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
February 23, 2021

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 2
  1.1 Propaedeutic definitions and lemmas 2
  1.2 Concurrent composition 4
  1.3 Auxiliary intransitive purge functions 9
  1.4 Conservation of noninterference security under concurrent composition 14
  1.5 Conservation of noninterference security in the absence of fake events 18
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]:
\( u \in \text{sinks-aux } I \ D \ U \ xs \rightarrow u \in U \lor (\exists x \in \text{set } xs. \ u = D \ x) \)
⟨proof⟩

lemma ipurge-ref-aux-cons:
\( \text{ipurge-ref-aux } I \ D \ U \ (x \# xs) \ X = \text{ipurge-ref-aux } I \ D \ (\text{sinks-aux } I \ D \ U \ [x]) \ xs \ X \)
⟨proof⟩

lemma process-rule-1-futures:
\( xs \in \text{traces } P \Rightarrow (\text{[]} , \{\}) \in \text{futures } P \ xs \)
⟨proof⟩

lemma process-rule-3-futures:
\( (ys, Y) \in \text{futures } P \ xs \Rightarrow Y' \subseteq Y \Rightarrow (ys, Y') \in \text{futures } P \ xs \)
⟨proof⟩

lemma process-rule-4-futures:
\( (ys, Y) \in \text{futures } P \ xs \Rightarrow(ys @ [x], \{\}) \in \text{futures } P \ xs \lor (ys, \text{insert } x \ Y) \in \text{futures } P \ xs \)
⟨proof⟩

lemma process-rule-5-general [rule-format]:
\( xs \in \text{divergences } P \rightarrow xs @ ys \in \text{divergences } P \)
⟨proof⟩

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 ⇒ ‘a list ⇒ ‘a list set where
future-divergences P xs ≡ \{ys. \ xs @ ys ∈ divergences P\}

definition after :: ‘a process ⇒ ‘a list ⇒ ‘a process (infixl \ 64) where
P \ xs ≡ Abs-process (futures P xs, future-divergences P xs)

lemma process-rule-5-futures:
\( ys \in \text{future-divergences } P \ xs \Rightarrow ys @ [x] \in \text{future-divergences } P \ xs \)
⟨proof⟩

lemma process-rule-6-futures:
\( ys \in \text{future-divergences } P \ xs \Rightarrow (ys, Y) \in \text{futures } P \ xs \)
⟨proof⟩

lemma after-rep:
assumes \( A \): \( xs ∈ \text{traces } P \)
shows Rep-process \((P \setminus xs) = (\text{futures } P \; xs, \text{future-divergences } P \; xs)\)

\(\text{(is } = ?X)\)

⟨proof⟩

lemma after-failures:

assumes \(A: \; xs \in \text{traces } P\)

shows \(\text{failures } (P \setminus xs) = \text{futures } P \; xs\)

⟨proof⟩

lemma after-futures:

assumes \(A: \; xs \in \text{traces } P\)

shows \(\text{futures } (P \setminus xs) \; ys = \text{futures } P \; (xs @ ys)\)

⟨proof⟩

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

lemma after-secure:

assumes \(A: \; xs \in \text{traces } P\)

shows \(\text{secure } P \; I \; D \implies \text{secure } (P \setminus xs) \; I \; D\)

⟨proof⟩

lemma ipurge-tr-ref-aux-futures:

\[\text{[secure } P \; I \; D; \; (ys, Y) \in \text{futures } P \; xs] \implies (\text{ipurge-tr-aux } I \; D \; U \; ys, \text{ipurge-ref-aux } I \; D \; U \; ys \; Y) \in \text{futures } P \; xs\]

⟨proof⟩

lemma ipurge-tr-ref-aux-failures-general:

\[\text{[secure } P \; I \; D; \; (xs @ ys, Y) \in \text{failures } P]\implies (xs @ \text{ipurge-tr-aux } I \; D \; U \; ys, \text{ipurge-ref-aux } I \; D \; U \; ys \; Y) \in \text{failures } P\]

⟨proof⟩

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 \(p \cup \text{range } q\) and shared events are those in range \(p \cap \text{range } q\).
The case where \(- (\text{range } p \cup \text{range } q) \neq \{\}\) constitutes a generalization of the definition given in [1], and the events in \(- (\text{range } p \cup \text{range } q)\), 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 \(- (\text{range } p \cup \text{range } q)\).

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 ::

\[\text{\'a process} \Rightarrow \text{\'b process} \Rightarrow \text{\'c} \Rightarrow \text{\'d} \Rightarrow \text{\'e} \Rightarrow \text{\'f list set} \text{ where} \]

\[
\text{con-comp-divergences } P Q p q \equiv \\
\{xs @ ys | xs ys. \\
\text{set } xs \subseteq \text{range } p \cup \text{range } q \land \\
\text{map (inv } p)[x \leftarrow xs. x \in \text{range } p] \in \text{divergences } P \land \\
\text{map (inv } q)[x \leftarrow xs. x \in \text{range } q] \in \text{divergences } Q\} \\
\]

**definition** con-comp-failures ::

\[\text{\'a process} \Rightarrow \text{\'b process} \Rightarrow \text{\'c} \Rightarrow \text{\'d} \Rightarrow \text{\'e} \Rightarrow \text{\'f failure set} \text{ where} \]

\[
\text{con-comp-failures } P Q p q \equiv \\
\{(xs, X \cup Y \cup Z) | xs X Y Z. \\
\text{set } xs \subseteq \text{range } p \cup \text{range } q \land \\
X \subseteq \text{range } p \land Y \subseteq \text{range } q \land Z \subseteq - (\text{range } p \cup \text{range } q) \land \\
\text{map (inv } p)[x \leftarrow xs. x \in \text{range } p], \text {inv } p \ ' X) \in \text{failures } P \land \\
\text{map (inv } q)[x \leftarrow xs. x \in \text{range } q], \text {inv } q \ ' Y) \in \text{failures } Q\} \cup \\
\]

5
\begin{enumerate}
\item \( (xs, X) \in \text{con-comp-divergences } P \ Q \ p \ q \}
\end{enumerate}

\textbf{definition} con-comp ::
\begin{align*}
\text{'a process } \Rightarrow \text{'b process } \Rightarrow \text{('a } \Rightarrow \text{'c) } \Rightarrow \text{('b } \Rightarrow \text{'c) } \Rightarrow \text{'c process where}
& \text{\text{Abs-process (con-comp-failures } P \ Q \ p \ q, \text{ con-comp-divergences } P \ Q \ p \ q) }
\end{align*}

\textbf{abbreviation} con-comp-syntax ::
\begin{align*}
\text{'a process } \Rightarrow \text{'b process } \Rightarrow \text{('a } \Rightarrow \text{'c) } \Rightarrow \text{('b } \Rightarrow \text{'c) } \Rightarrow \text{'c process}
& \text{((~\parallel <\cdot,\cdot>) 55)}
\end{align*}
\textbf{where}
\begin{align*}
P \parallel Q <p, q> \equiv \text{con-comp } P \ Q \ p \ q
\end{align*}

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

\textbf{lemma} con-comp-prop-1:
\begin{align*}
& ([\cdot], \{\cdot\}) \in \text{con-comp-failures } P \ Q \ p \ q \\
\text{\text{\langle proof\rangle }}
\end{align*}

\textbf{lemma} con-comp-prop-2:
\begin{align*}
& (xs @ [x], X) \in \text{con-comp-failures } P \ Q \ p \ q \implies \\
& (xs, \{\cdot\}) \in \text{con-comp-failures } P \ Q \ p \ q \\
\text{\text{\langle proof\rangle }}
\end{align*}

\textbf{lemma} con-comp-prop-3:
\begin{align*}
& [(xs, Y) \in \text{con-comp-failures } P \ Q \ p \ q; \ X \subseteq Y] \implies \\
& (xs, X) \in \text{con-comp-failures } P \ Q \ p \ q \\
\text{\text{\langle proof\rangle }}
\end{align*}

\textbf{lemma} con-comp-prop-4:
\begin{align*}
& (xs, X) \in \text{con-comp-failures } P \ Q \ p \ q \implies \\
& (xs @ [x], \{\cdot\}) \in \text{con-comp-failures } P \ Q \ p \ q \lor \\
& (xs, \text{insert } x X) \in \text{con-comp-failures } P \ Q \ p \ q \\
\text{\text{\langle proof\rangle }}
\end{align*}

\textbf{lemma} con-comp-prop-5:
\begin{align*}
& xs \in \text{con-comp-divergences } P \ Q \ p \ q \implies \\
& xs @ [x] \in \text{con-comp-divergences } P \ Q \ p \ q \\
\text{\text{\langle proof\rangle }}
\end{align*}

\textbf{lemma} con-comp-prop-6:
\begin{align*}
& xs \in \text{con-comp-divergences } P \ Q \ p \ q \implies \\
& (xs, X) \in \text{con-comp-failures } P \ Q \ p \ q \\
\text{\text{\langle proof\rangle }}
\end{align*}
lemma con-comp-rep:
\[
\text{Rep-process } (P \parallel Q <p, q>) = \\
(\text{con-comp-failures } P Q p q, \text{con-comp-divergences } P Q p q)
\] (is \(-= ?X))
\langle proof \rangle

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:
\[
\text{failures } (P \parallel Q <p, q>) = \text{con-comp-failures } P Q p q
\] (proof)

lemma con-comp-divergences:
\[
\text{divergences } (P \parallel Q <p, q>) = \text{con-comp-divergences } P Q p q
\] (proof)

lemma con-comp-futures:
\[
\text{futures } (P \parallel Q <p, q>) xs = \\
\{ (ys, Y), (xs @ ys, Y) \in \text{con-comp-failures } P Q p q \}
\] (proof)

lemma con-comp-traces:
\[
\text{traces } (P \parallel Q <p, q>) = \text{Domain } (\text{con-comp-failures } P Q p q)
\] (proof)

lemma con-comp-refusals:
\[
\text{refusals } (P \parallel Q <p, q>) xs \equiv \text{con-comp-failures } P Q p q {''} \{ xs \}
\] (proof)

lemma con-comp-next-events:
\[
\text{next-events } (P \parallel Q <p, q>) xs = \\
\{ x, xs @ [x] \in \text{Domain } (\text{con-comp-failures } P Q p q) \}
\] (proof)

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 \parallel Q <p, q>)

lemma con-comp-failures-traces:
\((xs, X) \in \text{con-comp-failures } P Q p q \implies \quad\)  
map (inv p) \[x\leftarrow xs.\ x \in \text{range } p\] \in \text{traces } P \land  
map (inv q) \[x\leftarrow xs.\ x \in \text{range } q\] \in \text{traces } Q

lemma con-comp-failures-divergences:
\((xs \circ y \neq ys, Y) \in \text{con-comp-failures } P Q p q \implies \quad\)  
y \notin \text{range } p \implies  
y \notin \text{range } q \implies  
\exists xs'.\quad (\exists ys'.\ x @ zs = xs' \circ ys') \land  
set xs' \subseteq \text{range } p \cup \text{range } q \land  
map (inv p) \[x\leftarrow xs'.\ x \in \text{range } p\] \in \text{divergences } P \land  
map (inv q) \[x\leftarrow xs'.\ x \in \text{range } q\] \in \text{divergences } Q

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 \(p \cap \text{range } q\) 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 \(d', e\), and for each \(x\) in range \(p \cap \text{range } q\), \(D\) 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 \(d', e\), the ordered pair comprised of the corresponding security domains for the output process will be included in \(I'\) just in case
(d, e) ∈ 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) ∈ 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 × d) set ⇒ ('d option × 'd option) set where

con-comp-pol I ≡ 

{(Some d, Some e) | d e. (d, e) ∈ I} ∪ {(u, v). v = None}

**function** con-comp-map ::

('a ⇒ 'd) ⇒ ('b ⇒ 'd) ⇒ ('a ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ 'c ⇒ 'd option where

x ∈ range p ⇒

con-comp-map D E p q x = Some (D (inv p x)) |

x /∈ range p ⇒ x ∈ range q ⇒

con-comp-map D E p q x = Some (E (inv q x)) |

x /∈ range p ⇒ x /∈ range q ⇒

con-comp-map D E p q x = None

⟨proof⟩

termination ⟨proof⟩

**definition** consistent-maps ::

('a ⇒ 'd) ⇒ ('b ⇒ 'd) ⇒ ('a ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ bool where

consistent-maps D E p q ≡

∀ x ∈ range p ∩ 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 $\text{sinks-aux } I\ D\ U\ [x]$ 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 $\text{ipurge-tr-aux-foldr}$ as its action resembles that of function $\text{foldr}$.

\begin{verbatim}
primrec ipurge-tr-aux-foldr ::
    (d x d set ⇒ (a ⇒ d) ⇒ (a ⇒ bool) ⇒ d set ⇒ a list ⇒ 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)
\end{verbatim}

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 $\text{ipurge-tr-aux-foldr } I\ D\ P\ (\text{sinks-aux } I\ D\ U\ [x])\ xs'$ 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 $\text{sinks-aux } I\ D\ U\ [x]$ 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 $\text{ipurge-tr-aux-foldr}$, it follows that the presence of list $\text{ipurge-tr-aux-foldr } I\ D\ P\ (\text{sinks-aux } I\ D\ U\ [x])\ xs'$ 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 $\text{ipurge-ref-aux-foldr}$, which at first glance just seems a uselessly complicate and inefficient way to compute the intransitive purge of an event set.

\begin{verbatim}
primrec ipurge-ref-aux-foldr ::
    (d x d set ⇒ (a ⇒ d) ⇒ (a ⇒ bool) ⇒ d set ⇒ a list ⇒ a set ⇒ a set
where
\end{verbatim}
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:

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

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 $\text{sinks-aux} \ (\text{con-comp-pol} \ I) \ (\text{con-comp-map} \ D E p q) \ U \ [y]$, respectively. The proof can then be accomplished by applying lemma ipurge-tr-ref-aux-failures-general to the resulting future of trace $\text{map} \ (\text{inv} \ p) \ (\text{filter} \ (\lambda x. \ x \in \text{range} \ p) \ xs)$, moving functions $\text{ipurge-tr-aux}$ and $\text{ipurge-ref-aux}$ into the arguments of $\text{map} \ (\text{inv} \ p)$ and $(\cdot) \ (\text{inv} \ p)$, 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:

$$U \subseteq V \Longrightarrow ipurge-tr-aux \ I D U \ (ipurge-tr-aux-foldr \ I D V \ xs) = ipurge-tr-aux-foldr \ I D V \ xs$$

(proof)

**Lemma** ipurge-tr-aux-foldr-eq:

$$[x \leftarrow ipurge-tr-aux \ I D U \ xs. \ P \ x] = ipurge-tr-aux-foldr \ I D V \ xs$$

(proof)
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]`:
\[
U \subseteq V \longrightarrow \text{sinks-aux } I D U (\text{ipurge-tr-aux-foldr } I D P V \ xs) = U
\]
(\text{proof})

**lemma** `ipurge-tr-aux-foldr-ref-aux`:
\[
\text{assumes } A: U \subseteq V
\]
\[
\text{shows } \text{ipurge-ref-aux } I D U (\text{ipurge-tr-aux-foldr } I D P V \ xs) X = \text{ipurge-ref-aux } I D U [] X
\]
(\text{proof})

**lemma** `ipurge-ref-aux-foldr-subset [rule-format]`:
\[
\text{sinks-aux } I D U \ ys \subseteq V \longrightarrow \text{ipurge-ref-aux } I D U \ ys (\text{ipurge-ref-aux-foldr } I D P V \ xs \ X) = \text{ipurge-ref-aux-foldr } I D P V \ xs \ X
\]
(\text{proof})

**lemma** `ipurge-ref-aux-foldr-eq`:
\[
\text{ipurge-ref-aux } I D U \ xs \ X = \text{ipurge-ref-aux-foldr } I D P U \ xs \ X
\]
(\text{proof})

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`:
\[
\text{assumes}
\]
\[
A: U \subseteq \text{range Some and } B: \text{set } xs \subseteq \text{range p } \cup \text{range q}
\]
\[
\text{shows } \text{sinks-aux } (\text{con-comp-pol } I) (\text{con-comp-map } D E p \ q) \ U \ xs \subseteq \text{range Some}
\]
\[
\text{(is sinks-aux } - ?D' - - \subseteq -)
\]
(\text{proof})

**lemma** `con-comp-sinks-aux [rule-format]`:
\[
\text{assumes } A: U \subseteq \text{range Some}
\]
\[
\text{shows } \text{set } xs \subseteq \text{range p } \longrightarrow \text{sinks-aux } I D (\text{the } ' U) (\text{map } (\text{inv } p) \ xs) = \text{the } ' \text{sinks-aux } (\text{con-comp-pol } I) (\text{con-comp-map } D E p \ q) \ U \ xs
\]
\[
\text{(is } - \longrightarrow - = \text{the } ' \text{sinks-aux } ?I' ?D' - -)
\]
(\text{proof})

**lemma** `con-comp-ipurge-tr-aux [rule-format]`:
\[
\text{assumes } A: U \subseteq \text{range Some}
\]
\[
\text{shows } \text{set } xs \subseteq \text{range p } \longrightarrow \text{ipurge-tr-aux } I D (\text{the } ' U) (\text{map } (\text{inv } p) \ xs) =
\]
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⟩

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⟩

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⟩

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⟩

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⟩

lemma con-comp-secure-aux [rule-format]:
assumes
A: secure P I D and
B: set ys ⊆ 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)
(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
⟨proof⟩
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 $P \parallel Q < p, q >$ is secure with respect to the noninterference policy $\text{con-comp-pol } I$ and the event-domain map $\text{con-comp-map } D E p q$, provided that condition $\text{consistent-maps } D E p q$ 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 $\text{con-comp-map } D E p q$ and then applying the corresponding lemmas proven for the former input process.

**lemma** $\text{con-comp-secure-del-aux-1}$:

assumes

A: secure $P I D$ and
B: $y \in \text{range } p \lor y \in \text{range } q$ and
C: set $ys \subseteq \text{range } p \cup \text{range } q$ and
D: $Y \subseteq \text{range } p$ and
E: $(\text{map (inv p)} [x \leftarrow xs @ y \neq ys. x \in \text{range } p], \text{inv } p \ ' Y) \in \text{failures } P$

shows

$(\text{map (inv p)} [x \leftarrow zs @ y \neq ys. x \in \text{range } p], \text{inv } p \ ' \text{ipurge-ref (con-comp-pol } I) \ (\text{con-comp-map } D E p q)\ )$

$(\text{map (inv p)} [x \leftarrow y # \text{ipurge-tr (con-comp-pol } I) \ (\text{con-comp-map } D E p q)\ )$

$(\text{is (map (inv p)} [x \leftarrow zs @ y \neq ys. x \in \text{range } p], \text{inv } p \ ' \text{ipurge-tr ?I' ?D' - -. -), -) } \in -$

(proof)

**lemma** $\text{con-comp-secure-add-aux-1}$:

assumes

A: secure $P I D$ and
B: $y \in \text{range } p \lor y \in \text{range } q$ and
C: set $zs \subseteq \text{range } p \cup \text{range } q$ and
D: $Z \subseteq \text{range } p$ and
E: $(\text{map (inv p)} [x \leftarrow xs @ zs. x \in \text{range } p], \text{inv } p \ ' Z) \in \text{failures } P$ and
F: $\text{map (inv p)} [x \leftarrow zs @ [y], x \in \text{range } p] \in \text{traces } P$

shows

$(\text{map (inv p)} [x \leftarrow xs @ y \neq zs. x \in \text{range } p], \text{inv } p \ ' \text{ipurge-ref (con-comp-pol } I) \ (\text{con-comp-map } D E p q)\ )$

$(\text{map (inv p)} [x \leftarrow zs @ [y], x \in \text{range } p] \in \text{traces } P$

$(\text{is (map (inv p)} [x \leftarrow zs @ y \neq zs. x \in \text{range } p], \text{inv } p \ ' \text{ipurge-ref ?I' ?D' - -. -), -) } \in -$

(proof)
lemma con-comp-consistent-maps:
consistent-maps D E p q \implies con-comp-map D E p q = con-comp-map E D q p
⟨proof⟩

lemma con-comp-secure-del-aux-2:
assumes A: consistent-maps D E p q
shows
secure Q I E \implies
y \in \text{range } p \lor y \in \text{range } q \implies
set ys \subseteq \text{range } p \cup \text{range } q \implies
Y \subseteq \text{range } q \implies
(map (\text{inv } q) [x\leftarrow xs @ y \# ys. x \in \text{range } q], \text{inv } q ' Y) \in \text{failures } Q \implies
(map (\text{inv } q) [x\leftarrow xs @ \text{ipurge-tr (con-comp-pol I)} (\text{con-comp-map } D E p q) (\text{map (\text{inv } q) [x\leftarrow xs @ y \# \text{ipurge-tr (con-comp-pol I)}] (\text{con-comp-map } D E p q) y) zs. x \in \text{range } q], \text{inv } q ' \text{ipurge-ref (con-comp-pol I)} (\text{con-comp-map } D E p q) (\text{map (\text{inv } q) [x\leftarrow xs @ y \# \text{ipurge-tr (con-comp-pol I)}] (\text{con-comp-map } D E p q) y) y z) \in \text{failures } Q
\langle \text{proof} \rangle

lemma con-comp-secure-add-aux-2:
assumes A: consistent-maps D E p q
shows
secure Q I E \implies
y \in \text{range } p \lor y \in \text{range } q \implies
set zs \subseteq \text{range } p \cup \text{range } q \implies
Z \subseteq \text{range } q \implies
(map (\text{inv } q) [x\leftarrow xs @ zs. x \in \text{range } q], \text{inv } q ' Z) \in \text{failures } Q \implies
(map (\text{inv } q) [x\leftarrow xs @ [y]. x \in \text{range } q] \in \text{traces } Q \implies
(map (\text{inv } q) [x\leftarrow xs @ y \# \text{ipurge-tr (con-comp-pol I)} (\text{con-comp-map } D E p q) (\text{map (\text{inv } q) [x\leftarrow xs @ y \# \text{ipurge-tr (con-comp-pol I)}] (\text{con-comp-map } D E p q) y) zs. x \in \text{range } q], \text{inv } q ' \text{ipurge-ref (con-comp-pol I)} (\text{con-comp-map } D E p q) (\text{map (\text{inv } q) [x\leftarrow xs @ y \# \text{ipurge-tr (con-comp-pol I)}] (\text{con-comp-map } D E p q) y) zs Z) \in \text{failures } Q
\langle \text{proof} \rangle

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
\exists R S T.
Y = R \cup S \cup T \land
(y \in \text{range } p \lor y \in \text{range } q) \land
set xs \subseteq \text{range } p \cup \text{range } q \land
set ys \subseteq \text{range } p \cup \text{range } q \land
R \subseteq \text{range } p \land
S \subseteq \text{range } q \land
T \subseteq - \text{range } p \land
T \subseteq - \text{range } q \land
(map (\text{inv } p) [x\leftarrow xs @ y \# ys. x \in \text{range } p], \text{inv } p ' R) \in \text{failures } P \land

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
shows
\exists R S T.
Y = R \cup S \cup T \land
(y \in \text{range } p \lor y \in \text{range } q) \land
set xs \subseteq \text{range } p \cup \text{range } q \land
set ys \subseteq \text{range } p \cup \text{range } q \land
R \subseteq \text{range } p \land
S \subseteq \text{range } q \land
T \subseteq - \text{range } p \land
T \subseteq - \text{range } q \land
(map (\text{inv } p) [x\leftarrow xs @ y \# ys. x \in \text{range } p], \text{inv } p ' R) \in \text{failures } P \land

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
\exists R S T.
Y = R \cup S \cup T \land
(y \in \text{range } p \lor y \in \text{range } q) \land
set xs \subseteq \text{range } p \cup \text{range } q \land
set ys \subseteq \text{range } p \cup \text{range } q \land
R \subseteq \text{range } p \land
S \subseteq \text{range } q \land
T \subseteq - \text{range } p \land
T \subseteq - \text{range } q \land
(map (\text{inv } p) [x\leftarrow xs @ y \# ys. x \in \text{range } p], \text{inv } p ' R) \in \text{failures } P \land

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
shows
\exists R S T.
Y = R \cup S \cup T \land
(y \in \text{range } p \lor y \in \text{range } q) \land
set xs \subseteq \text{range } p \cup \text{range } q \land
set ys \subseteq \text{range } p \cup \text{range } q \land
R \subseteq \text{range } p \land
S \subseteq \text{range } q \land
T \subseteq - \text{range } p \land
T \subseteq - \text{range } q \land
(map (\text{inv } p) [x\leftarrow xs @ y \# ys. x \in \text{range } p], \text{inv } p ' R) \in \text{failures } P \land
(map (inv q) [x ← xs @ y # ys. x ∈ range q], inv q ⊢ S) ∈ failures Q \implies
\exists 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 \cup S \cup T \land
 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 \cup range q \land
 R ⊆ range p \land
 S ⊆ range q \land
 T ⊆ − range p \land
 T ⊆ − range q \land
 (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 \land
 (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 - \implies ∃ - - . ipurge-ref ?I' ?D' - - = - ∧ - )
 ⟨proof⟩

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
 \exists xs'.
 (\exists ys'. xs @ y # ys = xs' @ ys') \land
 set xs' ⊆ range p \cup range q \land
 map (inv p) [x ← xs'. x ∈ range p] ∈ divergences P \land
 map (inv q) [x ← xs'. x ∈ range q] ∈ divergences Q \implies
 (\exists 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 \cup S \cup T \land
 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 \cup range q \land
 R ⊆ range p \land
 S ⊆ range q \land
 T ⊆ − range p \land
 T ⊆ − range q \land
 (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 \land
 (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) \lor
 (\exists xs'.
 (\exists 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') \land
 set xs' ⊆ range p \cup range q \land
 map (inv p) [x ← xs'. x ∈ range p] ∈ divergences P \land
 map (inv q) [x ← xs'. x ∈ range q] ∈ divergences Q)
(is - \implies (\exists R S T. ?F R S T ys) \lor ?G)

⟨proof⟩

**lemma con-comp-secure-add-case-1:**

**assumes**

A: consistent-maps \(DEpq\) **and**

B: secure \(PID\) **and**

C: secure \(QIE\) **and**

D: \((xs \odot y \# ys, Y) \in \text{con-comp-failures} Ppq\) **and**

E: \(y \in \text{range } p \lor y \in \text{range } q\)

**shows**

\(\exists RST.\)

\[Z = R \cup S \cup T \land\]

set \(xs \subseteq \text{range } p \cup \text{range } q \land\]

set \(zs \subseteq \text{range } p \cup \text{range } q \land\]

\(R \subseteq \text{range } p \land\]

\(S \subseteq \text{range } q \land\]

\(T \subseteq - \text{range } p \land\]

\(T \subseteq - \text{range } q \land\]

\((\text{map } (\text{inv } p) [x \leftarrow xs \odot zs \odot x \in \text{range } p], \text{inv } p \cdot R) \in \text{failures } P\land\]

\((\text{map } (\text{inv } q) [x \leftarrow xs \odot zs \odot x \in \text{range } q], \text{inv } q \cdot S) \in \text{failures } Q \implies\]

\(\exists RST.\)

\(\text{ipurge-ref } (\text{con-comp-pol } I) (\text{con-comp-map } DEpq)\)

\((\text{con-comp-map } DEpq) \; \text{zs} \; Z = R \cup S \cup T \land\]

set \(xs \subseteq \text{range } p \cup \text{range } q \land\]

set \((\text{ipurge-tr } (\text{con-comp-pol } I)) (\text{con-comp-map } DEpq) \; \text{zs} \subseteq \text{range } p \cup \text{range } q \land\]

\(R \subseteq \text{range } p \land\]

\(S \subseteq \text{range } q \land\]

\(T \subseteq - \text{range } p \land\]

\(T \subseteq - \text{range } q \land\]

\((\text{map } (\text{inv } p) [x \leftarrow zs \odot y \# \text{ipurge-tr } (\text{con-comp-pol } I)]

\((\text{con-comp-map } DEpq) \; \text{zs} \odot x \in \text{range } p],

\text{inv } p \cdot R) \in \text{failures } P\land\]

\((\text{map } (\text{inv } q) [x \leftarrow zs \odot y \# \text{ipurge-tr } (\text{con-comp-pol } I)]

\((\text{con-comp-map } DEpq) \; \text{zs} \odot x \in \text{range } q],

\text{inv } q \cdot S) \in \text{failures } Q\]

(is - \implies \exists - - . \text{ipurge-ref } ?I' ?D' - - = - \land -)\]

⟨proof⟩

**lemma con-comp-secure-add-case-2:**

**assumes**

A: consistent-maps \(DEpq\) **and**

B: secure \(PID\) **and**

C: secure \(QIE\) **and**

D: \((xs \odot y \# ys, Y) \in \text{con-comp-failures} Ppq\) **and**

E: \(y \in \text{range } p \lor y \in \text{range } q\)

**shows**

\(\exists xs'.\)
\[
(\exists \ y'. \ x @ y = x' @ y') \land \\
\text{set } x' \subseteq \text{range } p \cup \text{range } q \land \\
\text{map } (\text{inv } p) [x @ x'. \ x \in \text{range } p] \in \text{divergences } P \land \\
\text{map } (\text{inv } q) [x @ x'. \ x \in \text{range } q] \in \text{divergences } Q \Rightarrow \\
(\exists R S T).
\]

\text{ipurge-ref } (\text{con-comp-pol } I) (\text{con-comp-map } D \ E \ p \ q)
\text{ (con-comp-map } D \ E \ p \ q) \ y = \text{Z } R \cup S \cup T \land \\
\text{set } x' \subseteq \text{range } p \cup \text{range } q \land \\
\text{set } (\text{ipurge-tr } (\text{con-comp-pol } I) (\text{con-comp-map } D \ E \ p \ q)
\text{ (con-comp-map } D \ E \ p \ q) \ y \subseteq \text{range } p \cup \text{range } q \land \\
R \subseteq \text{range } p \land \\
S \subseteq \text{range } q \land \\
T \subseteq - \text{range } p \land \\
T \subseteq - \text{range } q \land \\
(\text{map } (\text{inv } p) [x @ x'. \ y # \text{ipurge-tr } (\text{con-comp-pol } I)
\text{ (con-comp-map } D \ E \ p \ q) \ y \subseteq \text{range } p],
\text{inv } p \cdot R] \in \text{failures } P \land \\
(\text{map } (\text{inv } q) [x @ x'. \ y # \text{ipurge-tr } (\text{con-comp-pol } I)
\text{ (con-comp-map } D \ E \ p \ q) \ y \subseteq \text{range } q],
\text{inv } q \cdot S] \in \text{failures } Q) \lor \\
(\exists x',
\text{ (con-comp-map } D \ E \ p \ q) \ y \in \text{range } p \land \\
\text{set } x' \subseteq \text{range } p \cup \text{range } q \land \\
\text{map } (\text{inv } p) [x @ x'. \ x \in \text{range } p] \in \text{divergences } P \land \\
\text{map } (\text{inv } q) [x @ x'. \ x \in \text{range } q] \in \text{divergences } Q)
\text{ (is - } \Rightarrow \ (\exists R S T. \ \#F R S T \ y \land \ #G)
\langle \text{proof} \rangle
\]

\text{theorem con-comp-secure:}
\text{assumes}
\text{A: consistent-maps } D \ E \ p \ q \text{ and}
\text{B: secure } P \ I \ D \text{ and}
\text{C: secure } Q \ I \ E
\text{shows secure } (P \ || \ Q < p, q>) \ (\text{con-comp-pol } I) (\text{con-comp-map } D \ E \ p \ q)
\langle \text{proof} \rangle
\]

### 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 \( \text{range } p \cup \text{range } q = \text{UNIV} \), 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 $P \parallel Q <p, q>$ is secure with respect to the same noninterference policy $I$ and the event-domain map $\text{the} \circ \text{con-comp-map} D E p q$, provided that conditions $\text{range } p \cup \text{range } q = \text{UNIV}$ and $\text{consistent-maps } D E p q$ are satisfied.

**Lemma con-comp-sinks-range:**

$u \in \text{range Some} \implies$

$\text{set } xs \subseteq \text{range } p \cup \text{range } q \implies$

$\text{sinks } (\text{con-comp-pol } I) (\text{con-comp-map } D E p q) u xs \subseteq \text{range Some}$

⟨proof⟩

**Lemma con-comp-sinks-no-fake:**

**Assumes**

$A: \text{range } p \cup \text{range } q = \text{UNIV}$ and

$B: u \in \text{range Some}$

**Shows** $\text{sinks } I (\text{the } \circ \text{con-comp-map } D E p q) (\text{the } u) xs =$

$\text{the } \text{sinks } (\text{con-comp-pol } I) (\text{con-comp-map } D E p q) u xs$

(is i = the ’ sinks ?I’ ?D’ - - )

⟨proof⟩

**Lemma con-comp-ipurge-tr-no-fake:**

**Assumes**

$A: \text{range } p \cup \text{range } q = \text{UNIV}$ and

$B: u \in \text{range Some}$

**Shows** $\text{ipurge-tr } (\text{con-comp-pol } I) (\text{con-comp-map } D E p q) u xs =$

$\text{ipurge-tr } I (\text{the } \circ \text{con-comp-map } D E p q) (\text{the } u) xs$

(is i purge-tr ?I’ ?D’ - - = - )

⟨proof⟩

**Lemma con-comp-ipurge-ref-no-fake:**

**Assumes**

$A: \text{range } p \cup \text{range } q = \text{UNIV}$ and

$B: u \in \text{range Some}$

**Shows** $\text{ipurge-ref } (\text{con-comp-pol } I) (\text{con-comp-map } D E p q) u xs X =$

$\text{ipurge-ref } I (\text{the } \circ \text{con-comp-map } D E p q) (\text{the } u) xs X$

(is i purge-ref ?I’ ?D’ - - - = - )

⟨proof⟩

**Theorem con-comp-secure-no-fake:**

**Assumes**

$A: \text{range } p \cup \text{range } q = \text{UNIV}$ and

$B: \text{consistent-maps } D E p q$ and

$C: \text{secure } P I D$ and

$D: \text{secure } Q I E$

**Shows** $\text{secure } (P \parallel Q <p, q>) I (\text{the } \circ \text{con-comp-map } D E p q)$

⟨proof⟩

end
References


