The Inductive Unwinding Theorem for CSP Noninterference Security

Pasquale Noce
Security Certification Specialist at Arjo Systems - Gep S.p.A.
pasquale dot noce dot lavoro at gmail dot com
pasquale dot noce at arjowiggins-it dot com

December 14, 2021

Abstract

The necessary and sufficient condition for CSP noninterference security stated by the Ipurge Unwinding Theorem is expressed in terms of a pair of event lists varying over the set of process traces. This does not render it suitable for the subsequent application of rule induction in the case of a process defined inductively, since rule induction may rather be applied to a single variable ranging over an inductively defined set.

Starting from the Ipurge Unwinding Theorem, this paper derives a necessary and sufficient condition for CSP noninterference security that involves a single event list varying over the set of process traces, and is thus suitable for rule induction; hence its name, Inductive Unwinding Theorem. Similarly to the Ipurge Unwinding Theorem, the new theorem only requires to consider individual accepted and refused events for each process trace, and applies to the general case of a possibly intransitive noninterference policy. Specific variants of this theorem are additionally proven for deterministic processes and trace set processes.

Contents

1 The Inductive Unwinding Theorem

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

The necessary and sufficient condition for CSP noninterference security [7] stated by the Ipurge Unwinding Theorem [8] is expressed in terms of a pair of event lists varying over the set of process traces. This does not render it suitable for the subsequent application of rule induction in the case of a process defined inductively, since rule induction may rather be applied to a single variable ranging over an inductively defined set (cf. [5]).

However, the formulation of an inductive definition is the standard way of defining a process that admits traces of unbounded length, indeed because it provides rule induction as a powerful method to prove process properties, particularly noninterference security, by considering any indefinitely long trace of the process. Therefore, it is essential to infer some condition equivalent to CSP noninterference security and suitable for being handled by means of rule induction.

Starting from the Ipurge Unwinding Theorem, this paper derives a necessary and sufficient condition for CSP noninterference security that involves a single event list varying over the set of process traces, and is thus suitable for rule induction; hence its name, *Inductive Unwinding Theorem*. Similarly to the Ipurge Unwinding Theorem, the new theorem only requires to consider individual accepted and refused events for each process trace, and applies to the general case of a possibly intransitive noninterference policy. Specific variants of this theorem are additionally proven for deterministic processes and trace set processes [8].

For details about the theory of Communicating Sequential Processes, to which the notion of process security defined in [7] and applied in this paper refers, cf. [1].

As regards the formal contents of 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 lemmas

Here below are the proofs of some lemmas on the constants defined in [7] and [8] which are propaedeutic to the demonstration of the Inductive Unwinding Theorem.

Among other things, the lemmas being proven formalize the following statements:

- A set of domains U may affect a set of domains V via an event list *xs*, as expressed through function *sinks-aux*, just in case V may be affected by U via *xs*, as expressed through function *sources-aux*.
- The event lists output by function *ipurge-tr* are not longer than the corresponding input ones.
- Function *ipurge-tr-rev* is idempotent.

lemma *sources-aux-single-dom*:
sources-aux I D {*u*} *xs* = *insert u* (*sources I D u xs*)
by (*simp add*: *sources-sinks sources-sinks-aux sinks-aux-single-dom*)

lemma *sources-interference-eq*:
((*D x*, *u*) ∈ *I* ∨ (∃ *v* ∈ *sources I D u xs*. (*D x*, *v*) ∈ *I*)) =
(*D x* ∈ *sources I D u* (*x* # *xs*))
proof (*simp only*: *sources-sinks rev.simps*, *subst* (*1 2*) *converse-iff* [*symmetric*])
qed (*rule sinks-interference-eq*)

lemma *ex-sinks-sources-aux-1* [*rule-format*]:
(∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*) ⟶
(∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D V xs*. (*u*, *v*) ∈ *I*)
proof (*induction xs arbitrary*: *V rule*: *rev-induct*, *simp*, *subst sources-aux-append*, *rule impI*)
fix *x xs V*
let
?V = *sources-aux I D V* [*x*] **and**
?V' = *insert* (*D x*) *V*
assume
A: ⋀*V*. (∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*) ⟶
(∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D V xs*. (*u*, *v*) ∈ *I*) **and**
B: ∃ *u* ∈ *sinks-aux I D U* (*xs* @ [*x*]). ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*
show ∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D ?V xs*. (*u*, *v*) ∈ *I*
proof (*cases* ∃ *u* ∈ *sinks-aux I D U xs*. (*u*, *D x*) ∈ *I*)
case *True*
hence (∃ *v* ∈ *V*. (*D x*, *v*) ∈ *I*) ∨
(∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*)
(**is** *?A* ∨ *?B*) **using** *B* **by** *simp*
moreover {

```
    assume ?A
    have (∃ u ∈ sinks-aux I D U xs. ∃ v ∈ ?V'. (u, v) ∈ I) ⟶
      (∃ u ∈ U. ∃ v ∈ sources-aux I D ?V' xs. (u, v) ∈ I)
     using A .
    moreover obtain u where
      C: u ∈ sinks-aux I D U xs and D: (u, D x) ∈ I
     using True ..
    have D x ∈ ?V' by simp
    with D have ∃ v ∈ ?V'. (u, v) ∈ I ..
    hence ∃ u ∈ sinks-aux I D U xs. ∃ v ∈ ?V'. (u, v) ∈ I using C ..
    ultimately have ∃ u ∈ U. ∃ v ∈ sources-aux I D ?V' xs. (u, v) ∈ I ..
    hence ?thesis using ‹?A› by simp
   }
   moreover {
    assume ?B
    have (∃ u ∈ sinks-aux I D U xs. ∃ v ∈ ?V. (u, v) ∈ I) ⟶
      (∃ u ∈ U. ∃ v ∈ sources-aux I D ?V xs. (u, v) ∈ I)
     using A .
    moreover obtain u where
      C: u ∈ sinks-aux I D U xs and D: ∃ v ∈ V. (u, v) ∈ I
     using ‹?B› ..
    have V ⊆ ?V by (rule sources-aux-subset)
    hence ∃ v ∈ ?V. (u, v) ∈ I using D by simp
    hence ∃ u ∈ sinks-aux I D U xs. ∃ v ∈ ?V. (u, v) ∈ I using C ..
    ultimately have ?thesis ..
   }
   ultimately show ?thesis ..
  next
   case False
   have (∃ u ∈ sinks-aux I D U xs. ∃ v ∈ ?V. (u, v) ∈ I) ⟶
     (∃ u ∈ U. ∃ v ∈ sources-aux I D ?V xs. (u, v) ∈ I)
    using A .
   moreover have ∃ u ∈ sinks-aux I D U xs. ∃ v ∈ V. (u, v) ∈ I
    using B and False by simp
   then obtain u where
     C: u ∈ sinks-aux I D U xs and D: ∃ v ∈ V. (u, v) ∈ I ..
   have V ⊆ ?V by (rule sources-aux-subset)
   hence ∃ v ∈ ?V. (u, v) ∈ I using D by simp
   hence ∃ u ∈ sinks-aux I D U xs. ∃ v ∈ ?V. (u, v) ∈ I using C ..
   ultimately show ?thesis ..
  qed
qed

lemma ex-sinks-sources-aux-2 [rule-format]:
 (∃ u ∈ U. ∃ v ∈ sources-aux I D V xs. (u, v) ∈ I) ⟶
  (∃ u ∈ sinks-aux I D U xs. ∃ v ∈ V. (u, v) ∈ I)
proof (induction xs arbitrary: V rule: rev-induct, simp, subst sources-aux-append,
 rule impI)
  fix x xs V
```

let
?V = *sources-aux I D V* [*x*] **and**
?V′ = *insert* (*D x*) *V*
assume
A: ⋀*V*. (∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D V xs*. (*u*, *v*) ∈ *I*) ⟶
(∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*) **and**
B: ∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D ?V xs*. (*u*, *v*) ∈ *I*
show ∃ *u* ∈ *sinks-aux I D U* (*xs* @ [*x*]). ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*
proof (*cases* ∃ *u* ∈ *sinks-aux I D U xs*. (*u*, *D x*) ∈ *I*,
cases ∃ *v* ∈ *V*. (*D x*, *v*) ∈ *I*, *simp-all* (*no-asm-simp*))
have (∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D V xs*. (*u*, *v*) ∈ *I*) ⟶
(∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*)
using *A* **.**
moreover assume ¬ (∃ *v* ∈ *V*. (*D x*, *v*) ∈ *I*)
hence ∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D V xs*. (*u*, *v*) ∈ *I* **using** *B* **by** *simp*
ultimately show ∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I* **..**
next
assume *C*: ¬ (∃ *u* ∈ *sinks-aux I D U xs*. (*u*, *D x*) ∈ *I*)
have (∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D ?V xs*. (*u*, *v*) ∈ *I*) ⟶
(∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *?V*. (*u*, *v*) ∈ *I*)
using *A* **.**
hence ∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *?V*. (*u*, *v*) ∈ *I* **using** *B* **..**
then obtain *u* **where**
D: *u* ∈ *sinks-aux I D U xs* **and** *E*: ∃ *v* ∈ *?V*. (*u*, *v*) ∈ *I* **..**
obtain *v* **where** *F*: *v* ∈ *?V* **and** *G*: (*u*, *v*) ∈ *I* **using** *E* **..**
have *v* = *D x* ∨ *v* ∈ *V* **using** *F* **by** (*cases* ∃ *v* ∈ *V*. (*D x*, *v*) ∈ *I*, *simp-all*)
moreover {
assume *v* = *D x*
hence (*u*, *D x*) ∈ *I* **using** *G* **by** *simp*
hence ∃ *u* ∈ *sinks-aux I D U xs*. (*u*, *D x*) ∈ *I* **using** *D* **..**
hence ∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*
using *C* **by** *contradiction*
}
moreover {
assume *v* ∈ *V*
with *G* **have** ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I* **..**
hence ∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I* **using** *D* **..**
}
ultimately show ∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I* **..**
qed
qed

lemma *ex-sinks-sources-aux*:
(∃ *u* ∈ *sinks-aux I D U xs*. ∃ *v* ∈ *V*. (*u*, *v*) ∈ *I*) =
(∃ *u* ∈ *U*. ∃ *v* ∈ *sources-aux I D V xs*. (*u*, *v*) ∈ *I*)
by (*rule iffI*, *erule ex-sinks-sources-aux-1*, *rule ex-sinks-sources-aux-2*)

lemma *ipurge-tr-rev-ipurge-tr-sources-aux-1* [*rule-format*]:
¬ (∃ *v* ∈ *D* ‘ *set ys*. ∃ *u* ∈ *sources-aux I D U zs*. (*v*, *u*) ∈ *I*) ⟶

```
  ipurge-tr-rev-aux I D U (xs @ ys @ zs) =
  ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs)
proof (induction zs arbitrary: U rule: rev-induct, rule-tac [!] impI,
 simp del: bex-simps)
  fix U
  assume ¬ (∃ v ∈ D ‘ set ys. ∃ u ∈ U. (v, u) ∈ I)
  hence ipurge-tr-rev-aux I D U ys = [] by (simp add: ipurge-tr-rev-aux-nil)
  thus ipurge-tr-rev-aux I D U (xs @ ys) = ipurge-tr-rev-aux I D U xs
   by (simp add: ipurge-tr-rev-aux-append-nil)
next
  fix z zs U
  let
    ?U = sources-aux I D U [z] and
    ?U' = insert (D z) U
  assume ⋀U. ¬ (∃ v ∈ D ‘ set ys. ∃ u ∈ sources-aux I D U zs. (v, u) ∈ I) ⟶
    ipurge-tr-rev-aux I D U (xs @ ys @ zs) =
    ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs)
  hence ¬ (∃ v ∈ D ‘ set ys. ∃ u ∈ sources-aux I D ?U zs. (v, u) ∈ I) ⟶
    ipurge-tr-rev-aux I D ?U (xs @ ys @ zs) =
    ipurge-tr-rev-aux I D ?U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs) .
  moreover assume
   ¬ (∃ v ∈ D ‘ set ys. ∃ u ∈ sources-aux I D U (zs @ [z]). (v, u) ∈ I)
  hence A:
   ¬ (∃ v ∈ D ‘ set ys. ∃ u ∈ sources-aux I D ?U zs. (v, u) ∈ I)
   by (subst (asm) sources-aux-append)
  ultimately have B:
   ipurge-tr-rev-aux I D ?U (xs @ ys @ zs) =
    ipurge-tr-rev-aux I D ?U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs) ..
  have
   ipurge-tr-rev-aux I D U (xs @ ys @ zs @ [z]) =
    ipurge-tr-rev-aux I D U ((xs @ ys @ zs) @ [z])
   by simp
  hence C:
   ipurge-tr-rev-aux I D U (xs @ ys @ zs @ [z]) =
    ipurge-tr-rev-aux I D ?U (xs @ ys @ zs) @ ipurge-tr-rev-aux I D U [z]
   (is - = - @ ?ws) by (simp only: ipurge-tr-rev-aux-append)
  show
   ipurge-tr-rev-aux I D U (xs @ ys @ zs @ [z]) =
    ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) (zs @ [z]))
  proof (subst C, cases ∃ u ∈ U. (D z, u) ∈ I,
   simp-all (no-asm-simp) del: ipurge-tr-aux.simps)
    case True
    have ¬ (∃ v ∈ sinks-aux I D (D ‘ set ys) zs. ∃ u ∈ ?U. (v, u) ∈ I)
     using A by (simp add: ex-sinks-sources-aux)
    hence ¬ (∃ v ∈ sinks-aux I D (D ‘ set ys) zs. (v, D z) ∈ I)
     using True by simp
    hence
     ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) (zs @ [z])) =
      ipurge-tr-rev-aux I D U ((xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ [z])
```

```
    by simp
   also have ... =
    ipurge-tr-rev-aux I D ?U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ ?ws
    by (simp only: ipurge-tr-rev-aux-append)
   also have ... =
    ipurge-tr-rev-aux I D ?U' (xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ [z]
    using True by simp
   finally have
    ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) (zs @ [z])) =
     ipurge-tr-rev-aux I D ?U' (xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ [z] .
   thus
    ipurge-tr-rev-aux I D ?U' (xs @ ys @ zs) @ [z] =
     ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) (zs @ [z]))
    using B and True by simp
  next
   case False
   have
    ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) (zs @ [z])) =
     ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs)
   proof (cases ∃ v ∈ sinks-aux I D (D ‘ set ys) zs. (v, D z) ∈ I, simp-all)
    have
     ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs @ [z]) =
      ipurge-tr-rev-aux I D U ((xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ [z])
     by simp
    also have ... =
     ipurge-tr-rev-aux I D ?U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ ?ws
     by (simp only: ipurge-tr-rev-aux-append)
    also have ... =
     ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs)
     using False by simp
    finally show
     ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs @ [z]) =
      ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs) .
   qed
   thus
    ipurge-tr-rev-aux I D U (xs @ ys @ zs) =
     ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) (zs @ [z]))
    using B and False by simp
  qed
qed

lemma ipurge-tr-rev-ipurge-tr-sources-1:
  assumes A: D y ∉ sources I D u (y # zs)
  shows
   ipurge-tr-rev I D u (xs @ y # zs) =
    ipurge-tr-rev I D u (xs @ ipurge-tr I D (D y) zs)
proof −
  have ¬ ((D y, u) ∈ I ∨ (∃ v ∈ sources I D u zs. (D y, v) ∈ I))
   using A by (simp only: sources-interference-eq, simp)
```

hence ¬ (∃ *v* ∈ *D* ‘ *set* [*y*]. ∃ *u* ∈ *sources-aux I D* {*u*} *zs*. (*v*, *u*) ∈ *I*)
by (*simp add*: *sources-aux-single-dom*)
hence
ipurge-tr-rev-aux I D {*u*} (*xs* @ [*y*] @ *zs*) =
ipurge-tr-rev-aux I D {*u*} (*xs* @ *ipurge-tr-aux I D* (*D* ‘ *set* [*y*]) *zs*)
by (*rule ipurge-tr-rev-ipurge-tr-sources-aux-1*)
thus *?thesis* **by** (*simp add*: *ipurge-tr-aux-single-dom ipurge-tr-rev-aux-single-dom*)
qed

lemma *ipurge-tr-length*:
length (*ipurge-tr I D u xs*) ≤ *length xs*
by (*induction xs rule*: *rev-induct*, *simp-all*)

lemma *sources-idem*:
sources I D u (*ipurge-tr-rev I D u xs*) = *sources I D u xs*
by (*induction xs*, *simp-all*)

lemma *ipurge-tr-rev-idem*:
ipurge-tr-rev I D u (*ipurge-tr-rev I D u xs*) = *ipurge-tr-rev I D u xs*
by (*induction xs*, *simp-all add*: *sources-idem*)

1.2 Closure of the traces of a secure process under reverse intransitive purge

The derivation of the Inductive Unwinding Theorem from the Ipurge Unwinding Theorem requires to prove that the set of the traces of a secure process is closed under reverse intransitive purge, i.e. function *ipurge-tr-rev* [8]. This can be expressed formally by means of the following statement:

⟦*secure P I D*; *xs* ∈ *traces P*⟧ ⟹ *ipurge-tr-rev I D u xs* ∈ *traces P*

The reason why such closure property holds is that the reverse intransitive purge of a list *xs* with regard to a policy *I*, an event-domain map *D*, and a domain *u* can equivalently be computed as follows: for each item *x* of *xs*, if *x* may affect *u*, retain *x* and go on recursively using as input the sublist of *xs* following *x*, say *xs′*; otherwise, discard *x* and go on recursively using *ipurge-tr I D* (*D x*) *xs′* [7] as input.

The result actually matches *ipurge-tr-rev I D u xs*. In fact, for each *x* not affecting *u*, *ipurge-tr I D* (*D x*) *xs′* retains any item of *xs′* not affected by *x*, which is the case for any item of *xs′* affecting *u*, since otherwise *x* would affect *u*.

Furthermore, if *xs* is a trace of a secure process, the result is still a trace. In fact, for each *x* not affecting *u*, if *ys* is the partial output for the sublist of *xs* preceding *x*, then *ys* @ *ipurge-tr I D* (*D x*) *xs′* is a trace provided such is *ys* @ *x* # *xs′*, by virtue of the definition of CSP noninterference security

[7]. Hence, the property of being a trace is conserved upon each recursive call by the concatenation of the partial output and the residual input, until the latter is nil and the former matches the total output.

This argument shows that in order to prove by induction, under the aforesaid assumptions, that the output of such a reverse intransitive purge function is a trace, the partial output has to be passed to the function as an argument, in addition to the residual input, in the recursive calls contained within the definition of the function. Therefore, the output of the function has to be accumulated into one of its parameters, 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 [6].

The starting point is to formulate a naive definition of the function, which will then be refined as specified by the proof method. The name of the refined function, from which the name of the naive function here below is derived, will be *ipurge-tr-rev-t*, where suffix *t* stands for *tail-recursive*.

function (*sequential*) *ipurge-tr-rev-t-naive* ::
($'d \times 'd$) *set* $\Rightarrow$ ($'a \Rightarrow 'd$) $\Rightarrow$ $'d \Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* **where**
ipurge-tr-rev-t-naive I D u (*x* # *xs*) *ys* =
(*if D x* $\in$ *sources I D u* (*x* # *xs*)
then ipurge-tr-rev-t-naive I D u xs (*ys* @ [*x*])
else ipurge-tr-rev-t-naive I D u (*ipurge-tr I D* (*D x*) *xs*) *ys*) |
ipurge-tr-rev-t-naive - - - - *ys* = *ys*
oops

The parameter into which the output is accumulated is the last one.

As shown by the previous argument, the properties of function *ipurge-tr-rev-t-naive* that would have to be proven are the following ones:

ipurge-tr-rev-t-naive I D u xs [] = *ipurge-tr-rev I D u xs*

⟦*secure P I D*; *xs* $\in$ *traces P*⟧ $\Longrightarrow$ *ipurge-tr-rev-t-naive I D u xs* [] $\in$ *traces P*

as they clearly entail the above formal statement of the target closure lemma.

1.2.1 Step 1

In the definition of the auxiliary tail-recursive function *ipurge-tr-rev-t-aux*, the Cartesian product of the input types of function *ipurge-tr-rev-t-naive* will be implemented as a record type.

```
record ('a, 'd) ipurge-rec =
  Pol :: ('d × 'd) set
  Map :: 'a ⇒ 'd
  Dom :: 'd
  In :: 'a list
  Out :: 'a list

function (sequential) ipurge-tr-rev-t-aux ::
  ('a, 'd) ipurge-rec ⇒ ('a, 'd) ipurge-rec where
ipurge-tr-rev-t-aux (|Pol = I, Map = D, Dom = u, In = x # xs, Out = ys|) =
 (if D x ∈ sources I D u (x # xs)
  then ipurge-tr-rev-t-aux
    (|Pol = I, Map = D, Dom = u, In = xs, Out = ys @ [x]|)
  else ipurge-tr-rev-t-aux
    (|Pol = I, Map = D, Dom = u, In = ipurge-tr I D (D x) xs, Out = ys|)) |
ipurge-tr-rev-t-aux X = X
proof (simp-all, atomize-elim)
  fix X :: ('a, 'd) ipurge-rec
  show
   (∃ I D u x xs ys.
      X = (|Pol = I, Map = D, Dom = u, In = x # xs, Out = ys|)) ∨
    (∃ I D u ys.
      X = (|Pol = I, Map = D, Dom = u, In = [], Out = ys|))
  proof (cases X, simp-all)
  qed (subst disj-commute, rule spec [OF list.nchotomy])
qed

termination ipurge-tr-rev-t-aux
proof (relation measure (λX. length (In X)), simp-all)
  fix D :: 'a ⇒ 'd and I x xs
  have length (ipurge-tr I D (D x) xs) ≤ length xs by (rule ipurge-tr-length)
  thus length (ipurge-tr I D (D x) xs) < Suc (length xs) by simp
qed
```

As shown by this proof, the termination of function *ipurge-tr-rev-t-aux* is guaranteed by the fact, proven previously, that the event lists output by function *ipurge-tr* are not longer than the corresponding input ones.

1.2.2 Step 2

```
definition ipurge-tr-rev-t-in ::
  ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ 'd ⇒ 'a list ⇒ ('a, 'd) ipurge-rec where
ipurge-tr-rev-t-in I D u xs ≡
  (|Pol = I, Map = D, Dom = u, In = xs, Out = []|)

definition ipurge-tr-rev-t-out ::
  ('a, 'd) ipurge-rec ⇒ 'a list where
ipurge-tr-rev-t-out ≡ Out
```

definition *ipurge-tr-rev-t* ::
  $('d \times 'd)$ *set* $\Rightarrow$ $('a \Rightarrow 'd) \Rightarrow 'd \Rightarrow 'a$ *list* $\Rightarrow$ $'a$ *list* **where**
ipurge-tr-rev-t I D u xs $\equiv$
  ipurge-tr-rev-t-out (*ipurge-tr-rev-t-aux* (*ipurge-tr-rev-t-in I D u xs*))

Since the significant inputs of function *ipurge-tr-rev-t-naive* match pattern -, -, -, -, [], those of function *ipurge-tr-rev-t-aux*, as returned by function *ipurge-tr-rev-t-in*, match pattern $(\!|Pol = -, Map = -, Dom = -, In = -, Out = []|\!)$.
In terms of function *ipurge-tr-rev-t*, the statements to be proven, henceforth respectively named *ipurge-tr-rev-t-equiv* and *ipurge-tr-rev-t-trace*, take the following form:

ipurge-tr-rev-t I D u xs $=$ *ipurge-tr-rev I D u xs*

$[\![$*secure P I D*; *xs* $\in$ *traces P*$]\!] \Longrightarrow$ *ipurge-tr-rev-t I D u xs* $\in$ *traces P*

1.2.3 Step 3

inductive-set *ipurge-tr-rev-t-set* :: $('a, 'd)$ *ipurge-rec* $\Rightarrow$ $('a, 'd)$ *ipurge-rec set*
  for *X* :: $('a, 'd)$ *ipurge-rec* **where**
R0: *X* $\in$ *ipurge-tr-rev-t-set X* |
R1: $[\![(\!|Pol = I, Map = D, Dom = u, In = x \# xs, Out = ys|\!)$
      $\in$ *ipurge-tr-rev-t-set X*;
    D x $\in$ *sources I D u* $(x \# xs)]\!] \Longrightarrow$
    $(\!|Pol = I, Map = D, Dom = u, In = xs, Out = ys$ @ $[x]|\!)$
      $\in$ *ipurge-tr-rev-t-set X* |
R2: $[\![(\!|Pol = I, Map = D, Dom = u, In = x \# xs, Out = ys|\!)$
      $\in$ *ipurge-tr-rev-t-set X*;
    D x $\notin$ *sources I D u* $(x \# xs)]\!] \Longrightarrow$
    $(\!|Pol = I, Map = D, Dom = u, In =$ *ipurge-tr I D* $(D\ x)$ *xs*, *Out* $= ys|\!)$
      $\in$ *ipurge-tr-rev-t-set X*

1.2.4 Step 4

lemma *ipurge-tr-rev-t-subset*:
  assumes *A*: *Y* $\in$ *ipurge-tr-rev-t-set X*
  shows *ipurge-tr-rev-t-set Y* $\subseteq$ *ipurge-tr-rev-t-set X*
proof (*rule subsetI*, *erule ipurge-tr-rev-t-set.induct*)
  show *Y* $\in$ *ipurge-tr-rev-t-set X* **using** *A* **.**
next
  fix *I D u x xs ys*
  assume
    $(\!|Pol = I, Map = D, Dom = u, In = x \# xs, Out = ys|\!)$
      $\in$ *ipurge-tr-rev-t-set X* **and**

```
    D x ∈ sources I D u (x # xs)
  thus (|Pol = I, Map = D, Dom = u, In = xs, Out = ys @ [x]|)
    ∈ ipurge-tr-rev-t-set X
   by (rule R1)
next
  fix I D u x xs ys
  assume
    (|Pol = I, Map = D, Dom = u, In = x # xs, Out = ys|)
      ∈ ipurge-tr-rev-t-set X and
    D x ∉ sources I D u (x # xs)
  thus (|Pol = I, Map = D, Dom = u, In = ipurge-tr I D (D x) xs, Out = ys|)
    ∈ ipurge-tr-rev-t-set X
   by (rule R2)
qed

lemma ipurge-tr-rev-t-aux-set:
 ipurge-tr-rev-t-aux X ∈ ipurge-tr-rev-t-set X
proof (induction rule: ipurge-tr-rev-t-aux.induct,
 simp-all only: ipurge-tr-rev-t-aux.simps(2) R0)
  fix I u x xs ys and D :: 'a ⇒ 'd
  assume
    A: D x ∈ sources I D u (x # xs) ⟹
        ipurge-tr-rev-t-aux
          (|Pol = I, Map = D, Dom = u, In = xs, Out = ys @ [x]|)
        ∈ ipurge-tr-rev-t-set
          (|Pol = I, Map = D, Dom = u, In = xs, Out = ys @ [x]|)
      (is - ⟹ ipurge-tr-rev-t-aux ?Y ∈ -) and
    B: D x ∉ sources I D u (x # xs) ⟹
        ipurge-tr-rev-t-aux
          (|Pol = I, Map = D, Dom = u, In = ipurge-tr I D (D x) xs, Out = ys|)
        ∈ ipurge-tr-rev-t-set
          (|Pol = I, Map = D, Dom = u, In = ipurge-tr I D (D x) xs, Out = ys|)
      (is - ⟹ ipurge-tr-rev-t-aux ?Z ∈ -)
  show
   ipurge-tr-rev-t-aux
      (|Pol = I, Map = D, Dom = u, In = x # xs, Out = ys|)
    ∈ ipurge-tr-rev-t-set
      (|Pol = I, Map = D, Dom = u, In = x # xs, Out = ys|)
    (is ipurge-tr-rev-t-aux ?X ∈ -)
  proof (cases D x ∈ sources I D u (x # xs), simp-all del: sources.simps)
    case True
    have ?X ∈ ipurge-tr-rev-t-set ?X by (rule R0)
    moreover have ?X ∈ ipurge-tr-rev-t-set ?X ⟹ ?Y ∈ ipurge-tr-rev-t-set ?X
     by (rule R1 [OF - True])
    ultimately have ?Y ∈ ipurge-tr-rev-t-set ?X by simp
    hence ipurge-tr-rev-t-set ?Y ⊆ ipurge-tr-rev-t-set ?X
     by (rule ipurge-tr-rev-t-subset)
    moreover have ipurge-tr-rev-t-aux ?Y ∈ ipurge-tr-rev-t-set ?Y
     using True by (rule A)
```

ultimately show *ipurge-tr-rev-t-aux ?Y ∈ ipurge-tr-rev-t-set ?X* **..**
next
case *False*
have *?X ∈ ipurge-tr-rev-t-set ?X* **by** (*rule R0*)
moreover have *?X ∈ ipurge-tr-rev-t-set ?X ⟹ ?Z ∈ ipurge-tr-rev-t-set ?X*
by (*rule R2* [*OF - False*])
ultimately have *?Z ∈ ipurge-tr-rev-t-set ?X* **by** *simp*
hence *ipurge-tr-rev-t-set ?Z ⊆ ipurge-tr-rev-t-set ?X*
by (*rule ipurge-tr-rev-t-subset*)
moreover have *ipurge-tr-rev-t-aux ?Z ∈ ipurge-tr-rev-t-set ?Z*
using *False* **by** (*rule B*)
ultimately show *ipurge-tr-rev-t-aux ?Z ∈ ipurge-tr-rev-t-set ?X* **..**
qed
qed

1.2.5 Step 5

definition *ipurge-tr-rev-t-inv-1* ::
(*'d* × *'d*) *set* ⇒ (*'a* ⇒ *'d*) ⇒ *'d* ⇒ *'a list* ⇒ (*'a*, *'d*) *ipurge-rec* ⇒ *bool*
where
ipurge-tr-rev-t-inv-1 I D u xs X ≡
Out X @ *ipurge-tr-rev I D u* (*In X*) = *ipurge-tr-rev I D u xs*

definition *ipurge-tr-rev-t-inv-2* ::
'a process ⇒ (*'d* × *'d*) *set* ⇒ (*'a* ⇒ *'d*) ⇒ *'a list* ⇒ (*'a*, *'d*) *ipurge-rec* ⇒ *bool*
where
ipurge-tr-rev-t-inv-2 P I D xs X ≡
secure P I D ⟶ *xs ∈ traces P* ⟶ *Out X* @ *In X ∈ traces P*

Two invariants have been defined, one for each of lemmas *ipurge-tr-rev-t-equiv*, *ipurge-tr-rev-t-trace*.

More precisely, the invariants are *ipurge-tr-rev-t-inv-1 I D u xs* and *ipurge-tr-rev-t-inv-2 P I D xs*, where the free variables are intended to match those appearing in the aforesaid lemmas.

1.2.6 Step 6

lemma *ipurge-tr-rev-t-input-1*:
ipurge-tr-rev-t-inv-1 I D u xs (|*Pol* = *I*, *Map* = *D*, *Dom* = *u*, *In* = *xs*, *Out* = []|)
by (*simp add*: *ipurge-tr-rev-t-inv-1-def*)

lemma *ipurge-tr-rev-t-input-2*:
ipurge-tr-rev-t-inv-2 P I D xs (|*Pol* = *I*, *Map* = *D*, *Dom* = *u*, *In* = *xs*, *Out* = []|)
by (*simp add*: *ipurge-tr-rev-t-inv-2-def*)

1.2.7 Step 7

definition *ipurge-tr-rev-t-form* :: (*'a*, *'d*) *ipurge-rec* ⇒ *bool* **where**

ipurge-tr-rev-t-form X ≡ *In X* = []

lemma *ipurge-tr-rev-t-intro-1*:
⟦*ipurge-tr-rev-t-inv-1 I D u xs X*; *ipurge-tr-rev-t-form X*⟧ ⟹
ipurge-tr-rev-t-out X = *ipurge-tr-rev I D u xs*
by (*simp add*: *ipurge-tr-rev-t-inv-1-def ipurge-tr-rev-t-form-def*
ipurge-tr-rev-t-out-def)

lemma *ipurge-tr-rev-t-intro-2*:
⟦*ipurge-tr-rev-t-inv-2 P I D xs X*; *ipurge-tr-rev-t-form X*⟧ ⟹
secure P I D ⟶ *xs* ∈ *traces P* ⟶ *ipurge-tr-rev-t-out X* ∈ *traces P*
by (*simp add*: *ipurge-tr-rev-t-inv-2-def ipurge-tr-rev-t-form-def*
ipurge-tr-rev-t-out-def)

1.2.8 Step 8

lemma *ipurge-tr-rev-t-form-aux*:
ipurge-tr-rev-t-form (*ipurge-tr-rev-t-aux X*)
by (*induction X rule*: *ipurge-tr-rev-t-aux.induct*,
simp-all add: *ipurge-tr-rev-t-form-def*)

1.2.9 Step 9

lemma *ipurge-tr-rev-t-invariance-aux*:
Y ∈ *ipurge-tr-rev-t-set X* ⟹
Pol Y = *Pol X* ∧ *Map Y* = *Map X* ∧ *Dom Y* = *Dom X*
by (*erule ipurge-tr-rev-t-set.induct*, *simp-all*)

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

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-tr-rev-t-equiv*, *ipurge-tr-rev-t-trace*, 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-tr-rev-t-invariance-1*:
⟦*Y* ∈ *ipurge-tr-rev-t-set* (*X*(|*Pol* := *I*, *Map* := *D*, *Dom* := *u*|));
ipurge-tr-rev-t-inv-1 I D u ws (*X*(|*Pol* := *I*, *Map* := *D*, *Dom* := *u*|))⟧ ⟹
ipurge-tr-rev-t-inv-1 I D u ws Y
proof (*erule ipurge-tr-rev-t-set.induct*, *assumption*,
drule-tac [!] *ipurge-tr-rev-t-invariance-aux*,
simp-all add: *ipurge-tr-rev-t-inv-1-def del*: *sources.simps*)

fix *x xs ys*
assume *A*: *D x* ∉ *sources I D u* (*x* # *xs*)
hence *ipurge-tr-rev I D u xs* = *ipurge-tr-rev I D u* ([] @ *x* # *xs*) **by** *simp*
also have ... = *ipurge-tr-rev I D u* ([] @ *ipurge-tr I D* (*D x*) *xs*)
using *A* **by** (*rule ipurge-tr-rev-ipurge-tr-sources-1*)
finally have
ipurge-tr-rev I D u xs = *ipurge-tr-rev I D u* (*ipurge-tr I D* (*D x*) *xs*)
by *simp*
moreover assume *ys* @ *ipurge-tr-rev I D u xs* = *ipurge-tr-rev I D u ws*
ultimately show
ys @ *ipurge-tr-rev I D u* (*ipurge-tr I D* (*D x*) *xs*) = *ipurge-tr-rev I D u ws*
by *simp*
qed

lemma *ipurge-tr-rev-t-invariance-2*:
⟦*Y* ∈ *ipurge-tr-rev-t-set* (*X*(|*Pol* := *I*, *Map* := *D*|));
ipurge-tr-rev-t-inv-2 P I D ws (*X*(|*Pol* := *I*, *Map* := *D*|))⟧ ⟹
ipurge-tr-rev-t-inv-2 P I D ws Y
proof (*erule ipurge-tr-rev-t-set.induct*, *assumption*,
drule-tac [!] *ipurge-tr-rev-t-invariance-aux*,
simp-all add: *ipurge-tr-rev-t-inv-2-def*, (*rule impI*)+)
fix *x xs ys*
assume
S: *secure P I D* **and**
ws ∈ *traces P* **and**
secure P I D ⟶ *ws* ∈ *traces P* ⟶ *ys* @ *x* # *xs* ∈ *traces P*
hence *ys* @ *x* # *xs* ∈ *traces P* **by** *simp*
hence (*ys* @ *x* # *xs*, {}) ∈ *failures P* **by** (*rule traces-failures*)
hence (*x* # *xs*, {}) ∈ *futures P ys* **by** (*simp add*: *futures-def*)
hence (*ipurge-tr I D* (*D x*) *xs*, *ipurge-ref I D* (*D x*) *xs* {}) ∈ *futures P ys*
using *S* **by** (*simp add*: *secure-def*)
hence (*ys* @ *ipurge-tr I D* (*D x*) *xs*, *ipurge-ref I D* (*D x*) *xs* {}) ∈ *failures P*
by (*simp add*: *futures-def*)
thus *ys* @ *ipurge-tr I D* (*D x*) *xs* ∈ *traces P* **by** (*rule failures-traces*)
qed

1.2.10 Step 10

Here below are the proofs of lemmas *ipurge-tr-rev-t-equiv*, *ipurge-tr-rev-t-trace*, which are then applied to demonstrate the target closure lemma.

lemma *ipurge-tr-rev-t-equiv*:
ipurge-tr-rev-t I D u xs = *ipurge-tr-rev I D u xs*
proof −
let *?X* = (|*Pol* = *I*, *Map* = *D*, *Dom* = *u*, *In* = *xs*, *Out* = []|)
have *ipurge-tr-rev-t-aux ?X*
∈ *ipurge-tr-rev-t-set* (*?X*(|*Pol* := *I*, *Map* := *D*, *Dom* := *u*|))
by (*simp add*: *ipurge-tr-rev-t-aux-set*)
moreover have

ipurge-tr-rev-t-inv-1 I D u xs (*?X*(|*Pol* := *I*, *Map* := *D*, *Dom* := *u*|))
by (*simp add*: *ipurge-tr-rev-t-input-1*)
ultimately have *ipurge-tr-rev-t-inv-1 I D u xs* (*ipurge-tr-rev-t-aux ?X*)
by (*rule ipurge-tr-rev-t-invariance-1*)
moreover have *ipurge-tr-rev-t-form* (*ipurge-tr-rev-t-aux ?X*)
by (*rule ipurge-tr-rev-t-form-aux*)
ultimately have
ipurge-tr-rev-t-out (*ipurge-tr-rev-t-aux ?X*) = *ipurge-tr-rev I D u xs*
by (*rule ipurge-tr-rev-t-intro-1*)
moreover have *?X* = *ipurge-tr-rev-t-in I D u xs*
by (*simp add*: *ipurge-tr-rev-t-in-def*)
ultimately show *?thesis* **by** (*simp add*: *ipurge-tr-rev-t-def*)
qed

lemma *ipurge-tr-rev-t-trace* [*rule-format*]:
secure P I D ⟶ *xs* ∈ *traces P* ⟶ *ipurge-tr-rev-t I D u xs* ∈ *traces P*
proof −
let *?X* = (|*Pol* = *I*, *Map* = *D*, *Dom* = *u*, *In* = *xs*, *Out* = []|)
have *ipurge-tr-rev-t-aux ?X*
∈ *ipurge-tr-rev-t-set* (*?X*(|*Pol* := *I*, *Map* := *D*|))
by (*simp add*: *ipurge-tr-rev-t-aux-set*)
moreover have *ipurge-tr-rev-t-inv-2 P I D xs* (*?X*(|*Pol* := *I*, *Map* := *D*|))
by (*simp add*: *ipurge-tr-rev-t-input-2*)
ultimately have *ipurge-tr-rev-t-inv-2 P I D xs* (*ipurge-tr-rev-t-aux ?X*)
by (*rule ipurge-tr-rev-t-invariance-2*)
moreover have *ipurge-tr-rev-t-form* (*ipurge-tr-rev-t-aux ?X*)
by (*rule ipurge-tr-rev-t-form-aux*)
ultimately have *secure P I D* ⟶ *xs* ∈ *traces P* ⟶
ipurge-tr-rev-t-out (*ipurge-tr-rev-t-aux ?X*) ∈ *traces P*
by (*rule ipurge-tr-rev-t-intro-2*)
moreover have *?X* = *ipurge-tr-rev-t-in I D u xs*
by (*simp add*: *ipurge-tr-rev-t-in-def*)
ultimately show *?thesis* **by** (*simp add*: *ipurge-tr-rev-t-def*)
qed

lemma *ipurge-tr-rev-trace*:
secure P I D ⟹ *xs* ∈ *traces P* ⟹ *ipurge-tr-rev I D u xs* ∈ *traces P*
by (*subst ipurge-tr-rev-t-equiv* [*symmetric*], *rule ipurge-tr-rev-t-trace*)

1.3 The Inductive Unwinding Theorem in its general form

In what follows, the Inductive Unwinding Theorem is proven, in the form applying to a generic process. The equivalence of the condition expressed by the theorem to CSP noninterference security, as defined in [7], is demonstrated by showing that it is necessary and sufficient for the verification of the condition expressed by the Ipurge Unwinding Theorem, under the same assumption that the sets of refusals of the process be closed under union (cf. [8]).

Particularly, the closure of the traces of a secure process under function *ipurge-tr-rev* and the idempotence of this function are used in the proof of condition necessity.

lemma *inductive-unwinding-1*:
assumes
R: *ref-union-closed P* **and**
S: *secure P I D*
shows $\forall\, xs \in traces\ P.\ \forall\, u \in range\ D \cap (-I)\ ``\ range\ D.$
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs* $\wedge$
ref-dom-events P D u (*ipurge-tr-rev I D u xs*) = *ref-dom-events P D u xs*
proof (*rule ballI*)+
fix *xs u*
from *R* **and** *S* **have** $\forall\, u \in range\ D \cap (-\ I)\ ``\ range\ D.\ \forall\, xs\ ys.$
$xs \in traces\ P \wedge ys \in traces\ P \wedge$
ipurge-tr-rev I D u xs = *ipurge-tr-rev I D u ys* $\longrightarrow$
next-dom-events P D u xs = *next-dom-events P D u ys* $\wedge$
ref-dom-events P D u xs = *ref-dom-events P D u ys*
by (*simp add*: *ipurge-unwinding weakly-future-consistent-def rel-ipurge-def*)
moreover assume $u \in range\ D \cap (-\ I)\ ``\ range\ D$
ultimately have $\forall\, xs\ ys.$
$xs \in traces\ P \wedge ys \in traces\ P \wedge$
ipurge-tr-rev I D u xs = *ipurge-tr-rev I D u ys* $\longrightarrow$
next-dom-events P D u xs = *next-dom-events P D u ys* $\wedge$
ref-dom-events P D u xs = *ref-dom-events P D u ys* **..**
hence
ipurge-tr-rev I D u xs $\in traces\ P \wedge xs \in traces\ P \wedge$
ipurge-tr-rev I D u (*ipurge-tr-rev I D u xs*) = *ipurge-tr-rev I D u xs* $\longrightarrow$
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs* $\wedge$
ref-dom-events P D u (*ipurge-tr-rev I D u xs*) = *ref-dom-events P D u xs*
by *blast*
moreover assume *xs*: $xs \in traces\ P$
moreover from *S* **and** *xs* **have** *ipurge-tr-rev I D u xs* $\in traces\ P$
by (*rule ipurge-tr-rev-trace*)
moreover have
ipurge-tr-rev I D u (*ipurge-tr-rev I D u xs*) = *ipurge-tr-rev I D u xs*
by (*rule ipurge-tr-rev-idem*)
ultimately show
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs* $\wedge$
ref-dom-events P D u (*ipurge-tr-rev I D u xs*) = *ref-dom-events P D u xs*
by *simp*
qed

lemma *inductive-unwinding-2*:
assumes
R: *ref-union-closed P* **and**
S: $\forall\, xs \in traces\ P.\ \forall\, u \in range\ D \cap (-I)\ ``\ range\ D.$
next-dom-events P D u (*ipurge-tr-rev I D u xs*) =
next-dom-events P D u xs $\wedge$

ref-dom-events P D u (ipurge-tr-rev I D u xs) =
ref-dom-events P D u xs
shows *secure P I D*
proof (*simp add*: *ipurge-unwinding* [*OF R*] *weakly-future-consistent-def rel-ipurge-def*, *rule ballI*, (*rule allI*)+, *rule impI*, (*erule conjE*)+)
fix *u xs ys*
assume *xs* ∈ *traces P*
with *S* **have** ∀ *u* ∈ *range D* ∩ (−*I*) `` *range D*.
next-dom-events P D u (ipurge-tr-rev I D u xs) = next-dom-events P D u xs ∧
ref-dom-events P D u (ipurge-tr-rev I D u xs) = ref-dom-events P D u xs **..**
moreover assume *A*: *u* ∈ *range D* ∩ (− *I*) `` *range D*
ultimately have *B*:
next-dom-events P D u (ipurge-tr-rev I D u xs) = next-dom-events P D u xs ∧
ref-dom-events P D u (ipurge-tr-rev I D u xs) = ref-dom-events P D u xs **..**
assume *ys* ∈ *traces P*
with *S* **have** ∀ *u* ∈ *range D* ∩ (−*I*) `` *range D*.
next-dom-events P D u (ipurge-tr-rev I D u ys) = next-dom-events P D u ys ∧
ref-dom-events P D u (ipurge-tr-rev I D u ys) = ref-dom-events P D u ys **..**
hence
next-dom-events P D u (ipurge-tr-rev I D u ys) = next-dom-events P D u ys ∧
ref-dom-events P D u (ipurge-tr-rev I D u ys) = ref-dom-events P D u ys
using *A* **..**
moreover assume *ipurge-tr-rev I D u xs = ipurge-tr-rev I D u ys*
ultimately show
next-dom-events P D u xs = next-dom-events P D u ys ∧
ref-dom-events P D u xs = ref-dom-events P D u ys
using *B* **by** *simp*
qed

theorem *inductive-unwinding*:
ref-union-closed P ⟹
secure P I D =
(∀ *xs* ∈ *traces P*. ∀ *u* ∈ *range D* ∩ (−*I*) `` *range D*.
next-dom-events P D u (ipurge-tr-rev I D u xs) = next-dom-events P D u xs ∧
ref-dom-events P D u (ipurge-tr-rev I D u xs) = ref-dom-events P D u xs)
by (*rule iffI*, *rule inductive-unwinding-1*, *assumption*+, *rule inductive-unwinding-2*)

Interestingly, this necessary and sufficient condition for the noninterference security of a process resembles the classical definition of noninterference security for a deterministic state machine with outputs formulated in [9], which is formalized in [7] as predicate *c-secure*.

Denoting with (1) the former and with (2) the latter, the differences between them can be summarized as follows:

- The event list appearing in (1) is constrained to vary over process traces, whereas the action list appearing in (2) is unconstrained.
  This comes as no surprise, since the state machines used as model of

computation in [9] accept any action list as a trace.

- The definition of function *ipurge-tr-rev*, used in (1), does not implicitly assume that the noninterference policy be reflexive, even though any policy of practical significance will be such. On the contrary, the definition of the intransitive purge function used in (2), which is formalized in [7] as function *c-ipurge*, makes this implicit assumption, as shown by the consideration that *c-ipurge* $I\ D\ (D\ x)\ [x] = [x]$ regardless of whether $(D\ x,\ D\ x) \in I$ or not.
  This is the mathematical reason why the equivalence between CSP noninterference security and classical noninterference security for deterministic state machines with outputs, proven in [7], is subordinated to the assumption that the noninterference policy be reflexive.

- The equality of action outputs appearing in (2) is replaced in (1) by the equality of accepted and refused events.

The binding of the universal quantification over domains contained in (1) does not constitute an actual difference, since in (2) the purge function is only applied to domains in the range of the event-domain map, and its output matches the entire input action list, thus rendering the equation trivial, for domains allowed to be affected by any event domain.

1.4 The Inductive Unwinding Theorem for deterministic and trace set processes

Here below are the proofs of specific variants of the Inductive Unwinding Theorem applying to deterministic processes and trace set processes [8]. The variant for deterministic processes is derived, following the above proof of the general form of the theorem, from the Ipurge Unwinding Theorem for deterministic processes [8]. Then, the variant for trace set processes is inferred from the variant for deterministic processes.

Similarly to what happens for the Ipurge Unwinding Theorem, the refusals union closure assumption that characterizes the general form of the Inductive Unwinding Theorem is replaced by the assumption that the process actually be deterministic in the variant for deterministic processes, and by the assumption that the set of traces actually be such in the variant for trace set processes. Moreover, these variants involve accepted events only, in accordance with the fact that in deterministic processes, refused events are completely specified by accepted events (cf. [1], [7]).

lemma *d-inductive-unwinding-1*:
assumes
D: *deterministic P* **and**

S: *secure P I D*
shows $\forall\, xs \in traces\ P.\ \forall\, u \in range\ D \cap (-I)\ ``\ range\ D.$
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs*
proof (*rule ballI*)+
fix *xs u*
from *D* **and** *S* **have** $\forall\, u \in range\ D \cap (-\ I)\ ``\ range\ D.\ \forall\, xs\ ys.$
$xs \in traces\ P \wedge ys \in traces\ P \wedge$
ipurge-tr-rev I D u xs = *ipurge-tr-rev I D u ys* $\longrightarrow$
next-dom-events P D u xs = *next-dom-events P D u ys*
by (*simp add*: *d-ipurge-unwinding d-weakly-future-consistent-def rel-ipurge-def*)
moreover assume $u \in range\ D \cap (-\ I)\ ``\ range\ D$
ultimately have $\forall\, xs\ ys.$
$xs \in traces\ P \wedge ys \in traces\ P \wedge$
ipurge-tr-rev I D u xs = *ipurge-tr-rev I D u ys* $\longrightarrow$
next-dom-events P D u xs = *next-dom-events P D u ys* **..**
hence
ipurge-tr-rev I D u xs $\in traces\ P \wedge xs \in traces\ P \wedge$
ipurge-tr-rev I D u (*ipurge-tr-rev I D u xs*) = *ipurge-tr-rev I D u xs* $\longrightarrow$
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs*
by *blast*
moreover assume *xs*: $xs \in traces\ P$
moreover from *S* **and** *xs* **have** *ipurge-tr-rev I D u xs* $\in traces\ P$
by (*rule ipurge-tr-rev-trace*)
moreover have
ipurge-tr-rev I D u (*ipurge-tr-rev I D u xs*) = *ipurge-tr-rev I D u xs*
by (*rule ipurge-tr-rev-idem*)
ultimately show
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs*
by *simp*
qed

lemma *d-inductive-unwinding-2*:
assumes
D: *deterministic P* **and**
S: $\forall\, xs \in traces\ P.\ \forall\, u \in range\ D \cap (-I)\ ``\ range\ D.$
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs*
shows *secure P I D*
proof (*simp add*: *d-ipurge-unwinding* [*OF D*] *d-weakly-future-consistent-def rel-ipurge-def*,
rule ballI, (*rule allI*)+, *rule impI*, (*erule conjE*)+)
fix *u xs ys*
assume $xs \in traces\ P$
with *S* **have** $\forall\, u \in range\ D \cap (-I)\ ``\ range\ D.$
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs* **..**
moreover assume *A*: $u \in range\ D \cap (-\ I)\ ``\ range\ D$
ultimately have *B*:
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs* **..**
assume $ys \in traces\ P$
with *S* **have** $\forall\, u \in range\ D \cap (-I)\ ``\ range\ D.$
next-dom-events P D u (*ipurge-tr-rev I D u ys*) = *next-dom-events P D u ys* **..**

hence
next-dom-events P D u (*ipurge-tr-rev I D u ys*) = *next-dom-events P D u ys*
using *A* **..**
moreover assume *ipurge-tr-rev I D u xs* = *ipurge-tr-rev I D u ys*
ultimately show *next-dom-events P D u xs* = *next-dom-events P D u ys*
using *B* **by** *simp*
qed

theorem *d-inductive-unwinding*:
deterministic P ⟹
secure P I D =
(∀ *xs* ∈ *traces P*. ∀ *u* ∈ *range D* ∩ (−*I*) `` *range D*.
next-dom-events P D u (*ipurge-tr-rev I D u xs*) = *next-dom-events P D u xs*)
by (*rule iffI*, *rule d-inductive-unwinding-1*, *assumption*+, *rule d-inductive-unwinding-2*)

theorem *ts-inductive-unwinding*:
assumes *T*: *trace-set T*
shows *secure* (*ts-process T*) *I D* =
(∀ *xs* ∈ *T*. ∀ *u* ∈ *range D* ∩ (−*I*) `` *range D*. ∀ *x* ∈ *D* −` {*u*}.
(*ipurge-tr-rev I D u xs* @ [*x*] ∈ *T*) = (*xs* @ [*x*] ∈ *T*))
(**is** *secure ?P I D* = -)
proof (*subst d-inductive-unwinding*, *rule ts-process-d* [*OF T*],
simp add: *next-dom-events-def ts-process-next-events* [*OF T*] *set-eq-iff*,
rule iffI, (*rule ballI*)+, (*rule-tac* [*2*] *ballI*)+, *rule-tac* [*2*] *allI*)
fix *xs u x*
assume *A*: ∀ *xs* ∈ *traces ?P*. ∀ *u* ∈ *range D* ∩ (− *I*) `` *range D*.
∀ *x*. (*u* = *D x* ∧ *ipurge-tr-rev I D u xs* @ [*x*] ∈ *T*) = (*u* = *D x* ∧ *xs* @ [*x*] ∈ *T*)
assume *xs* ∈ *T*
moreover have *traces ?P* = *T* **using** *T* **by** (*rule ts-process-traces*)
ultimately have *xs* ∈ *traces ?P* **by** *simp*
with *A* **have** ∀ *u* ∈ *range D* ∩ (− *I*) `` *range D*.
∀ *x*. (*u* = *D x* ∧ *ipurge-tr-rev I D u xs* @ [*x*] ∈ *T*) =
(*u* = *D x* ∧ *xs* @ [*x*] ∈ *T*) **..**
moreover assume *u* ∈ *range D* ∩ (− *I*) `` *range D*
ultimately have
∀ *x*. (*u* = *D x* ∧ *ipurge-tr-rev I D u xs* @ [*x*] ∈ *T*) =
(*u* = *D x* ∧ *xs* @ [*x*] ∈ *T*) **..**
hence (*u* = *D x* ∧ *ipurge-tr-rev I D u xs* @ [*x*] ∈ *T*) =
(*u* = *D x* ∧ *xs* @ [*x*] ∈ *T*) **..**
moreover assume *x* ∈ *D* −` {*u*}
hence *u* = *D x* **by** *simp*
ultimately show (*ipurge-tr-rev I D u xs* @ [*x*] ∈ *T*) = (*xs* @ [*x*] ∈ *T*) **by** *simp*
next
fix *xs u x*
assume *A*: ∀ *xs* ∈ *T*. ∀ *u* ∈ *range D* ∩ (− *I*) `` *range D*.
∀ *x* ∈ *D* −` {*u*}. (*ipurge-tr-rev I D u xs* @ [*x*] ∈ *T*) = (*xs* @ [*x*] ∈ *T*)
assume *xs* ∈ *traces ?P*
moreover have *traces ?P* = *T* **using** *T* **by** (*rule ts-process-traces*)
ultimately have *xs* ∈ *T* **by** *simp*

```
  with A have ∀u ∈ range D ∩ (− I) `` range D.
    ∀x ∈ D −` {u}. (ipurge-tr-rev I D u xs @ [x] ∈ T) = (xs @ [x] ∈ T) ..
  moreover assume u ∈ range D ∩ (− I) `` range D
  ultimately have B:
   ∀x ∈ D −` {u}. (ipurge-tr-rev I D u xs @ [x] ∈ T) = (xs @ [x] ∈ T) ..
  show (u = D x ∧ ipurge-tr-rev I D u xs @ [x] ∈ T) =
    (u = D x ∧ xs @ [x] ∈ T)
  proof (cases D x = u, simp-all)
    case True
    hence x ∈ D −` {u} by simp
    with B show (ipurge-tr-rev I D u xs @ [x] ∈ T) = (xs @ [x] ∈ T) ..
  qed
qed

end
```

References

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

[2] A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. http://isabelle.in.tum.de/website-Isabelle2015/dist/Isabelle2015/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*, May 2015. http://isabelle.in.tum.de/website-Isabelle2015/dist/Isabelle2015/doc/prog-prove.pdf.

[5] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, May 2015. http://isabelle.in.tum.de/website-Isabelle2015/dist/Isabelle2015/doc/tutorial.pdf.

[6] 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.

[7] 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.

[8] 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.

[9] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI International, 1992.