI)
      hence ipurge-tr-aux I D U
        (ipurge-tr-aux-foldr I D P (insert (D x) U) xs) =
          ipurge-tr-aux-foldr I D P (insert (D x) U) xs
       by (rule ipurge-tr-aux-foldr-subset)
      thus [x←ipurge-tr-aux I D (insert (D x) U) xs. P x] =
        ipurge-tr-aux I D U
          (ipurge-tr-aux-foldr I D P (insert (D x) U) xs)
       using B by simp
    qed
  next
    case False
    have B: [x←ipurge-tr-aux I D U xs. P x] = ipurge-tr-aux-foldr I D P U xs
     using A .
    show [x←x # ipurge-tr-aux I D U xs. P x] = ipurge-tr-aux I D U
      ((if P x then [x] else []) @ ipurge-tr-aux-foldr I D P U xs)
    proof (cases P x, simp-all add: ipurge-tr-aux-cons False
     del: con-comp-map.simps)
      have U ⊆ U ..
      hence ipurge-tr-aux I D U (ipurge-tr-aux-foldr I D P U xs) =
        ipurge-tr-aux-foldr I D P U xs
       by (rule ipurge-tr-aux-foldr-subset)
      thus [x←ipurge-tr-aux I D U xs. P x] =
        ipurge-tr-aux I D U (ipurge-tr-aux-foldr I D P U xs)
       using B by simp
    next
      have U ⊆ U ..
      hence ipurge-tr-aux I D U (ipurge-tr-aux-foldr I D P U xs) =
        ipurge-tr-aux-foldr I D P U xs
       by (rule ipurge-tr-aux-foldr-subset)
      thus [x←ipurge-tr-aux I D U xs. P x] =
        ipurge-tr-aux I D U (ipurge-tr-aux-foldr I D P U xs)
       using B by simp
    qed
  qed
qed
```

Here below is the proof of the equivalence between function *ipurge-ref-aux-foldr* and the intransitive purge of an event set.

```
lemma ipurge-tr-aux-foldr-sinks-aux [rule-format]:
 U ⊆ V ⟶ sinks-aux I D U (ipurge-tr-aux-foldr I D P V xs) = U
proof (induction xs arbitrary: V, simp, rule impI)
  fix x xs V
  assume
```

```
  A: ⋀V. U ⊆ V ⟶ sinks-aux I D U (ipurge-tr-aux-foldr I D P V xs) = U and
  B: U ⊆ V
show sinks-aux I D U (ipurge-tr-aux-foldr I D P V (x # xs)) = U
proof (cases P x, case-tac [!] ∃ v ∈ V. (v, D x) ∈ I,
 simp-all (no-asm-simp) add: sinks-aux-cons ipurge-tr-aux-cons)
  have U ⊆ insert (D x) V ⟶
    sinks-aux I D U (ipurge-tr-aux-foldr I D P (insert (D x) V) xs) = U
    (is - ⟶ sinks-aux I D U ?ys = U)
   using A .
  moreover have U ⊆ insert (D x) V
   using B by (rule subset-insertI2)
  ultimately have sinks-aux I D U ?ys = U ..
  moreover have insert (D x) V ⊆ insert (D x) V ..
  hence ipurge-tr-aux I D (insert (D x) V)
    (ipurge-tr-aux-foldr I D P (insert (D x) V) xs) = ?ys
    (is ?zs = -)
   by (rule ipurge-tr-aux-foldr-subset)
  ultimately show sinks-aux I D U ?zs = U
   by simp
next
  assume C: ¬ (∃ v ∈ V. (v, D x) ∈ I)
  have ¬ (∃ u ∈ U. (u, D x) ∈ I)
  proof
    assume ∃ u ∈ U. (u, D x) ∈ I
    then obtain u where D: u ∈ U and E: (u, D x) ∈ I ..
    have u ∈ V
     using B and D ..
    with E have ∃ v ∈ V. (v, D x) ∈ I ..
    thus False
     using C by contradiction
  qed
  thus
   ((∃ v ∈ U. (v, D x) ∈ I) ⟶ sinks-aux I D (insert (D x) U)
      (ipurge-tr-aux I D V (ipurge-tr-aux-foldr I D P V xs)) = U) ∧
   ((∀ v ∈ U. (v, D x) ∉ I) ⟶ sinks-aux I D U
      (ipurge-tr-aux I D V (ipurge-tr-aux-foldr I D P V xs)) = U)
  proof simp
    have U ⊆ V ⟶ sinks-aux I D U (ipurge-tr-aux-foldr I D P V xs) = U
     (is - ⟶ sinks-aux I D U ?ys = U)
     using A .
    hence sinks-aux I D U ?ys = U using B ..
    moreover have V ⊆ V ..
    hence ipurge-tr-aux I D V (ipurge-tr-aux-foldr I D P V xs) = ?ys
     (is ?zs = -)
     by (rule ipurge-tr-aux-foldr-subset)
    ultimately show sinks-aux I D U ?zs = U
     by simp
  qed
next
```

have *U* ⊆ *insert* (*D x*) *V* ⟶
sinks-aux I D U (*ipurge-tr-aux-foldr I D P* (*insert* (*D x*) *V*) *xs*) = *U*
(**is** - ⟶ *sinks-aux I D U ?ys* = *U*)
using *A* **.**
moreover have *U* ⊆ *insert* (*D x*) *V*
using *B* **by** (*rule subset-insertI2*)
ultimately have *sinks-aux I D U ?ys* = *U* **..**
moreover have *V* ⊆ *insert* (*D x*) *V*
by (*rule subset-insertI*)
hence *ipurge-tr-aux I D V*
(*ipurge-tr-aux-foldr I D P* (*insert* (*D x*) *V*) *xs*) = *?ys*
(**is** *?zs* = -)
by (*rule ipurge-tr-aux-foldr-subset*)
ultimately show *sinks-aux I D U ?zs* = *U*
by *simp*
next
have *U* ⊆ *V* ⟶ *sinks-aux I D U* (*ipurge-tr-aux-foldr I D P V xs*) = *U*
(**is** - ⟶ *sinks-aux I D U ?ys* = *U*)
using *A* **.**
hence *sinks-aux I D U ?ys* = *U* **using** *B* **..**
moreover have *V* ⊆ *V* **..**
hence *ipurge-tr-aux I D V* (*ipurge-tr-aux-foldr I D P V xs*) = *?ys*
(**is** *?zs* = -)
by (*rule ipurge-tr-aux-foldr-subset*)
ultimately show *sinks-aux I D U ?zs* = *U*
by *simp*
qed
qed

lemma *ipurge-tr-aux-foldr-ref-aux*:
assumes *A*: *U* ⊆ *V*
shows *ipurge-ref-aux I D U* (*ipurge-tr-aux-foldr I D P V xs*) *X* =
ipurge-ref-aux I D U [] *X*
by (*simp add*: *ipurge-ref-aux-def ipurge-tr-aux-foldr-sinks-aux* [*OF A*])

lemma *ipurge-ref-aux-foldr-subset* [*rule-format*]:
sinks-aux I D U ys ⊆ *V* ⟶
ipurge-ref-aux I D U ys (*ipurge-ref-aux-foldr I D P V xs X*) =
ipurge-ref-aux-foldr I D P V xs X
proof (*induction xs arbitrary*: *ys U V*, *rule-tac* [!] *impI*,
simp add: *ipurge-ref-aux-def*, *blast*)
fix *x xs ys U V*
assume
A: ⋀*ys U V*.
sinks-aux I D U ys ⊆ *V* ⟶
ipurge-ref-aux I D U ys (*ipurge-ref-aux-foldr I D P V xs X*) =
ipurge-ref-aux-foldr I D P V xs X **and**
B: *sinks-aux I D U ys* ⊆ *V*
show *ipurge-ref-aux I D U ys* (*ipurge-ref-aux-foldr I D P V* (*x* # *xs*) *X*) =

```
  ipurge-ref-aux-foldr I D P V (x # xs) X
proof (cases P x, simp-all add: ipurge-ref-aux-cons)
  have C: sinks-aux I D V [x] ⊆ sinks-aux I D V [x] ..
  show
   ipurge-ref-aux I D U ys (ipurge-ref-aux I D (sinks-aux I D V [x])
      (ipurge-tr-aux-foldr I D P (sinks-aux I D V [x]) xs)
      (ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X)) =
    ipurge-ref-aux I D (sinks-aux I D V [x])
      (ipurge-tr-aux-foldr I D P (sinks-aux I D V [x]) xs)
      (ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X)
  proof (simp add: ipurge-tr-aux-foldr-ref-aux [OF C])
    have sinks-aux I D (sinks-aux I D V [x]) [] ⊆ sinks-aux I D V [x] ⟶
      ipurge-ref-aux I D (sinks-aux I D V [x]) []
        (ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X) =
      ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X
      (is ?A ⟶ ?us = ?vs)
     using A .
    moreover have ?A
     by simp
    ultimately have ?us = ?vs ..
    thus ipurge-ref-aux I D U ys ?us = ?us
    proof simp
      have sinks-aux I D U ys ⊆ sinks-aux I D V [x] ⟶
        ipurge-ref-aux I D U ys
          (ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X) =
        ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X
        (is - ⟶ ?T)
       using A .
      moreover have V ⊆ sinks-aux I D V [x]
       by (rule sinks-aux-subset)
      hence sinks-aux I D U ys ⊆ sinks-aux I D V [x]
       using B by simp
      ultimately show ?T ..
    qed
  qed
next
  have C: V ⊆ sinks-aux I D V [x]
   by (rule sinks-aux-subset)
  show
   ipurge-ref-aux I D U ys (ipurge-ref-aux I D V
      (ipurge-tr-aux-foldr I D P (sinks-aux I D V [x]) xs)
      (ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X)) =
    ipurge-ref-aux I D V
      (ipurge-tr-aux-foldr I D P (sinks-aux I D V [x]) xs)
      (ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X)
  proof (simp add: ipurge-tr-aux-foldr-ref-aux [OF C])
    have sinks-aux I D V [] ⊆ sinks-aux I D V [x] ⟶
      ipurge-ref-aux I D V []
        (ipurge-ref-aux-foldr I D P (sinks-aux I D V [x]) xs X) =
```

ipurge-ref-aux-foldr I D P (*sinks-aux I D V* [*x*]) *xs X*
(**is** *?A* ⟶ *?us* = *?vs*)
using *A* **.**
moreover have *?A*
using *C* **by** *simp*
ultimately have *?us* = *?vs* **..**
thus *ipurge-ref-aux I D U ys ?us* = *?us*
proof *simp*
have *sinks-aux I D U ys* ⊆ *sinks-aux I D V* [*x*] ⟶
ipurge-ref-aux I D U ys
(*ipurge-ref-aux-foldr I D P* (*sinks-aux I D V* [*x*]) *xs X*) =
ipurge-ref-aux-foldr I D P (*sinks-aux I D V* [*x*]) *xs X*
(**is** - ⟶ *?T*)
using *A* **.**
moreover have *sinks-aux I D U ys* ⊆ *sinks-aux I D V* [*x*]
using *B* **and** *C* **by** *simp*
ultimately show *?T* **..**
qed
qed
qed
qed

lemma *ipurge-ref-aux-foldr-eq*:
ipurge-ref-aux I D U xs X = *ipurge-ref-aux-foldr I D P U xs X*
proof (*induction xs arbitrary*: *U*, *simp*)
fix *x xs U*
assume *A*: ⋀*U*. *ipurge-ref-aux I D U xs X* = *ipurge-ref-aux-foldr I D P U xs X*
show *ipurge-ref-aux I D U* (*x* # *xs*) *X* =
ipurge-ref-aux-foldr I D P U (*x* # *xs*) *X*
proof (*cases P x*, *simp-all add*: *ipurge-ref-aux-cons*)
have *sinks-aux I D U* [*x*] ⊆ *sinks-aux I D U* [*x*] **..**
hence
ipurge-ref-aux I D (*sinks-aux I D U* [*x*])
(*ipurge-tr-aux-foldr I D P* (*sinks-aux I D U* [*x*]) *xs*)
(*ipurge-ref-aux-foldr I D P* (*sinks-aux I D U* [*x*]) *xs X*) =
ipurge-ref-aux I D (*sinks-aux I D U* [*x*]) []
(*ipurge-ref-aux-foldr I D P* (*sinks-aux I D U* [*x*]) *xs X*)
(**is** *ipurge-ref-aux* - - - *?xs′ ?X′* = -)
by (*rule ipurge-tr-aux-foldr-ref-aux*)
also have *sinks-aux I D* (*sinks-aux I D U* [*x*]) [] ⊆ *sinks-aux I D U* [*x*]
by *simp*
hence *ipurge-ref-aux I D* (*sinks-aux I D U* [*x*]) [] *?X′* = *?X′*
by (*rule ipurge-ref-aux-foldr-subset*)
finally have *ipurge-ref-aux I D* (*sinks-aux I D U* [*x*]) *?xs′ ?X′* = *?X′* **.**
thus *ipurge-ref-aux I D* (*sinks-aux I D U* [*x*]) *xs X* =
ipurge-ref-aux I D (*sinks-aux I D U* [*x*]) *?xs′ ?X′*
proof *simp*
show *ipurge-ref-aux I D* (*sinks-aux I D U* [*x*]) *xs X* =
ipurge-ref-aux-foldr I D P (*sinks-aux I D U* [*x*]) *xs X*

```
      using A .
    qed
  next
    have U ⊆ sinks-aux I D U [x]
     by (rule sinks-aux-subset)
    hence
     ipurge-ref-aux I D U
        (ipurge-tr-aux-foldr I D P (sinks-aux I D U [x]) xs)
        (ipurge-ref-aux-foldr I D P (sinks-aux I D U [x]) xs X) =
      ipurge-ref-aux I D U []
        (ipurge-ref-aux-foldr I D P (sinks-aux I D U [x]) xs X)
      (is ipurge-ref-aux - - - ?xs' ?X' = -)
     by (rule ipurge-tr-aux-foldr-ref-aux)
    also have sinks-aux I D U [] ⊆ sinks-aux I D U [x]
     by (simp, rule sinks-aux-subset)
    hence ipurge-ref-aux I D U [] ?X' = ?X'
     by (rule ipurge-ref-aux-foldr-subset)
    finally have ipurge-ref-aux I D U ?xs' ?X' = ?X' .
    thus ipurge-ref-aux I D (sinks-aux I D U [x]) xs X =
      ipurge-ref-aux I D U ?xs' ?X'
    proof simp
      show ipurge-ref-aux I D (sinks-aux I D U [x]) xs X =
        ipurge-ref-aux-foldr I D P (sinks-aux I D U [x]) xs X
       using A .
    qed
  qed
qed
```

Finally, here below is the proof of the implication involving functions *ipurge-tr-aux-foldr* and *ipurge-ref-aux-foldr* discussed above.

```
lemma con-comp-sinks-aux-range:
  assumes
    A: U ⊆ range Some and
    B: set xs ⊆ range p ∪ range q
  shows sinks-aux (con-comp-pol I) (con-comp-map D E p q) U xs ⊆ range Some
    (is sinks-aux - ?D' - - ⊆ -)
proof (rule subsetI, drule sinks-aux-elem, erule disjE, erule-tac [2] bexE)
  fix u
  assume u ∈ U
  with A show u ∈ range Some ..
next
  fix u x
  assume x ∈ set xs
  with B have x ∈ range p ∪ range q ..
  hence ?D' x ∈ range Some
   by (cases x ∈ range p, simp-all)
  moreover assume u = ?D' x
```

```
  ultimately show u ∈ range Some
   by simp
qed

lemma con-comp-sinks-aux [rule-format]:
  assumes A: U ⊆ range Some
  shows set xs ⊆ range p ⟶
    sinks-aux I D (the ‘ U) (map (inv p) xs) =
    the ‘ sinks-aux (con-comp-pol I) (con-comp-map D E p q) U xs
    (is - ⟶ - = the ‘ sinks-aux ?I′ ?D′ - -)
proof (induction xs rule: rev-induct, simp, rule impI)
  fix x xs
  assume set xs ⊆ range p ⟶
    sinks-aux I D (the ‘ U) (map (inv p) xs) =
    the ‘ sinks-aux ?I′ ?D′ U xs
  moreover assume B: set (xs @ [x]) ⊆ range p
  ultimately have C: sinks-aux I D (the ‘ U) (map (inv p) xs) =
    the ‘ sinks-aux ?I′ ?D′ U xs
   by simp
  show sinks-aux I D (the ‘ U) (map (inv p) (xs @ [x])) =
    the ‘ sinks-aux ?I′ ?D′ U (xs @ [x])
  proof (cases ∃ u ∈ sinks-aux ?I′ ?D′ U xs. (u, ?D′ x) ∈ ?I′,
   simp-all (no-asm-simp) del: map-append)
    case True
    then obtain u where
      D: u ∈ sinks-aux ?I′ ?D′ U xs and E: (u, ?D′ x) ∈ ?I′ ..
    have (the u, D (inv p x)) ∈ I
     using B and E by (simp add: con-comp-pol-def, erule-tac exE, simp)
    moreover have the u ∈ the ‘ sinks-aux ?I′ ?D′ U xs
     using D by simp
    hence the u ∈ sinks-aux I D (the ‘ U) (map (inv p) xs)
     using C by simp
    ultimately have ∃ d ∈ sinks-aux I D (the ‘ U) (map (inv p) xs).
      (d, D (inv p x)) ∈ I ..
    hence sinks-aux I D (the ‘ U) (map (inv p) (xs @ [x])) =
      insert (D (inv p x)) (sinks-aux I D (the ‘ U) (map (inv p) xs))
     by simp
    thus sinks-aux I D (the ‘ U) (map (inv p) (xs @ [x])) =
      insert (the (?D′ x)) (the ‘ sinks-aux ?I′ ?D′ U xs)
     using B and C by simp
  next
    case False
    have ¬ (∃ d ∈ sinks-aux I D (the ‘ U) (map (inv p) xs).
      (d, D (inv p x)) ∈ I)
    proof (rule notI, erule bexE)
      fix d
      assume d ∈ sinks-aux I D (the ‘ U) (map (inv p) xs)
      hence d ∈ the ‘ sinks-aux ?I′ ?D′ U xs
       using C by simp
```

hence ∃ *u* ∈ *sinks-aux ?I′ ?D′ U xs*. *d* = *the u*
by (*simp add*: *image-iff*)
then obtain *u* **where**
D: *u* ∈ *sinks-aux ?I′ ?D′ U xs* **and** *E*: *d* = *the u* **..**
have *set xs* ⊆ *range p* ∪ *range q*
using *B* **by** (*simp*, *blast*)
with *A* **have** *sinks-aux ?I′ ?D′ U xs* ⊆ *range Some*
by (*rule con-comp-sinks-aux-range*)
hence *u* ∈ *range Some*
using *D* **..**
hence *u* = *Some d*
using *E* **by** (*simp add*: *image-iff*)
moreover assume (*d*, *D* (*inv p x*)) ∈ *I*
hence (*Some d*, *Some* (*D* (*inv p x*))) ∈ *?I′*
by (*simp add*: *con-comp-pol-def*)
ultimately have (*u*, *?D′ x*) ∈ *?I′*
using *B* **by** *simp*
hence ∃ *u* ∈ *sinks-aux ?I′ ?D′ U xs*. (*u*, *?D′ x*) ∈ *?I′*
using *D* **..**
thus *False*
using *False* **by** *contradiction*
qed
thus *sinks-aux I D* (*the* ‘ *U*) (*map* (*inv p*) (*xs* @ [*x*])) =
the ‘ *sinks-aux ?I′ ?D′ U xs*
using *C* **by** *simp*
qed
qed

lemma *con-comp-ipurge-tr-aux* [*rule-format*]:
assumes *A*: *U* ⊆ *range Some*
shows *set xs* ⊆ *range p* ⟶
ipurge-tr-aux I D (*the* ‘ *U*) (*map* (*inv p*) *xs*) =
map (*inv p*) (*ipurge-tr-aux* (*con-comp-pol I*) (*con-comp-map D E p q*) *U xs*)
(**is** - ⟶ - = *map* (*inv p*) (*ipurge-tr-aux ?I′ ?D′* - -))
proof (*induction xs rule*: *rev-induct*, *simp*, *rule impI*)
fix *x xs*
assume *set xs* ⊆ *range p* ⟶
ipurge-tr-aux I D (*the* ‘ *U*) (*map* (*inv p*) *xs*) =
map (*inv p*) (*ipurge-tr-aux ?I′ ?D′ U xs*)
moreover assume *B*: *set* (*xs* @ [*x*]) ⊆ *range p*
ultimately have *C*: *ipurge-tr-aux I D* (*the* ‘ *U*) (*map* (*inv p*) *xs*) =
map (*inv p*) (*ipurge-tr-aux ?I′ ?D′ U xs*)
by *simp*
show *ipurge-tr-aux I D* (*the* ‘ *U*) (*map* (*inv p*) (*xs* @ [*x*])) =
map (*inv p*) (*ipurge-tr-aux ?I′ ?D′ U* (*xs* @ [*x*]))
proof (*cases* ∃ *u* ∈ *sinks-aux ?I′ ?D′ U xs*. (*u*, *?D′ x*) ∈ *?I′*)
case *True*
then obtain *u* **where**
D: *u* ∈ *sinks-aux ?I′ ?D′ U xs* **and** *E*: (*u*, *?D′ x*) ∈ *?I′* **..**

```
  have (the u, D (inv p x)) ∈ I
   using B and E by (simp add: con-comp-pol-def, erule-tac exE, simp)
  moreover have F: the u ∈ the ` sinks-aux ?I' ?D' U xs
   using D by simp
  have set xs ⊆ range p
   using B by simp
  with A have sinks-aux I D (the ` U) (map (inv p) xs) =
    the ` sinks-aux ?I' ?D' U xs
   by (rule con-comp-sinks-aux)
  hence the u ∈ sinks-aux I D (the ` U) (map (inv p) xs)
   using F by simp
  ultimately have ∃ d ∈ sinks-aux I D (the ` U) (map (inv p) xs).
    (d, D (inv p x)) ∈ I ..
  hence ipurge-tr-aux I D (the ` U) (map (inv p) (xs @ [x])) =
    ipurge-tr-aux I D (the ` U) (map (inv p) xs)
   by simp
  moreover have map (inv p) (ipurge-tr-aux ?I' ?D' U (xs @ [x])) =
    map (inv p) (ipurge-tr-aux ?I' ?D' U xs)
   using True by simp
  ultimately show ?thesis
   using C by simp
next
  case False
  have ¬ (∃ d ∈ sinks-aux I D (the ` U) (map (inv p) xs).
    (d, D (inv p x)) ∈ I)
  proof (rule notI, erule bexE)
    fix d
    assume d ∈ sinks-aux I D (the ` U) (map (inv p) xs)
    moreover have set xs ⊆ range p
     using B by simp
    with A have sinks-aux I D (the ` U) (map (inv p) xs) =
      the ` sinks-aux ?I' ?D' U xs
     by (rule con-comp-sinks-aux)
    ultimately have d ∈ the ` sinks-aux ?I' ?D' U xs
     by simp
    hence ∃ u ∈ sinks-aux ?I' ?D' U xs. d = the u
     by (simp add: image-iff)
    then obtain u where
      D: u ∈ sinks-aux ?I' ?D' U xs and E: d = the u ..
    have set xs ⊆ range p ∪ range q
     using B by (simp, blast)
    with A have sinks-aux ?I' ?D' U xs ⊆ range Some
     by (rule con-comp-sinks-aux-range)
    hence u ∈ range Some
     using D ..
    hence u = Some d
     using E by (simp add: image-iff)
    moreover assume (d, D (inv p x)) ∈ I
    hence (Some d, Some (D (inv p x))) ∈ ?I'
```

by (*simp add*: *con-comp-pol-def*)
ultimately have (*u*, *?D′ x*) ∈ *?I′*
using *B* **by** *simp*
hence ∃ *u* ∈ *sinks-aux ?I′ ?D′ U xs*. (*u*, *?D′ x*) ∈ *?I′*
using *D* **..**
thus *False*
using *False* **by** *contradiction*
qed
hence *ipurge-tr-aux I D* (*the* ‘ *U*) (*map* (*inv p*) (*xs* @ [*x*])) =
ipurge-tr-aux I D (*the* ‘ *U*) (*map* (*inv p*) *xs*) @ [*inv p x*]
by *simp*
moreover have *map* (*inv p*) (*ipurge-tr-aux ?I′ ?D′ U* (*xs* @ [*x*])) =
map (*inv p*) (*ipurge-tr-aux ?I′ ?D′ U xs*) @ [*inv p x*]
using *False* **by** *simp*
ultimately show *?thesis*
using *C* **by** *simp*
qed
qed

lemma *con-comp-ipurge-ref-aux*:
assumes
A: *U* ⊆ *range Some* **and**
B: *set xs* ⊆ *range p* **and**
C: *X* ⊆ *range p*
shows *ipurge-ref-aux I D* (*the* ‘ *U*) (*map* (*inv p*) *xs*) (*inv p* ‘ *X*) =
inv p ‘ *ipurge-ref-aux* (*con-comp-pol I*) (*con-comp-map D E p q*) *U xs X*
(**is** - = *inv p* ‘ *ipurge-ref-aux ?I′ ?D′* - - -)
proof (*simp add*: *ipurge-ref-aux-def set-eq-iff image-iff*, *rule allI*, *rule iffI*,
erule conjE, *erule bexE*, *erule-tac* [*2*] *exE*, (*erule-tac* [*2*] *conjE*)+)
fix *a x*
assume
D: *x* ∈ *X* **and**
E: *a* = *inv p x* **and**
F: ∀ *d* ∈ *sinks-aux I D* (*the* ‘ *U*) (*map* (*inv p*) *xs*). (*d*, *D a*) ∉ *I*
show ∃ *x*. *x* ∈ *X* ∧ (∀ *u* ∈ *sinks-aux ?I′ ?D′ U xs*. (*u*, *?D′ x*) ∉ *?I′*) ∧
a = *inv p x*
proof (*rule-tac x* = *x* **in** *exI*, *simp add*: *D E*, *rule ballI*)
fix *u*
assume *G*: *u* ∈ *sinks-aux ?I′ ?D′ U xs*
moreover have *sinks-aux I D* (*the* ‘ *U*) (*map* (*inv p*) *xs*) =
the ‘ *sinks-aux ?I′ ?D′ U xs*
using *A* **and** *B* **by** (*rule con-comp-sinks-aux*)
ultimately have *the u* ∈ *sinks-aux I D* (*the* ‘ *U*) (*map* (*inv p*) *xs*)
by *simp*
with *F* **have** (*the u*, *D a*) ∉ *I* **..**
moreover have *set xs* ⊆ *range p* ∪ *range q*
using *B* **by** *blast*
with *A* **have** *sinks-aux ?I′ ?D′ U xs* ⊆ *range Some*
by (*rule con-comp-sinks-aux-range*)

```
    hence u ∈ range Some
     using G ..
    hence ∃ d. u = Some d
     by (simp add: image-iff)
    then obtain d where H: u = Some d ..
    ultimately have (d, D (inv p x)) ∉ I
     using E by simp
    hence (u, Some (D (inv p x))) ∉ ?I′
     using H by (simp add: con-comp-pol-def)
    moreover have x ∈ range p
     using C and D ..
    ultimately show (u, ?D′ x) ∉ ?I′
     by simp
  qed
next
  fix a x
  assume
    D: x ∈ X and
    E: a = inv p x and
    F: ∀ u ∈ sinks-aux ?I′ ?D′ U xs. (u, ?D′ x) ∉ ?I′
  show (∃ x ∈ X. a = inv p x) ∧
    (∀ u ∈ sinks-aux I D (the ‘ U) (map (inv p) xs). (u, D a) ∉ I)
  proof (rule conjI, rule-tac [2] ballI)
    show ∃ x ∈ X. a = inv p x
     using E and D ..
  next
    fix d
    assume d ∈ sinks-aux I D (the ‘ U) (map (inv p) xs)
    moreover have sinks-aux I D (the ‘ U) (map (inv p) xs) =
      the ‘ sinks-aux ?I′ ?D′ U xs
     using A and B by (rule con-comp-sinks-aux)
    ultimately have d ∈ the ‘ sinks-aux ?I′ ?D′ U xs
     by simp
    hence ∃ u ∈ sinks-aux ?I′ ?D′ U xs. d = the u
     by (simp add: image-iff)
    then obtain u where G: u ∈ sinks-aux ?I′ ?D′ U xs and H: d = the u ..
    have (u, ?D′ x) ∉ ?I′
     using F and G ..
    moreover have set xs ⊆ range p ∪ range q
     using B by blast
    with A have sinks-aux ?I′ ?D′ U xs ⊆ range Some
     by (rule con-comp-sinks-aux-range)
    hence u ∈ range Some
     using G ..
    hence u = Some d
     using H by (simp add: image-iff)
    moreover have x ∈ range p
     using C and D ..
    ultimately have (d, D (inv p x)) ∉ I
```

```
    by (simp add: con-comp-pol-def)
   thus (d, D a) ∉ I
    using E by simp
  qed
qed

lemma con-comp-sinks-filter:
 sinks (con-comp-pol I) (con-comp-map D E p q) u
    [x←xs. x ∈ range p ∪ range q] =
  sinks (con-comp-pol I) (con-comp-map D E p q) u xs ∩ range Some
  (is sinks ?I′ ?D′ - - = -)
proof (induction xs rule: rev-induct, simp)
  fix x xs
  assume A: sinks ?I′ ?D′ u [x←xs. x ∈ range p ∪ range q] =
    sinks ?I′ ?D′ u xs ∩ range Some
    (is sinks - - - ?xs′ = -)
  show sinks ?I′ ?D′ u [x←xs @ [x]. x ∈ range p ∪ range q] =
    sinks ?I′ ?D′ u (xs @ [x]) ∩ range Some
  proof (cases x ∈ range p ∪ range q, simp-all del: Un-iff sinks.simps,
   cases (u, ?D′ x) ∈ ?I′ ∨ (∃ v ∈ sinks ?I′ ?D′ u ?xs′. (v, ?D′ x) ∈ ?I′))
    assume
      B: x ∈ range p ∪ range q and
      C: (u, ?D′ x) ∈ ?I′ ∨ (∃ v ∈ sinks ?I′ ?D′ u ?xs′. (v, ?D′ x) ∈ ?I′)
    have sinks ?I′ ?D′ u (?xs′ @ [x]) =
      insert (?D′ x) (sinks ?I′ ?D′ u ?xs′)
     using C by simp
    also have ... =
      insert (?D′ x) (sinks ?I′ ?D′ u xs ∩ range Some)
     using A by simp
    also have ... =
      insert (?D′ x) (sinks ?I′ ?D′ u xs) ∩ insert (?D′ x) (range Some)
     by simp
    finally have sinks ?I′ ?D′ u (?xs′ @ [x]) =
      insert (?D′ x) (sinks ?I′ ?D′ u xs) ∩ insert (?D′ x) (range Some) .
    moreover have insert (?D′ x) (range Some) = range Some
     using B by (rule-tac insert-absorb, cases x ∈ range p, simp-all)
    ultimately have sinks ?I′ ?D′ u (?xs′ @ [x]) =
      insert (?D′ x) (sinks ?I′ ?D′ u xs) ∩ range Some
     by simp
    moreover have (u, ?D′ x) ∈ ?I′ ∨
     (∃ v ∈ sinks ?I′ ?D′ u xs. (v, ?D′ x) ∈ ?I′)
     using A and C by (simp, blast)
    ultimately show sinks ?I′ ?D′ u (?xs′ @ [x]) =
      sinks ?I′ ?D′ u (xs @ [x]) ∩ range Some
     by simp
  next
    assume
      B: x ∈ range p ∪ range q and
      C: ¬ ((u, ?D′ x) ∈ ?I′ ∨ (∃ v ∈ sinks ?I′ ?D′ u ?xs′. (v, ?D′ x) ∈ ?I′))
```

have *sinks ?I′ ?D′ u* (*?xs′* @ [*x*]) = *sinks ?I′ ?D′ u ?xs′*
 using *C* **by** *simp*
hence *sinks ?I′ ?D′ u* (*?xs′* @ [*x*]) = *sinks ?I′ ?D′ u xs* ∩ *range Some*
 using *A* **by** *simp*
moreover from *C* **have**
¬ ((*u*, *?D′ x*) ∈ *?I′* ∨ (∃ *v* ∈ *sinks ?I′ ?D′ u xs*. (*v*, *?D′ x*) ∈ *?I′*))
proof (*rule-tac notI*, *simp del*: *bex-simps*)
 assume ∃ *v* ∈ *sinks ?I′ ?D′ u xs*. (*v*, *?D′ x*) ∈ *?I′*
 then obtain *v* **where** *E*: *v* ∈ *sinks ?I′ ?D′ u xs* **and** *F*: (*v*, *?D′ x*) ∈ *?I′* **..**
 have ∃ *d*. *?D′ x* = *Some d*
  using *B* **by** (*cases x* ∈ *range p*, *simp-all*)
 then obtain *d* **where** *?D′ x* = *Some d* **..**
 hence (*v*, *Some d*) ∈ *?I′*
  using *F* **by** *simp*
 hence *v* ∈ *range Some*
  by (*cases v*, *simp-all add*: *con-comp-pol-def*)
 with *E* **have** *v* ∈ *sinks ?I′ ?D′ u xs* ∩ *range Some* **..**
 hence *v* ∈ *sinks ?I′ ?D′ u ?xs′*
  using *A* **by** *simp*
 with *F* **have** ∃ *v* ∈ *sinks ?I′ ?D′ u ?xs′*. (*v*, *?D′ x*) ∈ *?I′* **..**
 thus *False*
  using *C* **by** *simp*
qed
ultimately show *sinks ?I′ ?D′ u* (*?xs′* @ [*x*]) =
 sinks ?I′ ?D′ u (*xs* @ [*x*]) ∩ *range Some*
 by *simp*
next
assume *B*: *x* ∉ *range p* ∪ *range q*
hence (*u*, *?D′ x*) ∈ *?I′*
 by (*simp add*: *con-comp-pol-def*)
hence *sinks ?I′ ?D′ u* (*xs* @ [*x*]) = *insert* (*?D′ x*) (*sinks ?I′ ?D′ u xs*)
 by *simp*
moreover have *insert* (*?D′ x*) (*sinks ?I′ ?D′ u xs*) ∩ *range Some* =
 sinks ?I′ ?D′ u xs ∩ *range Some*
 using *B* **by** *simp*
ultimately have *sinks ?I′ ?D′ u* (*xs* @ [*x*]) ∩ *range Some* =
 sinks ?I′ ?D′ u xs ∩ *range Some*
 by *simp*
thus *sinks ?I′ ?D′ u ?xs′* = *sinks ?I′ ?D′ u* (*xs* @ [*x*]) ∩ *range Some*
 using *A* **by** *simp*
qed
qed

lemma *con-comp-ipurge-tr-filter*:
ipurge-tr (*con-comp-pol I*) (*con-comp-map D E p q*) *u*
 [*x*←*xs*. *x* ∈ *range p* ∪ *range q*] =
ipurge-tr (*con-comp-pol I*) (*con-comp-map D E p q*) *u xs*
(**is** *ipurge-tr ?I′ ?D′* - - = -)
proof (*induction xs rule*: *rev-induct*, *simp*)

fix *x xs*
assume *A*: *ipurge-tr ?I′ ?D′ u* [*x*←*xs*. *x* ∈ *range p* ∪ *range q*] =
ipurge-tr ?I′ ?D′ u xs
(**is** *ipurge-tr* - - - *?xs′* = -)
show *ipurge-tr ?I′ ?D′ u* [*x*←*xs* @ [*x*]. *x* ∈ *range p* ∪ *range q*] =
ipurge-tr ?I′ ?D′ u (*xs* @ [*x*])
proof (*cases x* ∈ *range p* ∪ *range q*, *simp-all del*: *Un-iff ipurge-tr.simps*,
cases ?D′ x ∈ *sinks ?I′ ?D′ u* (*?xs′* @ [*x*]))
assume
B: *x* ∈ *range p* ∪ *range q* **and**
C: *?D′ x* ∈ *sinks ?I′ ?D′ u* (*?xs′* @ [*x*])
have *ipurge-tr ?I′ ?D′ u* (*?xs′* @ [*x*]) = *ipurge-tr ?I′ ?D′ u ?xs′*
using *C* **by** *simp*
hence *ipurge-tr ?I′ ?D′ u* (*?xs′* @ [*x*]) = *ipurge-tr ?I′ ?D′ u xs*
using *A* **by** *simp*
moreover have *?D′ x* ∈ *sinks ?I′ ?D′ u* [*x*←*xs* @ [*x*]. *x* ∈ *range p* ∪ *range q*]
using *B* **and** *C* **by** *simp*
hence *?D′ x* ∈ *sinks ?I′ ?D′ u* (*xs* @ [*x*])
by (*simp only*: *con-comp-sinks-filter*, *blast*)
ultimately show *ipurge-tr ?I′ ?D′ u* (*?xs′* @ [*x*]) =
ipurge-tr ?I′ ?D′ u (*xs* @ [*x*])
by *simp*
next
assume
B: *x* ∈ *range p* ∪ *range q* **and**
C: ¬ (*?D′ x* ∈ *sinks ?I′ ?D′ u* (*?xs′* @ [*x*]))
have *ipurge-tr ?I′ ?D′ u* (*?xs′* @ [*x*]) = *ipurge-tr ?I′ ?D′ u ?xs′* @ [*x*]
using *C* **by** *simp*
hence *ipurge-tr ?I′ ?D′ u* (*?xs′* @ [*x*]) = *ipurge-tr ?I′ ?D′ u xs* @ [*x*]
using *A* **by** *simp*
moreover have *?D′ x* ∉ *sinks ?I′ ?D′ u* [*x*←*xs* @ [*x*]. *x* ∈ *range p* ∪ *range q*]
using *B* **and** *C* **by** *simp*
hence *?D′ x* ∉ *sinks ?I′ ?D′ u* (*xs* @ [*x*]) ∩ *range Some*
by (*simp only*: *con-comp-sinks-filter*, *simp*)
hence *?D′ x* ∉ *sinks ?I′ ?D′ u* (*xs* @ [*x*])
using *B* **by** (*cases x* ∈ *range p*, *simp-all*)
ultimately show *ipurge-tr ?I′ ?D′ u* (*?xs′* @ [*x*]) =
ipurge-tr ?I′ ?D′ u (*xs* @ [*x*])
by *simp*
next
assume *x* ∉ *range p* ∪ *range q*
hence (*u*, *?D′ x*) ∈ *?I′*
by (*simp add*: *con-comp-pol-def*)
hence *?D′ x* ∈ *sinks ?I′ ?D′ u* (*xs* @ [*x*])
by *simp*
thus *ipurge-tr ?I′ ?D′ u ?xs′* = *ipurge-tr ?I′ ?D′ u* (*xs* @ [*x*])
using *A* **by** *simp*
qed
qed

lemma *con-comp-ipurge-ref-filter*:
ipurge-ref (*con-comp-pol I*) (*con-comp-map D E p q*) *u*
[*x*←*xs*. *x* ∈ *range p* ∪ *range q*] *X* =
ipurge-ref (*con-comp-pol I*) (*con-comp-map D E p q*) *u xs X*
(**is** *ipurge-ref ?I′ ?D′* - - - = -)
proof (*simp add*: *ipurge-ref-def con-comp-sinks-filter set-eq-iff del*: *Un-iff*,
rule allI, *rule iffI*, *simp-all*, (*erule conjE*)+, *rule ballI*)
fix *x v*
assume
A: (*u*, *?D′ x*) ∉ *?I′* **and**
B: ∀ *v* ∈ *sinks ?I′ ?D′ u xs* ∩ *range Some*. (*v*, *?D′ x*) ∉ *?I′* **and**
C: *v* ∈ *sinks ?I′ ?D′ u xs*
show (*v*, *?D′ x*) ∉ *?I′*
proof (*cases v*, *simp*)
have *?D′ x* ∈ *range Some*
using *A* **by** (*cases ?D′ x*, *simp-all add*: *con-comp-pol-def*)
thus (*None*, *?D′ x*) ∉ *?I′*
by (*simp add*: *image-iff con-comp-pol-def*)
next
fix *d*
assume *v* = *Some d*
hence *v* ∈ *range Some*
by *simp*
with *C* **have** *v* ∈ *sinks ?I′ ?D′ u xs* ∩ *range Some* **..**
with *B* **show** (*v*, *?D′ x*) ∉ *?I′* **..**
qed
qed

lemma *con-comp-secure-aux* [*rule-format*]:
assumes
A: *secure P I D* **and**
B: *Y* ⊆ *range p*
shows *set ys* ⊆ *range p* ∪ *range q* ⟶ *U* ⊆ *range Some* ⟶
(*map* (*inv p*) [*x*←*xs* @ *ys*. *x* ∈ *range p*], *inv p* ‘ *Y*) ∈ *failures P* ⟶
(*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @
map (*inv p*) (*ipurge-tr-aux-foldr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(λ*x*. *x* ∈ *range p*) *U ys*),
inv p ‘ *ipurge-ref-aux-foldr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(λ*x*. *x* ∈ *range p*) *U ys Y*) ∈ *failures P*
proof (*induction ys arbitrary*: *xs U*, (*rule-tac* [!] *impI*)+, *simp*)
fix *xs U*
assume (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*], *inv p* ‘ *Y*) ∈ *failures P*
moreover have
ipurge-ref-aux (*con-comp-pol I*) (*con-comp-map D E p q*) *U* [] *Y* ⊆ *Y*
(**is** *?Y′* ⊆ -)
by (*rule ipurge-ref-aux-subset*)
hence *inv p* ‘ *?Y′* ⊆ *inv p* ‘ *Y*
by (*rule image-mono*)

```
  ultimately show (map (inv p) [x←xs. x ∈ range p], inv p ‘ ?Y′) ∈ failures P
   by (rule process-rule-3)
next
  fix y ys xs U
  assume ⋀xs U. set ys ⊆ range p ∪ range q ⟶ U ⊆ range Some ⟶
    (map (inv p) [x←xs @ ys. x ∈ range p], inv p ‘ Y) ∈ failures P ⟶
    (map (inv p) [x←xs. x ∈ range p] @
     map (inv p) (ipurge-tr-aux-foldr (con-comp-pol I) (con-comp-map D E p q)
       (λx. x ∈ range p) U ys),
     inv p ‘ ipurge-ref-aux-foldr (con-comp-pol I) (con-comp-map D E p q)
       (λx. x ∈ range p) U ys Y) ∈ failures P
  hence set ys ⊆ range p ∪ range q ⟶
    sinks-aux (con-comp-pol I) (con-comp-map D E p q) U [y] ⊆ range Some ⟶
    (map (inv p) [x←(xs @ [y]) @ ys. x ∈ range p], inv p ‘ Y) ∈ failures P ⟶
    (map (inv p) [x←xs @ [y]. x ∈ range p] @
     map (inv p) (ipurge-tr-aux-foldr (con-comp-pol I) (con-comp-map D E p q)
       (λx. x ∈ range p) (sinks-aux (con-comp-pol I) (con-comp-map D E p q)
         U [y]) ys),
     inv p ‘ ipurge-ref-aux-foldr (con-comp-pol I) (con-comp-map D E p q)
       (λx. x ∈ range p) (sinks-aux (con-comp-pol I) (con-comp-map D E p q)
         U [y]) ys Y) ∈ failures P
    (is - ⟶ - ⟶ - ⟶
      (- @ map (inv p) (ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ -), -) ∈ -) .
  moreover assume C: set (y # ys) ⊆ range p ∪ range q
  hence set ys ⊆ range p ∪ range q
   by simp
  ultimately have ?U′ ⊆ range Some ⟶
    (map (inv p) [x←(xs @ [y]) @ ys. x ∈ range p], inv p ‘ Y) ∈ failures P ⟶
    (map (inv p) [x←xs @ [y]. x ∈ range p] @
     map (inv p) (ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys),
     inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F ?U′ ys Y) ∈ failures P ..
  moreover assume D: U ⊆ range Some
  hence ?U′ ⊆ range Some
  proof (cases ∃ u ∈ U. (u, ?D′ y) ∈ ?I′, simp-all add: sinks-aux-single-event)
    have y ∈ range p ∪ range q
     using C by simp
    thus ?D′ y ∈ range Some
     by (cases y ∈ range p, simp-all)
  qed
  ultimately have
   (map (inv p) [x←(xs @ [y]) @ ys. x ∈ range p], inv p ‘ Y) ∈ failures P ⟶
    (map (inv p) [x←xs @ [y]. x ∈ range p] @
     map (inv p) (ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys),
     inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F ?U′ ys Y) ∈ failures P ..
  moreover assume
   (map (inv p) [x←xs @ y # ys. x ∈ range p], inv p ‘ Y) ∈ failures P
  ultimately have
   (map (inv p) [x←xs @ [y]. x ∈ range p] @
     map (inv p) (ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys),
```

```
    inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F ?U′ ys Y) ∈ failures P
  by simp
 hence
  (map (inv p) [x←xs. x ∈ range p] @
    map (inv p) ((if y ∈ range p then [y] else []) @
      ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys),
    inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F ?U′ ys Y) ∈ failures P
  by (cases y ∈ range p, simp-all)
 with A have
  (map (inv p) [x←xs. x ∈ range p] @
    ipurge-tr-aux I D (the ‘ U) (map (inv p) ((if y ∈ range p then [y] else []) @
      ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys)),
    ipurge-ref-aux I D (the ‘ U) (map (inv p) ((if y ∈ range p then [y] else []) @
      ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys))
    (inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F ?U′ ys Y)) ∈ failures P
  by (rule ipurge-tr-ref-aux-failures-general)
 moreover have
  ipurge-tr-aux I D (the ‘ U) (map (inv p) ((if y ∈ range p then [y] else []) @
     ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys)) =
   map (inv p) (ipurge-tr-aux ?I′ ?D′ U ((if y ∈ range p then [y] else []) @
     ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys))
  by (rule con-comp-ipurge-tr-aux, simp-all add:
   D ipurge-tr-aux-foldr-eq [symmetric], blast)
 moreover have
  ipurge-ref-aux I D (the ‘ U) (map (inv p) ((if y ∈ range p then [y] else []) @
     ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys))
     (inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F ?U′ ys Y) =
   inv p ‘ ipurge-ref-aux ?I′ ?D′ U ((if y ∈ range p then [y] else []) @
     ipurge-tr-aux-foldr ?I′ ?D′ ?F ?U′ ys)
     (ipurge-ref-aux-foldr ?I′ ?D′ ?F ?U′ ys Y)
 proof (rule con-comp-ipurge-ref-aux, simp-all add:
  D ipurge-tr-aux-foldr-eq [symmetric] ipurge-ref-aux-foldr-eq [symmetric], blast)
   have ipurge-ref-aux ?I′ ?D′ ?U′ ys Y ⊆ Y
    by (rule ipurge-ref-aux-subset)
   thus ipurge-ref-aux ?I′ ?D′ ?U′ ys Y ⊆ range p
    using B by simp
 qed
 ultimately show
  (map (inv p) [x←xs. x ∈ range p] @
    map (inv p) (ipurge-tr-aux-foldr ?I′ ?D′ ?F U (y # ys)),
    inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F U (y # ys) Y) ∈ failures P
  by simp
qed
```

1.4 Conservation of noninterference security under concurrent composition

Everything is now ready for proving the target security conservation theorem. It states that for any two processes P, Q being secure with respect to the noninterference policy *I* and the event-domain maps *D*, *E*, their concurrent composition *P* ∥ *Q* <*p*, *q*> is secure with respect to the noninterference policy *con-comp-pol I* and the event-domain map *con-comp-map D E p q*, provided that condition *consistent-maps D E p q* is satisfied.

The only assumption, in addition to the security of the input processes, is the consistency of the respective event-domain maps. Particularly, this assumption permits to solve the proof obligations concerning the latter input process by just swapping *D* for *E* and *p* for *q* in the term *con-comp-map D E p q* and then applying the corresponding lemmas proven for the former input process.

```
lemma con-comp-secure-del-aux-1:
  assumes
    A: secure P I D and
    B: y ∈ range p ∨ y ∈ range q and
    C: set ys ⊆ range p ∪ range q and
    D: Y ⊆ range p and
    E: (map (inv p) [x←xs @ y # ys. x ∈ range p], inv p ‘ Y) ∈ failures P
  shows
   (map (inv p) [x←xs @ ipurge-tr (con-comp-pol I) (con-comp-map D E p q)
      (con-comp-map D E p q y) ys. x ∈ range p],
     inv p ‘ ipurge-ref (con-comp-pol I) (con-comp-map D E p q)
      (con-comp-map D E p q y) ys Y) ∈ failures P
    (is (map (inv p) [x←xs @ ipurge-tr ?I′ ?D′ - -. -], -) ∈ -)
proof (simp add:
 ipurge-tr-aux-single-dom [symmetric] ipurge-ref-aux-single-dom [symmetric]
 ipurge-tr-aux-foldr-eq ipurge-ref-aux-foldr-eq [where P = λx. x ∈ range p])
  have (map (inv p) [x←xs @ [y]. x ∈ range p] @
    map (inv p) (ipurge-tr-aux-foldr ?I′ ?D′ (λx. x ∈ range p) {?D′ y} ys),
    inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ (λx. x ∈ range p) {?D′ y} ys Y)
      ∈ failures P
    (is (- @ map (inv p) (ipurge-tr-aux-foldr - - ?F - -), -) ∈ -)
  proof (rule con-comp-secure-aux [OF A D C])
    show {?D′ y} ⊆ range Some
     using B by (cases y ∈ range p, simp-all)
  next
    show (map (inv p) [x←(xs @ [y]) @ ys. x ∈ range p], inv p ‘ Y) ∈ failures P
     using E by simp
  qed
  thus (map (inv p) [x←xs. x ∈ range p] @
    map (inv p) (ipurge-tr-aux-foldr ?I′ ?D′ ?F {?D′ y} ys),
    inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F {?D′ y} ys Y)
      ∈ failures P
  proof (cases y ∈ range p, simp-all)
    assume (map (inv p) [x←xs. x ∈ range p] @ inv p y #
      map (inv p) (ipurge-tr-aux-foldr ?I′ ?D′ ?F {Some (D (inv p y))} ys),
      inv p ‘ ipurge-ref-aux-foldr ?I′ ?D′ ?F {Some (D (inv p y))} ys Y)
```

∈ *failures P*
hence (*inv p y* #
map (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*),
inv p ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*)
∈ *futures P* (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*])
by (*simp add*: *futures-def*)
hence
(*ipurge-tr I D* (*D* (*inv p y*))
(*map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*)),
ipurge-ref I D (*D* (*inv p y*))
(*map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*))
(*inv p* ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*))
∈ *futures P* (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*])
using *A* **by** (*simp add*: *secure-def*)
hence
(*ipurge-tr-aux I D* (*the* ‘ {*Some* (*D* (*inv p y*))})
(*map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*)),
ipurge-ref-aux I D (*the* ‘ {*Some* (*D* (*inv p y*))})
(*map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*))
(*inv p* ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*))
∈ *futures P* (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*])
by (*simp add*: *ipurge-tr-aux-single-dom ipurge-ref-aux-single-dom*)
moreover have
ipurge-tr-aux I D (*the* ‘ {*Some* (*D* (*inv p y*))})
(*map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*)) =
map (*inv p*) (*ipurge-tr-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*))
by (*rule con-comp-ipurge-tr-aux*, *simp-all add*:
ipurge-tr-aux-foldr-eq [*symmetric*], *blast*)
moreover have
ipurge-ref-aux I D (*the* ‘ {*Some* (*D* (*inv p y*))})
(*map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*))
(*inv p* ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*) =
inv p ‘ *ipurge-ref-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*)
(*ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*)
proof (*rule con-comp-ipurge-ref-aux*, *simp-all add*:
ipurge-tr-aux-foldr-eq [*symmetric*] *ipurge-ref-aux-foldr-eq* [*symmetric*], *blast*)
have *ipurge-ref-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))} *ys Y* ⊆ *Y*
by (*rule ipurge-ref-aux-subset*)
thus *ipurge-ref-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))} *ys Y* ⊆ *range p*
using *D* **by** *simp*
qed
ultimately have
(*map* (*inv p*) (*ipurge-tr-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*)),
inv p ‘ *ipurge-ref-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*)
(*ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*))

∈ *futures P* (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*])
by *simp*
moreover have
ipurge-tr-aux ?I′ ?D′ {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*) =
ipurge-tr-aux-foldr ?I′ ?D′ ?F {*Some* (*D* (*inv p y*))} *ys*
by (*rule ipurge-tr-aux-foldr-subset*, *simp*)
moreover have
ipurge-ref-aux ?I′ ?D′ {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*)
(*ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*) =
ipurge-ref-aux-foldr ?I′ ?D′ ?F {*Some* (*D* (*inv p y*))} *ys Y*
by (*rule ipurge-ref-aux-foldr-subset*, *subst ipurge-tr-aux-foldr-sinks-aux*, *simp*)
ultimately have
(*map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*),
inv p ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*)
∈ *futures P* (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*])
by *simp*
thus (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @
map (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys*),
inv p ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *ys Y*)
∈ *failures P*
by (*simp add*: *futures-def*)
qed
qed

lemma *con-comp-secure-add-aux-1*:
assumes
A: *secure P I D* **and**
B: *y* ∈ *range p* ∨ *y* ∈ *range q* **and**
C: *set zs* ⊆ *range p* ∪ *range q* **and**
D: *Z* ⊆ *range p* **and**
E: (*map* (*inv p*) [*x*←*xs* @ *zs*. *x* ∈ *range p*], *inv p* ‘ *Z*) ∈ *failures P* **and**
F: *map* (*inv p*) [*x*←*xs* @ [*y*]. *x* ∈ *range p*] ∈ *traces P*
shows
(*map* (*inv p*) [*x*←*xs* @ *y* # *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *zs*. *x* ∈ *range p*],
inv p ‘ *ipurge-ref* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *zs Z*) ∈ *failures P*
(**is** (*map* (*inv p*) [*x*←*xs* @ *y* # *ipurge-tr ?I′ ?D′* - -. -], -) ∈ -)
proof −
have
(*map* (*inv p*) [*x*←(*xs* @ [*y*]) @ *ipurge-tr ?I′ ?D′* (*?D′ y*) *zs*. *x* ∈ *range p*],
inv p ‘ *ipurge-ref ?I′ ?D′* (*?D′ y*) *zs Z*) ∈ *failures P*
proof (*subst filter-append*, *simp del*: *filter-append add*:
ipurge-tr-aux-single-dom [*symmetric*] *ipurge-ref-aux-single-dom* [*symmetric*]
ipurge-tr-aux-foldr-eq ipurge-ref-aux-foldr-eq [**where** *P* = λ*x*. *x* ∈ *range p*])
have (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @
map (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′* (λ*x*. *x* ∈ *range p*) {*?D′ y*} *zs*),

```
  inv p ` ipurge-ref-aux-foldr ?I' ?D' (λx. x ∈ range p) {?D' y} zs Z)
    ∈ failures P
  (is (- @ map (inv p) (ipurge-tr-aux-foldr - - ?F - -), -) ∈ -)
proof (rule con-comp-secure-aux [OF A D C])
  show {?D' y} ⊆ range Some
   using B by (cases y ∈ range p, simp-all)
next
  show (map (inv p) [x←xs @ zs. x ∈ range p], inv p ` Z) ∈ failures P
   using E .
qed
thus (map (inv p) [x←xs @ [y]. x ∈ range p] @
  map (inv p) (ipurge-tr-aux-foldr ?I' ?D' ?F {?D' y} zs),
  inv p ` ipurge-ref-aux-foldr ?I' ?D' ?F {?D' y} zs Z)
    ∈ failures P
proof (cases y ∈ range p, simp-all)
  case True
  assume (map (inv p) [x←xs. x ∈ range p] @
    map (inv p) (ipurge-tr-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs),
    inv p ` ipurge-ref-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs Z)
    ∈ failures P
  hence
   (map (inv p) (ipurge-tr-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs),
     inv p ` ipurge-ref-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs Z)
    ∈ futures P (map (inv p) [x←xs. x ∈ range p])
   by (simp add: futures-def)
  moreover have (map (inv p) [x←xs @ [y]. x ∈ range p], {}) ∈ failures P
   using F by (rule traces-failures)
  hence ([inv p y], {}) ∈ futures P (map (inv p) [x←xs. x ∈ range p])
   using True by (simp add: futures-def)
  ultimately have
   (inv p y # ipurge-tr I D (D (inv p y))
       (map (inv p) (ipurge-tr-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs)),
     ipurge-ref I D (D (inv p y))
       (map (inv p) (ipurge-tr-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs))
       (inv p ` ipurge-ref-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs Z))
    ∈ futures P (map (inv p) [x←xs. x ∈ range p])
   using A by (simp add: secure-def)
  hence
   (inv p y # ipurge-tr-aux I D (the ` {Some (D (inv p y))})
       (map (inv p) (ipurge-tr-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs)),
     ipurge-ref-aux I D (the ` {Some (D (inv p y))})
       (map (inv p) (ipurge-tr-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs))
       (inv p ` ipurge-ref-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs Z))
    ∈ futures P (map (inv p) [x←xs. x ∈ range p])
   by (simp add: ipurge-tr-aux-single-dom ipurge-ref-aux-single-dom)
  moreover have
   ipurge-tr-aux I D (the ` {Some (D (inv p y))})
     (map (inv p) (ipurge-tr-aux-foldr ?I' ?D' ?F {Some (D (inv p y))} zs)) =
    map (inv p) (ipurge-tr-aux ?I' ?D' {Some (D (inv p y))}
```

(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs*))
by (*rule con-comp-ipurge-tr-aux*, *simp-all add*:
ipurge-tr-aux-foldr-eq [*symmetric*], *blast*)
moreover have
ipurge-ref-aux I D (*the* ‘ {*Some* (*D* (*inv p y*))})
(*map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs*))
(*inv p* ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs Z*) =
inv p ‘ *ipurge-ref-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs*)
(*ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs Z*)
proof (*rule con-comp-ipurge-ref-aux*, *simp-all add*:
ipurge-tr-aux-foldr-eq [*symmetric*] *ipurge-ref-aux-foldr-eq* [*symmetric*], *blast*)
have *ipurge-ref-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))} *zs Z* ⊆ *Z*
by (*rule ipurge-ref-aux-subset*)
thus *ipurge-ref-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))} *zs Z* ⊆ *range p*
using *D* **by** *simp*
qed
ultimately have
(*inv p y* # *map* (*inv p*) (*ipurge-tr-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs*)),
inv p ‘ *ipurge-ref-aux ?I′ ?D′* {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs*)
(*ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs Z*))
∈ *futures P* (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*])
by *simp*
moreover have
ipurge-tr-aux ?I′ ?D′ {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs*) =
ipurge-tr-aux-foldr ?I′ ?D′ ?F {*Some* (*D* (*inv p y*))} *zs*
by (*rule ipurge-tr-aux-foldr-subset*, *simp*)
moreover have
ipurge-ref-aux ?I′ ?D′ {*Some* (*D* (*inv p y*))}
(*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs*)
(*ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs Z*) =
ipurge-ref-aux-foldr ?I′ ?D′ ?F {*Some* (*D* (*inv p y*))} *zs Z*
by (*rule ipurge-ref-aux-foldr-subset*, *subst ipurge-tr-aux-foldr-sinks-aux*, *simp*)
ultimately have
(*inv p y* # *map* (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F*
{*Some* (*D* (*inv p y*))} *zs*),
inv p ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F*
{*Some* (*D* (*inv p y*))} *zs Z*)
∈ *futures P* (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*])
by *simp*
thus (*map* (*inv p*) [*x*←*xs*. *x* ∈ *range p*] @ *inv p y* #
map (*inv p*) (*ipurge-tr-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs*),
inv p ‘ *ipurge-ref-aux-foldr ?I′ ?D′ ?F* {*Some* (*D* (*inv p y*))} *zs Z*)
∈ *failures P*
by (*simp add*: *futures-def*)
qed

  qed
  thus *?thesis*
   by *simp*
qed

lemma *con-comp-consistent-maps*:
 consistent-maps D E p q $\Longrightarrow$ *con-comp-map D E p q* = *con-comp-map E D q p*
using [[*simproc del*: *defined-all*]] **proof** (*simp add*: *consistent-maps-def*, *rule ext*)
  fix *x*
  assume *A*: $\forall x \in$ *range p* $\cap$ *range q*. *D* (*inv p x*) = *E* (*inv q x*)
  show *con-comp-map D E p q x* = *con-comp-map E D q p x*
  proof (*rule con-comp-map.cases* [*of* (*D*, *E*, *p*, *q*, *x*)], *simp-all*, (*erule conjE*)+)
    fix *p' q' D' E' x'*
    assume
      B: *p* = *p'* **and**
      C: *q* = *q'* **and**
      D: *D* = *D'* **and**
      E: *E* = *E'* **and**
      F: *x'* $\in$ *range p'*
    show *Some* (*D'* (*inv p' x'*)) = *con-comp-map E' D' q' p' x'*
    proof (*cases x'* $\in$ *range q'*, *simp-all add*: *F*)
      case *True*
      with *F* **have** *x'* $\in$ *range p'* $\cap$ *range q'* **..**
      hence *x'* $\in$ *range p* $\cap$ *range q*
       using *B* **and** *C* **by** *simp*
      with *A* **have** *D* (*inv p x'*) = *E* (*inv q x'*) **..**
      thus *D'* (*inv p' x'*) = *E'* (*inv q' x'*)
       using *B* **and** *C* **and** *D* **and** *E* **by** *simp*
    qed
  qed
qed

lemma *con-comp-secure-del-aux-2*:
  assumes *A*: *consistent-maps D E p q*
  shows
   secure Q I E $\Longrightarrow$
    y $\in$ *range p* $\vee$ *y* $\in$ *range q* $\Longrightarrow$
    set ys $\subseteq$ *range p* $\cup$ *range q* $\Longrightarrow$
    Y $\subseteq$ *range q* $\Longrightarrow$
    (*map* (*inv q*) [*x*←*xs* @ *y* # *ys*. *x* $\in$ *range q*], *inv q* ‘ *Y*) $\in$ *failures Q* $\Longrightarrow$
      (*map* (*inv q*) [*x*←*xs* @ *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
         (*con-comp-map D E p q y*) *ys*. *x* $\in$ *range q*],
       inv q ‘ *ipurge-ref* (*con-comp-pol I*) (*con-comp-map D E p q*)
         (*con-comp-map D E p q y*) *ys Y*) $\in$ *failures Q*
proof (*simp only*: *con-comp-consistent-maps* [*OF A*], *rule con-comp-secure-del-aux-1*)
qed (*simp-all*, *blast*+)

lemma *con-comp-secure-add-aux-2*:
  assumes *A*: *consistent-maps D E p q*

shows
secure Q I E ⟹
y ∈ *range p* ∨ *y* ∈ *range q* ⟹
set zs ⊆ *range p* ∪ *range q* ⟹
Z ⊆ *range q* ⟹
(*map* (*inv q*) [*x*←*xs* @ *zs*. *x* ∈ *range q*], *inv q* ‘ *Z*) ∈ *failures Q* ⟹
map (*inv q*) [*x*←*xs* @ [*y*]. *x* ∈ *range q*] ∈ *traces Q* ⟹
(*map* (*inv q*) [*x*←*xs* @ *y* # *ipurge-tr* (*con-comp-pol I*)
(*con-comp-map D E p q*) (*con-comp-map D E p q y*) *zs*. *x* ∈ *range q*],
inv q ‘ *ipurge-ref* (*con-comp-pol I*)
(*con-comp-map D E p q*) (*con-comp-map D E p q y*) *zs Z*) ∈ *failures Q*
proof (*simp only*: *con-comp-consistent-maps* [*OF A*], *rule con-comp-secure-add-aux-1*)
qed (*simp-all*, *blast*+)

lemma *con-comp-secure-del-case-1*:
assumes
A: *consistent-maps D E p q* **and**
B: *secure P I D* **and**
C: *secure Q I E*
shows
∃ *R S T*.
Y = *R* ∪ *S* ∪ *T* ∧
(*y* ∈ *range p* ∨ *y* ∈ *range q*) ∧
set xs ⊆ *range p* ∪ *range q* ∧
set ys ⊆ *range p* ∪ *range q* ∧
R ⊆ *range p* ∧
S ⊆ *range q* ∧
T ⊆ − *range p* ∧
T ⊆ − *range q* ∧
(*map* (*inv p*) [*x*←*xs* @ *y* # *ys*. *x* ∈ *range p*], *inv p* ‘ *R*) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs* @ *y* # *ys*. *x* ∈ *range q*], *inv q* ‘ *S*) ∈ *failures Q* ⟹
∃ *R S T*.
ipurge-ref (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys Y* = *R* ∪ *S* ∪ *T* ∧
set xs ⊆ *range p* ∪ *range q* ∧
set (*ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys*) ⊆ *range p* ∪ *range q* ∧
R ⊆ *range p* ∧
S ⊆ *range q* ∧
T ⊆ − *range p* ∧
T ⊆ − *range q* ∧
(*map* (*inv p*) [*x*←*xs* @ *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys*. *x* ∈ *range p*], *inv p* ‘ *R*) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs* @ *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys*. *x* ∈ *range q*], *inv q* ‘ *S*) ∈ *failures Q*
(**is** - ⟹ ∃ - - -. *ipurge-ref ?I′ ?D′* - - - = - ∧ -)
proof ((*erule exE*)+, (*erule conjE*)+)
fix *R S T*
assume

D: $Y = R \cup S \cup T$ **and**
E: $y \in range\ p \lor y \in range\ q$ **and**
F: $set\ xs \subseteq range\ p \cup range\ q$ **and**
G: $set\ ys \subseteq range\ p \cup range\ q$ **and**
H: $R \subseteq range\ p$ **and**
I: $S \subseteq range\ q$ **and**
J: $T \subseteq -\ range\ p$ **and**
K: $T \subseteq -\ range\ q$ **and**
L: $(map\ (inv\ p)\ [x \leftarrow xs\ @\ y\ \#\ ys.\ x \in range\ p],\ inv\ p\ `\ R) \in failures\ P$ **and**
M: $(map\ (inv\ q)\ [x \leftarrow xs\ @\ y\ \#\ ys.\ x \in range\ q],\ inv\ q\ `\ S) \in failures\ Q$
show *?thesis*
proof (*rule-tac x* = *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys R* **in** *exI*,
rule-tac x = *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys S* **in** *exI*, *rule-tac x* = {} **in** *exI*,
(*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+, *simp-all del*: *filter-append*)
have *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys Y* =
ipurge-ref ?I′ ?D′ (*?D′ y*) *ys* $(R \cup S \cup T)$
using *D* **by** *simp*
hence *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys Y* =
ipurge-ref ?I′ ?D′ (*?D′ y*) *ys R* ∪
ipurge-ref ?I′ ?D′ (*?D′ y*) *ys S* ∪
ipurge-ref ?I′ ?D′ (*?D′ y*) *ys T*
by (*simp add*: *ipurge-ref-distrib-union*)
moreover have *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys T* = {}
proof (*rule ipurge-ref-empty* [*of ?D′ y*], *simp*, *insert E*,
cases $y \in range\ p$, *simp-all*)
fix *x*
assume *N*: $x \in T$
with *J* **have** $x \in -\ range\ p$ **..**
moreover have $x \in -\ range\ q$
using *K* **and** *N* **..**
ultimately have *?D′ x* = *None*
by *simp*
thus (*Some* (*D* (*inv p y*)), *?D′ x*) ∈ *?I′*
by (*simp add*: *con-comp-pol-def*)
next
fix *x*
assume *N*: $x \in T$
with *J* **have** $x \in -\ range\ p$ **..**
moreover have $x \in -\ range\ q$
using *K* **and** *N* **..**
ultimately have *?D′ x* = *None*
by *simp*
thus (*Some* (*E* (*inv q y*)), *?D′ x*) ∈ *?I′*
by (*simp add*: *con-comp-pol-def*)
qed
ultimately show *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys Y* =
ipurge-ref ?I′ ?D′ (*?D′ y*) *ys R* ∪
ipurge-ref ?I′ ?D′ (*?D′ y*) *ys S*
by *simp*

next
  show *set xs* ⊆ *range p* ∪ *range q*
   using *F* **.**
next
  have *set* (*ipurge-tr ?I′ ?D′* (*?D′ y*) *ys*) ⊆ *set ys*
   by (*rule ipurge-tr-set*)
  thus *set* (*ipurge-tr ?I′ ?D′* (*?D′ y*) *ys*) ⊆ *range p* ∪ *range q*
   using *G* **by** *simp*
next
  have *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys R* ⊆ *R*
   by (*rule ipurge-ref-subset*)
  thus *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys R* ⊆ *range p*
   using *H* **by** *simp*
next
  have *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys S* ⊆ *S*
   by (*rule ipurge-ref-subset*)
  thus *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys S* ⊆ *range q*
   using *I* **by** *simp*
next
  show (*map* (*inv p*) [*x*←*xs* @ *ipurge-tr ?I′ ?D′* (*?D′ y*) *ys*. *x* ∈ *range p*],
    inv p ‘ *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys R*) ∈ *failures P*
   by (*rule con-comp-secure-del-aux-1* [*OF B E G H L*])
next
  show (*map* (*inv q*) [*x*←*xs* @ *ipurge-tr ?I′ ?D′* (*?D′ y*) *ys*. *x* ∈ *range q*],
    inv q ‘ *ipurge-ref ?I′ ?D′* (*?D′ y*) *ys S*) ∈ *failures Q*
   by (*rule con-comp-secure-del-aux-2* [*OF A C E G I M*])
qed
qed

lemma *con-comp-secure-del-case-2*:
 assumes
  A: *consistent-maps D E p q* **and**
  B: *secure P I D* **and**
  C: *secure Q I E*
 shows
  ∃ *xs′*.
   (∃ *ys′*. *xs* @ *y* # *ys* = *xs′* @ *ys′*) ∧
   set xs′ ⊆ *range p* ∪ *range q* ∧
   map (*inv p*) [*x*←*xs′*. *x* ∈ *range p*] ∈ *divergences P* ∧
   map (*inv q*) [*x*←*xs′*. *x* ∈ *range q*] ∈ *divergences Q* ⟹
  (∃ *R S T*.
   ipurge-ref (*con-comp-pol I*) (*con-comp-map D E p q*)
    (*con-comp-map D E p q y*) *ys Y* = *R* ∪ *S* ∪ *T* ∧
   set xs ⊆ *range p* ∪ *range q* ∧
   set (*ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
    (*con-comp-map D E p q y*) *ys*) ⊆ *range p* ∪ *range q* ∧
   R ⊆ *range p* ∧
   S ⊆ *range q* ∧
   T ⊆ − *range p* ∧

$T \subseteq -$ *range q* ∧
(*map* (*inv p*) [*x*←*xs* @ *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys*. *x* ∈ *range p*], *inv p* ' *R*) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs* @ *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys*. *x* ∈ *range q*], *inv q* ' *S*) ∈ *failures Q*) ∨
(∃ *xs'*.
(∃ *ys'*. *xs* @ *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys* = *xs'* @ *ys'*) ∧
set xs' ⊆ *range p* ∪ *range q* ∧
map (*inv p*) [*x*←*xs'*. *x* ∈ *range p*] ∈ *divergences P* ∧
map (*inv q*) [*x*←*xs'*. *x* ∈ *range q*] ∈ *divergences Q*)
(**is** - ⟹ (∃ *R S T*. *?F R S T ys*) ∨ *?G*)
proof (*erule exE*, (*erule conjE*)+, *erule exE*)
fix *xs' ys'*
assume
D: *set xs'* ⊆ *range p* ∪ *range q* **and**
E: *map* (*inv p*) [*x*←*xs'*. *x* ∈ *range p*] ∈ *divergences P* **and**
F: *map* (*inv q*) [*x*←*xs'*. *x* ∈ *range q*] ∈ *divergences Q* **and**
G: *xs* @ *y* # *ys* = *xs'* @ *ys'*
show *?thesis*
proof (*cases length xs* < *length xs'*, *rule disjI1*, *rule-tac* [*2*] *disjI2*)
case *True*
moreover have *take* (*length xs'*) (*xs* @ [*y*] @ *ys*) =
take (*length xs'*) (*xs* @ [*y*]) @ *take* (*length xs'* − *Suc* (*length xs*)) *ys*
by (*simp only*: *take-append*, *simp*)
ultimately have *take* (*length xs'*) (*xs* @ [*y*] @ *ys*) =
xs @ *y* # *take* (*length xs'* − *Suc* (*length xs*)) *ys*
(**is** - = - @ - # *?vs*)
by *simp*
moreover have *take* (*length xs'*) (*xs* @ [*y*] @ *ys*) =
take (*length xs'*) (*xs'* @ *ys'*)
using *G* **by** *simp*
ultimately have *H*: *xs* @ *y* # *?vs* = *xs'*
by *simp*
moreover have *y* ∈ *set* (*xs* @ *y* # *?vs*)
by *simp*
ultimately have *y* ∈ *set xs'*
by *simp*
with *D* **have** *I*: *y* ∈ *range p* ∪ *range q* **..**
have *set xs* ⊆ *set* (*xs* @ *y* # *?vs*)
by *auto*
hence *set xs* ⊆ *set xs'*
using *H* **by** *simp*
hence *J*: *set xs* ⊆ *range p* ∪ *range q*
using *D* **by** *simp*
have ∃ *R S T*. *?F R S T* [*x*←*ys*. *x* ∈ *range p* ∪ *range q*]
(**is** ∃ - - -. *ipurge-ref ?I' ?D'* - - - = - ∧ -)
proof (*rule con-comp-secure-del-case-1* [*OF A B C*],
rule-tac x = *range p* ∩ *Y* **in** *exI*, *rule-tac x* = *range q* ∩ *Y* **in** *exI*,

rule-tac x = − *range p* ∩ − *range q* ∩ *Y* **in** *exI*,
(*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+, *simp-all del*: *filter-append*)
show *Y* = *range p* ∩ *Y* ∪ *range q* ∩ *Y* ∪ − *range p* ∩ − *range q* ∩ *Y*
by *blast*
next
show *y* ∈ *range p* ∨ *y* ∈ *range q*
using *I* **by** *simp*
next
show *set xs* ⊆ *range p* ∪ *range q*
using *J* **.**
next
show {*x* ∈ *set ys*. *x* ∈ *range p* ∨ *x* ∈ *range q*} ⊆ *range p* ∪ *range q*
by *blast*
next
show − *range p* ∩ − *range q* ∩ *Y* ⊆ − *range p*
by *blast*
next
show − *range p* ∩ − *range q* ∩ *Y* ⊆ − *range q*
by *blast*
next
have *map* (*inv p*) [*x*←*xs* @ *y* # *?vs*. *x* ∈ *range p*] ∈ *divergences P*
using *E* **and** *H* **by** *simp*
hence *map* (*inv p*) [*x*←*xs* @ *y* # *?vs*. *x* ∈ *range p*] @
map (*inv p*) [*x*←*drop* (*length xs*′ − *Suc* (*length xs*)) *ys*. *x* ∈ *range p*]
∈ *divergences P*
(**is** - @ *map* (*inv p*) [*x*←*?ws*. -] ∈ -)
by (*rule process-rule-5-general*)
hence *map* (*inv p*) [*x*←(*xs* @ *y* # *?vs*) @ *?ws*. *x* ∈ *range p*] ∈ *divergences P*
by (*subst filter-append*, *simp*)
hence *map* (*inv p*) [*x*←(*xs* @ [*y*]) @ *ys*. *x* ∈ *range p*] ∈ *divergences P*
by *simp*
hence *map* (*inv p*) [*x*←(*xs* @ [*y*]) @ [*x*←*ys*. *x* ∈ *range p* ∨ *x* ∈ *range q*].
x ∈ *range p*] ∈ *divergences P*
proof (*subst* (*asm*) *filter-append*, *subst filter-append*, *subst filter-filter*)
qed (*subgoal-tac* (λ*x*. (*x* ∈ *range p* ∨ *x* ∈ *range q*) ∧ *x* ∈ *range p*) =
(λ*x*. *x* ∈ *range p*), *simp*, *blast*)
hence *map* (*inv p*) [*x*←*xs* @ *y* # [*x*←*ys*. *x* ∈ *range p* ∨ *x* ∈ *range q*].
x ∈ *range p*] ∈ *divergences P*
by *simp*
thus (*map* (*inv p*) [*x*←*xs* @ *y* # [*x*←*ys*. *x* ∈ *range p* ∨ *x* ∈ *range q*].
x ∈ *range p*], *inv p* ‘ (*range p* ∩ *Y*)) ∈ *failures P*
by (*rule process-rule-6*)
next
have *map* (*inv q*) [*x*←*xs* @ *y* # *?vs*. *x* ∈ *range q*] ∈ *divergences Q*
using *F* **and** *H* **by** *simp*
hence *map* (*inv q*) [*x*←*xs* @ *y* # *?vs*. *x* ∈ *range q*] @
map (*inv q*) [*x*←*drop* (*length xs*′ − *Suc* (*length xs*)) *ys*. *x* ∈ *range q*]
∈ *divergences Q*
(**is** - @ *map* (*inv q*) [*x*←*?ws*. -] ∈ -)

by (*rule process-rule-5-general*)
hence *map* (*inv q*) [*x*←(*xs* @ *y* # *?vs*) @ *?ws*. *x* ∈ *range q*] ∈ *divergences Q*
by (*subst filter-append*, *simp*)
hence *map* (*inv q*) [*x*←(*xs* @ [*y*]) @ *ys*. *x* ∈ *range q*] ∈ *divergences Q*
by *simp*
hence *map* (*inv q*) [*x*←(*xs* @ [*y*]) @ [*x*←*ys*. *x* ∈ *range p* ∨ *x* ∈ *range q*].
x ∈ *range q*] ∈ *divergences Q*
proof (*subst* (*asm*) *filter-append*, *subst filter-append*, *subst filter-filter*)
qed (*subgoal-tac* (λ*x*. (*x* ∈ *range p* ∨ *x* ∈ *range q*) ∧ *x* ∈ *range q*) =
(λ*x*. *x* ∈ *range q*), *simp*, *blast*)
hence *map* (*inv q*) [*x*←*xs* @ *y* # [*x*←*ys*. *x* ∈ *range p* ∨ *x* ∈ *range q*].
x ∈ *range q*] ∈ *divergences Q*
by *simp*
thus (*map* (*inv q*) [*x*←*xs* @ *y* # [*x*←*ys*. *x* ∈ *range p* ∨ *x* ∈ *range q*].
x ∈ *range q*], *inv q* ‘ (*range q* ∩ *Y*)) ∈ *failures Q*
by (*rule process-rule-6*)
qed
then obtain *R* **and** *S* **and** *T* **where**
?F R S T [*x*←*ys*. *x* ∈ *range p* ∪ *range q*]
by *blast*
thus ∃ *R S T*. *?F R S T ys*
proof (*rule-tac x* = *R* **in** *exI*, *rule-tac x* = *S* **in** *exI*, *rule-tac x* = *T* **in** *exI*)
qed (*simp only*: *con-comp-ipurge-tr-filter con-comp-ipurge-ref-filter*)
next
let
?I′ = *con-comp-pol I* **and**
?D′ = *con-comp-map D E p q*
case *False*
moreover have *xs* @ *ipurge-tr ?I′ ?D′* (*?D′ y*) *ys* =
take (*length xs′*) (*xs* @ *ipurge-tr ?I′ ?D′* (*?D′ y*) *ys*) @
drop (*length xs′*) (*xs* @ *ipurge-tr ?I′ ?D′* (*?D′ y*) *ys*)
(**is** - = - @ *?vs*)
by (*simp only*: *append-take-drop-id*)
ultimately have *xs* @ *ipurge-tr ?I′ ?D′* (*?D′ y*) *ys* =
take (*length xs′*) (*xs* @ *y* # *ys*) @ *?vs*
by *simp*
hence *H*: *xs* @ *ipurge-tr ?I′ ?D′* (*?D′ y*) *ys* = *xs′* @ *?vs*
using *G* **by** *simp*
show *?G*
proof (*rule-tac x* = *xs′* **in** *exI*, *rule conjI*, *rule-tac x* = *?vs* **in** *exI*)
qed (*subst H*, *simp-all add*: *D E F*)
qed
qed

lemma *con-comp-secure-add-case-1*:
assumes
A: *consistent-maps D E p q* **and**
B: *secure P I D* **and**
C: *secure Q I E* **and**

D: (*xs* @ *y* # *ys*, *Y*) ∈ *con-comp-failures P Q p q* **and**
E: *y* ∈ *range p* ∨ *y* ∈ *range q*
shows
∃ *R S T*.
Z = *R* ∪ *S* ∪ *T* ∧
set xs ⊆ *range p* ∪ *range q* ∧
set zs ⊆ *range p* ∪ *range q* ∧
R ⊆ *range p* ∧
S ⊆ *range q* ∧
T ⊆ − *range p* ∧
T ⊆ − *range q* ∧
(*map* (*inv p*) [*x*←*xs* @ *zs*. *x* ∈ *range p*], *inv p* ' *R*) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs* @ *zs*. *x* ∈ *range q*], *inv q* ' *S*) ∈ *failures Q* ⟹
∃ *R S T*.
ipurge-ref (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *zs Z* = *R* ∪ *S* ∪ *T* ∧
set xs ⊆ *range p* ∪ *range q* ∧
set (*ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *zs*) ⊆ *range p* ∪ *range q* ∧
R ⊆ *range p* ∧
S ⊆ *range q* ∧
T ⊆ − *range p* ∧
T ⊆ − *range q* ∧
(*map* (*inv p*) [*x*←*xs* @ *y* # *ipurge-tr* (*con-comp-pol I*)
(*con-comp-map D E p q*) (*con-comp-map D E p q y*) *zs*. *x* ∈ *range p*],
inv p ' *R*) ∈ *failures P* ∧
(*map* (*inv q*) [*x*←*xs* @ *y* # *ipurge-tr* (*con-comp-pol I*)
(*con-comp-map D E p q*) (*con-comp-map D E p q y*) *zs*. *x* ∈ *range q*],
inv q ' *S*) ∈ *failures Q*
(**is** - ⟹ ∃ - - -. *ipurge-ref ?I′ ?D′* - - - = - ∧ -)
proof ((*erule exE*)+, (*erule conjE*)+)
fix *R S T*
assume
F: *Z* = *R* ∪ *S* ∪ *T* **and**
G: *set xs* ⊆ *range p* ∪ *range q* **and**
H: *set zs* ⊆ *range p* ∪ *range q* **and**
I: *R* ⊆ *range p* **and**
J: *S* ⊆ *range q* **and**
K: *T* ⊆ − *range p* **and**
L: *T* ⊆ − *range q* **and**
M: (*map* (*inv p*) [*x*←*xs* @ *zs*. *x* ∈ *range p*], *inv p* ' *R*) ∈ *failures P* **and**
N: (*map* (*inv q*) [*x*←*xs* @ *zs*. *x* ∈ *range q*], *inv q* ' *S*) ∈ *failures Q*
show *?thesis*
proof (*rule-tac x* = *ipurge-ref ?I′ ?D′* (*?D′ y*) *zs R* **in** *exI*,
rule-tac x = *ipurge-ref ?I′ ?D′* (*?D′ y*) *zs S* **in** *exI*, *rule-tac x* = {} **in** *exI*,
(*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+, *simp-all del*: *filter-append*)
have *ipurge-ref ?I′ ?D′* (*?D′ y*) *zs Z* =
ipurge-ref ?I′ ?D′ (*?D′ y*) *zs* (*R* ∪ *S* ∪ *T*)
using *F* **by** *simp*

```
  hence ipurge-ref ?I' ?D' (?D' y) zs Z =
    ipurge-ref ?I' ?D' (?D' y) zs R ∪
    ipurge-ref ?I' ?D' (?D' y) zs S ∪
    ipurge-ref ?I' ?D' (?D' y) zs T
   by (simp add: ipurge-ref-distrib-union)
  moreover have ipurge-ref ?I' ?D' (?D' y) zs T = {}
  proof (rule ipurge-ref-empty [of ?D' y], simp, insert E,
   cases y ∈ range p, simp-all)
    fix x
    assume O: x ∈ T
    with K have x ∈ − range p ..
    moreover have x ∈ − range q
     using L and O ..
    ultimately have ?D' x = None
     by simp
    thus (Some (D (inv p y)), ?D' x) ∈ ?I'
     by (simp add: con-comp-pol-def)
  next
    fix x
    assume O: x ∈ T
    with K have x ∈ − range p ..
    moreover have x ∈ − range q
     using L and O ..
    ultimately have ?D' x = None
     by simp
    thus (Some (E (inv q y)), ?D' x) ∈ ?I'
     by (simp add: con-comp-pol-def)
  qed
  ultimately show ipurge-ref ?I' ?D' (?D' y) zs Z =
    ipurge-ref ?I' ?D' (?D' y) zs R ∪
    ipurge-ref ?I' ?D' (?D' y) zs S
   by simp
next
  show set xs ⊆ range p ∪ range q
   using G .
next
  have set (ipurge-tr ?I' ?D' (?D' y) zs) ⊆ set zs
   by (rule ipurge-tr-set)
  thus set (ipurge-tr ?I' ?D' (?D' y) zs) ⊆ range p ∪ range q
   using H by simp
next
  have ipurge-ref ?I' ?D' (?D' y) zs R ⊆ R
   by (rule ipurge-ref-subset)
  thus ipurge-ref ?I' ?D' (?D' y) zs R ⊆ range p
   using I by simp
next
  have ipurge-ref ?I' ?D' (?D' y) zs S ⊆ S
   by (rule ipurge-ref-subset)
  thus ipurge-ref ?I' ?D' (?D' y) zs S ⊆ range q
```

```
      using J by simp
  next
    have map (inv p) [x←xs @ y # ys. x ∈ range p] ∈ traces P ∧
      map (inv q) [x←xs @ y # ys. x ∈ range q] ∈ traces Q
     using D by (rule con-comp-failures-traces)
    hence map (inv p) [x←(xs @ [y]) @ ys. x ∈ range p] ∈ traces P
     by simp
    hence map (inv p) [x←xs @ [y]. x ∈ range p] @
      map (inv p) [x←ys. x ∈ range p] ∈ traces P
     by (subst (asm) filter-append, simp)
    hence map (inv p) [x←xs @ [y]. x ∈ range p] ∈ traces P
     by (rule process-rule-2-traces)
    thus (map (inv p) [x←xs @ y # ipurge-tr ?I' ?D' (?D' y) zs. x ∈ range p],
      inv p ‘ ipurge-ref ?I' ?D' (?D' y) zs R) ∈ failures P
     by (rule con-comp-secure-add-aux-1 [OF B E H I M])
  next
    have map (inv p) [x←xs @ y # ys. x ∈ range p] ∈ traces P ∧
      map (inv q) [x←xs @ y # ys. x ∈ range q] ∈ traces Q
     using D by (rule con-comp-failures-traces)
    hence map (inv q) [x←(xs @ [y]) @ ys. x ∈ range q] ∈ traces Q
     by simp
    hence map (inv q) [x←xs @ [y]. x ∈ range q] @
      map (inv q) [x←ys. x ∈ range q] ∈ traces Q
     by (subst (asm) filter-append, simp)
    hence map (inv q) [x←xs @ [y]. x ∈ range q] ∈ traces Q
     by (rule process-rule-2-traces)
    thus (map (inv q) [x←xs @ y # ipurge-tr ?I' ?D' (?D' y) zs. x ∈ range q],
      inv q ‘ ipurge-ref ?I' ?D' (?D' y) zs S) ∈ failures Q
     by (rule con-comp-secure-add-aux-2 [OF A C E H J N])
  qed
qed

lemma con-comp-secure-add-case-2:
  assumes
    A: consistent-maps D E p q and
    B: secure P I D and
    C: secure Q I E and
    D: (xs @ y # ys, Y) ∈ con-comp-failures P Q p q and
    E: y ∈ range p ∨ y ∈ range q
  shows
   ∃ xs'.
      (∃ ys'. xs @ zs = xs' @ ys') ∧
      set xs' ⊆ range p ∪ range q ∧
      map (inv p) [x←xs'. x ∈ range p] ∈ divergences P ∧
      map (inv q) [x←xs'. x ∈ range q] ∈ divergences Q ⟹
    (∃ R S T.
      ipurge-ref (con-comp-pol I) (con-comp-map D E p q)
        (con-comp-map D E p q y) zs Z = R ∪ S ∪ T ∧
      set xs ⊆ range p ∪ range q ∧
```

```
    set (ipurge-tr (con-comp-pol I) (con-comp-map D E p q)
      (con-comp-map D E p q y) zs) ⊆ range p ∪ range q ∧
    R ⊆ range p ∧
    S ⊆ range q ∧
    T ⊆ − range p ∧
    T ⊆ − range q ∧
    (map (inv p) [x←xs @ y # ipurge-tr (con-comp-pol I)
       (con-comp-map D E p q) (con-comp-map D E p q y) zs. x ∈ range p],
     inv p ‘ R) ∈ failures P ∧
    (map (inv q) [x←xs @ y # ipurge-tr (con-comp-pol I)
       (con-comp-map D E p q) (con-comp-map D E p q y) zs. x ∈ range q],
     inv q ‘ S) ∈ failures Q) ∨
   (∃ xs′.
    (∃ ys′. xs @ y # ipurge-tr (con-comp-pol I) (con-comp-map D E p q)
      (con-comp-map D E p q y) zs = xs′ @ ys′) ∧
    set xs′ ⊆ range p ∪ range q ∧
    map (inv p) [x←xs′. x ∈ range p] ∈ divergences P ∧
    map (inv q) [x←xs′. x ∈ range q] ∈ divergences Q)
  (is - ⟹ (∃ R S T. ?F R S T zs) ∨ ?G)
proof (erule exE, (erule conjE)+, erule exE)
  fix xs′ ys′
  assume
    F: set xs′ ⊆ range p ∪ range q and
    G: map (inv p) [x←xs′. x ∈ range p] ∈ divergences P and
    H: map (inv q) [x←xs′. x ∈ range q] ∈ divergences Q and
    I: xs @ zs = xs′ @ ys′
  show ?thesis
  proof (cases length xs < length xs′, rule disjI1, rule-tac [2] disjI2)
    case True
    moreover have take (length xs′) (xs @ zs) =
      take (length xs′) xs @ take (length xs′ − length xs) zs
     by simp
    ultimately have take (length xs′) (xs @ zs) =
      xs @ take (length xs′ − length xs) zs
      (is - = - @ ?vs)
     by simp
    moreover have take (length xs′) (xs @ zs) =
      take (length xs′) (xs′ @ ys′)
     using I by simp
    ultimately have J: xs @ ?vs = xs′
     by simp
    moreover have set xs ⊆ set (xs @ ?vs)
     by simp
    ultimately have set xs ⊆ set xs′
     by simp
    hence K: set xs ⊆ range p ∪ range q
     using F by simp
    have ∃ R S T. ?F R S T [x←zs. x ∈ range p ∪ range q]
     (is ∃ - - -. ipurge-ref ?I′ ?D′ - - - = - ∧ -)
```

```
proof (rule con-comp-secure-add-case-1 [OF A B C D E],
 rule-tac x = range p ∩ Z in exI, rule-tac x = range q ∩ Z in exI,
 rule-tac x = − range p ∩ − range q ∩ Z in exI,
 (subst conj-assoc [symmetric])+, (rule conjI)+, simp-all del: filter-append)
  show Z = range p ∩ Z ∪ range q ∩ Z ∪ − range p ∩ − range q ∩ Z
   by blast
next
  show set xs ⊆ range p ∪ range q
   using K .
next
  show {x ∈ set zs. x ∈ range p ∨ x ∈ range q} ⊆ range p ∪ range q
   by blast
next
  show − range p ∩ − range q ∩ Z ⊆ − range p
   by blast
next
  show − range p ∩ − range q ∩ Z ⊆ − range q
   by blast
next
  have map (inv p) [x←xs @ ?vs. x ∈ range p] ∈ divergences P
   using G and J by simp
  hence map (inv p) [x←xs @ ?vs. x ∈ range p] @
    map (inv p) [x←drop (length xs' − length xs) zs. x ∈ range p]
    ∈ divergences P
    (is - @ map (inv p) [x←?ws. -] ∈ -)
   by (rule process-rule-5-general)
  hence map (inv p) [x←(xs @ ?vs) @ ?ws. x ∈ range p] ∈ divergences P
   by (subst filter-append, simp)
  hence map (inv p) [x←xs @ zs. x ∈ range p] ∈ divergences P
   by simp
  hence map (inv p) [x←xs @ [x←zs. x ∈ range p ∨ x ∈ range q].
    x ∈ range p] ∈ divergences P
  proof (subst (asm) filter-append, subst filter-append, subst filter-filter)
  qed (subgoal-tac (λx. (x ∈ range p ∨ x ∈ range q) ∧ x ∈ range p) =
   (λx. x ∈ range p), simp, blast)
  thus (map (inv p) [x←xs @ [x←zs. x ∈ range p ∨ x ∈ range q].
    x ∈ range p], inv p ‘ (range p ∩ Z)) ∈ failures P
   by (rule process-rule-6)
next
  have map (inv q) [x←xs @ ?vs. x ∈ range q] ∈ divergences Q
   using H and J by simp
  hence map (inv q) [x←xs @ ?vs. x ∈ range q] @
    map (inv q) [x←drop (length xs' − length xs) zs. x ∈ range q]
    ∈ divergences Q
    (is - @ map (inv q) [x←?ws. -] ∈ -)
   by (rule process-rule-5-general)
  hence map (inv q) [x←(xs @ ?vs) @ ?ws. x ∈ range q] ∈ divergences Q
   by (subst filter-append, simp)
  hence map (inv q) [x←xs @ zs. x ∈ range q] ∈ divergences Q
```

by *simp*
hence *map* (*inv q*) [*x*←*xs* @ [*x*←*zs*. *x* ∈ *range p* ∨ *x* ∈ *range q*].
x ∈ *range q*] ∈ *divergences Q*
proof (*subst* (*asm*) *filter-append*, *subst filter-append*, *subst filter-filter*)
qed (*subgoal-tac* (λ*x*. (*x* ∈ *range p* ∨ *x* ∈ *range q*) ∧ *x* ∈ *range q*) =
(λ*x*. *x* ∈ *range q*), *simp*, *blast*)
thus (*map* (*inv q*) [*x*←*xs* @ [*x*←*zs*. *x* ∈ *range p* ∨ *x* ∈ *range q*].
x ∈ *range q*], *inv q* ' (*range q* ∩ *Z*)) ∈ *failures Q*
by (*rule process-rule-6*)
qed
then obtain *R* **and** *S* **and** *T* **where**
?F R S T [*x*←*zs*. *x* ∈ *range p* ∪ *range q*]
by *blast*
thus ∃ *R S T*. *?F R S T zs*
proof (*rule-tac x* = *R* **in** *exI*, *rule-tac x* = *S* **in** *exI*, *rule-tac x* = *T* **in** *exI*)
qed (*simp only*: *con-comp-ipurge-tr-filter con-comp-ipurge-ref-filter*)
next
let
?I′ = *con-comp-pol I* **and**
?D′ = *con-comp-map D E p q*
case *False*
moreover have *xs* @ *y* # *ipurge-tr ?I′ ?D′* (*?D′ y*) *zs* =
take (*length xs′*) (*xs* @ *y* # *ipurge-tr ?I′ ?D′* (*?D′ y*) *zs*) @
drop (*length xs′*) (*xs* @ *y* # *ipurge-tr ?I′ ?D′* (*?D′ y*) *zs*)
(**is** - = - @ *?vs*)
by (*simp only*: *append-take-drop-id*)
ultimately have *xs* @ *y* # *ipurge-tr ?I′ ?D′* (*?D′ y*) *zs* =
take (*length xs′*) (*xs* @ *zs*) @ *?vs*
by *simp*
hence *J*: *xs* @ *y* # *ipurge-tr ?I′ ?D′* (*?D′ y*) *zs* = *xs′* @ *?vs*
using *I* **by** *simp*
show *?G*
proof (*rule-tac x* = *xs′* **in** *exI*, *rule conjI*, *rule-tac x* = *?vs* **in** *exI*)
qed (*subst J*, *simp-all add*: *F G H*)
qed
qed

theorem *con-comp-secure*:
assumes
A: *consistent-maps D E p q* **and**
B: *secure P I D* **and**
C: *secure Q I E*
shows *secure* (*P* ∥ *Q* <*p*, *q*>) (*con-comp-pol I*) (*con-comp-map D E p q*)
proof (*simp add*: *secure-def con-comp-futures*, (*rule allI*)+, *rule impI*,
erule conjE, *rule conjI*, (*erule rev-mp*)+, *rotate-tac* [2], *erule-tac* [2] *rev-mp*)
fix *xs y ys Y zs Z*
show
(*xs* @ *zs*, *Z*) ∈ *con-comp-failures P Q p q* ⟶
(*xs* @ *y* # *ys*, *Y*) ∈ *con-comp-failures P Q p q* ⟶

(*xs* @ *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys*,
ipurge-ref (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *ys Y*)
∈ *con-comp-failures P Q p q*
(**is** - ⟶ - ⟶ (- @ *ipurge-tr ?I′ ?D′* - -, -) ∈ -)
proof ((*rule impI*)+, *thin-tac* (*xs* @ *zs*, *Z*) ∈ *con-comp-failures P Q p q*,
simp-all add: *con-comp-failures-def con-comp-divergences-def*
del: *filter-append*, *erule disjE*, *rule disjI1*)
qed (*erule con-comp-secure-del-case-1* [*OF A B C*],
rule con-comp-secure-del-case-2 [*OF A B C*])
next
fix *xs y ys Y zs Z*
assume *D*: (*xs* @ *y* # *ys*, *Y*) ∈ *con-comp-failures P Q p q*
show
(*xs* @ *zs*, *Z*) ∈ *con-comp-failures P Q p q* ⟶
(*xs* @ *y* # *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *zs*,
ipurge-ref (*con-comp-pol I*) (*con-comp-map D E p q*)
(*con-comp-map D E p q y*) *zs Z*)
∈ *con-comp-failures P Q p q*
(**is** - ⟶ (- @ - # *ipurge-tr ?I′ ?D′* - -, -) ∈ -)
proof (*rule impI*, *simp-all add*: *con-comp-failures-def con-comp-divergences-def*
del: *filter-append*, *cases y* ∈ *range p* ∨ *y* ∈ *range q*, *simp del*: *filter-append*,
erule disjE, *rule disjI1*, *rule-tac* [*3*] *disjI2*)
qed (*erule con-comp-secure-add-case-1* [*OF A B C D*], *assumption*,
erule con-comp-secure-add-case-2 [*OF A B C D*], *assumption*,
rule con-comp-failures-divergences [*OF D*], *simp-all*)
qed

1.5 Conservation of noninterference security in the absence of fake events

In what follows, it is proven that in the absence of fake events, namely if *range p* ∪ *range q* = *UNIV*, the output of the concurrent composition of two secure processes is secure with respect to the same noninterference policy enforced by the input processes, and to the event-domain map that simply associates each event to the same security domain as the corresponding events of the input processes.

More formally, for any two processes *P*, *Q* being secure with respect to the noninterference policy *I* and the event-domain maps *D*, *E*, their concurrent composition *P* ∥ *Q* <*p*, *q*> is secure with respect to the same noninterference policy *I* and the event-domain map *the* ∘ *con-comp-map D E p q*, provided that conditions *range p* ∪ *range q* = *UNIV* and *consistent-maps D E p q* are satisfied.

lemma *con-comp-sinks-range*:

```
 u ∈ range Some ⟹
  set xs ⊆ range p ∪ range q ⟹
    sinks (con-comp-pol I) (con-comp-map D E p q) u xs ⊆ range Some
by (insert con-comp-sinks-aux-range [of {u} xs p q I D E],
 simp add: sinks-aux-single-dom)

lemma con-comp-sinks-no-fake:
  assumes
    A: range p ∪ range q = UNIV and
    B: u ∈ range Some
  shows sinks I (the ∘ con-comp-map D E p q) (the u) xs =
    the ` sinks (con-comp-pol I) (con-comp-map D E p q) u xs
    (is - = the ` sinks ?I' ?D' - -)
proof (induction xs rule: rev-induct, simp)
  fix x xs
  assume C: sinks I (the ∘ ?D') (the u) xs = the ` sinks ?I' ?D' u xs
  have x ∈ range p ∪ range q
   using A by simp
  hence D: ?D' x = Some (the (?D' x))
   by (cases x ∈ range p, simp-all)
  have E: u = Some (the u)
   using B by (simp add: image-iff)
  show sinks I (the ∘ ?D') (the u) (xs @ [x]) = the ` sinks ?I' ?D' u (xs @ [x])
  proof (cases (u, ?D' x) ∈ ?I' ∨ (∃ v ∈ sinks ?I' ?D' u xs. (v, ?D' x) ∈ ?I'))
    case True
    hence sinks ?I' ?D' u (xs @ [x]) = insert (?D' x) (sinks ?I' ?D' u xs)
     by simp
    moreover have (the u, the (?D' x)) ∈ I ∨
      (∃ d ∈ sinks I (the ∘ ?D') (the u) xs. (d, the (?D' x)) ∈ I)
    proof (rule disjE [OF True], rule disjI1, rule-tac [2] disjI2)
      assume (u, ?D' x) ∈ ?I'
      hence (Some (the u), Some (the (?D' x))) ∈ ?I'
       using D and E by simp
      thus (the u, the (?D' x)) ∈ I
       by (simp add: con-comp-pol-def)
    next
      assume ∃ v ∈ sinks ?I' ?D' u xs. (v, ?D' x) ∈ ?I'
      then obtain v where F: v ∈ sinks ?I' ?D' u xs and G: (v, ?D' x) ∈ ?I' ..
      have sinks ?I' ?D' u xs ⊆ range Some
       by (rule con-comp-sinks-range, simp-all add: A B)
      hence v ∈ range Some
       using F ..
      hence v = Some (the v)
       by (simp add: image-iff)
      hence (Some (the v), Some (the (?D' x))) ∈ ?I'
       using D and G by simp
      hence (the v, the (?D' x)) ∈ I
       by (simp add: con-comp-pol-def)
      moreover have the v ∈ sinks I (the ∘ ?D') (the u) xs
```

```
      using C and F by simp
     ultimately show ∃ d ∈ sinks I (the ∘ ?D') (the u) xs.
       (d, the (?D' x)) ∈ I ..
   qed
   hence sinks I (the ∘ ?D') (the u) (xs @ [x]) =
    insert (the (?D' x)) (sinks I (the ∘ ?D') (the u) xs)
    by simp
   ultimately show ?thesis
    using C by simp
  next
   case False
   hence sinks ?I' ?D' u (xs @ [x]) = sinks ?I' ?D' u xs
    by simp
   moreover have ¬ ((the u, the (?D' x)) ∈ I ∨
    (∃ v ∈ sinks I (the ∘ ?D') (the u) xs. (v, the (?D' x)) ∈ I))
   proof (insert False, simp, erule conjE, rule conjI, rule-tac [2] ballI)
     assume (u, ?D' x) ∉ ?I'
     hence (Some (the u), Some (the (?D' x))) ∉ ?I'
      using D and E by simp
     thus (the u, the (?D' x)) ∉ I
      by (simp add: con-comp-pol-def)
   next
     fix d
     assume d ∈ sinks I (the ∘ ?D') (the u) xs
     hence d ∈ the ` sinks ?I' ?D' u xs
      using C by simp
     hence ∃ v ∈ sinks ?I' ?D' u xs. d = the v
      by (simp add: image-iff)
     then obtain v where F: v ∈ sinks ?I' ?D' u xs and G: d = the v ..
     have sinks ?I' ?D' u xs ⊆ range Some
      by (rule con-comp-sinks-range, simp-all add: A B)
     hence v ∈ range Some
      using F ..
     hence H: v = Some d
      using G by (simp add: image-iff)
     assume ∀ v ∈ sinks ?I' ?D' u xs. (v, ?D' x) ∉ ?I'
     hence (v, ?D' x) ∉ ?I'
      using F ..
     hence (Some d, Some (the (?D' x))) ∉ ?I'
      using D and H by simp
     thus (d, the (?D' x)) ∉ I
      by (simp add: con-comp-pol-def)
   qed
   hence sinks I (the ∘ ?D') (the u) (xs @ [x]) = sinks I (the ∘ ?D') (the u) xs
    by simp
   ultimately show ?thesis
    using C by simp
  qed
qed
```

lemma *con-comp-ipurge-tr-no-fake*:
assumes
A: *range p* ∪ *range q* = *UNIV* **and**
B: *u* ∈ *range Some*
shows *ipurge-tr* (*con-comp-pol I*) (*con-comp-map D E p q*) *u xs* =
ipurge-tr I (*the* ∘ *con-comp-map D E p q*) (*the u*) *xs*
(**is** *ipurge-tr ?I′ ?D′ - -* = -)
proof (*induction xs rule*: *rev-induct*, *simp*)
fix *x xs*
assume *C*: *ipurge-tr ?I′ ?D′ u xs* = *ipurge-tr I* (*the* ∘ *?D′*) (*the u*) *xs*
show *ipurge-tr ?I′ ?D′ u* (*xs* @ [*x*]) = *ipurge-tr I* (*the* ∘ *?D′*) (*the u*) (*xs* @ [*x*])
proof (*cases ?D′ x* ∈ *sinks ?I′ ?D′ u* (*xs* @ [*x*]))
case *True*
hence *ipurge-tr ?I′ ?D′ u* (*xs* @ [*x*]) = *ipurge-tr ?I′ ?D′ u xs*
by *simp*
moreover have *the* (*?D′ x*) ∈ *the* ‘ *sinks ?I′ ?D′ u* (*xs* @ [*x*])
using *True* **by** *simp*
hence *the* (*?D′ x*) ∈ *sinks I* (*the* ∘ *?D′*) (*the u*) (*xs* @ [*x*])
by (*subst con-comp-sinks-no-fake* [*OF A B*])
hence *ipurge-tr I* (*the* ∘ *?D′*) (*the u*) (*xs* @ [*x*]) =
ipurge-tr I (*the* ∘ *?D′*) (*the u*) *xs*
by *simp*
ultimately show *?thesis*
using *C* **by** *simp*
next
case *False*
hence *ipurge-tr ?I′ ?D′ u* (*xs* @ [*x*]) = *ipurge-tr ?I′ ?D′ u xs* @ [*x*]
by *simp*
moreover have *the* (*?D′ x*) ∉ *the* ‘ *sinks ?I′ ?D′ u* (*xs* @ [*x*])
proof
assume *the* (*?D′ x*) ∈ *the* ‘ *sinks ?I′ ?D′ u* (*xs* @ [*x*])
hence ∃ *v* ∈ *sinks ?I′ ?D′ u* (*xs* @ [*x*]). *the* (*?D′ x*) = *the v*
by (*simp add*: *image-iff*)
then obtain *v* **where**
D: *v* ∈ *sinks ?I′ ?D′ u* (*xs* @ [*x*]) **and** *E*: *the* (*?D′ x*) = *the v* **..**
have *x* ∈ *range p* ∪ *range q*
using *A* **by** *simp*
hence ∃ *d*. *?D′ x* = *Some d*
by (*cases x* ∈ *range p*, *simp-all*)
then obtain *d* **where** *?D′ x* = *Some d* **..**
moreover have *sinks ?I′ ?D′ u* (*xs* @ [*x*]) ⊆ *range Some*
by (*rule con-comp-sinks-range*, *simp-all add*: *A B*)
hence *v* ∈ *range Some*
using *D* **..**
hence ∃ *d′*. *v* = *Some d′*
by (*simp add*: *image-iff*)
then obtain *d′* **where** *v* = *Some d′* **..**
ultimately have *?D′ x* = *v*

using *E* **by** *simp*
hence *?D′ x* ∈ *sinks ?I′ ?D′ u* (*xs* @ [*x*])
using *D* **by** *simp*
thus *False*
using *False* **by** *contradiction*
qed
hence *the* (*?D′ x*) ∉ *sinks I* (*the* ∘ *?D′*) (*the u*) (*xs* @ [*x*])
by (*subst con-comp-sinks-no-fake* [*OF A B*])
hence *ipurge-tr I* (*the* ∘ *?D′*) (*the u*) (*xs* @ [*x*]) =
ipurge-tr I (*the* ∘ *?D′*) (*the u*) *xs* @ [*x*]
by *simp*
ultimately show *?thesis*
using *C* **by** *simp*
qed
qed

lemma *con-comp-ipurge-ref-no-fake*:
assumes
A: *range p* ∪ *range q* = *UNIV* **and**
B: *u* ∈ *range Some*
shows *ipurge-ref* (*con-comp-pol I*) (*con-comp-map D E p q*) *u xs X* =
ipurge-ref I (*the* ∘ *con-comp-map D E p q*) (*the u*) *xs X*
(**is** *ipurge-ref ?I′ ?D′* - - - = -)
proof (*simp add*: *ipurge-ref-def set-eq-iff*, *rule allI*,
simp-all add: *con-comp-sinks-no-fake* [*OF A B*])
fix *x*
have *x* ∈ *range p* ∪ *range q*
using *A* **by** *simp*
hence *C*: *?D′ x* = *Some* (*the* (*?D′ x*))
by (*cases x* ∈ *range p*, *simp-all*)
have *D*: *u* = *Some* (*the u*)
using *B* **by** (*simp add*: *image-iff*)
show
(*x* ∈ *X* ∧ (*u*, *?D′ x*) ∉ *con-comp-pol I* ∧
(∀ *v* ∈ *sinks ?I′ ?D′ u xs*. (*v*, *?D′ x*) ∉ *con-comp-pol I*)) =
(*x* ∈ *X* ∧ (*the u*, *the* (*?D′ x*)) ∉ *I* ∧
(∀ *v* ∈ *sinks ?I′ ?D′ u xs*. (*the v*, *the* (*?D′ x*)) ∉ *I*))
proof (*rule iffI*, (*erule-tac* [!] *conjE*)+, *simp-all*, *rule-tac* [!] *conjI*,
rule-tac [*2*] *ballI*, *rule-tac* [*4*] *ballI*)
assume (*u*, *?D′ x*) ∉ *?I′*
hence (*Some* (*the u*), *Some* (*the* (*?D′ x*))) ∉ *?I′*
using *C* **and** *D* **by** *simp*
thus (*the u*, *the* (*?D′ x*)) ∉ *I*
by (*simp add*: *con-comp-pol-def*)
next
fix *v*
assume ∀ *v* ∈ *sinks ?I′ ?D′ u xs*. (*v*, *?D′ x*) ∉ *?I′* **and**
E: *v* ∈ *sinks ?I′ ?D′ u xs*
hence (*v*, *?D′ x*) ∉ *?I′* **..**

```
    moreover have sinks ?I′ ?D′ u xs ⊆ range Some
     by (rule con-comp-sinks-range, simp-all add: A B)
    hence v ∈ range Some
     using E ..
    hence v = Some (the v)
     by (simp add: image-iff)
    ultimately have (Some (the v), Some (the (?D′ x))) ∉ ?I′
     using C by simp
    thus (the v, the (?D′ x)) ∉ I
     by (simp add: con-comp-pol-def)
  next
    assume (the u, the (?D′ x)) ∉ I
    hence (Some (the u), Some (the (?D′ x))) ∉ ?I′
     by (simp add: con-comp-pol-def)
    thus (u, ?D′ x) ∉ ?I′
     using C and D by simp
  next
    fix v
    assume ∀ v ∈ sinks ?I′ ?D′ u xs. (the v, the (?D′ x)) ∉ I and
      E: v ∈ sinks ?I′ ?D′ u xs
    hence (the v, the (?D′ x)) ∉ I ..
    hence (Some (the v), Some (the (?D′ x))) ∉ ?I′
     by (simp add: con-comp-pol-def)
    moreover have sinks ?I′ ?D′ u xs ⊆ range Some
     by (rule con-comp-sinks-range, simp-all add: A B)
    hence v ∈ range Some
     using E ..
    hence v = Some (the v)
     by (simp add: image-iff)
    ultimately show (v, ?D′ x) ∉ ?I′
     using C by simp
  qed
qed

theorem con-comp-secure-no-fake:
  assumes
    A: range p ∪ range q = UNIV and
    B: consistent-maps D E p q and
    C: secure P I D and
    D: secure Q I E
  shows secure (P ∥ Q <p, q>) I (the ∘ con-comp-map D E p q)
proof (insert con-comp-secure [OF B C D], simp add: secure-def,
 (rule allI)+, rule impI)
  fix xs y ys Y zs Z
  let
    ?I′ = con-comp-pol I and
    ?D′ = con-comp-map D E p q
  have y ∈ range p ∪ range q
   using A by simp
```

```
hence E: ?D′ y ∈ range Some
 by (cases y ∈ range p, simp-all)
assume ∀ xs y ys Y zs Z.
  (y # ys, Y) ∈ futures (P ∥ Q <p, q>) xs ∧
  (zs, Z) ∈ futures (P ∥ Q <p, q>) xs ⟶
    (ipurge-tr ?I′ ?D′ (?D′ y) ys, ipurge-ref ?I′ ?D′ (?D′ y) ys Y)
      ∈ futures (P ∥ Q <p, q>) xs ∧
    (y # ipurge-tr ?I′ ?D′ (?D′ y) zs, ipurge-ref ?I′ ?D′ (?D′ y) zs Z)
      ∈ futures (P ∥ Q <p, q>) xs and
 (y # ys, Y) ∈ futures (P ∥ Q <p, q>) xs ∧
  (zs, Z) ∈ futures (P ∥ Q <p, q>) xs
hence
 (ipurge-tr ?I′ ?D′ (?D′ y) ys, ipurge-ref ?I′ ?D′ (?D′ y) ys Y)
    ∈ futures (P ∥ Q <p, q>) xs ∧
  (y # ipurge-tr ?I′ ?D′ (?D′ y) zs, ipurge-ref ?I′ ?D′ (?D′ y) zs Z)
    ∈ futures (P ∥ Q <p, q>) xs
 by blast
thus
 (ipurge-tr I (the ∘ ?D′) (the (?D′ y)) ys,
    ipurge-ref I (the ∘ ?D′) (the (?D′ y)) ys Y) ∈ futures (P ∥ Q <p, q>) xs ∧
  (y # ipurge-tr I (the ∘ ?D′) (the (?D′ y)) zs,
    ipurge-ref I (the ∘ ?D′) (the (?D′ y)) zs Z) ∈ futures (P ∥ Q <p, q>) xs
 by (simp add: con-comp-ipurge-tr-no-fake [OF A E]
  con-comp-ipurge-ref-no-fake [OF A E])
qed

end
```

References

[1] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., 1985.

[2] A. Krauss. *Defining Recursive Functions in Isabelle/HOL.* http://isabelle.in.tum.de/website-Isabelle2016/dist/Isabelle2016/doc/functions.pdf.

[3] T. Nipkow. *A Tutorial Introduction to Structured Isar Proofs.* http://isabelle.in.tum.de/website-Isabelle2011/dist/Isabelle2011/doc/isar-overview.pdf.

[4] T. Nipkow. *Programming and Proving in Isabelle/HOL*, Feb. 2016. http://isabelle.in.tum.de/website-Isabelle2016/dist/Isabelle2016/doc/prog-prove.pdf.

[5] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Feb. 2016. http://isabelle.in.tum.de/website-Isabelle2016/dist/Isabelle2016/doc/tutorial.pdf.

[6] P. Noce. Noninterference security in communicating sequential processes. *Archive of Formal Proofs*, May 2014. http://isa-afp.org/entries/Noninterference_CSP.shtml, Formal proof development.

[7] P. Noce. The generic unwinding theorem for csp noninterference security. *Archive of Formal Proofs*, June 2015. http://isa-afp.org/entries/Noninterference_Generic_Unwinding.shtml, Formal proof development.

[8] P. Noce. The inductive unwinding theorem for csp noninterference security. *Archive of Formal Proofs*, Aug. 2015. http://isa-afp.org/entries/Noninterference_Inductive_Unwinding.shtml, Formal proof development.

[9] P. Noce. The ipurge unwinding theorem for csp noninterference security. *Archive of Formal Proofs*, June 2015. http://isa-afp.org/entries/Noninterference_Ipurge_Unwinding.shtml, Formal proof development.

[10] P. Noce. Conservation of csp noninterference security under sequential composition. *Archive of Formal Proofs*, Apr. 2016. http://isa-afp.org/entries/Noninterference_Sequential_Composition.shtml, Formal proof development.