Conservation of CSP Noninterference Security under Sequential 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 former operation is a process that initially behaves like the first operand, and then like the second operand once the execution of the first one has terminated successfully, as long as it does.

This paper formalizes Hoare's definition of sequential composition and proves, in the general case of a possibly intransitive policy, that CSP noninterference security is conserved under this operation, provided that successful termination cannot be affected by confidential events and cannot occur as an alternative to other events in the traces of the first operand. Both of these assumptions are shown, by means of counterexamples, to be necessary for the theorem to hold.

Contents

1 Propaedeutic definitions and lemmas

theory *Propaedeutics*
imports *Noninterference-Ipurge-Unwinding.DeterministicProcesses*
begin

To our Lord Jesus Christ, my dear parents, and my "little" sister, for the immense love with which they surround me.

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 former operation is a process that initially behaves like the first operand, and then like the second operand once the execution of the first one has terminated successfully, as long as it does. In order to distinguish it from deadlock, successful termination is regarded as a special event in the process alphabet (required to be the same for both the input processes and the output one).

This paper formalizes Hoare's definition of sequential composition and proves, in the general case of a possibly intransitive policy, that CSP noninterference security [8] is conserved under this operation, viz. the security of both of the input processes implies that of the output process.

This property is conditional on two nontrivial assumptions. The first assumption is that the policy do not allow successful termination to be affected by confidential events, viz. by other events not allowed to affect some event in the process alphabet. The second assumption is that successful termination do not occur as an alternative to other events in the traces of the first operand, viz. that whenever the process can terminate successfully, it cannot engage in any other event. Both of these assumptions are shown, by means of counterexamples, to be necessary for the theorem to hold.

From the above sketch of the sequential composition of two processes P and Q, notwithstanding its informal character, it clearly follows that any failure of the output process is either a failure of P (case A), or a pair (xs @ ys, Y), where xs is a trace of P and (ys, Y) is a failure of Q (case B). On the other hand, according to the definition of security given in [8], the output process is secure just in case, for each of its failures, any event x contained in the failure trace can be removed from the trace, or inserted into the trace of another failure after the same previous events as in the original trace, and the resulting pair is still a failure of the process, provided that the future of x is deprived of the events that may be affected by x.

In case A, this transformation is performed on a failure of process P; being it secure, the result is still a failure of P, and then of the output process. In case B, the transformation may involve either ys alone, or both xs and ys, depending on the position at which x is removed or inserted. In the former subcase, being Q secure, the result has the form (xs @ ys', Y') where (ys', Y') is a failure of Q, thus it is still a failure of the output process. In the latter subcase, ys has to be deprived of the events that may be affected by x, as well as by any event affected by x in the involved portion of xs, and a similar transformation applies to Y. In order that the output process be secure, the resulting pair (ys'', Y'') must still be a failure of Q, so that the pair (xs' @ ys'', Y''), where xs' results from the transformation of xs, be a failure of the output process.

The transformations bringing from ys and Y to ys'' and Y'' are implemented by the functions *ipurge-tr-aux* and *ipurge-ref-aux* defined in [9]. Therefore, the proof of the target security conservation theorem requires that of the following lemma: given a process P, a noninterference policy I, and an event-domain map D, if P is secure with respect to I and D and (xs, X) is a failure of P, then (*ipurge-tr-aux I D U xs*, *ipurge-ref-aux I D U xs X*) is still a failure of P. In other words, the lemma states that the failures of a secure process are closed under intransitive purge. This section contains a proof of such closure lemma, as well as further definitions and lemmas required for the proof of the target theorem.

Throughout this paper, the salient points of definitions and proofs are commented; for additional information, cf. Isabelle documentation, particularly [6], [4], [3], and [2].

1.1 Preliminary propaedeutic lemmas

In what follows, some lemmas required for the demonstration of the target closure lemma are proven.

Here below is the proof of some properties of functions *ipurge-tr* and *ipurge-ref*.

lemma *ipurge-tr-length*:
$length\ (ipurge\text{-}tr\ I\ D\ u\ xs) \leq length\ xs$
⟨*proof*⟩

lemma *ipurge-ref-swap*:
$ipurge\text{-}ref\ I\ D\ u\ xs\ \{x \in X.\ P\ x\} =$
$\{x \in ipurge\text{-}ref\ I\ D\ u\ xs\ X.\ P\ x\}$
⟨*proof*⟩

lemma *ipurge-ref-last*:
$ipurge\text{-}ref\ I\ D\ u\ (xs\ @\ [x])\ X =$
$(if\ (u,\ D\ x) \in I \vee (\exists v \in sinks\ I\ D\ u\ xs.\ (v,\ D\ x) \in I)$
$then\ ipurge\text{-}ref\ I\ D\ u\ xs\ \{x' \in X.\ (D\ x,\ D\ x') \notin I\}$
$else\ ipurge\text{-}ref\ I\ D\ u\ xs\ X)$
⟨*proof*⟩

Here below is the proof of some properties of function *sinks-aux*.

lemma *sinks-aux-append*:
$sinks\text{-}aux\ I\ D\ U\ (xs\ @\ ys) = sinks\text{-}aux\ I\ D\ (sinks\text{-}aux\ I\ D\ U\ xs)\ ys$
⟨*proof*⟩

lemma *sinks-aux-union*:
$sinks\text{-}aux\ I\ D\ (U \cup V)\ xs =$
$sinks\text{-}aux\ I\ D\ U\ xs \cup sinks\text{-}aux\ I\ D\ V\ (ipurge\text{-}tr\text{-}aux\ I\ D\ U\ xs)$
⟨*proof*⟩

lemma *sinks-aux-subset-dom*:
assumes *A*: $U \subseteq V$
shows $sinks\text{-}aux\ I\ D\ U\ xs \subseteq sinks\text{-}aux\ I\ D\ V\ xs$
⟨*proof*⟩

lemma *sinks-aux-subset-ipurge-tr-aux*:
$sinks\text{-}aux\ I\ D\ U\ (ipurge\text{-}tr\text{-}aux\ I'\ D'\ U'\ xs) \subseteq sinks\text{-}aux\ I\ D\ U\ xs$
⟨*proof*⟩

lemma *sinks-aux-subset-ipurge-tr*:
$sinks\text{-}aux\ I\ D\ U\ (ipurge\text{-}tr\ I'\ D'\ u'\ xs) \subseteq sinks\text{-}aux\ I\ D\ U\ xs$
⟨*proof*⟩

lemma *sinks-aux-member-ipurge-tr-aux* [*rule-format*]:

$u \in \mathit{sinks\text{-}aux}\ I\ D\ (U \cup V)\ xs \longrightarrow$
$(u,\ w) \in I \longrightarrow$
$\neg\ (\exists\, v \in \mathit{sinks\text{-}aux}\ I\ D\ V\ xs.\ (v,\ w) \in I) \longrightarrow$
$u \in \mathit{sinks\text{-}aux}\ I\ D\ U\ (\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ V\ xs)$
⟨*proof*⟩

lemma *sinks-aux-member-ipurge-tr*:
assumes
A: $u \in \mathit{sinks\text{-}aux}\ I\ D\ (\mathit{insert}\ v\ U)\ xs$ **and**
B: $(u,\ w) \in I$ **and**
C: $\neg\ ((v,\ w) \in I \vee (\exists\, v' \in \mathit{sinks}\ I\ D\ v\ xs.\ (v',\ w) \in I))$
shows $u \in \mathit{sinks\text{-}aux}\ I\ D\ U\ (\mathit{ipurge\text{-}tr}\ I\ D\ v\ xs)$
⟨*proof*⟩

Here below is the proof of some properties of functions *ipurge-tr-aux* and *ipurge-ref-aux*.

lemma *ipurge-tr-aux-append*:
$\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ U\ (xs\ @\ ys) =$
$\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ U\ xs\ @\ \mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ (\mathit{sinks\text{-}aux}\ I\ D\ U\ xs)\ ys$
⟨*proof*⟩

lemma *ipurge-tr-aux-single-event*:
$\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ U\ [x] = (\mathit{if}\ \exists\, v \in U.\ (v,\ D\ x) \in I$
$\mathit{then}\ []$
$\mathit{else}\ [x])$
⟨*proof*⟩

lemma *ipurge-tr-aux-cons*:
$\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ U\ (x\ \#\ xs) = (\mathit{if}\ \exists\, u \in U.\ (u,\ D\ x) \in I$
$\mathit{then}\ \mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ (\mathit{insert}\ (D\ x)\ U)\ xs$
$\mathit{else}\ x\ \#\ \mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ U\ xs)$
⟨*proof*⟩

lemma *ipurge-tr-aux-union*:
$\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ (U \cup V)\ xs =$
$\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ V\ (\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ U\ xs)$
⟨*proof*⟩

lemma *ipurge-tr-aux-insert*:
$\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ (\mathit{insert}\ v\ U)\ xs =$
$\mathit{ipurge\text{-}tr\text{-}aux}\ I\ D\ U\ (\mathit{ipurge\text{-}tr}\ I\ D\ v\ xs)$
⟨*proof*⟩

lemma *ipurge-ref-aux-subset*:
$\mathit{ipurge\text{-}ref\text{-}aux}\ I\ D\ U\ xs\ X \subseteq X$
⟨*proof*⟩

1.2 Intransitive purge of event sets with trivial base case

Here below are the definitions of variants of functions *sinks-aux* and *ipurge-ref-aux*, respectively named *sinks-aux-less* and *ipurge-ref-aux-less*, such that their base cases in correspondence with an empty input list are trivial, viz. such that *sinks-aux-less I D U* [] = {} and *ipurge-ref-aux-less I D U* [] *X* = *X*. These functions will prove to be useful in what follows.

function *sinks-aux-less* ::
$('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd\ set \Rightarrow 'a\ list \Rightarrow 'd\ set$ **where**
sinks-aux-less - - - [] = {} |
sinks-aux-less I D U (xs @ [x]) =
(if $\exists\, v \in U \cup$ *sinks-aux-less I D U xs.* $(v,\ D\ x) \in I$
then insert (D x) (sinks-aux-less I D U xs)
else sinks-aux-less I D U xs)
⟨*proof*⟩
termination ⟨*proof*⟩

definition *ipurge-ref-aux-less* ::
$('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd\ set \Rightarrow 'a\ list \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
ipurge-ref-aux-less I D U xs X ≡
$\{x \in X.\ \forall\, v \in$ *sinks-aux-less I D U xs.* $(v,\ D\ x) \notin I\}$

Here below is the proof of some properties of function *sinks-aux-less* used in what follows.

lemma *sinks-aux-sinks-aux-less*:
sinks-aux I D U xs = U ∪ sinks-aux-less I D U xs
⟨*proof*⟩

lemma *sinks-aux-less-single-dom*:
sinks-aux-less I D {u} xs = sinks I D u xs
⟨*proof*⟩

lemma *sinks-aux-less-single-event*:
sinks-aux-less I D U [x] = (if $\exists\, u \in U.\ (u,\ D\ x) \in I$ *then* $\{D\ x\}$ *else* $\{\}$)
⟨*proof*⟩

lemma *sinks-aux-less-append*:
sinks-aux-less I D U (xs @ ys) =
sinks-aux-less I D U xs ∪ sinks-aux-less I D (U ∪ sinks-aux-less I D U xs) ys
⟨*proof*⟩

lemma *sinks-aux-less-cons*:
sinks-aux-less I D U (x # xs) = (if $\exists\, u \in U.\ (u,\ D\ x) \in I$
then insert (D x) (sinks-aux-less I D (insert (D x) U) xs)
else sinks-aux-less I D U xs)

⟨*proof*⟩

Here below is the proof of some properties of function *ipurge-ref-aux-less* used in what follows.

lemma *ipurge-ref-aux-less-last*:
ipurge-ref-aux-less I D U (*xs* @ [*x*]) *X* =
(*if* ∃ *v* ∈ *U* ∪ *sinks-aux-less I D U xs*. (*v*, *D x*) ∈ *I*
then ipurge-ref-aux-less I D U xs {*x′* ∈ *X*. (*D x*, *D x′*) ∉ *I*}
else ipurge-ref-aux-less I D U xs X)
⟨*proof*⟩

lemma *ipurge-ref-aux-less-nil*:
ipurge-ref-aux-less I D U xs (*ipurge-ref-aux I D U* [] *X*) =
ipurge-ref-aux I D U xs X
⟨*proof*⟩

lemma *ipurge-ref-aux-less-cons-1*:
assumes *A*: ∃ *u* ∈ *U*. (*u*, *D x*) ∈ *I*
shows *ipurge-ref-aux-less I D U* (*x* # *xs*) *X* =
ipurge-ref-aux-less I D U (*ipurge-tr I D* (*D x*) *xs*) (*ipurge-ref I D* (*D x*) *xs X*)
⟨*proof*⟩

lemma *ipurge-ref-aux-less-cons-2*:
¬ (∃ *u* ∈ *U*. (*u*, *D x*) ∈ *I*) ⟹
ipurge-ref-aux-less I D U (*x* # *xs*) *X* =
ipurge-ref-aux-less I D U xs X
⟨*proof*⟩

1.3 Closure of the failures of a secure process under intransitive purge

The intransitive purge of an event list *xs* with regard to a policy *I*, an event-domain map *D*, and a set of domains *U* can equivalently be computed as follows: for each item *x* of *xs*, if *x* may be affected by some domain in *U*, discard *x* and go on recursively using *ipurge-tr I D* (*D x*) *xs′* as input, where *xs′* is the sublist of *xs* following *x*; otherwise, retain *x* and go on recursively using *xs′* as input.

In fact, in each recursive step, any item allowed to be indirectly affected by *U* through the effect of some item preceding *x* within *xs* has already been removed from the list. Hence, it is sufficient to check whether *x* may be directly affected by *U*, and remove *x*, as well as any residual item allowed to be affected by *x*, if this is the case.

Similarly, the intransitive purge of an event set *X* with regard to a policy *I*, an event-domain map *D*, a set of domains *U*, and an event list *xs* can be

computed as follows. First of all, compute *ipurge-ref-aux I D U* [] *X* and use this set, along with *xs*, as the input for the subsequent step. Then, for each item *x* of *xs*, if *x* may be affected by some domain in *U*, go on recursively using *ipurge-tr I D* (*D x*) *xs′* and *ipurge-ref I D* (*D x*) *xs′ X′* as input, where *X′* is the set input to the current recursive step; otherwise, go on recursively using *xs′* and *X′* as input.

In fact, in each recursive step, any item allowed to be affected by *U* either directly, or through the effect of some item preceding *x* within *xs*, has already been removed from the set (in the initial step and in subsequent steps, respectively). Thus, it is sufficient to check whether *x* may be directly affected by *U*, and remove any residual item allowed to be affected by *x* if this is the case.

Assume that the two computations be performed simultaneously by a single function, which will then take as input an event list-event set pair and return as output another such pair. Then, if the input pair is a failure of a secure process, the output pair is still a failure. In fact, for each item *x* of *xs* allowed to be affected by *U*, if *ys* is the partial output list for the sublist of *xs* preceding *x*, then (*ys* @ *ipurge-tr I D* (*D x*) *xs′*, *ipurge-ref I D* (*D x*) *xs′ X′*) is a failure provided that such is (*ys* @ *x* # *xs′*, *X′*), by virtue of the definition of CSP noninterference security [8]. Hence, the property of being a failure is conserved upon each recursive call by the event list-event set pair such that the list matches the concatenation of the partial output list with the residual input list, and the set matches the residual input set. This holds until the residual input list is nil, which is the base case determining the end of the computation.

As shown by this argument, a proof by induction that the output event list-event set pair, under the aforesaid assumptions, is still a failure, requires that the partial output list be passed to the function as a further argument, in addition to the residual input list, in the recursive calls contained within the definition of the function. Therefore, the output list has to be accumulated into a parameter of the function, viz. the function needs to be tail-recursive. This suggests to prove the properties of interest of the function by applying the ten-step proof method for theorems on tail-recursive functions described in [7].

The starting point is to formulate a naive definition of the function, which will then be refined as specified by the proof method. A slight complication is due to the preliminary replacement of the input event set *X* with *ipurge-ref-aux I D U* [] *X*, to be performed before the items of the input event list start to be consumed recursively. A simple solution to this problem is to nest the accumulator of the output list within data type *option*. In this way, the initial state can be distinguished from the subsequent one, in which the input event list starts to be consumed, by assigning the distinct values *None* and *Some* [], respectively, to the accumulator.

Everything is now ready for giving a naive definition of the function under consideration:

function (*sequential*) *ipurge-fail-aux-t-naive* ::
$('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd\ set \Rightarrow 'a\ list \Rightarrow 'a\ list\ option \Rightarrow 'a\ set \Rightarrow$
$'a\ failure$
where
ipurge-fail-aux-t-naive I D U xs None X =
ipurge-fail-aux-t-naive I D U xs (*Some* []) (*ipurge-ref-aux I D U* [] *X*) |
ipurge-fail-aux-t-naive I D U (*x* # *xs*) (*Some ys*) *X* =
(*if* $\exists\, u \in U.\ (u, D\ x) \in I$
then ipurge-fail-aux-t-naive I D U
(*ipurge-tr I D* (*D x*) *xs*) (*Some ys*) (*ipurge-ref I D* (*D x*) *xs X*)
else ipurge-fail-aux-t-naive I D U
xs (*Some* (*ys* @ [*x*])) *X*) |
ipurge-fail-aux-t-naive - - - - (*Some ys*) *X* = (*ys*, *X*)
⟨*proof*⟩

The parameter into which the output list is accumulated is the last but one.
As shown by the above informal argument, function *ipurge-fail-aux-t-naive* enjoys the following properties:

fst (*ipurge-fail-aux-t-naive I D U xs None X*) = *ipurge-tr-aux I D U xs*

snd (*ipurge-fail-aux-t-naive I D U xs None X*) = *ipurge-ref-aux I D U xs X*

⟦*secure P I D*; (*xs*, *X*) ∈ *failures P*⟧ ⟹ *ipurge-fail-aux-t-naive I D U xs None X* ∈ *failures P*

which altogether imply the target lemma, viz. the closure of the failures of a secure process under intransitive purge.
In what follows, the steps provided for by the aforesaid proof method will be dealt with one after the other, with the purpose of proving the target closure lemma in the final step. For more information on this proof method, cf. [7].

1.3.1 Step 1

In the definition of the auxiliary tail-recursive function *ipurge-fail-aux-t-aux*, the Cartesian product of the input parameter types of function *ipurge-fail-aux-t-naive* will be implemented as the following record type:

record ('*a*, '*d*) *ipurge-rec* =

Pol :: $('d \times 'd)$ *set*
Map :: $'a \Rightarrow 'd$
Doms :: $'d$ *set*
List :: $'a$ *list*
ListOp :: $'a$ *list option*
Set :: $'a$ *set*

Here below is the resulting definition of function *ipurge-fail-aux-t-aux*:

function *ipurge-fail-aux-t-aux* :: $('a, 'd)$ *ipurge-rec* $\Rightarrow$ $('a, 'd)$ *ipurge-rec*
where

ipurge-fail-aux-t-aux $(\!|Pol = I, Map = D, Doms = U, List = xs,$
ListOp = *None*, *Set* = $X|\!)$ =
ipurge-fail-aux-t-aux $(\!|Pol = I, Map = D, Doms = U, List = xs,$
ListOp = *Some* [], *Set* = *ipurge-ref-aux I D U* [] $X|\!)$ |

ipurge-fail-aux-t-aux $(\!|Pol = I, Map = D, Doms = U, List = x \# xs,$
ListOp = *Some ys*, *Set* = $X|\!)$ =
(*if* $\exists\, u \in U.\ (u, D\ x) \in I$
then ipurge-fail-aux-t-aux $(\!|Pol = I, Map = D, Doms = U,$
List = *ipurge-tr I D* $(D\ x)$ *xs*, *ListOp* = *Some ys*,
Set = *ipurge-ref I D* $(D\ x)$ *xs* $X|\!)$
else ipurge-fail-aux-t-aux $(\!|Pol = I, Map = D, Doms = U,$
List = *xs*, *ListOp* = *Some* $(ys \mathbin{@} [x])$, *Set* = $X|\!)$) |

ipurge-fail-aux-t-aux
$(\!|Pol = I, Map = D, Doms = U, List = [], ListOp = Some\ ys, Set = X|\!) =$
$(\!|Pol = I, Map = D, Doms = U, List = [], ListOp = Some\ ys, Set = X|\!)$

⟨*proof*⟩

The length of the input event list of function *ipurge-fail-aux-t-aux* decreases in every recursive call except for the first one, where the input list is left unchanged while the nested output list passes from *None* to *Some* []. A measure function decreasing in the first recursive call as well can then be obtained by increasing the length of the input list by one in case the nested output list matches *None*. Using such a measure function, the termination of function *ipurge-fail-aux-t-aux* is guaranteed by the fact that the event lists output by function *ipurge-tr* are not longer than the corresponding input ones.

termination *ipurge-fail-aux-t-aux*
⟨*proof*⟩

1.3.2 Step 2

definition *ipurge-fail-aux-t-in* ::
$('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd\ set \Rightarrow 'a\ list \Rightarrow 'a\ set \Rightarrow ('a, 'd)\ ipurge\text{-}rec$
where
ipurge-fail-aux-t-in I D U xs X ≡
$(\!|Pol = I, Map = D, Doms = U, List = xs, ListOp = None, Set = X|\!)$

definition *ipurge-fail-aux-t-out* :: $('a, 'd)\ ipurge\text{-}rec \Rightarrow 'a\ failure$ **where**
ipurge-fail-aux-t-out Y ≡ (*case ListOp Y of Some ys* ⇒ *ys*, *Set Y*)

definition *ipurge-fail-aux-t* ::
$('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd\ set \Rightarrow 'a\ list \Rightarrow 'a\ set \Rightarrow 'a\ failure$
where
ipurge-fail-aux-t I D U xs X ≡
ipurge-fail-aux-t-out (*ipurge-fail-aux-t-aux* (*ipurge-fail-aux-t-in I D U xs X*))

Since the significant inputs of function *ipurge-fail-aux-t-naive* match pattern -, -, -, -, *None*, -, those of function *ipurge-fail-aux-t-aux*, as returned by function *ipurge-fail-aux-t-in*, match pattern $(\!|Pol = \text{-}, Map = \text{-}, Doms = \text{-}, List = \text{-}, ListOp = None, Set = \text{-}|\!)$.

Likewise, since the nested output lists returned by function *ipurge-fail-aux-t-aux* match pattern *Some* -, function *ipurge-fail-aux-t-out* does not need to worry about dealing with nested output lists equal to *None*.

In terms of function *ipurge-fail-aux-t*, the statements to be proven in order to demonstrate the target closure lemma, previously expressed using function *ipurge-fail-aux-t-naive* and henceforth respectively named *ipurge-fail-aux-t-eq-tr*, *ipurge-fail-aux-t-eq-ref*, and *ipurge-fail-aux-t-failures*, take the following form:

fst (*ipurge-fail-aux-t I D U xs X*) = *ipurge-tr-aux I D U xs*

snd (*ipurge-fail-aux-t I D U xs X*) = *ipurge-ref-aux I D U xs X*

⟦*secure P I D*; (*xs*, *X*) ∈ *failures P*⟧ ⟹ *ipurge-fail-aux-t I D U xs X* ∈ *failures P*

1.3.3 Step 3

inductive-set *ipurge-fail-aux-t-set* ::
$('a, 'd)\ ipurge\text{-}rec \Rightarrow ('a, 'd)\ ipurge\text{-}rec\ set$
for *Y* :: $('a, 'd)\ ipurge\text{-}rec$ **where**

R0: *Y* ∈ *ipurge-fail-aux-t-set Y* |

R1: $(\!|Pol = I, Map = D, Doms = U, List = xs,$

ListOp = *None*, *Set* = *X*⦈ ∈ *ipurge-fail-aux-t-set Y* ⟹
⦇*Pol* = *I*, *Map* = *D*, *Doms* = *U*, *List* = *xs*,
ListOp = *Some* [], *Set* = *ipurge-ref-aux I D U* [] *X*⦈ ∈ *ipurge-fail-aux-t-set Y* |

R2: ⟦⦇*Pol* = *I*, *Map* = *D*, *Doms* = *U*, *List* = *x* # *xs*,
ListOp = *Some ys*, *Set* = *X*⦈ ∈ *ipurge-fail-aux-t-set Y*;
∃ *u* ∈ *U*. (*u*, *D x*) ∈ *I*⟧ ⟹
⦇*Pol* = *I*, *Map* = *D*, *Doms* = *U*, *List* = *ipurge-tr I D* (*D x*) *xs*,
ListOp = *Some ys*, *Set* = *ipurge-ref I D* (*D x*) *xs X*⦈ ∈ *ipurge-fail-aux-t-set Y* |

R3: ⟦⦇*Pol* = *I*, *Map* = *D*, *Doms* = *U*, *List* = *x* # *xs*,
ListOp = *Some ys*, *Set* = *X*⦈ ∈ *ipurge-fail-aux-t-set Y*;
¬ (∃ *u* ∈ *U*. (*u*, *D x*) ∈ *I*)⟧ ⟹
⦇*Pol* = *I*, *Map* = *D*, *Doms* = *U*, *List* = *xs*,
ListOp = *Some* (*ys* @ [*x*]), *Set* = *X*⦈ ∈ *ipurge-fail-aux-t-set Y*

1.3.4 Step 4

lemma *ipurge-fail-aux-t-subset*:
assumes *A*: *Z* ∈ *ipurge-fail-aux-t-set Y*
shows *ipurge-fail-aux-t-set Z* ⊆ *ipurge-fail-aux-t-set Y*
⟨*proof*⟩

lemma *ipurge-fail-aux-t-aux-set*:
ipurge-fail-aux-t-aux Y ∈ *ipurge-fail-aux-t-set Y*
⟨*proof*⟩

1.3.5 Step 5

definition *ipurge-fail-aux-t-inv-1* ::
('*d* × '*d*) *set* ⇒ ('*a* ⇒ '*d*) ⇒ '*d set* ⇒ '*a list* ⇒ ('*a*, '*d*) *ipurge-rec* ⇒ *bool*
where
ipurge-fail-aux-t-inv-1 I D U xs Y ≡
(*case ListOp Y of None* ⇒ [] | *Some ys* ⇒ *ys*) @ *ipurge-tr-aux I D U* (*List Y*) =
ipurge-tr-aux I D U xs

definition *ipurge-fail-aux-t-inv-2* ::
('*d* × '*d*) *set* ⇒ ('*a* ⇒ '*d*) ⇒ '*d set* ⇒ '*a list* ⇒ '*a set* ⇒
('*a*, '*d*) *ipurge-rec* ⇒ *bool*
where
ipurge-fail-aux-t-inv-2 I D U xs X Y ≡
if ListOp Y = *None*
then List Y = *xs* ∧ *Set Y* = *X*
else ipurge-ref-aux-less I D U (*List Y*) (*Set Y*) = *ipurge-ref-aux I D U xs X*

definition *ipurge-fail-aux-t-inv-3* ::
'*a process* ⇒ ('*d* × '*d*) *set* ⇒ ('*a* ⇒ '*d*) ⇒ '*a list* ⇒ '*a set* ⇒
('*a*, '*d*) *ipurge-rec* ⇒ *bool*
where
ipurge-fail-aux-t-inv-3 P I D xs X Y ≡

secure P I D $\longrightarrow$ (*xs*, *X*) $\in$ *failures P* $\longrightarrow$
((*case ListOp Y of None* $\Rightarrow$ [] | *Some ys* $\Rightarrow$ *ys*) @ *List Y*, *Set Y*) $\in$ *failures P*

Three invariants have been defined, one for each of lemmas *ipurge-fail-aux-t-eq-tr*, *ipurge-fail-aux-t-eq-ref*, and *ipurge-fail-aux-t-failures*. More precisely, the invariants are *ipurge-fail-aux-t-inv-1 I D U xs*, *ipurge-fail-aux-t-inv-2 I D U xs X*, and *ipurge-fail-aux-t-inv-3 P I D xs X*, where the free variables are intended to match those appearing in the aforesaid lemmas.
Particularly:

- The first invariant expresses the fact that in each recursive step, any item of the residual input list *List Y* indirectly affected by *U* through the effect of previous, already consumed items has already been removed from the list, so that applying function *ipurge-tr-aux I D U* to the list is sufficient to obtain the intransitive purge of the whole original list.
- The second invariant expresses the fact that in each recursive step, any item of the residual input set *Set Y* affected by *U* either directly, or through the effect of previous, already consumed items, has already been removed from the set, so that applying function *ipurge-ref-aux-less I D U* (*List Y*) to the set is sufficient to obtain the intransitive purge of the whole original set.
  The use of function *ipurge-ref-aux-less* ensures that the invariant implies the equality *Set Y* = *ipurge-ref-aux I D U xs X* for *List Y* = [], viz. for the output values of function *ipurge-fail-aux-t-aux*, which is the reason requiring the introduction of function *ipurge-ref-aux-less*.
- The third invariant expresses the fact that in each recursive step, the event list-event set pair such that the list matches the concatenation of the partial output list with *List Y*, and the set matches *Set Y*, is a failure provided that the original input pair is such as well.

1.3.6 Step 6

lemma *ipurge-fail-aux-t-input-1*:
ipurge-fail-aux-t-inv-1 I D U xs
(|*Pol* = *I*, *Map* = *D*, *Doms* = *U*, *List* = *xs*, *ListOp* = *None*, *Set* = *X*|)
⟨*proof*⟩

lemma *ipurge-fail-aux-t-input-2*:
ipurge-fail-aux-t-inv-2 I D U xs X
(|*Pol* = *I*, *Map* = *D*, *Doms* = *U*, *List* = *xs*, *ListOp* = *None*, *Set* = *X*|)
⟨*proof*⟩

lemma *ipurge-fail-aux-t-input-3*:
ipurge-fail-aux-t-inv-3 P I D xs X
(|Pol = I, Map = D, Doms = U, List = xs, ListOp = None, Set = X|)
⟨*proof*⟩

1.3.7 Step 7

definition *ipurge-fail-aux-t-form* :: ('*a*, '*d*) *ipurge-rec* ⇒ *bool* **where**
ipurge-fail-aux-t-form Y ≡
case ListOp Y of None ⇒ *False* | *Some ys* ⇒ *List Y* = []

lemma *ipurge-fail-aux-t-intro-1*:
⟦*ipurge-fail-aux-t-inv-1 I D U xs Y*; *ipurge-fail-aux-t-form Y*⟧ ⟹
fst (*ipurge-fail-aux-t-out Y*) = *ipurge-tr-aux I D U xs*
⟨*proof*⟩

lemma *ipurge-fail-aux-t-intro-2*:
⟦*ipurge-fail-aux-t-inv-2 I D U xs X Y*; *ipurge-fail-aux-t-form Y*⟧ ⟹
snd (*ipurge-fail-aux-t-out Y*) = *ipurge-ref-aux I D U xs X*
⟨*proof*⟩

lemma *ipurge-fail-aux-t-intro-3*:
⟦*ipurge-fail-aux-t-inv-3 P I D xs X Y*; *ipurge-fail-aux-t-form Y*⟧ ⟹
secure P I D ⟶ (*xs*, *X*) ∈ *failures P* ⟶
ipurge-fail-aux-t-out Y ∈ *failures P*
⟨*proof*⟩

1.3.8 Step 8

lemma *ipurge-fail-aux-t-form-aux*:
ipurge-fail-aux-t-form (*ipurge-fail-aux-t-aux Y*)
⟨*proof*⟩

1.3.9 Step 9

lemma *ipurge-fail-aux-t-invariance-aux*:
Z ∈ *ipurge-fail-aux-t-set Y* ⟹
Pol Z = *Pol Y* ∧ *Map Z* = *Map Y* ∧ *Doms Z* = *Doms Y*
⟨*proof*⟩

The lemma just proven, stating the invariance of the first three record fields over inductive set *ipurge-fail-aux-t-set Y*, is used in the following proofs of the invariance of predicates *ipurge-fail-aux-t-inv-1 I D U xs*, *ipurge-fail-aux-t-inv-2 I D U xs X*, and *ipurge-fail-aux-t-inv-3 P I D xs X*.

The equality between the free variables appearing in the predicates and the corresponding fields of the record generating the set, which is required for such invariance properties to hold, is asserted in the enunciation of the

properties by means of record updates. In the subsequent proofs of lemmas *ipurge-fail-aux-t-eq-tr*, *ipurge-fail-aux-t-eq-ref*, and *ipurge-fail-aux-t-failures*, the enforcement of this equality will be ensured by the identification of both predicate variables and record fields with the related free variables appearing in the lemmas.

lemma *ipurge-fail-aux-t-invariance-1*:
$[\![Z \in ipurge\text{-}fail\text{-}aux\text{-}t\text{-}set\ (Y(\!|Pol := I,\ Map := D,\ Doms := U|\!));$
$\quad ipurge\text{-}fail\text{-}aux\text{-}t\text{-}inv\text{-}1\ I\ D\ U\ xs\ (Y(\!|Pol := I,\ Map := D,\ Doms := U|\!))]\!] \Longrightarrow$
$\ ipurge\text{-}fail\text{-}aux\text{-}t\text{-}inv\text{-}1\ I\ D\ U\ xs\ Z$
⟨*proof*⟩

lemma *ipurge-fail-aux-t-invariance-2*:
$[\![Z \in ipurge\text{-}fail\text{-}aux\text{-}t\text{-}set\ (Y(\!|Pol := I,\ Map := D,\ Doms := U|\!));$
$\quad ipurge\text{-}fail\text{-}aux\text{-}t\text{-}inv\text{-}2\ I\ D\ U\ xs\ X\ (Y(\!|Pol := I,\ Map := D,\ Doms := U|\!))]\!] \Longrightarrow$
$\ ipurge\text{-}fail\text{-}aux\text{-}t\text{-}inv\text{-}2\ I\ D\ U\ xs\ X\ Z$
⟨*proof*⟩

lemma *ipurge-fail-aux-t-invariance-3*:
$[\![Z \in ipurge\text{-}fail\text{-}aux\text{-}t\text{-}set\ (Y(\!|Pol := I,\ Map := D|\!));$
$\quad ipurge\text{-}fail\text{-}aux\text{-}t\text{-}inv\text{-}3\ P\ I\ D\ xs\ X\ (Y(\!|Pol := I,\ Map := D|\!))]\!] \Longrightarrow$
$\ ipurge\text{-}fail\text{-}aux\text{-}t\text{-}inv\text{-}3\ P\ I\ D\ xs\ X\ Z$
⟨*proof*⟩

1.3.10 Step 10

Here below are the proofs of lemmas *ipurge-fail-aux-t-eq-tr*, *ipurge-fail-aux-t-eq-ref*, and *ipurge-fail-aux-t-failures*, which are then applied to demonstrate the target closure lemma.

lemma *ipurge-fail-aux-t-eq-tr*:
$fst\ (ipurge\text{-}fail\text{-}aux\text{-}t\ I\ D\ U\ xs\ X) = ipurge\text{-}tr\text{-}aux\ I\ D\ U\ xs$
⟨*proof*⟩

lemma *ipurge-fail-aux-t-eq-ref*:
$snd\ (ipurge\text{-}fail\text{-}aux\text{-}t\ I\ D\ U\ xs\ X) = ipurge\text{-}ref\text{-}aux\ I\ D\ U\ xs\ X$
⟨*proof*⟩

lemma *ipurge-fail-aux-t-failures* [*rule-format*]:
$secure\ P\ I\ D \longrightarrow (xs,\ X) \in failures\ P \longrightarrow$
$\quad ipurge\text{-}fail\text{-}aux\text{-}t\ I\ D\ U\ xs\ X \in failures\ P$
⟨*proof*⟩

lemma *ipurge-tr-ref-aux-failures*:
$[\![secure\ P\ I\ D;\ (xs,\ X) \in failures\ P]\!] \Longrightarrow$
$\quad (ipurge\text{-}tr\text{-}aux\ I\ D\ U\ xs,\ ipurge\text{-}ref\text{-}aux\ I\ D\ U\ xs\ X) \in failures\ P$
⟨*proof*⟩

1.4 Additional propaedeutic lemmas

In what follows, additional lemmas required for the demonstration of the target security conservation theorem are proven.

Here below is the proof of some properties of functions *ipurge-tr-aux* and *ipurge-ref-aux*. Particularly, it is shown that in case an event list and its intransitive purge for some set of domains are both traces of a secure process, and the purged list has a future not affected by any purged event, then that future is also a future for the full event list.

lemma *ipurge-tr-aux-idem*:
ipurge-tr-aux I D U (*ipurge-tr-aux I D U xs*) = *ipurge-tr-aux I D U xs*
⟨*proof*⟩

lemma *ipurge-tr-aux-set*:
set (*ipurge-tr-aux I D U xs*) ⊆ *set xs*
⟨*proof*⟩

lemma *ipurge-tr-aux-nil* [*rule-format*]:
assumes *A*: *u* ∈ *U*
shows (∀ *x* ∈ *set xs*. (*u*, *D x*) ∈ *I*) ⟶ *ipurge-tr-aux I D U xs* = []
⟨*proof*⟩

lemma *ipurge-tr-aux-del-failures* [*rule-format*]:
assumes *S*: *secure P I D*
shows (∀ *u* ∈ *sinks-aux-less I D U ys*. ∀ *z* ∈ *Z* ∪ *set zs*. (*u*, *D z*) ∉ *I*) ⟶
(*xs* @ *ipurge-tr-aux I D U ys* @ *zs*, *Z*) ∈ *failures P* ⟶
xs @ *ys* ∈ *traces P* ⟶
(*xs* @ *ys* @ *zs*, *Z*) ∈ *failures P*
⟨*proof*⟩

lemma *ipurge-ref-aux-append*:
ipurge-ref-aux I D U (*xs* @ *ys*) *X* = *ipurge-ref-aux I D* (*sinks-aux I D U xs*) *ys X*
⟨*proof*⟩

lemma *ipurge-ref-aux-empty* [*rule-format*]:
assumes
A: *u* ∈ *sinks-aux I D U xs* **and**
B: ∀ *x* ∈ *X*. (*u*, *D x*) ∈ *I*
shows *ipurge-ref-aux I D U xs X* = {}
⟨*proof*⟩

Here below is the proof of some properties of functions *sinks*, *ipurge-tr*, and *ipurge-ref*. Particularly, using the previous analogous result on function *ipurge-tr-aux*, it is shown that in case an event list and its intransitive purge for some domain are both traces of a secure process, and the purged list has a future not affected by any purged event, then that future is also a future

for the full event list.

lemma *sinks-idem*:
sinks I D u (ipurge-tr I D u xs) = {}
⟨*proof*⟩

lemma *sinks-elem* [*rule-format*]:
v ∈ sinks I D u xs ⟶ (∃ x ∈ set xs. v = D x)
⟨*proof*⟩

lemma *ipurge-tr-append*:
ipurge-tr I D u (xs @ ys) =
ipurge-tr I D u xs @ ipurge-tr-aux I D (insert u (sinks I D u xs)) ys
⟨*proof*⟩

lemma *ipurge-tr-idem*:
ipurge-tr I D u (ipurge-tr I D u xs) = ipurge-tr I D u xs
⟨*proof*⟩

lemma *ipurge-tr-set*:
set (ipurge-tr I D u xs) ⊆ set xs
⟨*proof*⟩

lemma *ipurge-tr-del-failures* [*rule-format*]:
assumes
S: *secure P I D* **and**
A: *∀ v ∈ sinks I D u ys. ∀ z ∈ Z ∪ set zs. (v, D z) ∉ I* **and**
B: *(xs @ ipurge-tr I D u ys @ zs, Z) ∈ failures P* **and**
C: *xs @ ys ∈ traces P*
shows *(xs @ ys @ zs, Z) ∈ failures P*
⟨*proof*⟩

lemma *ipurge-tr-del-traces* [*rule-format*]:
assumes
S: *secure P I D* **and**
A: *∀ v ∈ sinks I D u ys. ∀ z ∈ set zs. (v, D z) ∉ I* **and**
B: *xs @ ipurge-tr I D u ys @ zs ∈ traces P* **and**
C: *xs @ ys ∈ traces P*
shows *xs @ ys @ zs ∈ traces P*
⟨*proof*⟩

lemma *ipurge-ref-append*:
ipurge-ref I D u (xs @ ys) X =
ipurge-ref-aux I D (insert u (sinks I D u xs)) ys X
⟨*proof*⟩

lemma *ipurge-ref-distrib-inter*:
ipurge-ref I D u xs (X ∩ Y) = ipurge-ref I D u xs X ∩ ipurge-ref I D u xs Y
⟨*proof*⟩

lemma *ipurge-ref-distrib-union*:
ipurge-ref I D u xs $(X \cup Y)$ = *ipurge-ref I D u xs X* $\cup$ *ipurge-ref I D u xs Y*
⟨*proof*⟩

lemma *ipurge-ref-subset*:
ipurge-ref I D u xs X $\subseteq$ *X*
⟨*proof*⟩

lemma *ipurge-ref-subset-union*:
ipurge-ref I D u xs $(X \cup Y) \subseteq X \cup$ *ipurge-ref I D u xs Y*
⟨*proof*⟩

lemma *ipurge-ref-subset-insert*:
ipurge-ref I D u xs (*insert x X*) $\subseteq$ *insert x* (*ipurge-ref I D u xs X*)
⟨*proof*⟩

lemma *ipurge-ref-empty* [*rule-format*]:
assumes
A: $v = u \vee v \in$ *sinks I D u xs* **and**
B: $\forall x \in X.\ (v,\ D\ x) \in I$
shows *ipurge-ref I D u xs X* = {}
⟨*proof*⟩

Finally, in what follows, properties *process-prop-1*, *process-prop-5*, and *process-prop-6* of processes (cf. [8]) are put into the form of introduction rules.

lemma *process-rule-1*:
([], {}) $\in$ *failures P*
⟨*proof*⟩

lemma *process-rule-5* [*rule-format*]:
xs $\in$ *divergences P* $\longrightarrow$ *xs* @ [*x*] $\in$ *divergences P*
⟨*proof*⟩

lemma *process-rule-6* [*rule-format*]:
xs $\in$ *divergences P* $\longrightarrow$ (*xs*, *X*) $\in$ *failures P*
⟨*proof*⟩

end

2 Sequential composition and noninterference security

theory *SequentialComposition*
imports *Propaedeutics*
begin

This section formalizes the definitions of sequential processes and sequential composition given in [1], and then proves that under the assumptions discussed above, noninterference security is conserved under sequential composition for any pair of processes sharing an alphabet that contains successful termination. Finally, this result is generalized to an arbitrary list of processes.

2.1 Sequential processes

In [1], a *sequential process* is defined as a process whose alphabet contains successful termination. Since sequential composition applies to sequential processes, the first problem put by the formalization of this operation is that of finding a suitable way to represent such a process.

A simple but effective strategy is to identify it with a process having alphabet *'a option*, where *'a* is the native type of its ordinary (i.e. distinct from termination) events. Then, ordinary events will be those matching pattern *Some* -, whereas successful termination will be denoted by the special event *None*. This means that the *sentences* of a sequential process, defined in [1] as the traces after which the process can terminate successfully, will be nothing but the event lists *xs* such that *xs* @ [*None*] is a trace (which implies that *xs* is a trace as well).

Once a suitable representation of successful termination has been found, the next step is to formalize the properties of sequential processes related to this event, expressing them in terms of the selected representation. The first of the resulting predicates, *weakly-sequential*, is the minimum required for allowing the identification of event *None* with successful termination, namely that *None* may occur in a trace as its last event only. The second predicate, *sequential*, following what Hoare does in [1], extends the first predicate with an additional requirement, namely that whenever the process can engage in event *None*, it cannot engage in any other event. A simple counterexample shows that this requirement does not imply the first one: a process whose traces are {[], [*None*], [*None*, *None*]} satisfies the second requirement, but not the first one.

Moreover, here below is the definition of a further predicate, *secure-termination*, which applies to a security policy rather than to a process, and is satisfied just in case the policy does not allow event *None* to be affected by confidential events, viz. by ordinary events not allowed to affect some event in the alphabet. Interestingly, this property, which will prove to be necessary for the target theorem to hold, is nothing but the CSP counterpart of a condition required for a security type system to enforce termination-sensitive noninterference security of programs, namely that program termination must not depend on confidential data (cf. [5], section 9.2.6).

definition *sentences* :: *'a option process* ⇒ *'a option list set* **where**
sentences P ≡ {*xs*. *xs* @ [*None*] ∈ *traces P*}

definition *weakly-sequential* :: *'a option process* ⇒ *bool* **where**
weakly-sequential P ≡
∀ *xs* ∈ *traces P*. *None* ∉ *set* (*butlast xs*)

definition *sequential* :: *'a option process* ⇒ *bool* **where**
sequential P ≡
(∀ *xs* ∈ *traces P*. *None* ∉ *set* (*butlast xs*)) ∧
(∀ *xs* ∈ *sentences P*. *next-events P xs* = {*None*})

definition *secure-termination* :: (*'d* × *'d*) *set* ⇒ (*'a option* ⇒ *'d*) ⇒ *bool* **where**
secure-termination I D ≡
∀ *x*. (*D x*, *D None*) ∈ *I* ∧ *x* ≠ *None* ⟶ (∀ *u* ∈ *range D*. (*D x*, *u*) ∈ *I*)

Here below is the proof of some useful lemmas involving the constants just defined. Particularly, it is proven that process sequentiality is indeed stronger than weak sequentiality, and a sentence of a refusals union closed (cf. [9]), sequential process admits the set of all the ordinary events of the process as a refusal. The use of the latter lemma in the proof of the target security conservation theorem is the reason why the theorem requires to assume that the first of the processes to be composed be refusals union closed (cf. below).

lemma *seq-implies-weakly-seq*:
sequential P ⟹ *weakly-sequential P*
⟨*proof*⟩

lemma *weakly-seq-sentences-none*:
assumes
WS: *weakly-sequential P* **and**
A: *xs* ∈ *sentences P*
shows *None* ∉ *set xs*
⟨*proof*⟩

lemma *seq-sentences-none*:
assumes
S: *sequential P* **and**
A: *xs* ∈ *sentences P* **and**
B: *xs* @ *y* # *ys* ∈ *traces P*
shows *y* = *None*
⟨*proof*⟩

lemma *seq-sentences-ref*:
assumes

A: *ref-union-closed P* **and**
B: *sequential P* **and**
C: *xs* ∈ *sentences P*
shows (*xs*, {*x*. *x* ≠ *None*}) ∈ *failures P*
(**is** (-, *?X*) ∈ -)
⟨*proof*⟩

2.2 Sequential composition

In what follows, the definition of the failures resulting from the sequential composition of two processes P, Q given in [1] is formalized as the inductive definition of set *seq-comp-failures P Q*. Then, the sequential composition of P and Q, denoted by means of notation P ; Q following [1], is defined as the process having *seq-comp-failures P Q* as failures set and the empty set as divergences set.

For the sake of generality, this definition is based on the mere implicit assumption that the input processes be weakly sequential, rather than sequential. This slightly complicates things, since the sentences of process P may number further events in addition to *None* in their future.

Therefore, the resulting refusals of a sentence *xs* of P will have the form *insert None X* ∩ Y, where X is a refusal of *xs* in P and Y is an initial refusal of Q (cf. rule *SCF-R2*). In fact, after *xs*, process P ; Q must be able to refuse *None* if Q is, whereas it cannot refuse an ordinary event unless both P and Q, in their respective states, can.

Moreover, a trace *xs* of P ; Q may result from different combinations of a sentence of P with a trace of Q. Thus, in order that the refusals of P ; Q be closed under set union, the union of any two refusals of *xs* must still be a refusal (cf. rule *SCF-R4*). Indeed, this property will prove to be sufficient to ensure that for any two processes whose refusals are closed under set union, their sequential composition still be such, which is what is expected for any process of practical significance (cf. [9]).

According to the definition given in [1], a divergence of P ; Q is either a divergence of P, or the concatenation of a sentence of P with a divergence of Q. Apparently, this definition does not match the formal one stated here below, which identifies the divergences set of P ; Q with the empty set. Nonetheless, as remarked above, sequential composition does not make sense unless the input processes are weakly sequential, since this is the minimum required to confer the meaning of successful termination on the corresponding alphabet symbol. But a weakly sequential process cannot have any divergence, so that the two definitions are actually equivalent. In fact, a divergence is a trace such that, however it is extended with arbitrary additional events, the resulting event list is still a trace (cf. process properties *process-prop-5* and *process-prop-6* in [8]). Therefore, if *xs* were a divergence, then *xs* @ [*None*, *None*] would be a trace, which is impossible in case the process satisfies

predicate *weakly-sequential*.

inductive-set *seq-comp-failures* ::
'a option process ⇒ *'a option process* ⇒ *'a option failure set*
for *P* :: *'a option process* **and** *Q* :: *'a option process* **where**

SCF-R1: ⟦*xs* ∉ *sentences P*; (*xs*, *X*) ∈ *failures P*; *None* ∉ *set xs*⟧ ⟹
(*xs*, *X*) ∈ *seq-comp-failures P Q* |

SCF-R2: ⟦*xs* ∈ *sentences P*; (*xs*, *X*) ∈ *failures P*; ([], *Y*) ∈ *failures Q*⟧ ⟹
(*xs*, *insert None X* ∩ *Y*) ∈ *seq-comp-failures P Q* |

SCF-R3: ⟦*xs* ∈ *sentences P*; (*ys*, *Y*) ∈ *failures Q*; *ys* ≠ []⟧ ⟹
(*xs* @ *ys*, *Y*) ∈ *seq-comp-failures P Q* |

SCF-R4: ⟦(*xs*, *X*) ∈ *seq-comp-failures P Q*; (*xs*, *Y*) ∈ *seq-comp-failures P Q*⟧ ⟹
(*xs*, *X* ∪ *Y*) ∈ *seq-comp-failures P Q*

definition *seq-comp* ::
'a option process ⇒ *'a option process* ⇒ *'a option process* (**infixl** ‹;› *60*)
where
P ; *Q* ≡ *Abs-process* (*seq-comp-failures P Q*, {})

Here below is the proof that, for any two processes P, Q defined over the same alphabet containing successful termination, set *seq-comp-failures P Q* indeed enjoys the characteristic properties of the failures set of a process as defined in [8] provided that P is weakly sequential, which is what happens in any meaningful case.

lemma *seq-comp-prop-1*:
([], {}) ∈ *seq-comp-failures P Q*
⟨*proof*⟩

lemma *seq-comp-prop-2-aux* [*rule-format*]:
assumes *WS*: *weakly-sequential P*
shows (*ws*, *X*) ∈ *seq-comp-failures P Q* ⟹
ws = *xs* @ [*x*] ⟶ (*xs*, {}) ∈ *seq-comp-failures P Q*
⟨*proof*⟩

lemma *seq-comp-prop-2*:
assumes *WS*: *weakly-sequential P*
shows (*xs* @ [*x*], *X*) ∈ *seq-comp-failures P Q* ⟹
(*xs*, {}) ∈ *seq-comp-failures P Q*
⟨*proof*⟩

lemma *seq-comp-prop-3* [*rule-format*]:
(*xs*, *Y*) ∈ *seq-comp-failures P Q* ⟹ *X* ⊆ *Y* ⟶

$(xs,\ X) \in$ *seq-comp-failures* $P\ Q$
⟨*proof*⟩

lemma *seq-comp-prop-4*:
assumes *WS*: *weakly-sequential* P
shows $(xs,\ X) \in$ *seq-comp-failures* $P\ Q \Longrightarrow$
$(xs\ @\ [x],\ \{\}) \in$ *seq-comp-failures* $P\ Q \vee$
$(xs,\ insert\ x\ X) \in$ *seq-comp-failures* $P\ Q$
⟨*proof*⟩

lemma *seq-comp-rep*:
assumes *WS*: *weakly-sequential* P
shows *Rep-process* $(P\ ;\ Q) = (seq\text{-}comp\text{-}failures\ P\ Q,\ \{\})$
⟨*proof*⟩

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 [8] and [9], when acting on the sequential composition of a pair of processes.

lemma *seq-comp-failures*:
weakly-sequential $P \Longrightarrow$
$failures\ (P\ ;\ Q) = seq\text{-}comp\text{-}failures\ P\ Q$
⟨*proof*⟩

lemma *seq-comp-divergences*:
weakly-sequential $P \Longrightarrow$
$divergences\ (P\ ;\ Q) = \{\}$
⟨*proof*⟩

lemma *seq-comp-futures*:
weakly-sequential $P \Longrightarrow$
$futures\ (P\ ;\ Q)\ xs = \{(ys,\ Y).\ (xs\ @\ ys,\ Y) \in seq\text{-}comp\text{-}failures\ P\ Q\}$
⟨*proof*⟩

lemma *seq-comp-traces*:
weakly-sequential $P \Longrightarrow$
$traces\ (P\ ;\ Q) = Domain\ (seq\text{-}comp\text{-}failures\ P\ Q)$
⟨*proof*⟩

lemma *seq-comp-refusals*:
weakly-sequential $P \Longrightarrow$
$refusals\ (P\ ;\ Q)\ xs \equiv seq\text{-}comp\text{-}failures\ P\ Q\ ``\ \{xs\}$
⟨*proof*⟩

lemma *seq-comp-next-events*:
weakly-sequential $P \Longrightarrow$
$next\text{-}events\ (P\ ;\ Q)\ xs = \{x.\ xs\ @\ [x] \in Domain\ (seq\text{-}comp\text{-}failures\ P\ Q)\}$
⟨*proof*⟩

2.3 Conservation of refusals union closure and sequentiality under sequential composition

Here below is the proof that, for any two processes P, Q and any failure $(xs,\ X)$ of $P\ ;\ Q$, the refusal X is the union of a set of refusals where, for any such refusal W, $(xs,\ W)$ is a failure of $P\ ;\ Q$ by virtue of one of rules *SCF-R1*, *SCF-R2*, or *SCF-R3*.

The converse is also proven, under the assumption that the refusals of both P and Q be closed under union: namely, for any trace xs of $P\ ;\ Q$ and any set of refusals where, for any such refusal W, $(xs,\ W)$ is a failure of the aforesaid kind, the union of these refusals is still a refusal of xs.

The proof of the latter lemma makes use of the axiom of choice.

lemma *seq-comp-refusals-1*:
$(xs,\ X) \in \mathit{seq\text{-}comp\text{-}failures}\ P\ Q \Longrightarrow \exists\, R.$
$X = (\bigcup n \in \{..\mathit{length}\ xs\}.\ \bigcup W \in R\ n.\ W) \wedge$
$(\forall\, W \in R\ 0.$
$\mathit{xs} \notin \mathit{sentences}\ P \wedge \mathit{None} \notin \mathit{set}\ xs \wedge (xs,\ W) \in \mathit{failures}\ P \vee$
$\mathit{xs} \in \mathit{sentences}\ P \wedge (\exists\, U\ V.\ (xs,\ U) \in \mathit{failures}\ P \wedge ([],\ V) \in \mathit{failures}\ Q \wedge$
$W = \mathit{insert}\ \mathit{None}\ U \cap V)) \wedge$
$(\forall\, n \in \{0<..\mathit{length}\ xs\}.\ \forall\, W \in R\ n.$
$\mathit{take}\ (\mathit{length}\ xs - n)\ xs \in \mathit{sentences}\ P \wedge$
$(\mathit{drop}\ (\mathit{length}\ xs - n)\ xs,\ W) \in \mathit{failures}\ Q) \wedge$
$(\exists\, n \in \{..\mathit{length}\ xs\}.\ \exists\, W.\ W \in R\ n)$
(**is** - $\Longrightarrow \exists\, R.\ ?T\ R\ xs\ X$)
⟨*proof*⟩

lemma *seq-comp-refusals-finite* [*rule-format*]:
assumes A: $xs \in \mathit{Domain}\ (\mathit{seq\text{-}comp\text{-}failures}\ P\ Q)$
shows *finite* $A \Longrightarrow (\forall\, x \in A.\ (xs,\ F\ x) \in \mathit{seq\text{-}comp\text{-}failures}\ P\ Q) \longrightarrow$
$(xs,\ \bigcup x \in A.\ F\ x) \in \mathit{seq\text{-}comp\text{-}failures}\ P\ Q$
⟨*proof*⟩

lemma *seq-comp-refusals-2*:
assumes
A: *ref-union-closed* P **and**
B: *ref-union-closed* Q **and**
C: $xs \in \mathit{Domain}\ (\mathit{seq\text{-}comp\text{-}failures}\ P\ Q)$ **and**
D: $X = (\bigcup n \in \{..\mathit{length}\ xs\}.\ \bigcup W \in R\ n.\ W) \wedge$
$(\forall\, W \in R\ 0.$
$\mathit{xs} \notin \mathit{sentences}\ P \wedge \mathit{None} \notin \mathit{set}\ xs \wedge (xs,\ W) \in \mathit{failures}\ P \vee$
$\mathit{xs} \in \mathit{sentences}\ P \wedge (\exists\, U\ V.\ (xs,\ U) \in \mathit{failures}\ P \wedge ([],\ V) \in \mathit{failures}\ Q \wedge$
$W = \mathit{insert}\ \mathit{None}\ U \cap V)) \wedge$
$(\forall\, n \in \{0<..\mathit{length}\ xs\}.\ \forall\, W \in R\ n.$
$\mathit{take}\ (\mathit{length}\ xs - n)\ xs \in \mathit{sentences}\ P \wedge$
$(\mathit{drop}\ (\mathit{length}\ xs - n)\ xs,\ W) \in \mathit{failures}\ Q)$
shows $(xs,\ X) \in \mathit{seq\text{-}comp\text{-}failures}\ P\ Q$

⟨*proof*⟩

In what follows, the previous results are used to prove that refusals union closure, weak sequentiality, and sequentiality are conserved under sequential composition. The proof of the first of these lemmas makes use of the axiom of choice.

Since the target security conservation theorem, in addition to the security of both of the processes to be composed, also requires to assume that the first process be refusals union closed and sequential (cf. below), these further conservation lemmas will permit to generalize the theorem to the sequential composition of an arbitrary list of processes.

lemma *seq-comp-ref-union-closed*:
  assumes
    WS: *weakly-sequential P* **and**
    A: *ref-union-closed P* **and**
    B: *ref-union-closed Q*
  shows *ref-union-closed* (*P* ; *Q*)
⟨*proof*⟩

lemma *seq-comp-weakly-sequential*:
  assumes
    A: *weakly-sequential P* **and**
    B: *weakly-sequential Q*
  shows *weakly-sequential* (*P* ; *Q*)
⟨*proof*⟩

lemma *seq-comp-sequential*:
  assumes
    A: *sequential P* **and**
    B: *sequential Q*
  shows *sequential* (*P* ; *Q*)
⟨*proof*⟩

2.4 Conservation of noninterference security under sequential composition

Everything is now ready for proving the target security conservation theorem. The two closure properties that the definition of noninterference security requires process futures to satisfy, one for the addition of events into traces and the other for the deletion of events from traces (cf. [8]), will be faced separately; here below is the proof of the former property. Unsurprisingly, rule induction on set *seq-comp-failures* is applied, and the closure of the failures of a secure process under intransitive purge (proven in the previous section) is used to meet the proof obligations arising from

rule *SCF-R3*.

lemma *seq-comp-secure-aux-1-case-1*:
assumes
A: *secure-termination I D* **and**
B: *sequential P* **and**
C: *secure P I D* **and**
D: *xs @ y # ys ∉ sentences P* **and**
E: *(xs @ y # ys, X) ∈ failures P* **and**
F: *None ≠ y* **and**
G: *None ∉ set xs* **and**
H: *None ∉ set ys*
shows *(xs @ ipurge-tr I D (D y) ys, ipurge-ref I D (D y) ys X)*
∈ seq-comp-failures P Q
⟨*proof*⟩

lemma *seq-comp-secure-aux-1-case-2*:
assumes
A: *secure-termination I D* **and**
B: *sequential P* **and**
C: *secure P I D* **and**
D: *secure Q I D* **and**
E: *xs @ y # ys ∈ sentences P* **and**
F: *(xs @ y # ys, X) ∈ failures P* **and**
G: *([], Y) ∈ failures Q*
shows *(xs @ ipurge-tr I D (D y) ys,*
ipurge-ref I D (D y) ys (insert None X ∩ Y)) ∈ seq-comp-failures P Q
⟨*proof*⟩

lemma *seq-comp-secure-aux-1-case-3*:
assumes
A: *secure-termination I D* **and**
B: *ref-union-closed Q* **and**
C: *sequential Q* **and**
D: *secure Q I D* **and**
E: *secure R I D* **and**
F: *ws ∈ sentences Q* **and**
G: *(ys′, Y) ∈ failures R* **and**
H: *ws @ ys′ = xs @ y # ys*
shows *(xs @ ipurge-tr I D (D y) ys, ipurge-ref I D (D y) ys Y)*
∈ seq-comp-failures Q R
⟨*proof*⟩

lemma *seq-comp-secure-aux-1* [*rule-format*]:
assumes
A: *secure-termination I D* **and**
B: *ref-union-closed P* **and**
C: *sequential P* **and**
D: *secure P I D* **and**

E: *secure Q I D*
shows (*ws*, *Y*) ∈ *seq-comp-failures P Q* ⟹
ws = *xs* @ *y* # *ys* ⟶
(*xs* @ *ipurge-tr I D* (*D y*) *ys*, *ipurge-ref I D* (*D y*) *ys Y*)
∈ *seq-comp-failures P Q*
⟨*proof*⟩

lemma *seq-comp-secure-1*:
assumes
A: *secure-termination I D* **and**
B: *ref-union-closed P* **and**
C: *sequential P* **and**
D: *secure P I D* **and**
E: *secure Q I D*
shows (*xs* @ *y* # *ys*, *Y*) ∈ *seq-comp-failures P Q* ⟹
(*xs* @ *ipurge-tr I D* (*D y*) *ys*, *ipurge-ref I D* (*D y*) *ys Y*)
∈ *seq-comp-failures P Q*
⟨*proof*⟩

This completes the proof that the former requirement for noninterference security is satisfied, so it is the turn of the latter one. Again, rule induction on set *seq-comp-failures* is applied, and the closure of the failures of a secure process under intransitive purge is used to meet the proof obligations arising from rule *SCF-R3*. In more detail, rule induction is applied to the trace into which the event is inserted, and then a case distinction is performed on the trace from which the event is extracted, using the expression of its refusal as union of a set of refusals derived previously.

lemma *seq-comp-secure-aux-2-case-1*:
assumes
A: *secure-termination I D* **and**
B: *sequential P* **and**
C: *secure P I D* **and**
D: *xs* @ *zs* ∉ *sentences P* **and**
E: (*xs* @ *zs*, *X*) ∈ *failures P* **and**
F: *None* ∉ *set xs* **and**
G: *None* ∉ *set zs* **and**
H: (*xs* @ [*y*], {}) ∈ *seq-comp-failures P Q*
shows (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs X*)
∈ *seq-comp-failures P Q*
⟨*proof*⟩

lemma *seq-comp-secure-aux-2-case-2*:
assumes
A: *secure-termination I D* **and**
B: *sequential P* **and**
C: *secure P I D* **and**

D: *secure Q I D* **and**
E: *xs @ zs ∈ sentences P* **and**
F: *(xs @ zs, X) ∈ failures P* **and**
G: *([], Y) ∈ failures Q* **and**
H: *(xs @ [y], {}) ∈ seq-comp-failures P Q*
shows *(xs @ y # ipurge-tr I D (D y) zs,*
ipurge-ref I D (D y) zs (insert None X ∩ Y)) ∈ seq-comp-failures P Q
⟨*proof*⟩

lemma *seq-comp-secure-aux-2-case-3*:
assumes
A: *secure-termination I D* **and**
B: *ref-union-closed P* **and**
C: *sequential P* **and**
D: *secure P I D* **and**
E: *secure Q I D* **and**
F: *ws ∈ sentences P* **and**
G: *(ys, Y) ∈ failures Q* **and**
H: *ys ≠ []* **and**
I: *ws @ ys = xs @ zs* **and**
J: *(xs @ [y], {}) ∈ seq-comp-failures P Q*
shows *(xs @ y # ipurge-tr I D (D y) zs, ipurge-ref I D (D y) zs Y)*
∈ seq-comp-failures P Q
⟨*proof*⟩

lemma *seq-comp-secure-aux-2* [*rule-format*]:
assumes
A: *secure-termination I D* **and**
B: *ref-union-closed P* **and**
C: *sequential P* **and**
D: *secure P I D* **and**
E: *secure Q I D*
shows *(ws, Z) ∈ seq-comp-failures P Q ⟹*
ws = xs @ zs ⟶
(xs @ [y], {}) ∈ seq-comp-failures P Q ⟶
(xs @ y # ipurge-tr I D (D y) zs, ipurge-ref I D (D y) zs Z)
∈ seq-comp-failures P Q
⟨*proof*⟩

lemma *seq-comp-secure-2*:
assumes
A: *secure-termination I D* **and**
B: *ref-union-closed P* **and**
C: *sequential P* **and**
D: *secure P I D* **and**
E: *secure Q I D*
shows *(xs @ zs, Z) ∈ seq-comp-failures P Q ⟹*
(xs @ [y], {}) ∈ seq-comp-failures P Q ⟹
(xs @ y # ipurge-tr I D (D y) zs, ipurge-ref I D (D y) zs Z)

$\in$ *seq-comp-failures P Q*
⟨*proof*⟩

Finally, the target security conservation theorem can be enunciated and proven, which is done here below. The theorem states that for any two processes P, Q defined over the same alphabet containing successful termination, to which the noninterference policy I and the event-domain map D apply, if:

- I and D enforce termination security,
- P is refusals union closed and sequential, and
- both P and Q are secure with respect to I and D,

then P ; Q is secure as well.

theorem *seq-comp-secure*:
assumes
A: *secure-termination I D* **and**
B: *ref-union-closed P* **and**
C: *sequential P* **and**
D: *secure P I D* **and**
E: *secure Q I D*
shows *secure* (*P* ; *Q*) *I D*
⟨*proof*⟩

2.5 Generalization of the security conservation theorem to lists of processes

The target security conservation theorem, in the basic version just proven, applies to the sequential composition of a pair of processes. However, given an arbitrary list of processes where each process satisfies its assumptions, the theorem could be orderly applied to the composition of the first two processes in the list, then to the composition of the resulting process with the third process in the list, and so on, until the last process is reached. The final outcome would be that the sequential composition of all the processes in the list is secure.

Of course, this argument works provided that the assumptions of the theorem keep being satisfied by the composed processes produced in each step of the recursion. But this is what indeed happens, by virtue of the conservation of refusals union closure and sequentiality under sequential composition, proven previously, and of the conservation of security under sequential composition, ensured by the target theorem itself.

Therefore, the target security conservation theorem can be generalized to an arbitrary list of processes, which is done here below. The resulting theorem states that for any nonempty list of processes defined over the same alphabet containing successful termination, to which the noninterference policy I and the event-domain map D apply, if:

- I and D enforce termination security,
- each process in the list, with the possible exception of the last one, is refusals union closed and sequential, and
- each process in the list is secure with respect to I and D,

then the sequential composition of all the processes in the list is secure as well.

As a precondition, the above conservation lemmas for weak sequentiality, refusals union closure, and sequentiality are generalized, too.

lemma *seq-comp-list-weakly-sequential* [*rule-format*]:
$(\forall X \in set\ (P\ \#\ PS).\ weakly\text{-}sequential\ X) \longrightarrow$
$weakly\text{-}sequential\ (foldl\ (;)\ P\ PS)$
⟨*proof*⟩

lemma *seq-comp-list-ref-union-closed* [*rule-format*]:
$(\forall X \in set\ (butlast\ (P\ \#\ PS)).\ weakly\text{-}sequential\ X) \longrightarrow$
$(\forall X \in set\ (P\ \#\ PS).\ ref\text{-}union\text{-}closed\ X) \longrightarrow$
$ref\text{-}union\text{-}closed\ (foldl\ (;)\ P\ PS)$
⟨*proof*⟩

lemma *seq-comp-list-sequential* [*rule-format*]:
$(\forall X \in set\ (P\ \#\ PS).\ sequential\ X) \longrightarrow$
$sequential\ (foldl\ (;)\ P\ PS)$
⟨*proof*⟩

theorem *seq-comp-list-secure* [*rule-format*]:
assumes A: *secure-termination I D*
shows
$(\forall X \in set\ (butlast\ (P\ \#\ PS)).\ ref\text{-}union\text{-}closed\ X \wedge sequential\ X) \longrightarrow$
$(\forall X \in set\ (P\ \#\ PS).\ secure\ X\ I\ D) \longrightarrow$
$secure\ (foldl\ (;)\ P\ PS)\ I\ D$
⟨*proof*⟩

end

3 Necessity of nontrivial assumptions

theory *Counterexamples*

imports *SequentialComposition*
begin

The security conservation theorem proven in this paper contains two non-trivial assumptions; namely, the security policy must satisfy predicate *secure-termination*, and the first input process must satisfy predicate *sequential* instead of *weakly-sequential* alone. This section shows, by means of counterexamples, that both of these assumptions are necessary for the theorem to hold.

In more detail, two counterexamples will be constructed: the former drops the termination security assumption, whereas the latter drops the process sequentiality assumption, replacing it with weak sequentiality alone. In both cases, all the other assumptions of the theorem keep being satisfied.

Both counterexamples make use of reflexive security policies, which is the case for any policy of practical significance, and are based on trace set processes as defined in [9]. The security of the processes input to sequential composition, as well as the insecurity of the resulting process, are demonstrated by means of the Ipurge Unwinding Theorem proven in [9].

3.1 Preliminary definitions and lemmas

Both counterexamples will use the same type *event* as native type of ordinary events, as well as the same process Q as second input to sequential composition. Here below are the definitions of these constants, followed by few useful lemmas on process Q.

datatype *event* = *a* | *b*

definition *Q* :: *event option process* **where**
Q ≡ *ts-process* {[], [*Some b*]}

lemma *trace-set-snd*:
trace-set {[], [*Some b*]}
⟨*proof*⟩

lemmas *failures-snd* = *ts-process-failures* [*OF trace-set-snd*]

lemmas *traces-snd* = *ts-process-traces* [*OF trace-set-snd*]

lemmas *next-events-snd* = *ts-process-next-events* [*OF trace-set-snd*]

lemmas *unwinding-snd* = *ts-ipurge-unwinding* [*OF trace-set-snd*]

3.2 Necessity of termination security

The reason why the conservation of noninterference security under sequential composition requires the security policy to satisfy predicate *secure-termination* is that the second input process cannot engage in its events unless the first process has terminated successfully. Thus, the ordinary events of the first process can indirectly affect the events of the second process by affecting the successful termination of the first process. Therefore, if an ordinary event is allowed to affect successful termination, then the policy must allow it to affect any other event as well, which is exactly what predicate *secure-termination* states.

A counterexample showing the necessity of this assumption can then be constructed by defining a reflexive policy I_1 that allows event *Some a* to affect *None*, but not *Some b*, and a deterministic process P_1 that can engage in *None* only after engaging in *Some a*. The resulting process P_1 ; Q will number [*Some a*, *Some b*], but not [*Some b*], among its traces, so that event *Some a* affects the occurrence of event *Some b* in contrast with policy I_1, viz. P_1 ; Q is not secure with respect to I_1.

Here below are the definitions of constants I_1 and P_1, followed by few useful lemmas on process P_1.

definition I_1 :: (*event option* × *event option*) *set* **where**
$I_1 \equiv \{(Some\ a, None)\}^=$

definition P_1 :: *event option process* **where**
$P_1 \equiv$ *ts-process* {[], [*Some a*], [*Some a*, *None*]}

lemma *trace-set-fst-1*:
trace-set {[], [*Some a*], [*Some a*, *None*]}
⟨*proof*⟩

lemmas *failures-fst-1* = *ts-process-failures* [*OF trace-set-fst-1*]

lemmas *traces-fst-1* = *ts-process-traces* [*OF trace-set-fst-1*]

lemmas *next-events-fst-1* = *ts-process-next-events* [*OF trace-set-fst-1*]

lemmas *unwinding-fst-1* = *ts-ipurge-unwinding* [*OF trace-set-fst-1*]

Here below is the proof that policy I_1 does not satisfy predicate *secure-termination*, whereas the remaining assumptions of the security conservation theorem keep being satisfied. For the sake of simplicity, the identity function is used as event-domain map.

lemma *not-secure-termination-1*:

$\neg$ *secure-termination* I_1 *id*
⟨*proof*⟩

lemma *ref-union-closed-fst-1*:
ref-union-closed P_1
⟨*proof*⟩

lemma *sequential-fst-1*:
sequential P_1
⟨*proof*⟩

lemma *secure-fst-1*:
secure P_1 I_1 *id*
⟨*proof*⟩

lemma *secure-snd-1*:
secure Q I_1 *id*
⟨*proof*⟩

In what follows, the insecurity of process P_1 ; Q is demonstrated by proving that event list [*Some a*, *Some b*] is a trace of the process, whereas [*Some b*] is not.

lemma *traces-comp-1*:
traces (P_1 ; Q) = *Domain* (*seq-comp-failures* P_1 Q)
⟨*proof*⟩

lemma *ref-union-closed-comp-1*:
ref-union-closed (P_1 ; Q)
⟨*proof*⟩

lemma *not-secure-comp-1-aux-aux-1*:
(*xs*, X) ∈ *seq-comp-failures* P_1 Q ⟹ *xs* ≠ [*Some b*]
⟨*proof*⟩

lemma *not-secure-comp-1-aux-1*:
[*Some b*] ∉ *traces* (P_1 ; Q)
⟨*proof*⟩

lemma *not-secure-comp-1-aux-2*:
[*Some a*, *Some b*] ∈ *traces* (P_1 ; Q)
⟨*proof*⟩

lemma *not-secure-comp-1*:
$\neg$ *secure* (P_1 ; Q) I_1 *id*
⟨*proof*⟩

Here below, the previous results are used to show that constants I_1, P_1, Q, and *id* indeed constitute a counterexample to the statement obtained by removing termination security from the assumptions of the security conservation theorem.

lemma *counterexample-1*:
$\neg$ (*ref-union-closed* P_1 $\wedge$
sequential P_1 $\wedge$
secure P_1 I_1 *id* $\wedge$
secure Q I_1 *id* $\longrightarrow$
secure (P_1 ; Q) I_1 *id*)
⟨*proof*⟩

3.3 Necessity of process sequentiality

The reason why the conservation of noninterference security under sequential composition requires the first input process to satisfy predicate *sequential*, instead of the more permissive predicate *weakly-sequential*, is that the possibility for the first process to engage in events alternative to successful termination entails the possibility for the resulting process to engage in events alternative to the initial ones of the second process. Namely, the resulting process would admit some state in which events of the first process can occur in alternative to events of the second process. But neither process, though being secure on its own, will in general be prepared to handle securely the alternative events added by the other process. Therefore, the first process must not admit alternatives to successful termination, which is exactly what predicate *sequential* states in addition to *weakly-sequential*.

A counterexample showing the necessity of this assumption can then be constructed by defining a reflexive policy I_2 that does not allow event *Some b* to affect *Some a*, and a deterministic process P_2 that can engage in *Some a* in alternative to *None*. The resulting process P_2 ; Q will number both [*Some b*] and [*Some a*], but not [*Some b*, *Some a*], among its traces, so that event *Some b* affects the occurrence of event *Some a* in contrast with policy I_2, viz. P_2 ; Q is not secure with respect to I_2.

Here below are the definitions of constants I_2 and P_2, followed by few useful lemmas on process P_2.

definition I_2 :: (*event option* × *event option*) *set* **where**
$I_2 \equiv \{(None, Some\ a)\}^=$

definition P_2 :: *event option process* **where**
$P_2 \equiv$ *ts-process* {[], [*None*], [*Some a*], [*Some a*, *None*]}

lemma *trace-set-fst-2*:
trace-set {[], [*None*], [*Some a*], [*Some a*, *None*]}

⟨*proof*⟩

lemmas *failures-fst-2* = *ts-process-failures* [*OF trace-set-fst-2*]

lemmas *traces-fst-2* = *ts-process-traces* [*OF trace-set-fst-2*]

lemmas *next-events-fst-2* = *ts-process-next-events* [*OF trace-set-fst-2*]

lemmas *unwinding-fst-2* = *ts-ipurge-unwinding* [*OF trace-set-fst-2*]

Here below is the proof that process P_2 does not satisfy predicate *sequential*, but rather predicate *weakly-sequential* only, whereas the remaining assumptions of the security conservation theorem keep being satisfied. For the sake of simplicity, the identity function is used as event-domain map.

lemma *secure-termination-2*:
secure-termination I_2 *id*
⟨*proof*⟩

lemma *ref-union-closed-fst-2*:
ref-union-closed P_2
⟨*proof*⟩

lemma *weakly-sequential-fst-2*:
weakly-sequential P_2
⟨*proof*⟩

lemma *not-sequential-fst-2*:
¬ *sequential* P_2
⟨*proof*⟩

lemma *secure-fst-2*:
secure P_2 I_2 *id*
⟨*proof*⟩

lemma *secure-snd-2*:
secure Q I_2 *id*
⟨*proof*⟩

In what follows, the insecurity of process P_2 ; Q is demonstrated by proving that event lists [*Some b*] and [*Some a*] are traces of the process, whereas [*Some b*, *Some a*] is not.

lemma *traces-comp-2*:
traces (P_2 ; Q) = *Domain* (*seq-comp-failures* P_2 Q)
⟨*proof*⟩

lemma *ref-union-closed-comp-2*:
ref-union-closed $(P_2 \ ; \ Q)$
⟨*proof*⟩

lemma *not-secure-comp-2-aux-aux-1*:
$(xs, \ X) \in$ *seq-comp-failures* $P_2 \ Q \Longrightarrow xs \neq [$*Some* b, *Some* $a]$
⟨*proof*⟩

lemma *not-secure-comp-2-aux-1*:
$[$*Some* b, *Some* $a] \notin$ *traces* $(P_2 \ ; \ Q)$
⟨*proof*⟩

lemma *not-secure-comp-2-aux-2*:
$[$*Some* $a] \in$ *traces* $(P_2 \ ; \ Q)$
⟨*proof*⟩

lemma *not-secure-comp-2-aux-3*:
$[$*Some* $b] \in$ *traces* $(P_2 \ ; \ Q)$
⟨*proof*⟩

lemma *not-secure-comp-2*:
$\neg$ *secure* $(P_2 \ ; \ Q) \ I_2 \ id$
⟨*proof*⟩

Here below, the previous results are used to show that constants I_2, P_2, Q, and *id* indeed constitute a counterexample to the statement obtained by replacing process sequentiality with weak sequentiality in the assumptions of the security conservation theorem.

lemma *counterexample-2*:
$\neg$ (*secure-termination* I_2 *id* $\wedge$
ref-union-closed P_2 $\wedge$
weakly-sequential P_2 $\wedge$
secure P_2 I_2 *id* $\wedge$
secure Q I_2 *id* $\longrightarrow$
secure $(P_2 \ ; \ Q)$ I_2 *id*)
⟨*proof*⟩

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 and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. http://www.concrete-semantics.org/concrete-semantics.pdf.

[6] 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.

[7] P. Noce. A general method for the proof of theorems on tail-recursive functions. *Archive of Formal Proofs*, Dec. 2013. http://isa-afp.org/entries/Tail_Recursive_Functions.shtml, Formal proof development.

[8] 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.

[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.