# The Ipurge 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

March 17, 2025

**Abstract**

The definition of noninterference security for Communicating Sequential Processes requires to consider any possible future, i.e. any indefinitely long sequence of subsequent events and any indefinitely large set of refused events associated to that sequence, for each process trace. In order to render the verification of the security of a process more straightforward, there is a need of some sufficient condition for security such that just individual accepted and refused events, rather than unbounded sequences and sets of events, have to be considered.

Of course, if such a sufficient condition were necessary as well, it would be even more valuable, since it would permit to prove not only that a process is secure by verifying that the condition holds, but also that a process is not secure by verifying that the condition fails to hold.

This paper provides a necessary and sufficient condition for CSP noninterference security, which indeed requires to just consider individual accepted and refused events and applies to the general case of a possibly intransitive policy. This condition follows Rushby's output consistency for deterministic state machines with outputs, and has to be satisfied by a specific function mapping security domains into equivalence relations over process traces. The definition of this function makes use of an intransitive purge function following Rushby's one; hence the name given to the condition, Ipurge Unwinding Theorem.

Furthermore, in accordance with Hoare's formal definition of deterministic processes, it is shown that a process is deterministic just in case it is a trace set process, i.e. it may be identified by means of a trace set alone, matching the set of its traces, in place of a failures-divergences pair. Then, variants of the Ipurge Unwinding Theorem are proven for deterministic processes and trace set processes.

## Contents

# 1   The Ipurge Unwinding Theorem in its general form

**theory** *IpurgeUnwinding*
**imports** *Noninterference-CSP.CSPNoninterference List-Interleaving.ListInterleaving*
**begin**

The definition of noninterference security for Communicating Sequential Processes given in [6] requires to consider any possible future, i.e. any indefinitely long sequence of subsequent events and any indefinitely large set of refused events associated to that sequence, for each process trace. In order to render the verification of the security of a process more straightforward, there is a need of some sufficient condition for security such that just individual accepted and refused events, rather than unbounded sequences and sets of events, have to be considered.

Of course, if such a sufficient condition were necessary as well, it would be even more valuable, since it would permit to prove not only that a process is secure by verifying that the condition holds, but also that a process is not secure by verifying that the condition fails to hold.

This section provides a necessary and sufficient condition for CSP noninterference security, which indeed requires to just consider individual accepted and refused events and applies to the general case of a possibly intransitive policy. This condition follows Rushby's output consistency for deterministic state machines with outputs [8], and has to be satisfied by a specific function mapping security domains into equivalence relations over process traces. The definition of this function makes use of an intransitive purge function following Rushby's one; hence the name given to the condition, *Ipurge Unwinding Theorem*.

The contents of this paper are based on those of [6]. The salient points of definitions and proofs are commented; for additional information, cf. Isabelle documentation, particularly [5], [4], [3], and [2].

For the sake of brevity, given a function $F$ of type $'a_1 \Rightarrow \ldots \Rightarrow 'a_m \Rightarrow 'a_{m+1} \Rightarrow \ldots \Rightarrow 'a_n \Rightarrow 'b$, the explanatory text may discuss of $F$ using attributes that would more exactly apply to a term of type $'a_{m+1} \Rightarrow \ldots \Rightarrow 'a_n \Rightarrow 'b$. In this case, it shall be understood that strictly speaking, such attributes apply to a term matching pattern $F\ a_1\ \ldots\ a_m$.

## 1.1  Propaedeutic definitions and lemmas

The definition of CSP noninterference security formulated in [6] requires that some sets of events be refusals, i.e. sets of refused events, for some traces. Therefore, a sufficient condition for security just involving individual refused events will require that some single events be refused, viz. form singleton refusals, after the occurrence of some traces. However, such a statement may actually be a sufficient condition for security just in the case of a process such that the union of any set of singleton refusals for a given trace is itself a refusal for that trace.

This turns out to be true if and only if the union of any set $A$ of refusals, not necessarily singletons, is still a refusal. The direct implication is trivial. As regards the converse one, let $A$' be the set of the singletons included in some element of $A$. Then, each element of $A$' is a singleton refusal by virtue of rule $[\![$ (?xs, ?Y) $\in$ failures ?P; ?X $\subseteq$ ?Y $]\!] \Longrightarrow$ (?xs, ?X) $\in$ failures ?P, so that the union of the elements of $A$', which is equal to the union of the elements of $A$, is a refusal by hypothesis.

This property, henceforth referred to as *refusals union closure* and formalized in what follows, clearly holds for any process admitting a meaningful interpretation, as it would be a nonsense, in the case of a process modeling a real system, to say that some sets of events are refused after the occurrence of a trace, but their union is not. Thus, taking the refusals union closure of the process as an assumption for the equivalence between process security and a given condition, as will be done in the Ipurge Unwinding Theorem, does not give rise to any actual limitation on the applicability of such a result.

As for predicates *view partition* and *future consistent*, defined here below as well, they translate Rushby's predicates *view-partitioned* and *output consistent* [8], applying to deterministic state machines with outputs, into Hoare's Communicating Sequential Processes model of computation [1]. The reason for the verbal difference between the active form of predicate *view partition* and the passive form of predicate *view-partitioned* is that the implied subject of the former is a domain-relation map rather than a process, whose homologous in [8], viz. a machine, is the implied subject of the latter predicate instead.

More remarkably, the formal differences with respect to Rushby's original predicates are the following ones:

- The relations in the range of the domain-relation map hold between event lists rather than machine states.

- The domains appearing as inputs of the domain-relation map do not unnecessarily encompass all the possible values of the data type of domains, but just the domains in the range of the event-domain map.

- The equality of the outputs in domain $u$ produced by machine states equivalent for $u$, as required by output consistency, is replaced by the equality of the events in domain $u$ accepted or refused after the occurrence of event lists equivalent for $u$; hence the name of the property, *future consistency*.

An additional predicate, *weakly future consistent*, renders future consistency less strict by requiring the equality of subsequent accepted and refused events to hold only for event domains not allowed to be affected by some event domain.

**type-synonym** $('a, 'd)$ *dom-rel-map* $= 'd \Rightarrow ('a\ list \times 'a\ list)\ set$

**type-synonym** $('a, 'd)$ *domset-rel-map* $= 'd\ set \Rightarrow ('a\ list \times 'a\ list)\ set$

**definition** *ref-union-closed* $:: 'a\ process \Rightarrow bool$ **where**
*ref-union-closed* $P \equiv$
  $\forall xs\ A.\ (\exists X.\ X \in A) \longrightarrow (\forall X \in A.\ (xs,\ X) \in failures\ P) \longrightarrow$
  $(xs, \bigcup X \in A.\ X) \in failures\ P$

**definition** *view-partition* $::$
  $'a\ process \Rightarrow ('a \Rightarrow 'd) \Rightarrow ('a, 'd)\ dom\text{-}rel\text{-}map \Rightarrow bool$ **where**
*view-partition* $P\ D\ R \equiv \forall u \in range\ D.\ equiv\ (traces\ P)\ (R\ u)$

**definition** *next-dom-events* $::$
  $'a\ process \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd \Rightarrow 'a\ list \Rightarrow 'a\ set$ **where**
*next-dom-events* $P\ D\ u\ xs \equiv \{x.\ u = D\ x \land x \in next\text{-}events\ P\ xs\}$

**definition** *ref-dom-events* $::$
  $'a\ process \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd \Rightarrow 'a\ list \Rightarrow 'a\ set$ **where**
*ref-dom-events* $P\ D\ u\ xs \equiv \{x.\ u = D\ x \land \{x\} \in refusals\ P\ xs\}$

**definition** *future-consistent* $::$
  $'a\ process \Rightarrow ('a \Rightarrow 'd) \Rightarrow ('a, 'd)\ dom\text{-}rel\text{-}map \Rightarrow bool$ **where**
*future-consistent* $P\ D\ R \equiv$
  $\forall u \in range\ D.\ \forall xs\ ys.\ (xs,\ ys) \in R\ u \longrightarrow$
    $next\text{-}dom\text{-}events\ P\ D\ u\ xs = next\text{-}dom\text{-}events\ P\ D\ u\ ys\ \land$
    $ref\text{-}dom\text{-}events\ P\ D\ u\ xs = ref\text{-}dom\text{-}events\ P\ D\ u\ ys$

**definition** *weakly-future-consistent* $::$
  $'a\ process \Rightarrow ('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow ('a, 'd)\ dom\text{-}rel\text{-}map \Rightarrow bool$ **where**

*weakly-future-consistent P I D R ≡*
  *∀ u ∈ range D ∩ (−I) '' range D. ∀ xs ys. (xs, ys) ∈ R u ⟶*
    *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*

Here below are some lemmas propaedeutic for the proof of the Ipurge Unwinding Theorem, just involving constants defined in [6].

**lemma** *process-rule-2-traces*:
 *xs @ xs′ ∈ traces P ⟹ xs ∈ traces P*
⟨*proof*⟩

**lemma** *process-rule-4* [*rule-format*]:
 *(xs, X) ∈ failures P ⟶ (xs @ [x], {}) ∈ failures P ∨ (xs, insert x X) ∈ failures P*
⟨*proof*⟩

**lemma** *failures-traces*:
 *(xs, X) ∈ failures P ⟹ xs ∈ traces P*
⟨*proof*⟩

**lemma** *traces-failures*:
 *xs ∈ traces P ⟹ (xs, {}) ∈ failures P*
⟨*proof*⟩

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

**lemma** *sinks-interference-eq*:
 *((u, D x) ∈ I ∨ (∃ v ∈ sinks I D u xs. (v, D x) ∈ I)) =*
 *(D x ∈ sinks I D u (xs @ [x]))*
⟨*proof*⟩

In what follows, some lemmas concerning the constants defined above are proven.

In the definition of predicate *ref-union-closed*, the conclusion that the union of a set of refusals is itself a refusal for the same trace is subordinated to the condition that the set of refusals be nonempty. The first lemma shows that in the absence of this condition, the predicate could only be satisfied by a process admitting any event list as a trace, which proves that the condition must be present for the definition to be correct.

The subsequent lemmas prove that, for each domain $u$ in the ranges respectively taken into consideration, the image of $u$ under a future consistent or

weakly future consistent domain-relation map may only correlate a pair of event lists such that either both are traces, or both are not traces. Finally, it is demonstrated that future consistency implies weak future consistency.

**lemma**
  **assumes** $A$: $\forall xs\ A.\ (\forall X \in A.\ (xs,\ X) \in failures\ P) \longrightarrow$
    $(xs,\ \bigcup X \in A.\ X) \in failures\ P$
  **shows** $\forall xs.\ xs \in traces\ P$
$\langle proof \rangle$

**lemma** *traces-dom-events*:
  **assumes** $A$: $u \in range\ D$
  **shows** $xs \in traces\ P =$
    $(next\text{-}dom\text{-}events\ P\ D\ u\ xs \cup ref\text{-}dom\text{-}events\ P\ D\ u\ xs \neq \{\})$
    (**is** - = ($?S \neq \{\}$))
$\langle proof \rangle$

**lemma** *fc-traces*:
  **assumes**
    $A$: *future-consistent P D R* **and**
    $B$: $u \in range\ D$ **and**
    $C$: $(xs,\ ys) \in R\ u$
  **shows** $(xs \in traces\ P) = (ys \in traces\ P)$
$\langle proof \rangle$

**lemma** *wfc-traces*:
  **assumes**
    $A$: *weakly-future-consistent P I D R* **and**
    $B$: $u \in range\ D \cap (-I)\ ``\ range\ D$ **and**
    $C$: $(xs,\ ys) \in R\ u$
  **shows** $(xs \in traces\ P) = (ys \in traces\ P)$
$\langle proof \rangle$

**lemma** *fc-implies-wfc*:
 *future-consistent P D R* $\Longrightarrow$ *weakly-future-consistent P I D R*
$\langle proof \rangle$

Finally, the definition is given of an auxiliary function *singleton-set*, whose output is the set of the singleton subsets of a set taken as input, and then some basic properties of this function are proven.

**definition** *singleton-set* :: $'a\ set \Rightarrow\ 'a\ set\ set$ **where**
*singleton-set* $X \equiv \{Y.\ \exists x \in X.\ Y = \{x\}\}$

**lemma** *singleton-set-some*:
 $(\exists Y.\ Y \in singleton\text{-}set\ X) = (\exists x.\ x \in X)$
$\langle proof \rangle$

6

**lemma** *singleton-set-union*:
$(\bigcup Y \in$ *singleton-set X. Y) = X*
$\langle proof \rangle$

## 1.2 Additional intransitive purge functions and their properties

Functions *sinks-aux*, *ipurge-tr-aux*, and *ipurge-ref-aux*, defined here below, are auxiliary versions of functions *sinks*, *ipurge-tr*, and *ipurge-ref* taking as input a set of domains rather than a single domain. As shown below, these functions are useful for the study of single domain ones, involved in the definition of CSP noninterference security [6], since they distribute over list concatenation, while being susceptible to be expressed in terms of the corresponding single domain functions in case the input set of domains is a singleton.

A further function, *unaffected-domains*, takes as inputs a set of domains $U$ and an event list *xs*, and outputs the set of the event domains not allowed to be affected by $U$ after the occurrence of *xs*.

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

**function** *ipurge-tr-aux* ::
$('d \times 'd)$ *set* $\Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd$ *set* $\Rightarrow 'a$ *list* $\Rightarrow 'a$ *list* **where**
*ipurge-tr-aux - - -* [] = [] |
*ipurge-tr-aux I D U* (*xs* @ [*x*]) = (*if* $\exists v \in$ *sinks-aux I D U xs.* (*v, D x*) $\in I$
  *then ipurge-tr-aux I D U xs*
  *else ipurge-tr-aux I D U xs* @ [*x*])
$\langle proof \rangle$
**termination** $\langle proof \rangle$

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

**definition** *unaffected-domains* ::
$('d \times 'd)$ *set* $\Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd$ *set* $\Rightarrow 'a$ *list* $\Rightarrow 'd$ *set* **where**
*unaffected-domains I D U xs* $\equiv$
  $\{u \in$ *range D.* $\forall v \in$ *sinks-aux I D U xs.* (*v, u*) $\notin I\}$

Function *ipurge-tr-rev*, defined here below in terms of function *sources*, is the reverse of function *ipurge-tr* with regard to both the order in which events are considered, and the criterion by which they are purged.

In some detail, both functions *sources* and *ipurge-tr-rev* take as inputs a domain $u$ and an event list $xs$, whose recursive decomposition is performed by item prepending rather than appending. Then:

- *sources* outputs the set of the domains of the events in $xs$ allowed to affect $u$;

- *ipurge-tr-rev* outputs the sublist of $xs$ obtained by recursively deleting the events not allowed to affect $u$, as detected via function *sources*.

In other words, these functions follow Rushby's ones *sources* and *ipurge* [8], formalized in [6] as *c-sources* and *c-ipurge*. The only difference consists of dropping the implicit supposition that the noninterference policy be reflexive, as done in the definition of CPS noninterference security [6]. This goal is achieved by defining the output of function *sources*, when it is applied to the empty list, as being the empty set rather than the singleton comprised of the input domain.

As for functions *sources-aux* and *ipurge-tr-rev-aux*, they are auxiliary versions of functions *sources* and *ipurge-tr-rev* taking as input a set of domains rather than a single domain. As shown below, these functions distribute over list concatenation, while being susceptible to be expressed in terms of the corresponding single domain functions in case the input set of domains is a singleton.

**primrec** *sources* :: $('d \times 'd)$ *set* $\Rightarrow$ $('a \Rightarrow 'd)$ $\Rightarrow$ $'d$ $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'d$ *set* **where**
*sources* - - - $[] = \{\}$ |
*sources* $I$ $D$ $u$ $(x \ \# \ xs) =$
  $(if \ (D \ x, \ u) \in I \vee (\exists v \in sources \ I \ D \ u \ xs. \ (D \ x, \ v) \in I)$
  *then insert* $(D \ x)$ $(sources \ I \ D \ u \ xs)$
  *else sources* $I$ $D$ $u$ $xs)$

**primrec** *ipurge-tr-rev* :: $('d \times 'd)$ *set* $\Rightarrow$ $('a \Rightarrow 'd)$ $\Rightarrow$ $'d$ $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* **where**
*ipurge-tr-rev* - - - $[] = []$ |
*ipurge-tr-rev* $I$ $D$ $u$ $(x \ \# \ xs) = (if \ D \ x \in sources \ I \ D \ u \ (x \ \# \ xs)$
  *then* $x \ \#$ *ipurge-tr-rev* $I$ $D$ $u$ $xs$
  *else ipurge-tr-rev* $I$ $D$ $u$ $xs)$

**primrec** *sources-aux* ::
  $('d \times 'd)$ *set* $\Rightarrow$ $('a \Rightarrow 'd)$ $\Rightarrow$ $'d$ *set* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'d$ *set* **where**
*sources-aux* - - $U \ [] = U$ |
*sources-aux* $I$ $D$ $U$ $(x \ \# \ xs) = (if \ \exists v \in sources\text{-}aux \ I \ D \ U \ xs. \ (D \ x, \ v) \in I$

*then insert* $(D\ x)$ *(sources-aux I D U xs)*
*else sources-aux I D U xs)*

**primrec** *ipurge-tr-rev-aux* ::
$('d \times {}'d)\ set \Rightarrow ('a \Rightarrow {}'d) \Rightarrow {}'d\ set \Rightarrow {}'a\ list \Rightarrow {}'a\ list$ **where**
*ipurge-tr-rev-aux - - - [] = [] |*
*ipurge-tr-rev-aux I D U* $(x\ \#\ xs) = ($*if* $\exists v \in$ *sources-aux I D U xs.* $(D\ x,\ v) \in I$
  *then x # ipurge-tr-rev-aux I D U xs*
  *else ipurge-tr-rev-aux I D U xs)*

Here below are some lemmas on functions *sinks-aux*, *ipurge-tr-aux*, *ipurge-ref-aux*, and *unaffected-domains*. As anticipated above, these lemmas essentially concern distributivity over list concatenation and expressions in terms of single domain functions in the degenerate case of a singleton set of domains.

**lemma** *sinks-aux-subset*:
$U \subseteq$ *sinks-aux I D U xs*
$\langle proof \rangle$

**lemma** *sinks-aux-single-dom*:
*sinks-aux I D* $\{u\}$ *xs = insert u (sinks I D u xs)*
$\langle proof \rangle$

**lemma** *sinks-aux-single-event*:
*sinks-aux I D U* $[x] = ($*if* $\exists v \in U.\ (v,\ D\ x) \in I$
  *then insert* $(D\ x)\ U$
  *else U)*
$\langle proof \rangle$

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

**lemma** *ipurge-tr-aux-single-dom*:
*ipurge-tr-aux I D* $\{u\}$ *xs = ipurge-tr I D u xs*
$\langle proof \rangle$

**lemma** *ipurge-ref-aux-single-dom*:
*ipurge-ref-aux I D* $\{u\}$ *xs X = ipurge-ref I D u xs X*
$\langle proof \rangle$

**lemma** *ipurge-ref-aux-all* [*rule-format*]:
$(\forall u \in U.\ \neg\ (\exists v \in D\ {}^\backprime\ (X \cup set\ xs).\ (u,\ v) \in I)) \longrightarrow$
  *ipurge-ref-aux I D U xs X = X*
$\langle proof \rangle$

**lemma** *ipurge-ref-all*:
  $\neg\ (\exists\, v \in D\ `\ (X \cup set\ xs).\ (u,\ v) \in I) \Longrightarrow ipurge\text{-}ref\ I\ D\ u\ xs\ X = X$
  $\langle proof \rangle$

**lemma** *unaffected-domains-single-dom*:
  $\{x \in X.\ D\ x \in unaffected\text{-}domains\ I\ D\ \{u\}\ xs\} = ipurge\text{-}ref\ I\ D\ u\ xs\ X$
  $\langle proof \rangle$


Here below are some lemmas on functions *sources*, *ipurge-tr-rev*, *sources-aux*, and *ipurge-tr-rev-aux*. As anticipated above, the lemmas on the last two functions basically concern distributivity over list concatenation and expressions in terms of single domain functions in the degenerate case of a singleton set of domains.


**lemma** *sources-sinks*:
  $sources\ I\ D\ u\ xs = sinks\ (I^{-1})\ D\ u\ (rev\ xs)$
  $\langle proof \rangle$

**lemma** *sources-sinks-aux*:
  $sources\text{-}aux\ I\ D\ U\ xs = sinks\text{-}aux\ (I^{-1})\ D\ U\ (rev\ xs)$
  $\langle proof \rangle$

**lemma** *sources-aux-subset*:
  $U \subseteq sources\text{-}aux\ I\ D\ U\ xs$
  $\langle proof \rangle$

**lemma** *sources-aux-append*:
  $sources\text{-}aux\ I\ D\ U\ (xs\ @\ ys) = sources\text{-}aux\ I\ D\ (sources\text{-}aux\ I\ D\ U\ ys)\ xs$
  $\langle proof \rangle$

**lemma** *sources-aux-append-nil* [*rule-format*]:
  $sources\text{-}aux\ I\ D\ U\ ys = U \longrightarrow$
    $sources\text{-}aux\ I\ D\ U\ (xs\ @\ ys) = sources\text{-}aux\ I\ D\ U\ xs$
  $\langle proof \rangle$

**lemma** *ipurge-tr-rev-aux-append*:
  $ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ U\ (xs\ @\ ys) =$
    $ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ (sources\text{-}aux\ I\ D\ U\ ys)\ xs\ @\ ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ U\ ys$
  $\langle proof \rangle$

**lemma** *ipurge-tr-rev-aux-nil-1* [*rule-format*]:
  $ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ U\ xs = [] \longrightarrow (\forall\, u \in U.\ \neg\ (\exists\, v \in D\ `\ set\ xs.\ (v,\ u) \in I))$
  $\langle proof \rangle$

**lemma** *ipurge-tr-rev-aux-nil-2* [*rule-format*]:
  $(\forall\, u \in U.\ \neg\ (\exists\, v \in D\ `\ set\ xs.\ (v,\ u) \in I)) \longrightarrow ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ U\ xs = []$
  $\langle proof \rangle$

**lemma** *ipurge-tr-rev-aux-nil*:
 (*ipurge-tr-rev-aux I D U xs* = []) = (∀ *u* ∈ *U*. ¬ (∃ *v* ∈ *D* ' *set xs*. (*v*, *u*) ∈ *I*))
⟨*proof*⟩

**lemma** *ipurge-tr-rev-aux-nil-sources* [*rule-format*]:
 *ipurge-tr-rev-aux I D U xs* = [] ⟶ *sources-aux I D U xs* = *U*
⟨*proof*⟩

**lemma** *ipurge-tr-rev-aux-append-nil-1* [*rule-format*]:
 *ipurge-tr-rev-aux I D U ys* = [] ⟶
  *ipurge-tr-rev-aux I D U* (*xs* @ *ys*) = *ipurge-tr-rev-aux I D U xs*
⟨*proof*⟩

**lemma** *ipurge-tr-rev-aux-first* [*rule-format*]:
 *ipurge-tr-rev-aux I D U xs* = *x* # *ws* ⟶
  (∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧
    *ipurge-tr-rev-aux I D* (*sources-aux I D U* (*x* # *zs*)) *ys* = [] ∧
    (∃ *v* ∈ *sources-aux I D U zs*. (*D x*, *v*) ∈ *I*))
⟨*proof*⟩

**lemma** *ipurge-tr-rev-aux-last-1* [*rule-format*]:
 *ipurge-tr-rev-aux I D U xs* = *ws* @ [*x*] ⟶ (∃ *v* ∈ *U*. (*D x*, *v*) ∈ *I*)
⟨*proof*⟩

**lemma** *ipurge-tr-rev-aux-last-2* [*rule-format*]:
 *ipurge-tr-rev-aux I D U xs* = *ws* @ [*x*] ⟶
  (∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *ipurge-tr-rev-aux I D U zs* = [])
⟨*proof*⟩

**lemma** *ipurge-tr-rev-aux-all* [*rule-format*]:
 (∀ *v* ∈ *D* ' *set xs*. ∃ *u* ∈ *U*. (*v*, *u*) ∈ *I*) ⟶ *ipurge-tr-rev-aux I D U xs* = *xs*
⟨*proof*⟩


Here below, further properties of the functions defined above are investigated
thanks to the introduction of function *offset*, which searches a list for a given
item and returns the offset of its first occurrence, if any, from the first item
of the list.


**primrec** *offset* :: *nat* ⇒ ′*a* ⇒ ′*a list* ⇒ *nat option* **where**
*offset - -* [] = *None* |
*offset n x* (*y* # *ys*) = (*if y* = *x then Some n else offset* (*Suc n*) *x ys*)

**lemma** *offset-not-none-1* [*rule-format*]:
 *offset k x xs* ≠ *None* ⟶ (∃ *ys zs*. *xs* = *ys* @ *x* # *zs*)
⟨*proof*⟩

**lemma** *offset-not-none-2* [*rule-format*]:

11

$xs = ys \mathbin{@} x \mathbin{\#} zs \longrightarrow \text{offset } k\ x\ xs \neq None$
$\langle proof \rangle$

**lemma** *offset-not-none*:
$(\text{offset } k\ x\ xs \neq None) = (\exists\, ys\ zs.\ xs = ys \mathbin{@} x \mathbin{\#} zs)$
$\langle proof \rangle$

**lemma** *offset-addition* [*rule-format*]:
$\text{offset } k\ x\ xs \neq None \longrightarrow \text{offset } (n + m)\ x\ xs = Some\ (the\ (\text{offset } n\ x\ xs) + m)$
$\langle proof \rangle$

**lemma** *offset-suc*:
  **assumes** *A*: $\text{offset } k\ x\ xs \neq None$
  **shows** $\text{offset } (Suc\ n)\ x\ xs = Some\ (Suc\ (the\ (\text{offset } n\ x\ xs)))$
$\langle proof \rangle$

**lemma** *ipurge-tr-rev-aux-first-offset* [*rule-format*]:
$xs = ys \mathbin{@} x \mathbin{\#} zs \land \text{ipurge-tr-rev-aux } I\ D\ (\text{sources-aux } I\ D\ U\ (x \mathbin{\#} zs))\ ys = [\,] \land$
  $(\exists\, v \in \text{sources-aux } I\ D\ U\ zs.\ (D\ x,\ v) \in I) \longrightarrow$
 $ys = take\ (the\ (\text{offset } 0\ x\ xs))\ xs$
$\langle proof \rangle$

**lemma** *ipurge-tr-rev-aux-append-nil-2* [*rule-format*]:
$\text{ipurge-tr-rev-aux } I\ D\ U\ (xs \mathbin{@} ys) = \text{ipurge-tr-rev-aux } I\ D\ V\ xs \longrightarrow$
 $\text{ipurge-tr-rev-aux } I\ D\ U\ ys = [\,]$
$\langle proof \rangle$

**lemma** *ipurge-tr-rev-aux-append-nil*:
$(\text{ipurge-tr-rev-aux } I\ D\ U\ (xs \mathbin{@} ys) = \text{ipurge-tr-rev-aux } I\ D\ U\ xs) =$
 $(\text{ipurge-tr-rev-aux } I\ D\ U\ ys = [\,])$
$\langle proof \rangle$

In what follows, it is proven by induction that the lists output by functions *ipurge-tr* and *ipurge-tr-rev*, as well as those output by *ipurge-tr-aux* and *ipurge-tr-rev-aux*, satisfy predicate *Interleaves* (cf. [7]), in correspondence with suitable input predicates expressed in terms of functions *sinks* and *sinks-aux*, respectively. Then, some lemmas on the aforesaid functions are demonstrated without induction, using previous lemmas along with the properties of predicate *Interleaves*.

**lemma** *Interleaves-ipurge-tr*:
$xs \cong \{\text{ipurge-tr-rev } I\ D\ u\ xs,\ rev\ (\text{ipurge-tr } (I^{-1})\ D\ u\ (rev\ xs)),$
  $\lambda y\ ys.\ D\ y \in \text{sinks } (I^{-1})\ D\ u\ (rev\ (y \mathbin{\#} ys))\}$
$\langle proof \rangle$

**lemma** *Interleaves-ipurge-tr-aux*:
$xs \cong \{\text{ipurge-tr-rev-aux } I\ D\ U\ xs,\ rev\ (\text{ipurge-tr-aux } (I^{-1})\ D\ U\ (rev\ xs)),$

$\lambda y\ ys.\ \exists\,v \in sinks\text{-}aux\ (I^{-1})\ D\ U\ (rev\ ys).\ (D\ y,\ v) \in I\}$
$\langle proof \rangle$

**lemma** *ipurge-tr-aux-all*:
 $(ipurge\text{-}tr\text{-}aux\ I\ D\ U\ xs = xs) = (\forall\,u \in U.\ \neg\ (\exists\,v \in D\ `\ set\ xs.\ (u,\ v) \in I))$
$\langle proof \rangle$

**lemma** *ipurge-tr-rev-aux-single-dom*:
 $ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ \{u\}\ xs = ipurge\text{-}tr\text{-}rev\ I\ D\ u\ xs$ (**is** *?ys = ?ys′*)
$\langle proof \rangle$

**lemma** *ipurge-tr-all*:
 $(ipurge\text{-}tr\ I\ D\ u\ xs = xs) = (\neg\ (\exists\,v \in D\ `\ set\ xs.\ (u,\ v) \in I))$
$\langle proof \rangle$

**lemma** *ipurge-tr-rev-all*:
 $\forall\,v \in D\ `\ set\ xs.\ (v,\ u) \in I \implies ipurge\text{-}tr\text{-}rev\ I\ D\ u\ xs = xs$
$\langle proof \rangle$

## 1.3   A domain-relation map based on intransitive purge

In what follows, constant *rel-ipurge* is defined as the domain-relation map that associates each domain *u* to the relation comprised of the pairs of traces whose images under function *ipurge-tr-rev I D u* are equal, viz. whose events affecting *u* are the same.

An auxiliary domain set-relation map, *rel-ipurge-aux*, is also defined by replacing *ipurge-tr-rev* with *ipurge-tr-rev-aux*, so as to exploit the distributivity of the latter function over list concatenation. Unsurprisingly, since *ipurge-tr-rev-aux* degenerates into *ipurge-tr-rev* for a singleton set of domains, the same happens for *rel-ipurge-aux* and *rel-ipurge*.

Subsequently, some basic properties of domain-relation map *rel-ipurge* are proven, namely that it is a view partition, and is future consistent if and only if it is weakly future consistent. The nontrivial implication, viz. the direct one, derives from the fact that for each domain *u* allowed to be affected by any event domain, function *ipurge-tr-rev I D u* matches the identity function, so that two traces are correlated by the image of *rel-ipurge* under *u* just in case they are equal.

**definition** *rel-ipurge* ::
 *′a process* $\Rightarrow$ *(′d × ′d) set* $\Rightarrow$ *(′a* $\Rightarrow$ *′d)* $\Rightarrow$ *(′a, ′d) dom-rel-map* **where**
*rel-ipurge P I D u* $\equiv$ *{(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}*

**definition** *rel-ipurge-aux* ::
 *′a process* $\Rightarrow$ *(′d × ′d) set* $\Rightarrow$ *(′a* $\Rightarrow$ *′d)* $\Rightarrow$ *(′a, ′d) domset-rel-map* **where**
*rel-ipurge-aux P I D U* $\equiv$ *{(xs, ys). xs* $\in$ *traces P* $\wedge$ *ys* $\in$ *traces P* $\wedge$

*ipurge-tr-rev-aux I D U xs = ipurge-tr-rev-aux I D U ys*}

**lemma** *rel-ipurge-aux-single-dom*:
 *rel-ipurge-aux P I D {u} = rel-ipurge P I D u*
⟨*proof*⟩

**lemma** *view-partition-rel-ipurge*:
 *view-partition P D (rel-ipurge P I D)*
⟨*proof*⟩

**lemma** *fc-equals-wfc-rel-ipurge*:
 *future-consistent P D (rel-ipurge P I D) =*
  *weakly-future-consistent P I D (rel-ipurge P I D)*
⟨*proof*⟩

## 1.4 The Ipurge Unwinding Theorem: proof of condition sufficiency

The Ipurge Unwinding Theorem, formalized in what follows as theorem *ipurge-unwinding*, states that a necessary and sufficient condition for the CSP noninterference security [6] of a process being refusals union closed is that domain-relation map *rel-ipurge* be weakly future consistent. Notwithstanding the equivalence of future consistency and weak future consistency for *rel-ipurge* (cf. above), expressing the theorem in terms of the latter reduces the range of the domains to be considered in order to prove or disprove the security of a process, and then is more convenient.

According to the definition of CSP noninterference security formulated in [6], a process is regarded as being secure just in case the occurrence of an event $e$ may only affect future events allowed to be affected by $e$. Identifying security with the weak future consistency of *rel-ipurge* means reversing the view of the problem with respect to the direction of time. In fact, from this view, a process is secure just in case the occurrence of an event $e$ may only be affected by past events allowed to affect $e$. Therefore, what the Ipurge Unwinding Theorem proves is that ultimately, opposite perspectives with regard to the direction of time give rise to equivalent definitions of the noninterference security of a process.

Here below, it is proven that the condition expressed by the Ipurge Unwinding Theorem is sufficient for security.

**lemma** *ipurge-tr-rev-ipurge-tr-aux-1* [*rule-format*]:
 *U ⊆ unaffected-domains I D (D ' set ys) zs ⟶*
 *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*⟩

**lemma** *ipurge-tr-rev-ipurge-tr-aux-2* [*rule-format*]:

$U \subseteq$ *unaffected-domains I D* $(D \, ' \, set \, ys) \, zs \longrightarrow$
*ipurge-tr-rev-aux I D U* $(xs \, @ \, zs) =$
*ipurge-tr-rev-aux I D U* $(xs \, @ \, ys \, @ \, ipurge\text{-}tr\text{-}aux \, I \, D \, (D \, ' \, set \, ys) \, zs)$
$\langle proof \rangle$

**lemma** *ipurge-tr-rev-ipurge-tr-1*:
  **assumes** $A$: $u \in$ *unaffected-domains I D* $\{D \, y\} \, zs$
  **shows** *ipurge-tr-rev I D u* $(xs \, @ \, y \, \# \, zs) =$
    *ipurge-tr-rev I D u* $(xs \, @ \, ipurge\text{-}tr \, I \, D \, (D \, y) \, zs)$
$\langle proof \rangle$

**lemma** *ipurge-tr-rev-ipurge-tr-2*:
  **assumes** $A$: $u \in$ *unaffected-domains I D* $\{D \, y\} \, zs$
  **shows** *ipurge-tr-rev I D u* $(xs \, @ \, zs) =$
    *ipurge-tr-rev I D u* $(xs \, @ \, y \, \# \, ipurge\text{-}tr \, I \, D \, (D \, y) \, zs)$
$\langle proof \rangle$

**lemma** *iu-condition-imply-secure-aux-1*:
  **assumes**
    $RUC$: *ref-union-closed P* **and**
    $IU$: *weakly-future-consistent P I D* (*rel-ipurge P I D*) **and**
    $A$: $(xs \, @ \, y \, \# \, ys, \, Y) \in$ *failures P* **and**
    $B$: $xs \, @ \, ipurge\text{-}tr \, I \, D \, (D \, y) \, ys \in$ *traces P* **and**
    $C$: $\exists \, y'. \, y' \in$ *ipurge-ref I D* $(D \, y) \, ys \, Y$
  **shows** $(xs \, @ \, ipurge\text{-}tr \, I \, D \, (D \, y) \, ys, \, ipurge\text{-}ref \, I \, D \, (D \, y) \, ys \, Y) \in$ *failures P*
$\langle proof \rangle$

**lemma** *iu-condition-imply-secure-aux-2*:
  **assumes**
    $RUC$: *ref-union-closed P* **and**
    $IU$: *weakly-future-consistent P I D* (*rel-ipurge P I D*) **and**
    $A$: $(xs \, @ \, zs, \, Z) \in$ *failures P* **and**
    $B$: $xs \, @ \, y \, \# \, ipurge\text{-}tr \, I \, D \, (D \, y) \, zs \in$ *traces P* **and**
    $C$: $\exists \, z'. \, z' \in$ *ipurge-ref I D* $(D \, y) \, zs \, Z$
  **shows** $(xs \, @ \, y \, \# \, ipurge\text{-}tr \, I \, D \, (D \, y) \, zs, \, ipurge\text{-}ref \, I \, D \, (D \, y) \, zs \, Z) \in$ *failures P*
$\langle proof \rangle$

**lemma** *iu-condition-imply-secure-1* [*rule-format*]:
  **assumes**
    $RUC$: *ref-union-closed P* **and**
    $IU$: *weakly-future-consistent P I D* (*rel-ipurge P I D*)
  **shows** $(xs \, @ \, y \, \# \, ys, \, Y) \in$ *failures P* $\longrightarrow$
    $(xs \, @ \, ipurge\text{-}tr \, I \, D \, (D \, y) \, ys, \, ipurge\text{-}ref \, I \, D \, (D \, y) \, ys \, Y) \in$ *failures P*
$\langle proof \rangle$

**lemma** *iu-condition-imply-secure-2* [*rule-format*]:
  **assumes**
    $RUC$: *ref-union-closed P* **and**
    $IU$: *weakly-future-consistent P I D* (*rel-ipurge P I D*) **and**

*Y*: *xs @ [y] ∈ traces P*
  **shows** (*xs @ zs, Z*) ∈ *failures P* ⟶
    (*xs @ y # ipurge-tr I D (D y) zs, ipurge-ref I D (D y) zs Z*) ∈ *failures P*
⟨*proof*⟩

**theorem** *iu-condition-imply-secure*:
  **assumes**
    *RUC*: *ref-union-closed P* **and**
    *IU*: *weakly-future-consistent P I D (rel-ipurge P I D)*
  **shows** *secure P I D*
⟨*proof*⟩

## 1.5 The Ipurge Unwinding Theorem: proof of condition necessity

Here below, it is proven that the condition expressed by the Ipurge Unwinding Theorem is necessary for security. Finally, the lemmas concerning condition sufficiency and necessity are gathered in the main theorem.

**lemma** *secure-implies-failure-consistency-aux* [*rule-format*]:
  **assumes** *S*: *secure P I D*
  **shows** (*xs @ ys @ zs, X*) ∈ *failures P* ⟶
    *ipurge-tr-rev-aux I D (D ' (X ∪ set zs)) ys = []* ⟶ (*xs @ zs, X*) ∈ *failures P*
⟨*proof*⟩

**lemma** *secure-implies-failure-consistency* [*rule-format*]:
  **assumes** *S*: *secure P I D*
  **shows** (*xs, ys*) ∈ *rel-ipurge-aux P I D (D ' (X ∪ set zs))* ⟶
    (*xs @ zs, X*) ∈ *failures P* ⟶ (*ys @ zs, X*) ∈ *failures P*
⟨*proof*⟩

**lemma** *secure-implies-trace-consistency*:
  *secure P I D* ⟹ (*xs, ys*) ∈ *rel-ipurge-aux P I D (D ' set zs)* ⟹
  *xs @ zs ∈ traces P* ⟹ *ys @ zs ∈ traces P*
⟨*proof*⟩

**lemma** *secure-implies-next-event-consistency*:
  *secure P I D* ⟹ (*xs, ys*) ∈ *rel-ipurge P I D (D x)* ⟹
  *x ∈ next-events P xs* ⟹ *x ∈ next-events P ys*
  ⟨*proof*⟩

**lemma** *secure-implies-refusal-consistency*:
  *secure P I D* ⟹ (*xs, ys*) ∈ *rel-ipurge-aux P I D (D ' X)* ⟹
  *X ∈ refusals P xs* ⟹ *X ∈ refusals P ys*
⟨*proof*⟩

**lemma** *secure-implies-ref-event-consistency*:
  *secure P I D* ⟹ (*xs, ys*) ∈ *rel-ipurge P I D (D x)* ⟹

$\{x\} \in$ *refusals P xs* $\implies \{x\} \in$ *refusals P ys*
⟨*proof*⟩

**theorem** *secure-implies-iu-condition*:
  **assumes** *S*: *secure P I D*
  **shows** *future-consistent P D* (*rel-ipurge P I D*)
⟨*proof*⟩

**theorem** *ipurge-unwinding*:
 *ref-union-closed P* $\implies$
  *secure P I D* = *weakly-future-consistent P I D* (*rel-ipurge P I D*)
⟨*proof*⟩

**end**

# 2 The Ipurge Unwinding Theorem for deterministic and trace set processes

**theory** *DeterministicProcesses*
**imports** *IpurgeUnwinding*
**begin**


In accordance with Hoare's formal definition of deterministic processes [1], this section shows that a process is deterministic just in case it is a *trace set process*, i.e. it may be identified by means of a trace set alone, matching the set of its traces, in place of a failures-divergences pair. Then, variants of the Ipurge Unwinding Theorem are proven for deterministic processes and trace set processes.


## 2.1 Deterministic processes

Here below are the definitions of predicates *d-future-consistent* and *d-weakly-future-consistent*, which are variants of predicates *future-consistent* and *weakly-future-consistent* meant for applying to deterministic processes. In some detail, being deterministic processes such that refused events are completely specified by accepted events (cf. [1], [6]), the new predicates are such that their truth values can be determined by just considering the accepted events of the process taken as input.

Then, it is proven that these predicates are characterized by the same connection as that of their general-purpose counterparts, viz. *d-future-consistent* implies *d-weakly-future-consistent*, and they are equivalent for domain-relation map *rel-ipurge*. Finally, the predicates are shown to be equivalent to their general-purpose counterparts in the case of a deterministic process.

**definition** *d-future-consistent* ::
 *'a process ⇒ ('a ⇒ 'd) ⇒ ('a, 'd) dom-rel-map ⇒ bool* **where**
*d-future-consistent P D R ≡*
  *∀ u ∈ range D. ∀ xs ys. (xs, ys) ∈ R u ⟶*
   *(xs ∈ traces P) = (ys ∈ traces P) ∧*
   *next-dom-events P D u xs = next-dom-events P D u ys*

**definition** *d-weakly-future-consistent* ::
 *'a process ⇒ ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ ('a, 'd) dom-rel-map ⇒ bool* **where**
*d-weakly-future-consistent P I D R ≡*
  *∀ u ∈ range D ∩ (−I) '' range D. ∀ xs ys. (xs, ys) ∈ R u ⟶*
   *(xs ∈ traces P) = (ys ∈ traces P) ∧*
   *next-dom-events P D u xs = next-dom-events P D u ys*

**lemma** *dfc-implies-dwfc*:
 *d-future-consistent P D R ⟹ d-weakly-future-consistent P I D R*
⟨*proof*⟩

**lemma** *dfc-equals-dwfc-rel-ipurge*:
 *d-future-consistent P D (rel-ipurge P I D) =*
  *d-weakly-future-consistent P I D (rel-ipurge P I D)*
⟨*proof*⟩

**lemma** *d-fc-equals-dfc*:
  **assumes** *A*: *deterministic P*
  **shows** *future-consistent P D R = d-future-consistent P D R*
⟨*proof*⟩

**lemma** *d-wfc-equals-dwfc*:
  **assumes** *A*: *deterministic P*
  **shows** *weakly-future-consistent P I D R = d-weakly-future-consistent P I D R*
⟨*proof*⟩


Here below is the proof of a variant of the Ipurge Unwinding Theorem applying to deterministic processes. Unsurprisingly, its enunciation contains predicate *d-weakly-future-consistent* in place of *weakly-future-consistent*. Furthermore, the assumption that the process be refusals union closed is replaced by the assumption that it be deterministic, since the former property is shown to be entailed by the latter.

**lemma** *d-implies-ruc*:
  **assumes** *A*: *deterministic P*
  **shows** *ref-union-closed P*
⟨*proof*⟩

**theorem** *d-ipurge-unwinding*:
  **assumes** *A*: *deterministic P*

**shows** *secure P I D = d-weakly-future-consistent P I D (rel-ipurge P I D)*
⟨*proof*⟩

## 2.2 Trace set processes

In [1], section 2.8, Hoare formulates a simplified definition of a deterministic process, identified with a *trace set*, i.e. a set of event lists containing the empty list and any prefix of each of its elements. Of course, this is consistent with the definition of determinism applying to processes identified with failures-divergences pairs, which implies that their refusals are completely specified by their traces (cf. [1], [6]).

Here below are the definitions of a function *ts-process*, converting the input set of lists into a process, and a predicate *trace-set*, returning *True* just in case the input set of lists has the aforesaid properties. An analysis is then conducted about the output of the functions defined in [6], section 1.1, when acting on a *trace set process*, i.e. a process that may be expressed as *ts-process T* where *trace-set T* matches *True*.

**definition** *ts-process* :: *'a list set ⇒ 'a process* **where**
*ts-process T ≡ Abs-process ({(xs, X). xs ∈ T ∧ (∀ x ∈ X. xs @ [x] ∉ T)}, {})*

**definition** *trace-set* :: *'a list set ⇒ bool* **where**
*trace-set T ≡ [] ∈ T ∧ (∀ xs x. xs @ [x] ∈ T ⟶ xs ∈ T)*

**lemma** *ts-process-rep*:
  **assumes** *A*: *trace-set T*
  **shows** *Rep-process (ts-process T) =*
    *({(xs, X). xs ∈ T ∧ (∀ x ∈ X. xs @ [x] ∉ T)}, {})*
⟨*proof*⟩

**lemma** *ts-process-failures*:
 *trace-set T ⟹*
 *failures (ts-process T) = {(xs, X). xs ∈ T ∧ (∀ x ∈ X. xs @ [x] ∉ T)}*
⟨*proof*⟩

**lemma** *ts-process-futures*:
 *trace-set T ⟹*
 *futures (ts-process T) xs =*
 *{(ys, Y). xs @ ys ∈ T ∧ (∀ y ∈ Y. xs @ ys @ [y] ∉ T)}*
⟨*proof*⟩

**lemma** *ts-process-traces*:
 *trace-set T ⟹ traces (ts-process T) = T*
⟨*proof*⟩

**lemma** *ts-process-refusals*:
 *trace-set T ⟹ xs ∈ T ⟹*

*refusals (ts-process T) xs = {X. ∀ x ∈ X. xs @ [x] ∉ T}*
⟨*proof*⟩

**lemma** *ts-process-next-events*:
 *trace-set T ⟹ (x ∈ next-events (ts-process T) xs) = (xs @ [x] ∈ T)*
⟨*proof*⟩

In what follows, the proof is given of two results which provide a connection between the notions of deterministic and trace set processes: any trace set process is deterministic, and any process is deterministic just in case it is equal to the trace set process corresponding to the set of its traces.

**lemma** *ts-process-d*:
 *trace-set T ⟹ deterministic (ts-process T)*
⟨*proof*⟩

**definition** *divergences* :: *′a process ⇒ ′a list set* **where**
*divergences P ≡ snd (Rep-process P)*

**lemma** *d-divergences*:
  **assumes** *A*: *deterministic P*
  **shows** *divergences P = {}*
⟨*proof*⟩

**lemma** *trace-set-traces*:
 *trace-set (traces P)*
⟨*proof*⟩

**lemma** *d-implies-ts-process-traces*:
 *deterministic P ⟹ ts-process (traces P) = P*
⟨*proof*⟩

**lemma** *ts-process-traces-implies-d*:
 *ts-process (traces P) = P ⟹ deterministic P*
⟨*proof*⟩

**lemma** *d-equals-ts-process-traces*:
 *deterministic P = (ts-process (traces P) = P)*
⟨*proof*⟩

Finally, a variant of the Ipurge Unwinding Theorem applying to trace set processes is derived from the variant for deterministic processes. Particularly, the assumption that the process be deterministic is replaced by the assumption that it be a trace set process, since the former property is entailed by the latter (cf. above).

**theorem** *ts-ipurge-unwinding*:
 *trace-set* $T \Longrightarrow$
 *secure* (*ts-process* $T$) $I\ D =$
 *d-weakly-future-consistent* (*ts-process* $T$) $I\ D$ (*rel-ipurge* (*ts-process* $T$) $I\ D$)
$\langle proof \rangle$

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

[7] P. Noce. Reasoning about lists via list interleaving. *Archive of Formal Proofs*, June 2015. http://isa-afp.org/entries/List_Interleaving.shtml, Formal proof development.

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