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

October 13, 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 $[\![(?xs,\ ?Y) \in \mathit{failures}\ ?P;\ ?X \subseteq\ ?Y]\!] \Longrightarrow (?xs,\ ?X) \in \mathit{failures}\ ?P$, 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 $P\ D\ u\ xs \equiv \{x.\ u = D\ x \wedge x \in next\text{-}events\ P\ xs\}$

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

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

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**

$\mathit{weakly\text{-}future\text{-}consistent}\ P\ I\ D\ R \equiv$
$\quad \forall u \in \mathit{range}\ D \cap (-I)\ ``\ \mathit{range}\ D.\ \forall \mathit{xs}\ \mathit{ys}.\ (\mathit{xs}, \mathit{ys}) \in R\ u \longrightarrow$
$\qquad \mathit{next\text{-}dom\text{-}events}\ P\ D\ u\ \mathit{xs} = \mathit{next\text{-}dom\text{-}events}\ P\ D\ u\ \mathit{ys} \wedge$
$\qquad \mathit{ref\text{-}dom\text{-}events}\ P\ D\ u\ \mathit{xs} = \mathit{ref\text{-}dom\text{-}events}\ P\ D\ u\ \mathit{ys}$

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*:
$\mathit{xs}\ @\ \mathit{xs}' \in \mathit{traces}\ P \Longrightarrow \mathit{xs} \in \mathit{traces}\ P$
proof (*simp add*: *traces-def Domain-iff*, *erule exE*, *rule-tac* $x = \{\}$ **in** *exI*)
qed (*rule process-rule-2-failures*)

lemma *process-rule-4* [*rule-format*]:
$(\mathit{xs}, X) \in \mathit{failures}\ P \longrightarrow (\mathit{xs}\ @\ [x], \{\}) \in \mathit{failures}\ P \vee (\mathit{xs}, \mathit{insert}\ x\ X) \in \mathit{failures}\ P$
proof (*simp add*: *failures-def*)
 have $\mathit{Rep\text{-}process}\ P \in \mathit{process\text{-}set}$ (**is** $?P' \in$ -) **by** (*rule Rep-process*)
 hence $\forall \mathit{xs}\ x\ X.\ (\mathit{xs}, X) \in \mathit{fst}\ ?P' \longrightarrow$
  $(\mathit{xs}\ @\ [x], \{\}) \in \mathit{fst}\ ?P' \vee (\mathit{xs}, \mathit{insert}\ x\ X) \in \mathit{fst}\ ?P'$
  by (*simp add*: *process-set-def process-prop-4-def*)
 thus $(\mathit{xs}, X) \in \mathit{fst}\ ?P' \longrightarrow$
  $(\mathit{xs}\ @\ [x], \{\}) \in \mathit{fst}\ ?P' \vee (\mathit{xs}, \mathit{insert}\ x\ X) \in \mathit{fst}\ ?P'$
  by *blast*
qed

lemma *failures-traces*:
$(\mathit{xs}, X) \in \mathit{failures}\ P \Longrightarrow \mathit{xs} \in \mathit{traces}\ P$
by (*simp add*: *traces-def Domain-iff*, *rule exI*)

lemma *traces-failures*:
$\mathit{xs} \in \mathit{traces}\ P \Longrightarrow (\mathit{xs}, \{\}) \in \mathit{failures}\ P$
proof (*simp add*: *traces-def Domain-iff*, *erule exE*)
qed (*erule process-rule-3*, *simp*)

lemma *sinks-interference* [*rule-format*]:
$D\ x \in \mathit{sinks}\ I\ D\ u\ \mathit{xs} \longrightarrow$
 $(u, D\ x) \in I \vee (\exists v \in \mathit{sinks}\ I\ D\ u\ \mathit{xs}.\ (v, D\ x) \in I)$
proof (*induction xs rule*: *rev-induct*, *simp*, *rule impI*)
 fix $x'\ \mathit{xs}$
 assume
  A: $D\ x \in \mathit{sinks}\ I\ D\ u\ \mathit{xs} \longrightarrow$
   $(u, D\ x) \in I \vee (\exists v \in \mathit{sinks}\ I\ D\ u\ \mathit{xs}.\ (v, D\ x) \in I)$ **and**
  B: $D\ x \in \mathit{sinks}\ I\ D\ u\ (\mathit{xs}\ @\ [x'])$
 show $(u, D\ x) \in I \vee (\exists v \in \mathit{sinks}\ I\ D\ u\ (\mathit{xs}\ @\ [x']).\ (v, D\ x) \in I)$
 proof (*cases* $(u, D\ x') \in I \vee (\exists v \in \mathit{sinks}\ I\ D\ u\ \mathit{xs}.\ (v, D\ x') \in I)$)
  case *True*
  hence $D\ x = D\ x' \vee D\ x \in \mathit{sinks}\ I\ D\ u\ \mathit{xs}$ **using** B **by** *simp*

```
    moreover {
      assume C: D x = D x'
      have ?thesis using True
      proof (rule disjE, erule-tac [2] bexE)
        assume (u, D x') ∈ I
        hence (u, D x) ∈ I using C by simp
        thus ?thesis ..
      next
        fix v
        assume (v, D x') ∈ I
        hence (v, D x) ∈ I using C by simp
        moreover assume v ∈ sinks I D u xs
        hence v ∈ sinks I D u (xs @ [x']) by simp
        ultimately have ∃ v ∈ sinks I D u (xs @ [x']). (v, D x) ∈ I ..
        thus ?thesis ..
      qed
    }
    moreover {
      assume D x ∈ sinks I D u xs
      with A have (u, D x) ∈ I ∨ (∃ v ∈ sinks I D u xs. (v, D x) ∈ I) ..
      hence ?thesis
      proof (rule disjE, erule-tac [2] bexE)
        assume (u, D x) ∈ I
        thus ?thesis ..
      next
        fix v
        assume (v, D x) ∈ I
        moreover assume v ∈ sinks I D u xs
        hence v ∈ sinks I D u (xs @ [x']) by simp
        ultimately have ∃ v ∈ sinks I D u (xs @ [x']). (v, D x) ∈ I ..
        thus ?thesis ..
      qed
    }
    ultimately show ?thesis ..
  next
    case False
    hence C: sinks I D u (xs @ [x']) = sinks I D u xs by simp
    hence D x ∈ sinks I D u xs using B by simp
    with A have (u, D x) ∈ I ∨ (∃ v ∈ sinks I D u xs. (v, D x) ∈ I) ..
    thus ?thesis using C by simp
  qed
qed

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 (rule iffI, erule-tac [2] contrapos-pp, simp-all (no-asm-simp))
qed (erule contrapos-nn, rule sinks-interference)
```

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: $\forall xs\ A.\ (\forall X \in A.\ (xs,\ X) \in \mathit{failures}\ P) \longrightarrow$
    $(xs,\ \bigcup X \in A.\ X) \in \mathit{failures}\ P$
  shows $\forall xs.\ xs \in \mathit{traces}\ P$
proof
  fix xs
  have $(\forall X \in \{\}.\ (xs,\ X) \in \mathit{failures}\ P) \longrightarrow (xs,\ \bigcup X \in \{\}.\ X) \in \mathit{failures}\ P$
   using A **by** *blast*
  moreover have $\forall X \in \{\}.\ (xs,\ X) \in \mathit{failures}\ P$ **by** *simp*
  ultimately have $(xs,\ \bigcup X \in \{\}.\ X) \in \mathit{failures}\ P$ **..**
  thus $xs \in \mathit{traces}\ P$ **by** (*rule failures-traces*)
qed

lemma *traces-dom-events*:
  assumes A: $u \in \mathit{range}\ D$
  shows $xs \in \mathit{traces}\ P =$
    $(\mathit{next\text{-}dom\text{-}events}\ P\ D\ u\ xs \cup \mathit{ref\text{-}dom\text{-}events}\ P\ D\ u\ xs \neq \{\})$
    (**is** - = $(?S \neq \{\})$)
proof
  have $\exists x.\ u = D\ x$ **using** A **by** (*simp add*: *image-def*)
  then obtain x **where** B: $u = D\ x$ **..**
  assume $xs \in \mathit{traces}\ P$
  hence $(xs,\ \{\}) \in \mathit{failures}\ P$ **by** (*rule traces-failures*)
  hence $(xs\ @\ [x],\ \{\}) \in \mathit{failures}\ P \vee (xs,\ \{x\}) \in \mathit{failures}\ P$ **by** (*rule process-rule-4*)
  moreover {
    assume $(xs\ @\ [x],\ \{\}) \in \mathit{failures}\ P$
    hence $xs\ @\ [x] \in \mathit{traces}\ P$ **by** (*rule failures-traces*)
    hence $x \in \mathit{next\text{-}dom\text{-}events}\ P\ D\ u\ xs$
     using B **by** (*simp add*: *next-dom-events-def next-events-def*)
    hence $x \in ?S$ **..**
  }
  moreover {
    assume $(xs,\ \{x\}) \in \mathit{failures}\ P$

```
    hence x ∈ ref-dom-events P D u xs
     using B by (simp add: ref-dom-events-def refusals-def)
    hence x ∈ ?S ..
  }
  ultimately have x ∈ ?S ..
  hence ∃ x. x ∈ ?S ..
  thus ?S ≠ {} by (subst ex-in-conv [symmetric])
next
  assume ?S ≠ {}
  hence ∃ x. x ∈ ?S by (subst ex-in-conv)
  then obtain x where x ∈ ?S ..
  moreover {
    assume x ∈ next-dom-events P D u xs
    hence xs @ [x] ∈ traces P by (simp add: next-dom-events-def next-events-def)
    hence xs ∈ traces P by (rule process-rule-2-traces)
  }
  moreover {
    assume x ∈ ref-dom-events P D u xs
    hence (xs, {x}) ∈ failures P by (simp add: ref-dom-events-def refusals-def)
    hence xs ∈ traces P by (rule failures-traces)
  }
  ultimately show xs ∈ traces P ..
qed

lemma fc-traces:
  assumes
    A: future-consistent P D R and
    B: u ∈ range D and
    C: (xs, ys) ∈ R u
  shows (xs ∈ traces P) = (ys ∈ traces P)
proof −
  have ∀ u ∈ 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
   using A by (simp add: future-consistent-def)
  hence ∀ 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
   using B ..
  hence (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
   by blast
  hence 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 C ..
  hence next-dom-events P D u xs ∪ ref-dom-events P D u xs ≠ {} =
    (next-dom-events P D u ys ∪ ref-dom-events P D u ys ≠ {})
   by simp
```

moreover have *xs* ∈ *traces P* =
(*next-dom-events P D u xs* ∪ *ref-dom-events P D u xs* ≠ {})
using *B* **by** (*rule traces-dom-events*)
moreover have *ys* ∈ *traces P* =
(*next-dom-events P D u ys* ∪ *ref-dom-events P D u ys* ≠ {})
using *B* **by** (*rule traces-dom-events*)
ultimately show *?thesis* **by** *simp*
qed

lemma *wfc-traces*:
assumes
A: *weakly-future-consistent P I D R* **and**
B: *u* ∈ *range D* ∩ (−*I*) `` *range D* **and**
C: (*xs*, *ys*) ∈ *R u*
shows (*xs* ∈ *traces P*) = (*ys* ∈ *traces P*)
proof −
have ∀ *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*
using *A* **by** (*simp add*: *weakly-future-consistent-def*)
hence ∀ *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*
using *B* **..**
hence (*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*
by *blast*
hence *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 *C* **..**
hence *next-dom-events P D u xs* ∪ *ref-dom-events P D u xs* ≠ {} =
(*next-dom-events P D u ys* ∪ *ref-dom-events P D u ys* ≠ {})
by *simp*
moreover have *B'*: *u* ∈ *range D* **using** *B* **..**
hence *xs* ∈ *traces P* =
(*next-dom-events P D u xs* ∪ *ref-dom-events P D u xs* ≠ {})
by (*rule traces-dom-events*)
moreover have *ys* ∈ *traces P* =
(*next-dom-events P D u ys* ∪ *ref-dom-events P D u ys* ≠ {})
using *B'* **by** (*rule traces-dom-events*)
ultimately show *?thesis* **by** *simp*
qed

lemma *fc-implies-wfc*:
future-consistent P D R ⟹ *weakly-future-consistent P I D R*
by (*simp only*: *future-consistent-def weakly-future-consistent-def*, *blast*)

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-set* $X) = (\exists x.\ x \in X)$
proof (*rule iffI*, *simp-all add*: *singleton-set-def*, *erule-tac* [!] *exE*, *erule bexE*)
  fix x
  assume $x \in X$
  thus $\exists x.\ x \in X$ **..**
next
  fix x
  assume A: $x \in X$
  have $\{x\} = \{x\}$ **..**
  hence $\exists x' \in X.\ \{x\} = \{x'\}$ **using** A **..**
  thus $\exists Y.\ \exists x' \in X.\ Y = \{x'\}$ **by** (*rule exI*)
qed

lemma *singleton-set-union*:
$(\bigcup Y \in$ *singleton-set* $X.\ Y) = X$
proof (*subst singleton-set-def*, *rule equalityI*, *rule-tac* [!] *subsetI*)
  fix x
  assume A: $x \in (\bigcup Y \in \{Y'.\ \exists x' \in X.\ Y' = \{x'\}\}.\ Y)$
  show $x \in X$
  proof (*rule UN-E* [*OF A*], *simp*)
  qed (*erule bexE*, *simp*)
next
  fix x
  assume A: $x \in X$
  show $x \in (\bigcup Y \in \{Y'.\ \exists x' \in X.\ Y' = \{x'\}\}.\ Y)$
  proof (*rule UN-I* [*of* $\{x\}$])
  qed (*simp-all add*: A)
qed

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*)
proof (*atomize-elim*, *simp-all add*: *split-paired-all*)
qed (*rule rev-cases*, *rule disjI1*, *assumption*, *simp*)
termination by *lexicographic-order*

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* @ $[x]$)
proof (*atomize-elim*, *simp-all add*: *split-paired-all*)
qed (*rule rev-cases*, *rule disjI1*, *assumption*, *simp*)
termination by *lexicographic-order*

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**
$sources\ \text{-}\ \text{-}\ \text{-}\ [] = \{\}\ |$
$sources\ I\ D\ u\ (x\ \#\ xs) =$
$\quad (if\ (D\ x,\ u) \in I \vee (\exists\, v \in sources\ I\ D\ u\ xs.\ (D\ x,\ v) \in I)$
$\quad then\ insert\ (D\ x)\ (sources\ I\ D\ u\ xs)$
$\quad else\ sources\ I\ D\ u\ xs)$

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

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

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

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*
proof (*induction xs rule*: *rev-induct*, *simp-all*, *rule impI*)
qed (*rule subset-insertI2*)

lemma *sinks-aux-single-dom*:
 sinks-aux I D $\{u\}$ *xs* = *insert u* (*sinks I D u xs*)
by (*induction xs rule*: *rev-induct*, *simp-all add*: *insert-commute*)

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)
proof −
 have *sinks-aux I D U* $[x]$ = *sinks-aux I D U* ([] @ $[x]$) **by** *simp*
 thus *?thesis* **by** (*simp only*: *sinks-aux.simps*)
qed

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)
proof (*induction xs rule*: *rev-induct*, *case-tac* [!] $\exists\, v \in U.\ (v, D\ x) \in I$,
 simp-all add: *sinks-aux-single-event del*: *sinks-aux.simps*(*2*))
 fix x' *xs*
 assume *A*: *sinks-aux I D U* (x # *xs*) = *sinks-aux I D* (*insert* (*D x*) *U*) *xs*
  (**is** *?S* = *?S′*)
 show *sinks-aux I D U* (x # *xs* @ $[x']$) =
  sinks-aux I D (*insert* (*D x*) *U*) (*xs* @ $[x']$)
 proof (*cases* $\exists\, v \in$ *?S*. $(v, D\ x') \in I$)
  case *True*
  hence *sinks-aux I D U* ((x # *xs*) @ $[x']$) = *insert* ($D\ x'$) *?S*
   by (*simp only*: *sinks-aux.simps*, *simp*)
  moreover have $\exists\, v \in$ *?S′*. $(v, D\ x') \in I$ **using** *A* **and** *True* **by** *simp*
  hence *sinks-aux I D* (*insert* (*D x*) *U*) (*xs* @ $[x']$) = *insert* ($D\ x'$) *?S′*
   by *simp*
  ultimately show *?thesis* **using** *A* **by** *simp*
 next
  case *False*
  hence *sinks-aux I D U* ((x # *xs*) @ $[x']$) = *?S*
   by (*simp only*: *sinks-aux.simps*, *simp*)
  moreover have $\neg$ ($\exists\, v \in$ *?S′*. $(v, D\ x') \in I$) **using** *A* **and** *False* **by** *simp*
  hence *sinks-aux I D* (*insert* (*D x*) *U*) (*xs* @ $[x']$) = *?S′* **by** *simp*
  ultimately show *?thesis* **using** *A* **by** *simp*
 qed
next
 fix x' *xs*
 assume *A*: *sinks-aux I D U* (x # *xs*) = *sinks-aux I D U xs*
  (**is** *?S* = *?S′*)

show *sinks-aux I D U* (*x* # *xs* @ [*x'*]) = *sinks-aux I D U* (*xs* @ [*x'*])
proof (*cases* ∃ *v* ∈ *?S*. (*v*, *D x'*) ∈ *I*)
case *True*
hence *sinks-aux I D U* ((*x* # *xs*) @ [*x'*]) = *insert* (*D x'*) *?S*
by (*simp only*: *sinks-aux.simps*, *simp*)
moreover have ∃ *v* ∈ *?S'*. (*v*, *D x'*) ∈ *I* **using** *A* **and** *True* **by** *simp*
hence *sinks-aux I D U* (*xs* @ [*x'*]) = *insert* (*D x'*) *?S'* **by** *simp*
ultimately show *?thesis* **using** *A* **by** *simp*
next
case *False*
hence *sinks-aux I D U* ((*x* # *xs*) @ [*x'*]) = *?S*
by (*simp only*: *sinks-aux.simps*, *simp*)
moreover have ¬ (∃ *v* ∈ *?S'*. (*v*, *D x'*) ∈ *I*) **using** *A* **and** *False* **by** *simp*
hence *sinks-aux I D U* (*xs* @ [*x'*]) = *?S'* **by** *simp*
ultimately show *?thesis* **using** *A* **by** *simp*
qed
qed

lemma *ipurge-tr-aux-single-dom*:
ipurge-tr-aux I D {*u*} *xs* = *ipurge-tr I D u xs*
proof (*induction xs rule*: *rev-induct*, *simp*)
fix *x xs*
assume *A*: *ipurge-tr-aux I D* {*u*} *xs* = *ipurge-tr I D u xs*
show *ipurge-tr-aux I D* {*u*} (*xs* @ [*x*]) = *ipurge-tr I D u* (*xs* @ [*x*])
proof (*cases* ∃ *v* ∈ *sinks-aux I D* {*u*} *xs*. (*v*, *D x*) ∈ *I*,
simp-all only: *ipurge-tr-aux.simps if-True if-False*)
case *True*
hence (*u*, *D x*) ∈ *I* ∨ (∃ *v* ∈ *sinks I D u xs*. (*v*, *D x*) ∈ *I*)
by (*simp add*: *sinks-aux-single-dom*)
hence *ipurge-tr I D u* (*xs* @ [*x*]) = *ipurge-tr I D u xs* **by** *simp*
thus *ipurge-tr-aux I D* {*u*} *xs* = *ipurge-tr I D u* (*xs* @ [*x*])
using *A* **by** *simp*
next
case *False*
hence ¬ ((*u*, *D x*) ∈ *I* ∨ (∃ *v* ∈ *sinks I D u xs*. (*v*, *D x*) ∈ *I*))
by (*simp add*: *sinks-aux-single-dom*)
hence *D x* ∉ *sinks I D u* (*xs* @ [*x*])
by (*simp only*: *sinks-interference-eq*, *simp*)
hence *ipurge-tr I D u* (*xs* @ [*x*]) = *ipurge-tr I D u xs* @ [*x*] **by** *simp*
thus *ipurge-tr-aux I D* {*u*} *xs* @ [*x*] = *ipurge-tr I D u* (*xs* @ [*x*])
using *A* **by** *simp*
qed
qed

lemma *ipurge-ref-aux-single-dom*:
ipurge-ref-aux I D {*u*} *xs X* = *ipurge-ref I D u xs X*
by (*simp add*: *ipurge-ref-aux-def ipurge-ref-def sinks-aux-single-dom*)

lemma *ipurge-ref-aux-all* [*rule-format*]:

$(\forall u \in U.\ \neg (\exists v \in D\ `\ (X \cup set\ xs).\ (u,\ v) \in I)) \longrightarrow$
$ipurge\text{-}ref\text{-}aux\ I\ D\ U\ xs\ X = X$
proof (*induction xs*, *simp-all add*: *ipurge-ref-aux-def sinks-aux-cons*)
qed (*rule impI*, *rule equalityI*, *rule-tac* [!] *subsetI*, *simp-all*)

lemma *ipurge-ref-all*:
$\neg (\exists v \in D\ `\ (X \cup set\ xs).\ (u,\ v) \in I) \Longrightarrow ipurge\text{-}ref\ I\ D\ u\ xs\ X = X$
by (*subst ipurge-ref-aux-single-dom* [*symmetric*], *rule ipurge-ref-aux-all*, *simp*)

lemma *unaffected-domains-single-dom*:
$\{x \in X.\ D\ x \in unaffected\text{-}domains\ I\ D\ \{u\}\ xs\} = ipurge\text{-}ref\ I\ D\ u\ xs\ X$
by (*simp add*: *ipurge-ref-def unaffected-domains-def sinks-aux-single-dom*)

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*:
$sources\ I\ D\ u\ xs = sinks\ (I^{-1})\ D\ u\ (rev\ xs)$
by (*induction xs*, *simp-all*)

lemma *sources-sinks-aux*:
$sources\text{-}aux\ I\ D\ U\ xs = sinks\text{-}aux\ (I^{-1})\ D\ U\ (rev\ xs)$
by (*induction xs*, *simp-all*)

lemma *sources-aux-subset*:
$U \subseteq sources\text{-}aux\ I\ D\ U\ xs$
by (*subst sources-sinks-aux*, *rule sinks-aux-subset*)

lemma *sources-aux-append*:
$sources\text{-}aux\ I\ D\ U\ (xs\ @\ ys) = sources\text{-}aux\ I\ D\ (sources\text{-}aux\ I\ D\ U\ ys)\ xs$
by (*induction xs*, *simp-all*)

lemma *sources-aux-append-nil* [*rule-format*]:
$sources\text{-}aux\ I\ D\ U\ ys = U \longrightarrow$
$sources\text{-}aux\ I\ D\ U\ (xs\ @\ ys) = sources\text{-}aux\ I\ D\ U\ xs$
by (*induction xs*, *simp-all*)

lemma *ipurge-tr-rev-aux-append*:
$ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ U\ (xs\ @\ ys) =$
$ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ (sources\text{-}aux\ I\ D\ U\ ys)\ xs\ @\ ipurge\text{-}tr\text{-}rev\text{-}aux\ I\ D\ U\ ys$
by (*induction xs*, *simp-all add*: *sources-aux-append*)

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

by (*induction xs rule*: *rev-induct*, *simp-all add*: *ipurge-tr-rev-aux-append*)

lemma *ipurge-tr-rev-aux-nil-2* [*rule-format*]:
(∀ *u* ∈ *U*. ¬ (∃ *v* ∈ *D* ‘ *set xs*. (*v*, *u*) ∈ *I*)) ⟶ *ipurge-tr-rev-aux I D U xs* = []
by (*induction xs rule*: *rev-induct*, *simp-all add*: *ipurge-tr-rev-aux-append*)

lemma *ipurge-tr-rev-aux-nil*:
(*ipurge-tr-rev-aux I D U xs* = []) = (∀ *u* ∈ *U*. ¬ (∃ *v* ∈ *D* ‘ *set xs*. (*v*, *u*) ∈ *I*))
proof (*rule iffI*, *rule ballI*, *erule ipurge-tr-rev-aux-nil-1*, *assumption*)
qed (*rule ipurge-tr-rev-aux-nil-2*, *erule bspec*)

lemma *ipurge-tr-rev-aux-nil-sources* [*rule-format*]:
ipurge-tr-rev-aux I D U xs = [] ⟶ *sources-aux I D U xs* = *U*
by (*induction xs*, *simp-all*)

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*
by (*induction xs*, *simp-all add*: *ipurge-tr-rev-aux-nil-sources sources-aux-append-nil*)

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 (*induction xs*, *simp*, *rule impI*)
fix *x′ xs*
assume
A: *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*)) **and**
B: *ipurge-tr-rev-aux I D U* (*x′* # *xs*) = *x* # *ws*
show ∃ *ys zs*. *x′* # *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 (*cases* ∃ *v* ∈ *sources-aux I D U xs*. (*D x′*, *v*) ∈ *I*)
case *True*
then have *x′* = *x* **using** *B* **by** *simp*
with *True* **have** *x′* # *xs* = *x* # *xs* ∧
ipurge-tr-rev-aux I D (*sources-aux I D U* (*x* # *xs*)) [] = [] ∧
(∃ *v* ∈ *sources-aux I D U xs*. (*D x*, *v*) ∈ *I*)
by *simp*
thus *?thesis* **by** *blast*
next
case *False*
hence *ipurge-tr-rev-aux I D U xs* = *x* # *ws* **using** *B* **by** *simp*
with *A* **have** ∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧
ipurge-tr-rev-aux I D (*sources-aux I D U* (*x* # *zs*)) *ys* = [] ∧

($\exists$ v $\in$ *sources-aux I D U zs*. (D x, v) $\in$ I) **..**
then obtain *ys* **and** *zs* **where** *xs*: $xs = ys$ @ x # zs $\wedge$
ipurge-tr-rev-aux I D (*sources-aux I D U* (x # *zs*)) *ys* = [] $\wedge$
($\exists$ v $\in$ *sources-aux I D U zs*. (D x, v) $\in$ I)
by *blast*
then have
$\neg$ ($\exists$ v $\in$ *sources-aux I D* (*sources-aux I D U* (x # *zs*)) *ys*. (D x', v) $\in$ I)
using *False* **by** (*simp add*: *sources-aux-append*)
hence *ipurge-tr-rev-aux I D* (*sources-aux I D U* (x # *zs*)) (x' # *ys*) =
ipurge-tr-rev-aux I D (*sources-aux I D U* (x # *zs*)) *ys*
by *simp*
with *xs* **have** x' # *xs* = (x' # *ys*) @ x # *zs* $\wedge$
ipurge-tr-rev-aux I D (*sources-aux I D U* (x # *zs*)) (x' # *ys*) = [] $\wedge$
($\exists$ v $\in$ *sources-aux I D U zs*. (D x, v) $\in$ I)
by (*simp del*: *sources-aux.simps*)
thus *?thesis* **by** *blast*
qed
qed

lemma *ipurge-tr-rev-aux-last-1* [*rule-format*]:
ipurge-tr-rev-aux I D U xs = *ws* @ [x] $\longrightarrow$ ($\exists$ v $\in$ U. (D x, v) $\in$ I)
proof (*induction xs rule*: *rev-induct*, *simp*, *rule impI*)
fix *xs* x'
assume
A: *ipurge-tr-rev-aux I D U xs* = *ws* @ [x] $\longrightarrow$ ($\exists$ v $\in$ U. (D x, v) $\in$ I) **and**
B: *ipurge-tr-rev-aux I D U* (*xs* @ [x']) = *ws* @ [x]
show $\exists$ v $\in$ U. (D x, v) $\in$ I
proof (*cases* $\exists$ v $\in$ U. (D x', v) $\in$ I)
case *True*
hence *ipurge-tr-rev-aux I D U* (*xs* @ [x']) =
ipurge-tr-rev-aux I D (*insert* (D x') U) *xs* @ [x']
by (*simp add*: *ipurge-tr-rev-aux-append*)
hence $x' = x$ **using** *B* **by** *simp*
thus *?thesis* **using** *True* **by** *simp*
next
case *False*
hence *ipurge-tr-rev-aux I D U* (*xs* @ [x']) = *ipurge-tr-rev-aux I D U xs*
by (*simp add*: *ipurge-tr-rev-aux-append*)
hence *ipurge-tr-rev-aux I D U xs* = *ws* @ [x] **using** *B* **by** *simp*
with *A* **show** *?thesis* **..**
qed
qed

lemma *ipurge-tr-rev-aux-last-2* [*rule-format*]:
ipurge-tr-rev-aux I D U xs = *ws* @ [x] $\longrightarrow$
($\exists$ *ys zs*. *xs* = *ys* @ x # *zs* $\wedge$ *ipurge-tr-rev-aux I D U zs* = [])
proof (*induction xs rule*: *rev-induct*, *simp*, *rule impI*)
fix *xs* x'
assume

A: *ipurge-tr-rev-aux I D U xs* = *ws* @ [*x*] ⟶
(∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *ipurge-tr-rev-aux I D U zs* = []) **and**
B: *ipurge-tr-rev-aux I D U* (*xs* @ [*x'*]) = *ws* @ [*x*]
show ∃ *ys zs*. *xs* @ [*x'*] = *ys* @ *x* # *zs* ∧ *ipurge-tr-rev-aux I D U zs* = []
proof (*cases* ∃ *v* ∈ *U*. (*D x'*, *v*) ∈ *I*)
case *True*
hence *ipurge-tr-rev-aux I D U* (*xs* @ [*x'*]) =
ipurge-tr-rev-aux I D (*insert* (*D x'*) *U*) *xs* @ [*x'*]
by (*simp add*: *ipurge-tr-rev-aux-append*)
hence *xs* @ [*x'*] = *xs* @ *x* # [] ∧ *ipurge-tr-rev-aux I D U* [] = []
using *B* **by** *simp*
thus *?thesis* **by** *blast*
next
case *False*
hence *ipurge-tr-rev-aux I D U* (*xs* @ [*x'*]) = *ipurge-tr-rev-aux I D U xs*
by (*simp add*: *ipurge-tr-rev-aux-append*)
hence *ipurge-tr-rev-aux I D U xs* = *ws* @ [*x*] **using** *B* **by** *simp*
with *A* **have** ∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *ipurge-tr-rev-aux I D U zs* = [] **..**
then obtain *ys* **and** *zs* **where**
C: *xs* = *ys* @ *x* # *zs* ∧ *ipurge-tr-rev-aux I D U zs* = []
by *blast*
hence *xs* @ [*x'*] = *ys* @ *x* # *zs* @ [*x'*] **by** *simp*
moreover have
ipurge-tr-rev-aux I D U (*zs* @ [*x'*]) = *ipurge-tr-rev-aux I D U zs*
using *False* **by** (*simp add*: *ipurge-tr-rev-aux-append*)
hence *ipurge-tr-rev-aux I D U* (*zs* @ [*x'*]) = [] **using** *C* **by** *simp*
ultimately have *xs* @ [*x'*] = *ys* @ *x* # *zs* @ [*x'*] ∧
ipurge-tr-rev-aux I D U (*zs* @ [*x'*]) = [] **..**
thus *?thesis* **by** *blast*
qed
qed

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 (*induction xs*, *simp*, *rule impI*, *simp*, *erule conjE*)
fix *x xs*
assume ∃ *u* ∈ *U*. (*D x*, *u*) ∈ *I*
then obtain *u* **where** *A*: *u* ∈ *U* **and** *B*: (*D x*, *u*) ∈ *I* **..**
have *U* ⊆ *sources-aux I D U xs* **by** (*rule sources-aux-subset*)
hence *u* ∈ *sources-aux I D U xs* **using** *A* **..**
with *B* **show** ∃ *u* ∈ *sources-aux I D U xs*. (*D x*, *u*) ∈ *I* **..**
qed

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 (induction xs arbitrary: k, simp, rule impI)
  fix w xs k
  assume
    A: ⋀k. offset k x xs ≠ None ⟶ (∃ ys zs. xs = ys @ x # zs) and
    B: offset k x (w # xs) ≠ None
  show ∃ ys zs. w # xs = ys @ x # zs
  proof (cases w = x, simp)
    case True
    hence x # xs = [] @ x # xs by simp
    thus ∃ ys zs. x # xs = ys @ x # zs by blast
  next
    case False
    hence offset k x (w # xs) = offset (Suc k) x xs by simp
    hence offset (Suc k) x xs ≠ None using B by simp
    moreover have offset (Suc k) x xs ≠ None ⟶ (∃ ys zs. xs = ys @ x # zs)
     using A .
    ultimately have ∃ ys zs. xs = ys @ x # zs by simp
    then obtain ys and zs where xs = ys @ x # zs by blast
    hence w # xs = (w # ys) @ x # zs by simp
    thus ∃ ys zs. w # xs = ys @ x # zs by blast
  qed
qed

lemma offset-not-none-2 [rule-format]:
 xs = ys @ x # zs ⟶ offset k x xs ≠ None
proof (induction xs arbitrary: ys k, simp-all del: not-None-eq, rule impI)
  fix w xs ys k
  assume
    A: ⋀ys' k'. xs = ys' @ x # zs ⟶ offset k' x (ys' @ x # zs) ≠ None and
    B: w # xs = ys @ x # zs
  show offset k x (ys @ x # zs) ≠ None
  proof (cases ys, simp-all del: not-None-eq, rule impI)
    fix y' ys'
    have xs = ys' @ x # zs ⟶ offset (Suc k) x (ys' @ x # zs) ≠ None
     using A .
    moreover assume ys = y' # ys'
    hence xs = ys' @ x # zs using B by simp
    ultimately show offset (Suc k) x (ys' @ x # zs) ≠ None ..
  qed
qed

lemma offset-not-none:
 (offset k x xs ≠ None) = (∃ ys zs. xs = ys @ x # zs)
```

by (*rule iffI*, *erule offset-not-none-1*, (*erule exE*)+, *rule offset-not-none-2*)

lemma *offset-addition* [*rule-format*]:
offset k x xs $\neq$ *None* $\longrightarrow$ *offset* (*n* + *m*) *x xs* = *Some* (*the* (*offset n x xs*) + *m*)
proof (*induction xs arbitrary*: *k n*, *simp*, *rule impI*)
fix *w xs k n*
assume
A: $\bigwedge$*k n*. *offset k x xs* $\neq$ *None* $\longrightarrow$
offset (*n* + *m*) *x xs* = *Some* (*the* (*offset n x xs*) + *m*) **and**
B: *offset k x* (*w* # *xs*) $\neq$ *None*
show *offset* (*n* + *m*) *x* (*w* # *xs*) = *Some* (*the* (*offset n x* (*w* # *xs*)) + *m*)
proof (*cases w* = *x*, *simp-all*)
case *False*
hence *offset k x* (*w* # *xs*) = *offset* (*Suc k*) *x xs* **by** *simp*
hence *offset* (*Suc k*) *x xs* $\neq$ *None* **using** *B* **by** *simp*
moreover have *offset* (*Suc k*) *x xs* $\neq$ *None* $\longrightarrow$
offset (*Suc n* + *m*) *x xs* = *Some* (*the* (*offset* (*Suc n*) *x xs*) + *m*)
using *A* **.**
ultimately show *offset* (*Suc* (*n* + *m*)) *x xs* =
Some (*the* (*offset* (*Suc n*) *x xs*) + *m*)
by *simp*
qed
qed

lemma *offset-suc*:
assumes *A*: *offset k x xs* $\neq$ *None*
shows *offset* (*Suc n*) *x xs* = *Some* (*Suc* (*the* (*offset n x xs*)))
proof −
have *offset* (*Suc n*) *x xs* = *offset* (*n* + *Suc 0*) *x xs* **by** *simp*
also have ... = *Some* (*the* (*offset n x xs*) + *Suc 0*) **using** *A* **by** (*rule offset-addition*)
also have ... = *Some* (*Suc* (*the* (*offset n x xs*))) **by** *simp*
finally show *?thesis* **.**
qed

lemma *ipurge-tr-rev-aux-first-offset* [*rule-format*]:
xs = *ys* @ *x* # *zs* $\wedge$ *ipurge-tr-rev-aux I D* (*sources-aux I D U* (*x* # *zs*)) *ys* = [] $\wedge$
($\exists$ *v* $\in$ *sources-aux I D U zs*. (*D x*, *v*) $\in$ *I*) $\longrightarrow$
ys = *take* (*the* (*offset 0 x xs*)) *xs*
proof (*induction xs arbitrary*: *ys*, *simp*, *rule impI*, (*erule conjE*)+)
fix *x′ xs ys*
assume
A: $\bigwedge$*ys*. *xs* = *ys* @ *x* # *zs* $\wedge$
ipurge-tr-rev-aux I D (*sources-aux I D U* (*x* # *zs*)) *ys* = [] $\wedge$
($\exists$ *v* $\in$ *sources-aux I D U zs*. (*D x*, *v*) $\in$ *I*) $\longrightarrow$
ys = *take* (*the* (*offset 0 x xs*)) *xs* **and**
B: *x′* # *xs* = *ys* @ *x* # *zs* **and**
C: *ipurge-tr-rev-aux I D* (*sources-aux I D U* (*x* # *zs*)) *ys* = [] **and**
D: $\exists$ *v* $\in$ *sources-aux I D U zs*. (*D x*, *v*) $\in$ *I*

```
show ys = take (the (offset 0 x (x' # xs))) (x' # xs)
proof (cases ys)
  case Nil
  then have x' = x using B by simp
  with Nil show ?thesis by simp
next
  case (Cons y ys')
  hence E: xs = ys' @ x # zs using B by simp
  moreover have
    F: ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) (y # ys') = []
   using Cons and C by simp
  hence
    G: ¬ (∃ v ∈ sources-aux I D (sources-aux I D U (x # zs)) ys'. (D y, v) ∈ I)
   by (rule-tac notI, simp)
  hence ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys' = []
   using F by simp
  ultimately have 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)
   using D by blast
  with A have H: ys' = take (the (offset 0 x xs)) xs ..
  have I: x' = y using Cons and B by simp
  hence
    J: ¬ (∃ v ∈ sources-aux I D (sources-aux I D U zs) (ys' @ [x]). (D x', v) ∈ I)
   using G by (simp add: sources-aux-append)
  have x' ≠ x
  proof
    assume x' = x
    hence ∃ v ∈ sources-aux I D U zs. (D x', v) ∈ I using D by simp
    then obtain v where K: v ∈ sources-aux I D U zs and L: (D x', v) ∈ I ..
    have sources-aux I D U zs ⊆
      sources-aux I D (sources-aux I D U zs) (ys' @ [x])
     by (rule sources-aux-subset)
    hence v ∈ sources-aux I D (sources-aux I D U zs) (ys' @ [x]) using K ..
    with L have
     ∃ v ∈ sources-aux I D (sources-aux I D U zs) (ys' @ [x]). (D x', v) ∈ I ..
    thus False using J by contradiction
  qed
  hence offset 0 x (x' # xs) = offset (Suc 0) x xs by simp
  also have ... = Some (Suc (the (offset 0 x xs)))
  proof –
    have ∃ ys zs. xs = ys @ x # zs using E by blast
    hence offset 0 x xs ≠ None by (simp only: offset-not-none)
    thus ?thesis by (rule offset-suc)
  qed
  finally have take (the (offset 0 x (x' # xs))) (x' # xs) =
    x' # take (the (offset 0 x xs)) xs
   by simp
  thus ?thesis using Cons and H and I by simp
```

  qed
qed

lemma *ipurge-tr-rev-aux-append-nil-2* [*rule-format*]:
 ipurge-tr-rev-aux I D U (*xs* @ *ys*) = *ipurge-tr-rev-aux I D V xs* ⟶
  ipurge-tr-rev-aux I D U ys = []
proof (*induction xs*, *simp*, *simp only*: *append-Cons*, *rule impI*)
  fix *x xs*
  assume
    A: *ipurge-tr-rev-aux I D U* (*xs* @ *ys*) = *ipurge-tr-rev-aux I D V xs* ⟶
      ipurge-tr-rev-aux I D U ys = [] **and**
    B: *ipurge-tr-rev-aux I D U* (*x* # *xs* @ *ys*) = *ipurge-tr-rev-aux I D V* (*x* # *xs*)
  show *ipurge-tr-rev-aux I D U ys* = []
  proof (*cases* ∃ *v* ∈ *sources-aux I D V xs*. (*D x*, *v*) ∈ *I*)
    case *True*
    hence *C*: *ipurge-tr-rev-aux I D U* (*x* # *xs* @ *ys*) =
      x # *ipurge-tr-rev-aux I D V xs*
     using *B* **by** *simp*
    hence ∃ *vs ws*. *x* # *xs* @ *ys* = *vs* @ *x* # *ws* ∧
      ipurge-tr-rev-aux I D (*sources-aux I D U* (*x* # *ws*)) *vs* = [] ∧
      (∃ *v* ∈ *sources-aux I D U ws*. (*D x*, *v*) ∈ *I*)
     by (*rule ipurge-tr-rev-aux-first*)
    then obtain *vs* **and** *ws* **where** ∗: *x* # *xs* @ *ys* = *vs* @ *x* # *ws* ∧
      ipurge-tr-rev-aux I D (*sources-aux I D U* (*x* # *ws*)) *vs* = [] ∧
      (∃ *v* ∈ *sources-aux I D U ws*. (*D x*, *v*) ∈ *I*)
     by *blast*
    then have *vs* = *take* (*the* (*offset 0 x* (*x* # *xs* @ *ys*))) (*x* # *xs* @ *ys*)
      by (*rule ipurge-tr-rev-aux-first-offset*)
    hence *vs* = [] **by** *simp*
    with ∗ **have** ∃ *v* ∈ *sources-aux I D U* (*xs* @ *ys*). (*D x*, *v*) ∈ *I* **by** *simp*
    hence *ipurge-tr-rev-aux I D U* (*xs* @ *ys*) = *ipurge-tr-rev-aux I D V xs*
     using *C* **by** *simp*
    with *A* **show** *?thesis* **..**
  next
    case *False*
    moreover have ¬ (∃ *v* ∈ *sources-aux I D U* (*xs* @ *ys*). (*D x*, *v*) ∈ *I*)
    proof
      assume ∃ *v* ∈ *sources-aux I D U* (*xs* @ *ys*). (*D x*, *v*) ∈ *I*
      hence *ipurge-tr-rev-aux I D V* (*x* # *xs*) =
        x # *ipurge-tr-rev-aux I D U* (*xs* @ *ys*)
       using *B* **by** *simp*
      hence ∃ *vs ws*. *x* # *xs* = *vs* @ *x* # *ws* ∧
        ipurge-tr-rev-aux I D (*sources-aux I D V* (*x* # *ws*)) *vs* = [] ∧
        (∃ *v* ∈ *sources-aux I D V ws*. (*D x*, *v*) ∈ *I*)
       by (*rule ipurge-tr-rev-aux-first*)
      then obtain *vs* **and** *ws* **where** ∗: *x* # *xs* = *vs* @ *x* # *ws* ∧
        ipurge-tr-rev-aux I D (*sources-aux I D V* (*x* # *ws*)) *vs* = [] ∧
        (∃ *v* ∈ *sources-aux I D V ws*. (*D x*, *v*) ∈ *I*)
       by *blast*

then have *vs* = *take* (*the* (*offset 0 x* (*x* # *xs*))) (*x* # *xs*)
by (*rule ipurge-tr-rev-aux-first-offset*)
hence *vs* = [] **by** *simp*
with ∗ **have** ∃ *v* ∈ *sources-aux I D V xs*. (*D x*, *v*) ∈ *I* **by** *simp*
thus *False* **using** *False* **by** *contradiction*
qed
ultimately have *ipurge-tr-rev-aux I D U* (*xs* @ *ys*) =
ipurge-tr-rev-aux I D V xs
using *B* **by** *simp*
with *A* **show** *?thesis* **..**
qed
qed

lemma *ipurge-tr-rev-aux-append-nil*:
(*ipurge-tr-rev-aux I D U* (*xs* @ *ys*) = *ipurge-tr-rev-aux I D U xs*) =
(*ipurge-tr-rev-aux I D U ys* = [])
by (*rule iffI*, *erule ipurge-tr-rev-aux-append-nil-2*, *rule ipurge-tr-rev-aux-append-nil-1*)

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 ≅ {*ipurge-tr-rev I D u xs*, *rev* (*ipurge-tr* (I^{-1}) *D u* (*rev xs*)),
λ*y ys*. *D y* ∈ *sinks* (I^{-1}) *D u* (*rev* (*y* # *ys*))}
proof (*induction xs*, *simp*, *simp only*: *rev.simps*)
fix *x xs*
assume *A*: *xs* ≅ {*ipurge-tr-rev I D u xs*, *rev* (*ipurge-tr* (I^{-1}) *D u* (*rev xs*)),
λ*y ys*. *D y* ∈ *sinks* (I^{-1}) *D u* (*rev ys* @ [*y*])}
(**is** - ≅ {*?ys*, *?zs*, *?P*})
show *x* # *xs* ≅
{*ipurge-tr-rev I D u* (*x* # *xs*), *rev* (*ipurge-tr* (I^{-1}) *D u* (*rev xs* @ [*x*])), *?P*}
proof (*cases ?P x xs*, *simp-all add*: *sources-sinks del*: *sinks.simps*)
case *True*
thus *x* # *xs* ≅ {*x* # *?ys*, *?zs*, *?P*} **using** *A* **by** (*cases ?zs*, *simp-all*)
next
case *False*
thus *x* # *xs* ≅ {*?ys*, *x* # *?zs*, *?P*} **using** *A* **by** (*cases ?ys*, *simp-all*)
qed
qed

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

λy ys. ∃ v ∈ sinks-aux (I^{-1}) *D U* (*rev ys*). (*D y*, *v*) ∈ *I*}
proof (*induction xs*, *simp*, *simp only*: *rev.simps*)
fix *x xs*
assume *A*: *xs* ≅ {*ipurge-tr-rev-aux I D U xs*,
rev (*ipurge-tr-aux* (I^{-1}) *D U* (*rev xs*)),
λy ys. ∃ v ∈ sinks-aux (I^{-1}) *D U* (*rev ys*). (*D y*, *v*) ∈ *I*}
(**is** - ≅ {*?ys*, *?zs*, *?P*})
show *x # xs* ≅
{*ipurge-tr-rev-aux I D U* (*x # xs*),
rev (*ipurge-tr-aux* (I^{-1}) *D U* (*rev xs* @ [*x*])), *?P*}
proof (*cases ?P x xs*, *simp-all* (*no-asm-simp*) *add*: *sources-sinks-aux*)
case *True*
thus *x # xs* ≅ {*x # ?ys*, *?zs*, *?P*} **using** *A* **by** (*cases ?zs*, *simp-all*)
next
case *False*
thus *x # xs* ≅ {*?ys*, *x # ?zs*, *?P*} **using** *A* **by** (*cases ?ys*, *simp-all*)
qed
qed

lemma *ipurge-tr-aux-all*:
(*ipurge-tr-aux I D U xs* = *xs*) = (∀ *u* ∈ *U*. ¬ (∃ *v* ∈ *D* ‘ *set xs*. (*u*, *v*) ∈ *I*))
proof −
have *A*: *rev xs* ≅ {*ipurge-tr-rev-aux* (I^{-1}) *D U* (*rev xs*),
rev (*ipurge-tr-aux* ($(I^{-1})^{-1}$) *D U* (*rev* (*rev xs*))),
λy ys. ∃ v ∈ sinks-aux ($(I^{-1})^{-1}$) *D U* (*rev ys*). (*D y*, *v*) ∈ (I^{-1})}
(**is** - ≅ {-, -, *?P*})
by (*rule Interleaves-ipurge-tr-aux*)
show *?thesis*
proof
assume *ipurge-tr-aux I D U xs* = *xs*
hence *rev xs* ≅ {*ipurge-tr-rev-aux* (I^{-1}) *D U* (*rev xs*), *rev xs*, *?P*}
using *A* **by** *simp*
hence *rev xs* ≃ {*ipurge-tr-rev-aux* (I^{-1}) *D U* (*rev xs*), *rev xs*, *?P*}
by (*rule Interleaves-interleaves*)
moreover have *rev xs* ≃ {[], *rev xs*, *?P*} **by** (*rule interleaves-nil-all*)
ultimately have *ipurge-tr-rev-aux* (I^{-1}) *D U* (*rev xs*) = []
by (*rule interleaves-equal-fst*)
thus ∀ *u* ∈ *U*. ¬ (∃ *v* ∈ *D* ‘ *set xs*. (*u*, *v*) ∈ *I*)
by (*simp add*: *ipurge-tr-rev-aux-nil*)
next
assume ∀ *u* ∈ *U*. ¬ (∃ *v* ∈ *D* ‘ *set xs*. (*u*, *v*) ∈ *I*)
hence *ipurge-tr-rev-aux* (I^{-1}) *D U* (*rev xs*) = []
by (*simp add*: *ipurge-tr-rev-aux-nil*)
hence *rev xs* ≅ {[], *rev* (*ipurge-tr-aux I D U xs*), *?P*} **using** *A* **by** *simp*
hence *rev xs* ≃ {[], *rev* (*ipurge-tr-aux I D U xs*), *?P*}
by (*rule Interleaves-interleaves*)
hence *rev xs* ≃ {*rev* (*ipurge-tr-aux I D U xs*), [], *λw ws.* ¬ *?P w ws*}
by (*subst* (*asm*) *interleaves-swap*)
moreover have *rev xs* ≃ {*rev xs*, [], *λw ws.* ¬ *?P w ws*}

by (*rule interleaves-all-nil*)
ultimately have *rev* (*ipurge-tr-aux I D U xs*) = *rev xs*
by (*rule interleaves-equal-fst*)
thus *ipurge-tr-aux I D U xs* = *xs* **by** *simp*
qed
qed

lemma *ipurge-tr-rev-aux-single-dom*:
ipurge-tr-rev-aux I D {*u*} *xs* = *ipurge-tr-rev I D u xs* (**is** *?ys* = *?ys'*)
proof −
have *xs* ≅ {*?ys*, *rev* (*ipurge-tr-aux* (I^{-1}) *D* {*u*} (*rev xs*)),
λ*y ys*. ∃ *v* ∈ *sinks-aux* (I^{-1}) *D* {*u*} (*rev ys*). (*D y*, *v*) ∈ *I*}
by (*rule Interleaves-ipurge-tr-aux*)
hence *xs* ≅ {*?ys*, *rev* (*ipurge-tr* (I^{-1}) *D u* (*rev xs*)),
λ*y ys*. (*u*, *D y*) ∈ I^{-1} ∨ (∃ *v* ∈ *sinks* (I^{-1}) *D u* (*rev ys*). (*v*, *D y*) ∈ I^{-1})}
by (*simp add*: *ipurge-tr-aux-single-dom sinks-aux-single-dom*)
hence *xs* ≅ {*?ys*, *rev* (*ipurge-tr* (I^{-1}) *D u* (*rev xs*)),
λ*y ys*. *D y* ∈ *sinks* (I^{-1}) *D u* (*rev* (*y* # *ys*))}
(**is** - ≅ {-, *?zs*, *?P*})
by (*simp only*: *sinks-interference-eq*, *simp*)
moreover have *xs* ≅ {*?ys'*, *?zs*, *?P*} **by** (*rule Interleaves-ipurge-tr*)
ultimately show *?thesis* **by** (*rule Interleaves-equal-fst*)
qed

lemma *ipurge-tr-all*:
(*ipurge-tr I D u xs* = *xs*) = (¬ (∃ *v* ∈ *D* ‘ *set xs*. (*u*, *v*) ∈ *I*))
by (*subst ipurge-tr-aux-single-dom* [*symmetric*], *simp add*: *ipurge-tr-aux-all*)

lemma *ipurge-tr-rev-all*:
∀ *v* ∈ *D* ‘ *set xs*. (*v*, *u*) ∈ *I* ⟹ *ipurge-tr-rev I D u xs* = *xs*
proof (*subst ipurge-tr-rev-aux-single-dom* [*symmetric*], *rule ipurge-tr-rev-aux-all*)
qed (*simp* (*no-asm-simp*))

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 ⇒ ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ ('a, 'd) dom-rel-map **where**
rel-ipurge P I D u ≡ {(xs, ys). xs ∈ traces P ∧ ys ∈ traces P ∧
ipurge-tr-rev I D u xs = ipurge-tr-rev I D u ys}

definition *rel-ipurge-aux* ::
'a process ⇒ ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ ('a, 'd) domset-rel-map **where**
rel-ipurge-aux P I D U ≡ {(xs, ys). xs ∈ traces P ∧ ys ∈ traces P ∧
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
by (*simp add*: *rel-ipurge-def rel-ipurge-aux-def ipurge-tr-rev-aux-single-dom*)

lemma *view-partition-rel-ipurge*:
view-partition P D (rel-ipurge P I D)
proof (*subst view-partition-def*, *rule ballI*, *rule equivI*)
fix *u*
show *rel-ipurge P I D u ⊆ traces P × traces P*
by (*rule subsetI*) (*simp add*: *rel-ipurge-def split-paired-all*)
next
fix *u*
show *refl-on (traces P) (rel-ipurge P I D u)*
by (*rule refl-onI*) (*simp add*: *rel-ipurge-def*)
next
fix *u*
show *sym (rel-ipurge P I D u)*
by (*rule symI*, *simp add*: *rel-ipurge-def*)
next
fix *u*
show *trans (rel-ipurge P I D u)*
by (*rule transI*, *simp add*: *rel-ipurge-def*)
qed

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 (*rule iffI*, *erule fc-implies-wfc*,
simp only: *future-consistent-def weakly-future-consistent-def*,
rule ballI, (*rule allI*)+, *rule impI*)
fix *u xs ys*
assume
A: *∀ u ∈ range D ∩ (−I) ‘‘ range D. ∀ xs ys. (xs, ys) ∈ rel-ipurge P I D 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 **and**
B: *u ∈ range D* **and**
C: (*xs*, *ys*) ∈ *rel-ipurge P I D u*
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
proof (*cases u ∈ range D ∩ (−I) ‘‘ range D*)
case *True*
with *A* **have** ∀ *xs ys*. (*xs*, *ys*) ∈ *rel-ipurge P I D 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 **..**
hence (*xs*, *ys*) ∈ *rel-ipurge P I D 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
by *blast*
thus *?thesis* **using** *C* **..**
next
case *False*
hence *D*: *u ∉ (−I) ‘‘ range D* **using** *B* **by** *simp*
have *ipurge-tr-rev I D u xs = ipurge-tr-rev I D u ys*
using *C* **by** (*simp add*: *rel-ipurge-def*)
moreover have ∀ *zs*. *ipurge-tr-rev I D u zs = zs*
proof (*rule allI*, *rule ipurge-tr-rev-all*, *rule ballI*, *erule imageE*, *rule ccontr*)
fix *v x*
assume (*v*, *u*) ∉ *I*
hence (*v*, *u*) ∈ −*I* **by** *simp*
moreover assume *v = D x*
hence *v ∈ range D* **by** *simp*
ultimately have *u ∈ (−I) ‘‘ range D* **..**
thus *False* **using** *D* **by** *contradiction*
qed
ultimately show *?thesis* **by** *simp*
qed
qed

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 (*induction zs arbitrary*: *U rule*: *rev-induct*, *rule-tac* [!] *impI*, *simp*)
  fix *U*
  assume *A*: *U* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) []
  have ∀ *u* ∈ *U*. ∀ *v* ∈ *D* ‘ *set ys*. (*v*, *u*) ∉ *I*
  proof
    fix *u*
    assume *u* ∈ *U*
    with *A* **have** *u* ∈ *unaffected-domains I D* (*D* ‘ *set ys*) [] **..**
    thus ∀ *v* ∈ *D* ‘ *set ys*. (*v*, *u*) ∉ *I* **by** (*simp add*: *unaffected-domains-def*)
  qed
  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'* = *insert* (*D z*) *U*
  assume
    A: ⋀*U*. *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*) **and**
    B: *U* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) (*zs* @ [*z*])
  have *C*: *U* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) *zs*
  proof
    fix *u*
    assume *u* ∈ *U*
    with *B* **have** *u* ∈ *unaffected-domains I D* (*D* ‘ *set ys*) (*zs* @ [*z*]) **..**
    thus *u* ∈ *unaffected-domains I D* (*D* ‘ *set ys*) *zs*
     by (*simp add*: *unaffected-domains-def*)
  qed
  have *D*: *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 −

have $U \subseteq$ *unaffected-domains I D* $(D$ ‘ *set ys*$)$ *zs* $\longrightarrow$
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*)
using *A* **.**
thus *?thesis* **using** *C* **..**
qed
have *E*: ¬ (∃ *v* ∈ *sinks-aux I D* (*D* ‘ *set ys*) *zs*. (*v*, *D z*) ∈ *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*)
(**is** *?P* ⟶ *?Q*)
proof
assume *?P*
have *?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*)
using *A* **.**
moreover have *?U′* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) *zs*
by (*simp add*: *C*, *simp add*: *unaffected-domains-def* ‹*?P*› [*simplified*])
ultimately show *?Q* **..**
qed
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 (*cases* ∃ *v* ∈ *sinks-aux I D* (*D* ‘ *set ys*) *zs*. (*v*, *D z*) ∈ *I*,
simp-all (*no-asm-simp*))
case *True*
have ¬ (∃ *u* ∈ *U*. (*D z*, *u*) ∈ *I*)
proof
assume ∃ *u* ∈ *U*. (*D z*, *u*) ∈ *I*
then obtain *u* **where** *F*: *u* ∈ *U* **and** *G*: (*D z*, *u*) ∈ *I* **..**
have *D z* ∈ *sinks-aux I D* (*D* ‘ *set ys*) (*zs* @ [*z*]) **using** *True* **by** *simp*
with *G* **have** ∃ *v* ∈ *sinks-aux I D* (*D* ‘ *set ys*) (*zs* @ [*z*]). (*v*, *u*) ∈ *I* **..**
moreover have *u* ∈ *unaffected-domains I D* (*D* ‘ *set ys*) (*zs* @ [*z*])
using *B* **and** *F* **..**
hence ¬ (∃ *v* ∈ *sinks-aux I D* (*D* ‘ *set ys*) (*zs* @ [*z*]). (*v*, *u*) ∈ *I*)
by (*simp add*: *unaffected-domains-def*)
ultimately show *False* **by** *contradiction*
qed
hence *ipurge-tr-rev-aux I D U* ((*xs* @ *ys* @ *zs*) @ [*z*]) =
ipurge-tr-rev-aux I D U (*xs* @ *ys* @ *zs*)
by (*subst ipurge-tr-rev-aux-append*, *simp*)
also have … = *ipurge-tr-rev-aux I D U*
(*xs* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*)
using *D* **.**
finally 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*)
by *simp*
next
case *False*
note *F* = *this*

```
    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 (cases ∃ u ∈ U. (D z, u) ∈ I)
      case True
      hence ipurge-tr-rev-aux I D U ((xs @ ys @ zs) @ [z]) =
        ipurge-tr-rev-aux I D ?U' (xs @ ys @ zs) @ [z]
       by (subst ipurge-tr-rev-aux-append, simp)
      also have ... =
        ipurge-tr-rev-aux I D ?U' (xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ [z]
       using E and F by simp
      also have ... =
        ipurge-tr-rev-aux I D U ((xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ [z])
       using True by (subst ipurge-tr-rev-aux-append, simp)
      finally show ?thesis by simp
    next
      case False
      hence ipurge-tr-rev-aux I D U ((xs @ ys @ zs) @ [z]) =
        ipurge-tr-rev-aux I D U (xs @ ys @ zs)
       by (subst ipurge-tr-rev-aux-append, simp)
      also have ... =
        ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ‘ set ys) zs)
       using D .
      also have ... =
        ipurge-tr-rev-aux I D U ((xs @ ipurge-tr-aux I D (D ‘ set ys) zs) @ [z])
       using False by (subst ipurge-tr-rev-aux-append, simp)
      finally show ?thesis by simp
    qed
  qed
qed

lemma ipurge-tr-rev-ipurge-tr-aux-2 [rule-format]:
 U ⊆ unaffected-domains I D (D ‘ set ys) zs ⟶
  ipurge-tr-rev-aux I D U (xs @ zs) =
  ipurge-tr-rev-aux I D U (xs @ ys @ ipurge-tr-aux I D (D ‘ set ys) zs)
proof (induction zs arbitrary: U rule: rev-induct, rule-tac [!] impI, simp)
  fix U
  assume A: U ⊆ unaffected-domains I D (D ‘ set ys) []
  have ∀ u ∈ U. ∀ v ∈ D ‘ set ys. (v, u) ∉ I
  proof
    fix u
    assume u ∈ U
    with A have u ∈ unaffected-domains I D (D ‘ set ys) [] ..
    thus ∀ v ∈ D ‘ set ys. (v, u) ∉ I by (simp add: unaffected-domains-def)
  qed
  hence ipurge-tr-rev-aux I D U ys = [] by (simp add: ipurge-tr-rev-aux-nil)
  hence 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)
  thus ipurge-tr-rev-aux I D U xs = ipurge-tr-rev-aux I D U (xs @ ys) ..
next
```

fix *z zs U*
let *?U′* = *insert* (*D z*) *U*
assume
A: ⋀*U*. *U* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) *zs* ⟶
ipurge-tr-rev-aux I D U (*xs* @ *zs*) =
ipurge-tr-rev-aux I D U (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*) **and**
B: *U* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) (*zs* @ [*z*])
have *C*: *U* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) *zs*
proof
fix *u*
assume *u* ∈ *U*
with *B* **have** *u* ∈ *unaffected-domains I D* (*D* ‘ *set ys*) (*zs* @ [*z*]) **..**
thus *u* ∈ *unaffected-domains I D* (*D* ‘ *set ys*) *zs*
by (*simp add*: *unaffected-domains-def*)
qed
have *D*: *ipurge-tr-rev-aux I D U* (*xs* @ *zs*) =
ipurge-tr-rev-aux I D U (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*)
proof −
have *U* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) *zs* ⟶
ipurge-tr-rev-aux I D U (*xs* @ *zs*) =
ipurge-tr-rev-aux I D U (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*)
using *A* **.**
thus *?thesis* **using** *C* **..**
qed
have *E*: ¬ (∃ *v* ∈ *sinks-aux I D* (*D* ‘ *set ys*) *zs*. (*v*, *D z*) ∈ *I*) ⟶
ipurge-tr-rev-aux I D ?U′ (*xs* @ *zs*) =
ipurge-tr-rev-aux I D ?U′ (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*)
(**is** *?P* ⟶ *?Q*)
proof
assume *?P*
have *?U′* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) *zs* ⟶
ipurge-tr-rev-aux I D ?U′ (*xs* @ *zs*) =
ipurge-tr-rev-aux I D ?U′ (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*)
using *A* **.**
moreover have *?U′* ⊆ *unaffected-domains I D* (*D* ‘ *set ys*) *zs*
by (*simp add*: *C*, *simp add*: *unaffected-domains-def* ‹*?P*› [*simplified*])
ultimately show *?Q* **..**
qed
show *ipurge-tr-rev-aux I D U* (*xs* @ *zs* @ [*z*]) =
ipurge-tr-rev-aux I D U (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) (*zs* @ [*z*]))
proof (*cases* ∃ *v* ∈ *sinks-aux I D* (*D* ‘ *set ys*) *zs*. (*v*, *D z*) ∈ *I*,
simp-all (*no-asm-simp*))
case *True*
have ¬ (∃ *u* ∈ *U*. (*D z*, *u*) ∈ *I*)
proof
assume ∃ *u* ∈ *U*. (*D z*, *u*) ∈ *I*
then obtain *u* **where** *F*: *u* ∈ *U* **and** *G*: (*D z*, *u*) ∈ *I* **..**
have *D z* ∈ *sinks-aux I D* (*D* ‘ *set ys*) (*zs* @ [*z*]) **using** *True* **by** *simp*
with *G* **have** ∃ *v* ∈ *sinks-aux I D* (*D* ‘ *set ys*) (*zs* @ [*z*]). (*v*, *u*) ∈ *I* **..**

moreover have *u* ∈ *unaffected-domains I D* (*D* ‘ *set ys*) (*zs* @ [*z*])
using *B* **and** *F* **..**
hence ¬ (∃ *v* ∈ *sinks-aux I D* (*D* ‘ *set ys*) (*zs* @ [*z*]). (*v*, *u*) ∈ *I*)
by (*simp add*: *unaffected-domains-def*)
ultimately show *False* **by** *contradiction*
qed
hence *ipurge-tr-rev-aux I D U* ((*xs* @ *zs*) @ [*z*]) =
ipurge-tr-rev-aux I D U (*xs* @ *zs*)
by (*subst ipurge-tr-rev-aux-append*, *simp*)
also have
... = *ipurge-tr-rev-aux I D U* (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*)
using *D* **.**
finally show *ipurge-tr-rev-aux I D U* (*xs* @ *zs* @ [*z*]) =
ipurge-tr-rev-aux I D U (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*)
by *simp*
next
case *False*
note *F* = *this*
show *ipurge-tr-rev-aux I D U* (*xs* @ *zs* @ [*z*]) =
ipurge-tr-rev-aux I D U (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs* @ [*z*])
proof (*cases* ∃ *u* ∈ *U*. (*D z*, *u*) ∈ *I*)
case *True*
hence *ipurge-tr-rev-aux I D U* ((*xs* @ *zs*) @ [*z*]) =
ipurge-tr-rev-aux I D ?U′ (*xs* @ *zs*) @ [*z*]
by (*subst ipurge-tr-rev-aux-append*, *simp*)
also have ... =
ipurge-tr-rev-aux I D ?U′
(*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*) @ [*z*]
using *E* **and** *F* **by** *simp*
also have ... =
ipurge-tr-rev-aux I D U
((*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*) @ [*z*])
using *True* **by** (*subst ipurge-tr-rev-aux-append*, *simp*)
finally show *?thesis* **by** *simp*
next
case *False*
hence *ipurge-tr-rev-aux I D U* ((*xs* @ *zs*) @ [*z*]) =
ipurge-tr-rev-aux I D U (*xs* @ *zs*)
by (*subst ipurge-tr-rev-aux-append*, *simp*)
also have ... =
ipurge-tr-rev-aux I D U (*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*)
using *D* **.**
also have ... =
ipurge-tr-rev-aux I D U
((*xs* @ *ys* @ *ipurge-tr-aux I D* (*D* ‘ *set ys*) *zs*) @ [*z*])
using *False* **by** (*subst ipurge-tr-rev-aux-append*, *simp*)
finally show *?thesis* **by** *simp*
qed
qed

qed

lemma *ipurge-tr-rev-ipurge-tr-1*:
  assumes *A*: *u* ∈ *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-tr I D* (*D y*) *zs*)
proof −
  have *ipurge-tr-rev I D u* (*xs* @ *y* # *zs*) =
    ipurge-tr-rev-aux I D {*u*} (*xs* @ [*y*] @ *zs*)
   by (*simp add*: *ipurge-tr-rev-aux-single-dom*)
  also have ... = *ipurge-tr-rev-aux I D* {*u*}
    (*xs* @ *ipurge-tr-aux I D* (*D* ‘ *set* [*y*]) *zs*)
   by (*rule ipurge-tr-rev-ipurge-tr-aux-1*, *simp add*: *A*)
  also have ... = *ipurge-tr-rev I D u* (*xs* @ *ipurge-tr I D* (*D y*) *zs*)
   by (*simp add*: *ipurge-tr-aux-single-dom ipurge-tr-rev-aux-single-dom*)
  finally show *?thesis* **.**
qed

lemma *ipurge-tr-rev-ipurge-tr-2*:
  assumes *A*: *u* ∈ *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-tr I D* (*D y*) *zs*)
proof −
  have *ipurge-tr-rev I D u* (*xs* @ *zs*) = *ipurge-tr-rev-aux I D* {*u*} (*xs* @ *zs*)
   by (*simp add*: *ipurge-tr-rev-aux-single-dom*)
  also have
   ... = *ipurge-tr-rev-aux I D* {*u*} (*xs* @ [*y*] @ *ipurge-tr-aux I D* (*D* ‘ *set* [*y*]) *zs*)
   by (*rule ipurge-tr-rev-ipurge-tr-aux-2*, *simp add*: *A*)
  also have ... = *ipurge-tr-rev I D u* (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*)
   by (*simp add*: *ipurge-tr-aux-single-dom ipurge-tr-rev-aux-single-dom*)
  finally show *?thesis* **.**
qed

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*) ∈ *failures P* **and**
    B: *xs* @ *ipurge-tr I D* (*D y*) *ys* ∈ *traces P* **and**
    C: ∃ *y*′. *y*′ ∈ *ipurge-ref I D* (*D y*) *ys Y*
  shows (*xs* @ *ipurge-tr I D* (*D y*) *ys*, *ipurge-ref I D* (*D y*) *ys Y*) ∈ *failures P*
proof −
  let *?A* = *singleton-set* (*ipurge-ref I D* (*D y*) *ys Y*)
  have (∃ *X*. *X* ∈ *?A*) ⟶
    (∀ *X* ∈ *?A*. (*xs* @ *ipurge-tr I D* (*D y*) *ys*, *X*) ∈ *failures P*) ⟶
    (*xs* @ *ipurge-tr I D* (*D y*) *ys*, ⋃ *X* ∈ *?A*. *X*) ∈ *failures P*
   using *RUC* **by** (*simp add*: *ref-union-closed-def*)
  moreover obtain *y*′ **where** *D*: *y*′ ∈ *ipurge-ref I D* (*D y*) *ys Y* **using** *C* **..**
  hence ∃ *X*. *X* ∈ *?A* **by** (*simp add*: *singleton-set-some*, *rule exI*)

ultimately have ($\forall X \in$ *?A*. (*xs* @ *ipurge-tr I D* (*D y*) *ys*, *X*) $\in$ *failures P*) $\longrightarrow$
(*xs* @ *ipurge-tr I D* (*D y*) *ys*, $\bigcup X \in$ *?A*. *X*) $\in$ *failures P* **..**
moreover have $\forall X \in$ *?A*. (*xs* @ *ipurge-tr I D* (*D y*) *ys*, *X*) $\in$ *failures P*
proof (*rule ballI*, *simp add*: *singleton-set-def*, *erule bexE*, *simp*)
fix y'
have $\forall u \in$ *range D* $\cap$ $(-I)$ `` *range D*.
$\forall$ *xs ys*. (*xs*, *ys*) $\in$ *rel-ipurge P I D u* $\longrightarrow$
ref-dom-events P D u xs = *ref-dom-events P D u ys*
using *IU* **by** (*simp add*: *weakly-future-consistent-def*)
moreover assume *E*: $y' \in$ *ipurge-ref I D* (*D y*) *ys Y*
hence (*D y*, *D* y') $\notin$ *I* **by** (*simp add*: *ipurge-ref-def*)
hence *D* $y' \in$ *range D* $\cap$ $(-I)$ `` *range D* **by** (*simp add*: *Image-iff*, *rule exI*)
ultimately have $\forall$ *xs ys*. (*xs*, *ys*) $\in$ *rel-ipurge P I D* (*D* y') $\longrightarrow$
ref-dom-events P D (*D* y') *xs* = *ref-dom-events P D* (*D* y') *ys* **..**
hence
F: (*xs* @ *y* # *ys*, *xs* @ *ipurge-tr I D* (*D y*) *ys*) $\in$ *rel-ipurge P I D* (*D* y') $\longrightarrow$
ref-dom-events P D (*D* y') (*xs* @ *y* # *ys*) =
ref-dom-events P D (*D* y') (*xs* @ *ipurge-tr I D* (*D y*) *ys*)
by *blast*
have $y' \in \{x \in Y.$ *D x* $\in$ *unaffected-domains I D* {*D y*} *ys*}
using *E* **by** (*simp add*: *unaffected-domains-single-dom*)
hence *D* $y' \in$ *unaffected-domains I D* {*D y*} *ys* **by** *simp*
hence *ipurge-tr-rev I D* (*D* y') (*xs* @ *y* # *ys*) =
ipurge-tr-rev I D (*D* y') (*xs* @ *ipurge-tr I D* (*D y*) *ys*)
by (*rule ipurge-tr-rev-ipurge-tr-1*)
moreover have *xs* @ *y* # *ys* $\in$ *traces P* **using** *A* **by** (*rule failures-traces*)
ultimately have
(*xs* @ *y* # *ys*, *xs* @ *ipurge-tr I D* (*D y*) *ys*) $\in$ *rel-ipurge P I D* (*D* y')
using *B* **by** (*simp add*: *rel-ipurge-def*)
with *F* **have** *ref-dom-events P D* (*D* y') (*xs* @ *y* # *ys*) =
ref-dom-events P D (*D* y') (*xs* @ *ipurge-tr I D* (*D y*) *ys*) **..**
moreover have $y' \in$ *ref-dom-events P D* (*D* y') (*xs* @ *y* # *ys*)
proof (*simp add*: *ref-dom-events-def refusals-def*)
have $\{y'\} \subseteq Y$ **using** *E* **by** (*simp add*: *ipurge-ref-def*)
with *A* **show** (*xs* @ *y* # *ys*, $\{y'\}$) $\in$ *failures P* **by** (*rule process-rule-3*)
qed
ultimately have $y' \in$ *ref-dom-events P D* (*D* y')
(*xs* @ *ipurge-tr I D* (*D y*) *ys*)
by *simp*
thus (*xs* @ *ipurge-tr I D* (*D y*) *ys*, $\{y'\}$) $\in$ *failures P*
by (*simp add*: *ref-dom-events-def refusals-def*)
qed
ultimately have (*xs* @ *ipurge-tr I D* (*D y*) *ys*, $\bigcup X \in$ *?A*. *X*) $\in$ *failures P* **..**
thus *?thesis* **by** (*simp only*: *singleton-set-union*)
qed

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*) ∈ *failures P* **and**
B: *xs* @ *y* # *ipurge-tr I D* (*D y*) *zs* ∈ *traces P* **and**
C: ∃ *z'*. *z'* ∈ *ipurge-ref I D* (*D y*) *zs Z*
shows (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs Z*) ∈ *failures P*
proof −
let *?A* = *singleton-set* (*ipurge-ref I D* (*D y*) *zs Z*)
have (∃ *X*. *X* ∈ *?A*) ⟶
(∀ *X* ∈ *?A*. (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *X*) ∈ *failures P*) ⟶
(*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, ⋃ *X* ∈ *?A*. *X*) ∈ *failures P*
using *RUC* **by** (*simp add*: *ref-union-closed-def*)
moreover obtain *z'* **where** *D*: *z'* ∈ *ipurge-ref I D* (*D y*) *zs Z* **using** *C* **..**
hence ∃ *X*. *X* ∈ *?A* **by** (*simp add*: *singleton-set-some*, *rule exI*)
ultimately have
(∀ *X* ∈ *?A*. (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *X*) ∈ *failures P*) ⟶
(*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, ⋃ *X* ∈ *?A*. *X*) ∈ *failures P* **..**
moreover have ∀ *X* ∈ *?A*. (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *X*) ∈ *failures P*
proof (*rule ballI*, *simp add*: *singleton-set-def*, *erule bexE*, *simp*)
fix *z'*
have ∀ *u* ∈ *range D* ∩ (−*I*) `` *range D*.
∀ *xs ys*. (*xs*, *ys*) ∈ *rel-ipurge P I D u* ⟶
ref-dom-events P D u xs = *ref-dom-events P D u ys*
using *IU* **by** (*simp add*: *weakly-future-consistent-def*)
moreover assume *E*: *z'* ∈ *ipurge-ref I D* (*D y*) *zs Z*
hence (*D y*, *D z'*) ∉ *I* **by** (*simp add*: *ipurge-ref-def*)
hence *D z'* ∈ *range D* ∩ (−*I*) `` *range D* **by** (*simp add*: *Image-iff*, *rule exI*)
ultimately have ∀ *xs ys*. (*xs*, *ys*) ∈ *rel-ipurge P I D* (*D z'*) ⟶
ref-dom-events P D (*D z'*) *xs* = *ref-dom-events P D* (*D z'*) *ys* **..**
hence
F: (*xs* @ *zs*, *xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*) ∈ *rel-ipurge P I D* (*D z'*) ⟶
ref-dom-events P D (*D z'*) (*xs* @ *zs*) =
ref-dom-events P D (*D z'*) (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*)
by *blast*
have *z'* ∈ {*x* ∈ *Z*. *D x* ∈ *unaffected-domains I D* {*D y*} *zs*}
using *E* **by** (*simp add*: *unaffected-domains-single-dom*)
hence *D z'* ∈ *unaffected-domains I D* {*D y*} *zs* **by** *simp*
hence *ipurge-tr-rev I D* (*D z'*) (*xs* @ *zs*) =
ipurge-tr-rev I D (*D z'*) (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*)
by (*rule ipurge-tr-rev-ipurge-tr-2*)
moreover have *xs* @ *zs* ∈ *traces P* **using** *A* **by** (*rule failures-traces*)
ultimately have
(*xs* @ *zs*, *xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*) ∈ *rel-ipurge P I D* (*D z'*)
using *B* **by** (*simp add*: *rel-ipurge-def*)
with *F* **have** *ref-dom-events P D* (*D z'*) (*xs* @ *zs*) =
ref-dom-events P D (*D z'*) (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*) **..**
moreover have *z'* ∈ *ref-dom-events P D* (*D z'*) (*xs* @ *zs*)
proof (*simp add*: *ref-dom-events-def refusals-def*)
have {*z'*} ⊆ *Z* **using** *E* **by** (*simp add*: *ipurge-ref-def*)
with *A* **show** (*xs* @ *zs*, {*z'*}) ∈ *failures P* **by** (*rule process-rule-3*)

qed
ultimately have $z' \in$ *ref-dom-events P D* $(D\ z')$
$(xs\ @\ y\ \#\ ipurge\text{-}tr\ I\ D\ (D\ y)\ zs)$
by *simp*
thus $(xs\ @\ y\ \#\ ipurge\text{-}tr\ I\ D\ (D\ y)\ zs,\ \{z'\}) \in failures\ P$
by (*simp add*: *ref-dom-events-def refusals-def*)
qed
ultimately have
$(xs\ @\ y\ \#\ ipurge\text{-}tr\ I\ D\ (D\ y)\ zs,\ \bigcup X \in ?A.\ X) \in failures\ P$ **..**
thus *?thesis* **by** (*simp only*: *singleton-set-union*)
qed

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 $(xs\ @\ y\ \#\ ys,\ Y) \in failures\ P \longrightarrow$
$(xs\ @\ ipurge\text{-}tr\ I\ D\ (D\ y)\ ys,\ ipurge\text{-}ref\ I\ D\ (D\ y)\ ys\ Y) \in failures\ P$
proof (*induction ys arbitrary*: *Y rule*: *rev-induct*, *rule-tac* [!] *impI*)
fix *Y*
assume *A*: $(xs\ @\ [y],\ Y) \in failures\ P$
show $(xs\ @\ ipurge\text{-}tr\ I\ D\ (D\ y)\ [],\ ipurge\text{-}ref\ I\ D\ (D\ y)\ []\ Y) \in failures\ P$
proof (*cases* $\exists y'.\ y' \in ipurge\text{-}ref\ I\ D\ (D\ y)\ []\ Y$)
case *True*
have $xs\ @\ [y] \in traces\ P$ **using** *A* **by** (*rule failures-traces*)
hence $xs \in traces\ P$ **by** (*rule process-rule-2-traces*)
hence $xs\ @\ ipurge\text{-}tr\ I\ D\ (D\ y)\ [] \in traces\ P$ **by** *simp*
with *RUC* **and** *IU* **and** *A* **show** *?thesis*
using *True* **by** (*rule iu-condition-imply-secure-aux-1*)
next
case *False*
moreover have $(xs,\ \{\}) \in failures\ P$ **using** *A* **by** (*rule process-rule-2*)
ultimately show *?thesis* **by** *simp*
qed
next
fix *y' ys Y*
assume
A: $\bigwedge Y'.\ (xs\ @\ y\ \#\ ys,\ Y') \in failures\ P \longrightarrow$
$(xs\ @\ ipurge\text{-}tr\ I\ D\ (D\ y)\ ys,\ ipurge\text{-}ref\ I\ D\ (D\ y)\ ys\ Y') \in failures\ P$ **and**
B: $(xs\ @\ y\ \#\ ys\ @\ [y'],\ Y) \in failures\ P$
have $(xs\ @\ y\ \#\ ys,\ \{\}) \in failures\ P \longrightarrow$
$(xs\ @\ ipurge\text{-}tr\ I\ D\ (D\ y)\ ys,\ ipurge\text{-}ref\ I\ D\ (D\ y)\ ys\ \{\}) \in failures\ P$
(**is** - $\longrightarrow$ (-, *?Y'*) $\in$ -)
using *A* **.**
moreover have $((xs\ @\ y\ \#\ ys)\ @\ [y'],\ Y) \in failures\ P$ **using** *B* **by** *simp*
hence *C*: $(xs\ @\ y\ \#\ ys,\ \{\}) \in failures\ P$ **by** (*rule process-rule-2*)
ultimately have $(xs\ @\ ipurge\text{-}tr\ I\ D\ (D\ y)\ ys,\ ?Y') \in failures\ P$ **..**
moreover have $\{\} \subseteq ?Y'$ **..**
ultimately have *D*: $(xs\ @\ ipurge\text{-}tr\ I\ D\ (D\ y)\ ys,\ \{\}) \in failures\ P$

by (*rule process-rule-3*)
have *E*: *xs* @ *ipurge-tr I D* (*D y*) (*ys* @ [*y'*]) ∈ *traces P*
proof (*cases D y'* ∈ *sinks I D* (*D y*) (*ys* @ [*y'*]))
case *True*
hence (*xs* @ *ipurge-tr I D* (*D y*) (*ys* @ [*y'*]), {}) ∈ *failures P* **using** *D* **by** *simp*
thus *?thesis* **by** (*rule failures-traces*)
next
case *False*
have ∀ *u* ∈ *range D* ∩ (−*I*) `` *range D*.
∀ *xs ys*. (*xs*, *ys*) ∈ *rel-ipurge P I D u* ⟶
next-dom-events P D u xs = *next-dom-events P D u ys*
using *IU* **by** (*simp add*: *weakly-future-consistent-def*)
moreover have (*D y*, *D y'*) ∉ *I*
using *False* **by** (*simp add*: *sinks-interference-eq* [*symmetric*] *del*: *sinks.simps*)
hence *D y'* ∈ *range D* ∩ (−*I*) `` *range D* **by** (*simp add*: *Image-iff*, *rule exI*)
ultimately have ∀ *xs ys*. (*xs*, *ys*) ∈ *rel-ipurge P I D* (*D y'*) ⟶
next-dom-events P D (*D y'*) *xs* = *next-dom-events P D* (*D y'*) *ys* **..**
hence
F: (*xs* @ *y* # *ys*, *xs* @ *ipurge-tr I D* (*D y*) *ys*) ∈ *rel-ipurge P I D* (*D y'*) ⟶
next-dom-events P D (*D y'*) (*xs* @ *y* # *ys*) =
next-dom-events P D (*D y'*) (*xs* @ *ipurge-tr I D* (*D y*) *ys*)
by *blast*
have ∀ *v* ∈ *insert* (*D y*) (*sinks I D* (*D y*) *ys*). (*v*, *D y'*) ∉ *I*
using *False* **by** (*simp add*: *sinks-interference-eq* [*symmetric*] *del*: *sinks.simps*)
hence ∀ *v* ∈ *sinks-aux I D* {*D y*} *ys*. (*v*, *D y'*) ∉ *I*
by (*simp add*: *sinks-aux-single-dom*)
hence *D y'* ∈ *unaffected-domains I D* {*D y*} *ys*
by (*simp add*: *unaffected-domains-def*)
hence *ipurge-tr-rev I D* (*D y'*) (*xs* @ *y* # *ys*) =
ipurge-tr-rev I D (*D y'*) (*xs* @ *ipurge-tr I D* (*D y*) *ys*)
by (*rule ipurge-tr-rev-ipurge-tr-1*)
moreover have *xs* @ *y* # *ys* ∈ *traces P* **using** *C* **by** (*rule failures-traces*)
moreover have *xs* @ *ipurge-tr I D* (*D y*) *ys* ∈ *traces P*
using *D* **by** (*rule failures-traces*)
ultimately have
(*xs* @ *y* # *ys*, *xs* @ *ipurge-tr I D* (*D y*) *ys*) ∈ *rel-ipurge P I D* (*D y'*)
by (*simp add*: *rel-ipurge-def*)
with *F* **have** *next-dom-events P D* (*D y'*) (*xs* @ *y* # *ys*) =
next-dom-events P D (*D y'*) (*xs* @ *ipurge-tr I D* (*D y*) *ys*) **..**
moreover have *y'* ∈ *next-dom-events P D* (*D y'*) (*xs* @ *y* # *ys*)
proof (*simp add*: *next-dom-events-def next-events-def*)
qed (*rule failures-traces* [*OF B*])
ultimately have *y'* ∈ *next-dom-events P D* (*D y'*)
(*xs* @ *ipurge-tr I D* (*D y*) *ys*)
by *simp*
hence *xs* @ *ipurge-tr I D* (*D y*) *ys* @ [*y'*] ∈ *traces P*
by (*simp add*: *next-dom-events-def next-events-def*)
thus *?thesis* **using** *False* **by** *simp*
qed

show (*xs* @ *ipurge-tr I D* (*D y*) (*ys* @ [*y'*]), *ipurge-ref I D* (*D y*) (*ys* @ [*y'*]) *Y*)
∈ *failures P*
proof (*cases* ∃ *x*. *x* ∈ *ipurge-ref I D* (*D y*) (*ys* @ [*y'*]) *Y*)
case *True*
with *RUC* **and** *IU* **and** *B* **and** *E* **show** *?thesis* **by** (*rule iu-condition-imply-secure-aux-1*)
next
case *False*
moreover have (*xs* @ *ipurge-tr I D* (*D y*) (*ys* @ [*y'*]), {}) ∈ *failures P*
using *E* **by** (*rule traces-failures*)
ultimately show *?thesis* **by** *simp*
qed
qed

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 (*induction zs arbitrary*: *Z rule*: *rev-induct*, *rule-tac* [!] *impI*)
fix *Z*
assume *A*: (*xs* @ [], *Z*) ∈ *failures P*
show (*xs* @ *y* # *ipurge-tr I D* (*D y*) [], *ipurge-ref I D* (*D y*) [] *Z*) ∈ *failures P*
proof (*cases* ∃ *z'*. *z'* ∈ *ipurge-ref I D* (*D y*) [] *Z*)
case *True*
have *xs* @ *y* # *ipurge-tr I D* (*D y*) [] ∈ *traces P* **using** *Y* **by** *simp*
with *RUC* **and** *IU* **and** *A* **show** *?thesis*
using *True* **by** (*rule iu-condition-imply-secure-aux-2*)
next
case *False*
moreover have (*xs* @ [*y*], {}) ∈ *failures P* **using** *Y* **by** (*rule traces-failures*)
ultimately show *?thesis* **by** *simp*
qed
next
fix *z zs Z*
assume
A: ⋀*Z*. (*xs* @ *zs*, *Z*) ∈ *failures P* ⟶
(*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs Z*) ∈ *failures P* **and**
B: (*xs* @ *zs* @ [*z*], *Z*) ∈ *failures P*
have (*xs* @ *zs*, {}) ∈ *failures P* ⟶
(*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs* {}) ∈ *failures P*
(**is** - ⟶ (-, *?Z'*) ∈ -)
using *A* **.**
moreover have ((*xs* @ *zs*) @ [*z*], *Z*) ∈ *failures P* **using** *B* **by** *simp*
hence *C*: (*xs* @ *zs*, {}) ∈ *failures P* **by** (*rule process-rule-2*)
ultimately have (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *?Z'*) ∈ *failures P* **..**
moreover have {} ⊆ *?Z'* **..**
ultimately have *D*: (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, {}) ∈ *failures P*

by (*rule process-rule-3*)
have *E*: *xs* @ *y* # *ipurge-tr I D* (*D y*) (*zs* @ [*z*]) ∈ *traces P*
proof (*cases D z* ∈ *sinks I D* (*D y*) (*zs* @ [*z*]))
  case *True*
  hence (*xs* @ *y* # *ipurge-tr I D* (*D y*) (*zs* @ [*z*]), {}) ∈ *failures P*
   using *D* **by** *simp*
  thus *?thesis* **by** (*rule failures-traces*)
next
  case *False*
  have ∀ *u* ∈ *range D* ∩ (−*I*) " *range D*.
    ∀ *xs ys*. (*xs*, *ys*) ∈ *rel-ipurge P I D u* ⟶
    next-dom-events P D u xs = *next-dom-events P D u ys*
   using *IU* **by** (*simp add*: *weakly-future-consistent-def*)
  moreover have (*D y*, *D z*) ∉ *I*
   using *False* **by** (*simp add*: *sinks-interference-eq* [*symmetric*] *del*: *sinks.simps*)
  hence *D z* ∈ *range D* ∩ (−*I*) " *range D* **by** (*simp add*: *Image-iff*, *rule exI*)
  ultimately have ∀ *xs ys*. (*xs*, *ys*) ∈ *rel-ipurge P I D* (*D z*) ⟶
    next-dom-events P D (*D z*) *xs* = *next-dom-events P D* (*D z*) *ys* **..**
  hence
    F: (*xs* @ *zs*, *xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*) ∈ *rel-ipurge P I D* (*D z*) ⟶
      next-dom-events P D (*D z*) (*xs* @ *zs*) =
      next-dom-events P D (*D z*) (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*)
   by *blast*
  have ∀ *v* ∈ *insert* (*D y*) (*sinks I D* (*D y*) *zs*). (*v*, *D z*) ∉ *I*
   using *False* **by** (*simp add*: *sinks-interference-eq* [*symmetric*] *del*: *sinks.simps*)
  hence ∀ *v* ∈ *sinks-aux I D* {*D y*} *zs*. (*v*, *D z*) ∉ *I*
   by (*simp add*: *sinks-aux-single-dom*)
  hence *D z* ∈ *unaffected-domains I D* {*D y*} *zs*
   by (*simp add*: *unaffected-domains-def*)
  hence *ipurge-tr-rev I D* (*D z*) (*xs* @ *zs*) =
    ipurge-tr-rev I D (*D z*) (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*)
   by (*rule ipurge-tr-rev-ipurge-tr-2*)
  moreover have *xs* @ *zs* ∈ *traces P* **using** *C* **by** (*rule failures-traces*)
  moreover have *xs* @ *y* # *ipurge-tr I D* (*D y*) *zs* ∈ *traces P*
   using *D* **by** (*rule failures-traces*)
  ultimately have
   (*xs* @ *zs*, *xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*) ∈ *rel-ipurge P I D* (*D z*)
   by (*simp add*: *rel-ipurge-def*)
  with *F* **have** *next-dom-events P D* (*D z*) (*xs* @ *zs*) =
    next-dom-events P D (*D z*) (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*) **..**
  moreover have *z* ∈ *next-dom-events P D* (*D z*) (*xs* @ *zs*)
  proof (*simp add*: *next-dom-events-def next-events-def*)
  qed (*rule failures-traces* [*OF B*])
  ultimately have *z* ∈ *next-dom-events P D* (*D z*)
   (*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*)
   by *simp*
  hence *xs* @ *y* # *ipurge-tr I D* (*D y*) *zs* @ [*z*] ∈ *traces P*
   by (*simp add*: *next-dom-events-def next-events-def*)
  thus *?thesis* **using** *False* **by** *simp*

qed
show (*xs* @ *y* # *ipurge-tr I D* (*D y*) (*zs* @ [*z*]),
ipurge-ref I D (*D y*) (*zs* @ [*z*]) *Z*)
∈ *failures P*
proof (*cases* ∃ *x*. *x* ∈ *ipurge-ref I D* (*D y*) (*zs* @ [*z*]) *Z*)
case *True*
with *RUC* **and** *IU* **and** *B* **and** *E* **show** *?thesis* **by** (*rule iu-condition-imply-secure-aux-2*)
next
case *False*
moreover have (*xs* @ *y* # *ipurge-tr I D* (*D y*) (*zs* @ [*z*]), {}) ∈ *failures P*
using *E* **by** (*rule traces-failures*)
ultimately show *?thesis* **by** *simp*
qed
qed

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 (*simp add*: *secure-def futures-def*, (*rule allI*)+, *rule impI*, *erule conjE*)
fix *xs y ys Y zs Z*
assume
A: (*xs* @ *y* # *ys*, *Y*) ∈ *failures P* **and**
B: (*xs* @ *zs*, *Z*) ∈ *failures P*
show (*xs* @ *ipurge-tr I D* (*D y*) *ys*, *ipurge-ref I D* (*D y*) *ys Y*) ∈ *failures P* ∧
(*xs* @ *y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs Z*) ∈ *failures P*
(**is** *?P* ∧ *?Q*)
proof
show *?P* **using** *RUC* **and** *IU* **and** *A* **by** (*rule iu-condition-imply-secure-1*)
next
have ((*xs* @ [*y*]) @ *ys*, *Y*) ∈ *failures P* **using** *A* **by** *simp*
hence (*xs* @ [*y*], {}) ∈ *failures P* **by** (*rule process-rule-2-failures*)
hence *xs* @ [*y*] ∈ *traces P* **by** (*rule failures-traces*)
with *RUC* **and** *IU* **show** *?Q* **using** *B* **by** (*rule iu-condition-imply-secure-2*)
qed
qed

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 (*induction ys rule*: *rev-induct*, *simp-all*, (*rule impI*)+)
fix *y ys*
assume ∗: *ipurge-tr-rev-aux I D* (*D* ' (*X* ∪ *set zs*)) (*ys* @ [*y*]) = []
then have *A*: ¬ (∃ *v* ∈ *D* ' (*X* ∪ *set zs*). (*D y*, *v*) ∈ *I*)
by (*cases* ∃ *v* ∈ *D* ' (*X* ∪ *set zs*). (*D y*, *v*) ∈ *I*,
simp-all add: *ipurge-tr-rev-aux-append*)
with ∗ **have** *B*: *ipurge-tr-rev-aux I D* (*D* ' (*X* ∪ *set zs*)) *ys* = []
by (*simp add*: *ipurge-tr-rev-aux-append*)
assume (*xs* @ *ys* @ *y* # *zs*, *X*) ∈ *failures P*
hence (*y* # *zs*, *X*) ∈ *futures P* (*xs* @ *ys*) **by** (*simp add*: *futures-def*)
hence (*ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs X*)
∈ *futures P* (*xs* @ *ys*)
using *S* **by** (*simp add*: *secure-def*)
moreover have *ipurge-tr I D* (*D y*) *zs* = *zs* **using** *A* **by** (*simp add*: *ipurge-tr-all*)
moreover have *ipurge-ref I D* (*D y*) *zs X* = *X* **using** *A* **by** (*rule ipurge-ref-all*)
ultimately have (*zs*, *X*) ∈ *futures P* (*xs* @ *ys*) **by** *simp*
hence *C*: (*xs* @ *ys* @ *zs*, *X*) ∈ *failures P* **by** (*simp add*: *futures-def*)
assume (*xs* @ *ys* @ *zs*, *X*) ∈ *failures P* ⟶
ipurge-tr-rev-aux I D (*D* ' (*X* ∪ *set zs*)) *ys* = [] ⟶
(*xs* @ *zs*, *X*) ∈ *failures P*
hence *ipurge-tr-rev-aux I D* (*D* ' (*X* ∪ *set zs*)) *ys* = [] ⟶
(*xs* @ *zs*, *X*) ∈ *failures P*
using *C* **..**
thus (*xs* @ *zs*, *X*) ∈ *failures P* **using** *B* **..**
qed

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 (*induction ys arbitrary*: *xs zs rule*: *rev-induct*,
simp-all add: *rel-ipurge-aux-def*, (*rule-tac* [!] *impI*)+, (*erule-tac* [!] *conjE*)+)
fix *xs zs*
assume (*xs* @ *zs*, *X*) ∈ *failures P*
hence ([] @ *xs* @ *zs*, *X*) ∈ *failures P* **by** *simp*
moreover assume *ipurge-tr-rev-aux I D* (*D* ' (*X* ∪ *set zs*)) *xs* = []
ultimately have ([] @ *zs*, *X*) ∈ *failures P*
using *S* **by** (*rule-tac secure-implies-failure-consistency-aux*)
thus (*zs*, *X*) ∈ *failures P* **by** *simp*
next
fix *y ys xs zs*
assume
A: ⋀*xs' zs'*. *xs'* ∈ *traces P* ∧ *ys* ∈ *traces P* ∧
ipurge-tr-rev-aux I D (*D* ' (*X* ∪ *set zs'*)) *xs'* =
ipurge-tr-rev-aux I D (*D* ' (*X* ∪ *set zs'*)) *ys* ⟶
(*xs'* @ *zs'*, *X*) ∈ *failures P* ⟶ (*ys* @ *zs'*, *X*) ∈ *failures P* **and**
B: (*xs* @ *zs*, *X*) ∈ *failures P* **and**
C: *xs* ∈ *traces P* **and**

D: *ys* @ [*y*] ∈ *traces P* **and**
E: *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *xs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set zs*)) (*ys* @ [*y*])
show (*ys* @ *y* # *zs*, *X*) ∈ *failures P*
proof (*cases* ∃ *v* ∈ *D* ‘ (*X* ∪ *set zs*). (*D y*, *v*) ∈ *I*)
case *True*
hence *F*: *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *xs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set* (*y* # *zs*))) *ys* @ [*y*]
using *E* **by** (*simp add*: *ipurge-tr-rev-aux-append*)
hence
∃ *vs ws*. *xs* = *vs* @ *y* # *ws* ∧ *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *ws* = []
by (*rule ipurge-tr-rev-aux-last-2*)
then obtain *vs* **and** *ws* **where**
G: *xs* = *vs* @ *y* # *ws* ∧ *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *ws* = []
by *blast*
hence *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *xs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set zs*)) ((*vs* @ [*y*]) @ *ws*)
by *simp*
hence *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *xs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set zs*)) (*vs* @ [*y*])
using *G* **by** (*simp only*: *ipurge-tr-rev-aux-append-nil*)
moreover have ∃ *v* ∈ *D* ‘ (*X* ∪ *set zs*). (*D y*, *v*) ∈ *I*
using *F* **by** (*rule ipurge-tr-rev-aux-last-1*)
ultimately have *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *xs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set* (*y* # *zs*))) *vs* @ [*y*]
by (*simp add*: *ipurge-tr-rev-aux-append*)
hence *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set* (*y* # *zs*))) *vs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set* (*y* # *zs*))) *ys*
using *F* **by** *simp*
moreover have *vs* @ *y* # *ws* ∈ *traces P* **using** *C* **and** *G* **by** *simp*
hence *vs* ∈ *traces P* **by** (*rule process-rule-2-traces*)
moreover have *ys* ∈ *traces P* **using** *D* **by** (*rule process-rule-2-traces*)
moreover have *vs* ∈ *traces P* ∧ *ys* ∈ *traces P* ∧
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set* (*y* # *zs*))) *vs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set* (*y* # *zs*))) *ys* ⟶
(*vs* @ *y* # *zs*, *X*) ∈ *failures P* ⟶ (*ys* @ *y* # *zs*, *X*) ∈ *failures P*
using *A* **.**
ultimately have *H*: (*vs* @ *y* # *zs*, *X*) ∈ *failures P* ⟶
(*ys* @ *y* # *zs*, *X*) ∈ *failures P*
by *simp*
have ((*vs* @ [*y*]) @ *ws* @ *zs*, *X*) ∈ *failures P* **using** *B* **and** *G* **by** *simp*
moreover have *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *ws* = [] **using** *G* **..**
ultimately have ((*vs* @ [*y*]) @ *zs*, *X*) ∈ *failures P*
using *S* **by** (*rule-tac secure-implies-failure-consistency-aux*)
thus *?thesis* **using** *H* **by** *simp*
next
case *False*
hence *ipurge-tr-rev-aux I D* (*D* ‘ (*X* ∪ *set zs*)) *xs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set zs*)) *ys*

using *E* **by** (*simp add*: *ipurge-tr-rev-aux-append*)
moreover have *ys* ∈ *traces P* **using** *D* **by** (*rule process-rule-2-traces*)
moreover have *xs* ∈ *traces P* ∧ *ys* ∈ *traces P* ∧
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set zs*)) *xs* =
ipurge-tr-rev-aux I D (*D* ‘ (*X* ∪ *set zs*)) *ys* ⟶
(*xs* @ *zs*, *X*) ∈ *failures P* ⟶ (*ys* @ *zs*, *X*) ∈ *failures P*
using *A* **.**
ultimately have (*ys* @ *zs*, *X*) ∈ *failures P* **using** *B* **and** *C* **by** *simp*
hence (*zs*, *X*) ∈ *futures P ys* **by** (*simp add*: *futures-def*)
moreover have ∃ *Y*. ([*y*], *Y*) ∈ *futures P ys*
using *D* **by** (*simp add*: *traces-def Domain-iff futures-def*)
then obtain *Y* **where** ([*y*], *Y*) ∈ *futures P ys* **..**
ultimately have
(*y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs X*) ∈ *futures P ys*
using *S* **by** (*simp add*: *secure-def*)
moreover have *ipurge-tr I D* (*D y*) *zs* = *zs*
using *False* **by** (*simp add*: *ipurge-tr-all*)
moreover have *ipurge-ref I D* (*D y*) *zs X* = *X*
using *False* **by** (*rule ipurge-ref-all*)
ultimately show *?thesis* **by** (*simp add*: *futures-def*)
qed
qed

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 (*simp add*: *traces-def Domain-iff*, *rule-tac x* = {} **in** *exI*,
rule secure-implies-failure-consistency, *simp-all*)
qed (*erule exE*, *erule process-rule-3*, *simp*)

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*
by (*auto simp add*: *next-events-def rel-ipurge-aux-single-dom intro*: *secure-implies-trace-consistency*)

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*
by (*simp add*: *refusals-def*, *subst append-Nil2* [*symmetric*],
rule secure-implies-failure-consistency, *simp-all*)

lemma *secure-implies-ref-event-consistency*:
secure P I D ⟹ (*xs*, *ys*) ∈ *rel-ipurge P I D* (*D x*) ⟹
{*x*} ∈ *refusals P xs* ⟹ {*x*} ∈ *refusals P ys*
by (*rule secure-implies-refusal-consistency*, *simp-all add*: *rel-ipurge-aux-single-dom*)

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

proof (*simp add*: *future-consistent-def next-dom-events-def ref-dom-events-def*,
(*rule allI*)+, *rule impI*, *rule conjI*, *rule-tac* [!] *equalityI*, *rule-tac* [!] *subsetI*,
simp-all, *erule-tac* [!] *conjE*)
fix *xs ys x*
assume (*xs*, *ys*) ∈ *rel-ipurge P I D* (*D x*) **and** *x* ∈ *next-events P xs*
with *S* **show** *x* ∈ *next-events P ys* **by** (*rule secure-implies-next-event-consistency*)
next
fix *xs ys x*
have ∀ *u* ∈ *range D*. *equiv* (*traces P*) (*rel-ipurge P I D u*)
using *view-partition-rel-ipurge* **by** (*simp add*: *view-partition-def*)
hence *sym* (*rel-ipurge P I D* (*D x*)) **by** (*simp add*: *equiv-def*)
moreover assume (*xs*, *ys*) ∈ *rel-ipurge P I D* (*D x*)
ultimately have (*ys*, *xs*) ∈ *rel-ipurge P I D* (*D x*) **by** (*rule symE*)
moreover assume *x* ∈ *next-events P ys*
ultimately show *x* ∈ *next-events P xs*
using *S* **by** (*rule-tac secure-implies-next-event-consistency*)
next
fix *xs ys x*
assume (*xs*, *ys*) ∈ *rel-ipurge P I D* (*D x*) **and** {*x*} ∈ *refusals P xs*
with *S* **show** {*x*} ∈ *refusals P ys* **by** (*rule secure-implies-ref-event-consistency*)
next
fix *xs ys x*
have ∀ *u* ∈ *range D*. *equiv* (*traces P*) (*rel-ipurge P I D u*)
using *view-partition-rel-ipurge* **by** (*simp add*: *view-partition-def*)
hence *sym* (*rel-ipurge P I D* (*D x*)) **by** (*simp add*: *equiv-def*)
moreover assume (*xs*, *ys*) ∈ *rel-ipurge P I D* (*D x*)
ultimately have (*ys*, *xs*) ∈ *rel-ipurge P I D* (*D x*) **by** (*rule symE*)
moreover assume {*x*} ∈ *refusals P ys*
ultimately show {*x*} ∈ *refusals P xs*
using *S* **by** (*rule-tac secure-implies-ref-event-consistency*)
qed

theorem *ipurge-unwinding*:
ref-union-closed P ⟹
secure P I D = *weakly-future-consistent P I D* (*rel-ipurge P I D*)
proof (*rule iffI*, *subst fc-equals-wfc-rel-ipurge* [*symmetric*])
qed (*erule secure-implies-iu-condition*, *rule iu-condition-imply-secure*)

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

definition *d-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**
d-weakly-future-consistent P I D R $\equiv$
$\forall u \in range\ D \cap (-I)\ ``\ range\ D.\ \forall xs\ ys.\ (xs,\ ys) \in R\ u \longrightarrow$
$(xs \in traces\ P) = (ys \in traces\ P) \wedge$
$next\text{-}dom\text{-}events\ P\ D\ u\ xs = next\text{-}dom\text{-}events\ P\ D\ u\ ys$

lemma *dfc-implies-dwfc*:
d-future-consistent P D R $\Longrightarrow$ *d-weakly-future-consistent P I D R*
by (*simp only*: *d-future-consistent-def d-weakly-future-consistent-def*, *blast*)

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 (*rule iffI*, *erule dfc-implies-dwfc*,
simp only: *d-future-consistent-def d-weakly-future-consistent-def*,
rule ballI, (*rule allI*)+, *rule impI*)

```
  fix u xs ys
  assume
    A: ∀ u ∈ range D ∩ (−I) `` range D. ∀ xs ys. (xs, ys) ∈ rel-ipurge P I D u −→
      (xs ∈ traces P) = (ys ∈ traces P) ∧
      next-dom-events P D u xs = next-dom-events P D u ys and
    B: u ∈ range D and
    C: (xs, ys) ∈ rel-ipurge P I D u
  show (xs ∈ traces P) = (ys ∈ traces P) ∧
    next-dom-events P D u xs = next-dom-events P D u ys
  proof (cases u ∈ range D ∩ (−I) `` range D)
    case True
    with A have ∀ xs ys. (xs, ys) ∈ rel-ipurge P I D u −→
      (xs ∈ traces P) = (ys ∈ traces P) ∧
      next-dom-events P D u xs = next-dom-events P D u ys ..
    hence (xs, ys) ∈ rel-ipurge P I D u −→
      (xs ∈ traces P) = (ys ∈ traces P) ∧
      next-dom-events P D u xs = next-dom-events P D u ys
     by blast
    thus ?thesis using C ..
  next
    case False
    hence D: u ∉ (−I) `` range D using B by simp
    have ipurge-tr-rev I D u xs = ipurge-tr-rev I D u ys
     using C by (simp add: rel-ipurge-def)
    moreover have ∀ zs. ipurge-tr-rev I D u zs = zs
    proof (rule allI, rule ipurge-tr-rev-all, rule ballI, erule imageE, rule ccontr)
      fix v x
      assume (v, u) ∉ I
      hence (v, u) ∈ −I by simp
      moreover assume v = D x
      hence v ∈ range D by simp
      ultimately have u ∈ (−I) `` range D ..
      thus False using D by contradiction
    qed
    ultimately show ?thesis by simp
  qed
qed

lemma d-fc-equals-dfc:
  assumes A: deterministic P
  shows future-consistent P D R = d-future-consistent P D R
proof (rule iffI, simp-all only: d-future-consistent-def,
 rule ballI, (rule allI)+, rule impI, rule conjI, rule fc-traces, assumption+,
 simp-all add: future-consistent-def del: ball-simps)
  assume B: ∀ 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
  show ∀ u ∈ range D. ∀ xs ys. (xs, ys) ∈ R u −→
    ref-dom-events P D u xs = ref-dom-events P D u ys
```

proof (*rule ballI*, (*rule allI*)+, *rule impI*,
simp add: *ref-dom-events-def set-eq-iff*, *rule allI*)
fix *u xs ys x*
assume *u* ∈ *range D*
with *B* **have** ∀ *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* **..**
hence (*xs*, *ys*) ∈ *R u* ⟶
(*xs* ∈ *traces P*) = (*ys* ∈ *traces P*) ∧
next-dom-events P D u xs = *next-dom-events P D u ys*
by *blast*
moreover assume (*xs*, *ys*) ∈ *R u*
ultimately have *C*: (*xs* ∈ *traces P*) = (*ys* ∈ *traces P*) ∧
next-dom-events P D u xs = *next-dom-events P D u ys* **..**
show (*u* = *D x* ∧ {*x*} ∈ *refusals P xs*) = (*u* = *D x* ∧ {*x*} ∈ *refusals P ys*)
proof (*cases u* = *D x*, *simp-all*, *cases xs* ∈ *traces P*)
assume *D*: *u* = *D x* **and** *E*: *xs* ∈ *traces P*
have
A′: ∀ *xs* ∈ *traces P*. ∀ *X*. *X* ∈ *refusals P xs* = (*X* ∩ *next-events P xs* = {})
using *A* **by** (*simp add*: *deterministic-def*)
hence ∀ *X*. *X* ∈ *refusals P xs* = (*X* ∩ *next-events P xs* = {}) **using** *E* **..**
hence {*x*} ∈ *refusals P xs* = ({*x*} ∩ *next-events P xs* = {}) **..**
moreover have *ys* ∈ *traces P* **using** *C* **and** *E* **by** *simp*
with *A′* **have** ∀ *X*. *X* ∈ *refusals P ys* = (*X* ∩ *next-events P ys* = {}) **..**
hence {*x*} ∈ *refusals P ys* = ({*x*} ∩ *next-events P ys* = {}) **..**
moreover have {*x*} ∩ *next-events P xs* = {*x*} ∩ *next-events P ys*
proof (*simp add*: *set-eq-iff*, *rule allI*, *rule iffI*, *erule-tac* [!] *conjE*, *simp-all*)
assume *x* ∈ *next-events P xs*
hence *x* ∈ *next-dom-events P D u xs* **using** *D* **by** (*simp add*: *next-dom-events-def*)
hence *x* ∈ *next-dom-events P D u ys* **using** *C* **by** *simp*
thus *x* ∈ *next-events P ys* **by** (*simp add*: *next-dom-events-def*)
next
assume *x* ∈ *next-events P ys*
hence *x* ∈ *next-dom-events P D u ys* **using** *D* **by** (*simp add*: *next-dom-events-def*)
hence *x* ∈ *next-dom-events P D u xs* **using** *C* **by** *simp*
thus *x* ∈ *next-events P xs* **by** (*simp add*: *next-dom-events-def*)
qed
ultimately show ({*x*} ∈ *refusals P xs*) = ({*x*} ∈ *refusals P ys*) **by** *simp*
next
assume *D*: *xs* ∉ *traces P*
hence ∀ *X*. (*xs*, *X*) ∉ *failures P* **by** (*simp add*: *traces-def Domain-iff*)
hence *refusals P xs* = {} **by** (*rule-tac equals0I*, *simp add*: *refusals-def*)
moreover have *ys* ∉ *traces P* **using** *C* **and** *D* **by** *simp*
hence ∀ *X*. (*ys*, *X*) ∉ *failures P* **by** (*simp add*: *traces-def Domain-iff*)
hence *refusals P ys* = {} **by** (*rule-tac equals0I*, *simp add*: *refusals-def*)
ultimately show ({*x*} ∈ *refusals P xs*) = ({*x*} ∈ *refusals P ys*) **by** *simp*
qed
qed
qed

```
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 (rule iffI, simp-all only: d-weakly-future-consistent-def,
 rule ballI, (rule allI)+, rule impI, rule conjI, rule wfc-traces, assumption+,
 simp-all add: weakly-future-consistent-def del: ball-simps)
  assume B: ∀ 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
  show ∀ u ∈ range D ∩ (− I) ‘‘ range D. ∀ xs ys. (xs, ys) ∈ R u ⟶
    ref-dom-events P D u xs = ref-dom-events P D u ys
  proof (rule ballI, (rule allI)+, rule impI,
   simp (no-asm-simp) add: ref-dom-events-def set-eq-iff, rule allI)
    fix u xs ys x
    assume u ∈ range D ∩ (− I) ‘‘ range D
    with B have ∀ 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 ..
    hence (xs, ys) ∈ R u ⟶
      (xs ∈ traces P) = (ys ∈ traces P) ∧
      next-dom-events P D u xs = next-dom-events P D u ys
     by blast
    moreover assume (xs, ys) ∈ R u
    ultimately have C: (xs ∈ traces P) = (ys ∈ traces P) ∧
      next-dom-events P D u xs = next-dom-events P D u ys ..
    show (u = D x ∧ {x} ∈ refusals P xs) = (u = D x ∧ {x} ∈ refusals P ys)
    proof (cases u = D x, simp-all, cases xs ∈ traces P)
      assume D: u = D x and E: xs ∈ traces P
      have A′: ∀ xs ∈ traces P. ∀ X.
        X ∈ refusals P xs = (X ∩ next-events P xs = {})
       using A by (simp add: deterministic-def)
      hence ∀ X. X ∈ refusals P xs = (X ∩ next-events P xs = {}) using E ..
      hence {x} ∈ refusals P xs = ({x} ∩ next-events P xs = {}) ..
      moreover have ys ∈ traces P using C and E by simp
      with A′ have ∀ X. X ∈ refusals P ys = (X ∩ next-events P ys = {}) ..
      hence {x} ∈ refusals P ys = ({x} ∩ next-events P ys = {}) ..
      moreover have {x} ∩ next-events P xs = {x} ∩ next-events P ys
      proof (simp add: set-eq-iff, rule allI, rule iffI, erule-tac [!] conjE, simp-all)
        assume x ∈ next-events P xs
      hence x ∈ next-dom-events P D u xs using D by (simp add: next-dom-events-def)
        hence x ∈ next-dom-events P D u ys using C by simp
        thus x ∈ next-events P ys by (simp add: next-dom-events-def)
      next
        assume x ∈ next-events P ys
      hence x ∈ next-dom-events P D u ys using D by (simp add: next-dom-events-def)
        hence x ∈ next-dom-events P D u xs using C by simp
        thus x ∈ next-events P xs by (simp add: next-dom-events-def)
      qed
```

ultimately show $(\{x\} \in$ *refusals P xs*$) = (\{x\} \in$ *refusals P ys*$)$ **by** *simp*
next
assume *D*: $xs \notin$ *traces P*
hence $\forall X.$ $(xs, X) \notin$ *failures P* **by** (*simp add*: *traces-def Domain-iff*)
hence *refusals P xs* = {} **by** (*rule-tac equals0I*, *simp add*: *refusals-def*)
moreover have $ys \notin$ *traces P* **using** *C* **and** *D* **by** *simp*
hence $\forall X.$ $(ys, X) \notin$ *failures P* **by** (*simp add*: *traces-def Domain-iff*)
hence *refusals P ys* = {} **by** (*rule-tac equals0I*, *simp add*: *refusals-def*)
ultimately show $(\{x\} \in$ *refusals P xs*$) = (\{x\} \in$ *refusals P ys*$)$ **by** *simp*
qed
qed
qed

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 (*subst ref-union-closed-def*, (*rule allI*)+, (*rule impI*)+, *erule exE*)
fix *xs A X*
have $\forall xs \in$ *traces P*. $\forall X.$ $X \in$ *refusals P xs* = $(X \cap$ *next-events P xs* = {})
using *A* **by** (*simp add*: *deterministic-def*)
moreover assume *B*: $\forall X \in A.$ $(xs, X) \in$ *failures P* **and** $X \in A$
hence $(xs, X) \in$ *failures P* **..**
hence $xs \in$ *traces P* **by** (*rule failures-traces*)
ultimately have *C*: $\forall X.$ $X \in$ *refusals P xs* = $(X \cap$ *next-events P xs* = {}) **..**
have *D*: $\forall X \in A.$ $X \cap$ *next-events P xs* = {}
proof
fix *X*
assume $X \in A$
with *B* **have** $(xs, X) \in$ *failures P* **..**
hence $X \in$ *refusals P xs* **by** (*simp add*: *refusals-def*)
thus $X \cap$ *next-events P xs* = {} **using** *C* **by** *simp*
qed
have $(\bigcup X \in A. X) \in$ *refusals P xs* = $((\bigcup X \in A. X) \cap$ *next-events P xs* = {})
using *C* **..**
hence *E*: $(xs, \bigcup X \in A. X) \in$ *failures P* =
$((\bigcup X \in A. X) \cap$ *next-events P xs* = {})
by (*simp add*: *refusals-def*)
show $(xs, \bigcup X \in A. X) \in$ *failures P*
proof (*rule ssubst* [*OF E*], *rule equals0I*, *erule IntE*, *erule UN-E*)
fix *x X*

assume $X \in A$
with D **have** $X \cap$ *next-events* P $xs = \{\}$ **..**
moreover assume $x \in X$ **and** $x \in$ *next-events* P xs
hence $x \in X \cap$ *next-events* P xs **..**
hence $\exists x.\ x \in X \cap$ *next-events* P xs **..**
hence $X \cap$ *next-events* P $xs \neq \{\}$ **by** (*subst ex-in-conv* [*symmetric*])
ultimately show *False* **by** *contradiction*
qed
qed

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 (*insert d-wfc-equals-dwfc* [*of P I D rel-ipurge P I D*, *OF A*], *erule subst*)
qed (*insert d-implies-ruc* [*OF A*], *rule ipurge-unwinding*)

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* $\Rightarrow$ *'a process* **where**
ts-process T $\equiv$ *Abs-process* $(\{(xs, X).\ xs \in T \wedge (\forall x \in X.\ xs\ @\ [x] \notin T)\}, \{\})$

definition *trace-set* :: *'a list set* $\Rightarrow$ *bool* **where**
trace-set T $\equiv [] \in T \wedge (\forall xs\ x.\ xs\ @\ [x] \in T \longrightarrow xs \in T)$

lemma *ts-process-rep*:
assumes *A*: *trace-set T*
shows *Rep-process* (*ts-process T*) =
$(\{(xs, X).\ xs \in T \wedge (\forall x \in X.\ xs\ @\ [x] \notin T)\}, \{\})$
proof (*subst ts-process-def*, *rule Abs-process-inverse*, *simp add*: *process-set-def*,
(*subst conj-assoc* [*symmetric*])+, (*rule conjI*)+, *simp-all add*:
process-prop-1-def
process-prop-2-def
process-prop-3-def
process-prop-4-def

process-prop-5-def
process-prop-6-def)
show [] $\in$ T **using** A **by** (*simp add*: *trace-set-def*)
next
show $\forall$ *xs*. ($\exists$ *x*. *xs* @ [*x*] $\in$ T $\wedge$ ($\exists$ X. $\forall$ x' $\in$ X. *xs* @ [*x*, *x'*] $\notin$ T)) $\longrightarrow$ *xs* $\in$ T
proof (*rule allI*, *rule impI*, *erule exE*, *erule conjE*)
fix *xs x*
have $\forall$ *xs x*. *xs* @ [*x*] $\in$ T $\longrightarrow$ *xs* $\in$ T **using** A **by** (*simp add*: *trace-set-def*)
hence *xs* @ [*x*] $\in$ T $\longrightarrow$ *xs* $\in$ T **by** *blast*
moreover assume *xs* @ [*x*] $\in$ T
ultimately show *xs* $\in$ T **..**
qed
next
show $\forall$ *xs* X. *xs* $\in$ T $\wedge$ ($\exists$ Y. ($\forall$ x $\in$ Y. *xs* @ [*x*] $\notin$ T) $\wedge$ $X \subseteq Y$) $\longrightarrow$
($\forall$ x $\in$ X. *xs* @ [*x*] $\notin$ T)
proof ((*rule allI*)+, *rule impI*, (*erule conjE*, (*erule exE*)?)+, *rule ballI*)
fix *xs x X Y*
assume $\forall$ x $\in$ Y. *xs* @ [*x*] $\notin$ T
moreover assume $X \subseteq Y$ **and** $x \in X$
hence $x \in Y$ **..**
ultimately show *xs* @ [*x*] $\notin$ T **..**
qed
qed

lemma *ts-process-failures*:
trace-set T $\Longrightarrow$
failures (*ts-process* T) = {(*xs*, X). *xs* $\in$ T $\wedge$ ($\forall$ x $\in$ X. *xs* @ [*x*] $\notin$ T)}
by (*drule ts-process-rep*, *simp add*: *failures-def*)

lemma *ts-process-futures*:
trace-set T $\Longrightarrow$
futures (*ts-process* T) *xs* =
{(*ys*, Y). *xs* @ *ys* $\in$ T $\wedge$ ($\forall$ y $\in$ Y. *xs* @ *ys* @ [*y*] $\notin$ T)}
by (*simp add*: *futures-def ts-process-failures*)

lemma *ts-process-traces*:
trace-set T $\Longrightarrow$ *traces* (*ts-process* T) = T
proof (*drule ts-process-failures*, *simp add*: *traces-def*, *rule set-eqI*, *rule iffI*, *simp-all*)
qed (*rule-tac* x = {} **in** *exI*, *simp*)

lemma *ts-process-refusals*:
trace-set T $\Longrightarrow$ *xs* $\in$ T $\Longrightarrow$
refusals (*ts-process* T) *xs* = {X. $\forall$ x $\in$ X. *xs* @ [*x*] $\notin$ T}
by (*drule ts-process-failures*, *simp add*: *refusals-def*)

lemma *ts-process-next-events*:
trace-set T $\Longrightarrow$ (x $\in$ *next-events* (*ts-process* T) *xs*) = (*xs* @ [*x*] $\in$ T)
by (*drule ts-process-traces*, *simp add*: *next-events-def*)

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 $T \Longrightarrow$ *deterministic* (*ts-process* T)
proof (*frule ts-process-traces*, *simp add*: *deterministic-def*, *rule ballI*,
drule ts-process-refusals, *assumption*, *simp add*: *next-events-def*,
rule allI, *rule iffI*)
  fix *xs* X
  assume $\forall x \in X.\ xs\ @\ [x] \notin T$
  thus $X \cap \{x.\ xs\ @\ [x] \in T\} = \{\}$
  by (*rule-tac equals0I*, *erule-tac IntE*, *simp*)
next
  fix *xs* X
  assume A: $X \cap \{x.\ xs\ @\ [x] \in T\} = \{\}$
  show $\forall x \in X.\ xs\ @\ [x] \notin T$
  proof (*rule ballI*, *rule notI*)
    fix x
    assume $x \in X$ **and** $xs\ @\ [x] \in T$
    hence $x \in X \cap \{x.\ xs\ @\ [x] \in T\}$ **by** *simp*
    moreover have $x \notin X \cap \{x.\ xs\ @\ [x] \in T\}$ **using** A **by** (*rule equals0D*)
    ultimately show *False* **by** *contradiction*
  qed
qed

definition *divergences* :: *'a process* $\Rightarrow$ *'a list set* **where**
divergences $P \equiv$ *snd* (*Rep-process* P)

lemma *d-divergences*:
  assumes A: *deterministic* P
  shows *divergences* $P = \{\}$
proof (*subst divergences-def*, *rule equals0I*)
  fix *xs*
  have B: *Rep-process* $P \in$ *process-set* (**is** *?P'* $\in$ -) **by** (*rule Rep-process*)
  hence $\forall xs.\ \exists x.\ xs \in snd\ ?P' \longrightarrow xs\ @\ [x] \in snd\ ?P'$
   by (*simp add*: *process-set-def process-prop-5-def*)
  hence $\exists x.\ xs \in snd\ ?P' \longrightarrow xs\ @\ [x] \in snd\ ?P'$ **..**
  then obtain x **where** $xs \in snd\ ?P' \longrightarrow xs\ @\ [x] \in snd\ ?P'$ **..**
  moreover assume C: $xs \in snd\ ?P'$
  ultimately have D: $xs\ @\ [x] \in snd\ ?P'$ **..**
  have E: $\forall xs\ X.\ xs \in snd\ ?P' \longrightarrow (xs, X) \in fst\ ?P'$
   using B **by** (*simp add*: *process-set-def process-prop-6-def*)
  hence $xs \in snd\ ?P' \longrightarrow (xs, \{x\}) \in fst\ ?P'$ **by** *blast*
  hence $\{x\} \in$ *refusals* P *xs*
   using C **by** (*drule-tac mp*, *simp-all add*: *failures-def refusals-def*)
  moreover have $xs\ @\ [x] \in snd\ ?P' \longrightarrow (xs\ @\ [x], \{\}) \in fst\ ?P'$

using *E* **by** *blast*
hence (*xs* @ [*x*], {}) ∈ *failures P*
using *D* **by** (*drule-tac mp*, *simp-all add*: *failures-def*)
hence *F*: *xs* @ [*x*] ∈ *traces P* **by** (*rule failures-traces*)
hence {*x*} ∩ *next-events P xs* ≠ {} **by** (*simp add*: *next-events-def*)
ultimately have *G*: ({*x*} ∈ *refusals P xs*) ≠ ({*x*} ∩ *next-events P xs* = {})
by *simp*
have ∀ *xs* ∈ *traces P*. ∀ *X*. *X* ∈ *refusals P xs* = (*X* ∩ *next-events P xs* = {})
using *A* **by** (*simp add*: *deterministic-def*)
moreover have *xs* ∈ *traces P* **using** *F* **by** (*rule process-rule-2-traces*)
ultimately have ∀ *X*. *X* ∈ *refusals P xs* = (*X* ∩ *next-events P xs* = {}) **..**
hence {*x*} ∈ *refusals P xs* = ({*x*} ∩ *next-events P xs* = {}) **..**
thus *False* **using** *G* **by** *contradiction*
qed

lemma *trace-set-traces*:
trace-set (*traces P*)
proof (*simp only*: *trace-set-def traces-def failures-def Domain-iff*,
rule conjI, (*rule-tac* [*2*] *allI*)+, *rule-tac* [*2*] *impI*, *erule-tac* [*2*] *exE*)
have *Rep-process P* ∈ *process-set* (**is** *?P′* ∈ -) **by** (*rule Rep-process*)
hence ([], {}) ∈ *fst ?P′* **by** (*simp add*: *process-set-def process-prop-1-def*)
thus ∃ *X*. ([], *X*) ∈ *fst ?P′* **..**
next
fix *xs x X*
have *Rep-process P* ∈ *process-set* (**is** *?P′* ∈ -) **by** (*rule Rep-process*)
hence ∀ *xs x X*. (*xs* @ [*x*], *X*) ∈ *fst ?P′* ⟶ (*xs*, {}) ∈ *fst ?P′*
by (*simp add*: *process-set-def process-prop-2-def*)
hence (*xs* @ [*x*], *X*) ∈ *fst ?P′* ⟶ (*xs*, {}) ∈ *fst ?P′* **by** *blast*
moreover assume (*xs* @ [*x*], *X*) ∈ *fst ?P′*
ultimately have (*xs*, {}) ∈ *fst ?P′* **..**
thus ∃ *X*. (*xs*, *X*) ∈ *fst ?P′* **..**
qed

lemma *d-implies-ts-process-traces*:
deterministic P ⟹ *ts-process* (*traces P*) = *P*
proof (*simp add*: *Rep-process-inject* [*symmetric*] *prod-eq-iff failures-def* [*symmetric*],
insert trace-set-traces [*of P*], *frule ts-process-rep*, *frule d-divergences*,
simp add: *divergences-def deterministic-def*)
assume *A*: ∀ *xs* ∈ *traces P*. ∀ *X*.
(*X* ∈ *refusals P xs*) = (*X* ∩ *next-events P xs* = {})
assume *B*: *trace-set* (*traces P*)
hence *C*: *traces* (*ts-process* (*traces P*)) = *traces P* **by** (*rule ts-process-traces*)
show *failures* (*ts-process* (*traces P*)) = *failures P*
proof (*rule equalityI*, *rule-tac* [!] *subsetI*, *simp-all only*: *split-paired-all*)
fix *xs X*
assume *D*: (*xs*, *X*) ∈ *failures* (*ts-process* (*traces P*))
hence *xs* ∈ *traces* (*ts-process* (*traces P*)) **by** (*rule failures-traces*)
hence *E*: *xs* ∈ *traces P* **using** *C* **by** *simp*
with *B* **have**

refusals (*ts-process* (*traces P*)) *xs* = {*X*. ∀ *x* ∈ *X*. *xs* @ [*x*] ∉ *traces P*}
by (*rule ts-process-refusals*)
moreover have *X* ∈ *refusals* (*ts-process* (*traces P*)) *xs*
using *D* **by** (*simp add*: *refusals-def*)
ultimately have ∀ *x* ∈ *X*. *xs* @ [*x*] ∉ *traces P* **by** *simp*
hence *X* ∩ *next-events P xs* = {}
by (*rule-tac equals0I*, *erule-tac IntE*, *simp add*: *next-events-def*)
moreover have ∀ *X*. (*X* ∈ *refusals P xs*) = (*X* ∩ *next-events P xs* = {})
using *A* **and** *E* **..**
hence (*X* ∈ *refusals P xs*) = (*X* ∩ *next-events P xs* = {}) **..**
ultimately have *X* ∈ *refusals P xs* **by** *simp*
thus (*xs*, *X*) ∈ *failures P* **by** (*simp add*: *refusals-def*)
next
fix *xs X*
assume *D*: (*xs*, *X*) ∈ *failures P*
hence *E*: *xs* ∈ *traces P* **by** (*rule failures-traces*)
with *A* **have** ∀ *X*. (*X* ∈ *refusals P xs*) = (*X* ∩ *next-events P xs* = {}) **..**
hence (*X* ∈ *refusals P xs*) = (*X* ∩ *next-events P xs* = {}) **..**
moreover have *X* ∈ *refusals P xs* **using** *D* **by** (*simp add*: *refusals-def*)
ultimately have *F*: *X* ∩ {*x*. *xs* @ [*x*] ∈ *traces P*} = {}
by (*simp add*: *next-events-def*)
have ∀ *x* ∈ *X*. *xs* @ [*x*] ∉ *traces P*
proof (*rule ballI*, *rule notI*)
fix *x*
assume *x* ∈ *X* **and** *xs* @ [*x*] ∈ *traces P*
hence *x* ∈ *X* ∩ {*x*. *xs* @ [*x*] ∈ *traces P*} **by** *simp*
moreover have *x* ∉ *X* ∩ {*x*. *xs* @ [*x*] ∈ *traces P*} **using** *F* **by** (*rule equals0D*)
ultimately show *False* **by** *contradiction*
qed
moreover have
refusals (*ts-process* (*traces P*)) *xs* = {*X*. ∀ *x* ∈ *X*. *xs* @ [*x*] ∉ *traces P*}
using *B* **and** *E* **by** (*rule ts-process-refusals*)
ultimately have *X* ∈ *refusals* (*ts-process* (*traces P*)) *xs* **by** *simp*
thus (*xs*, *X*) ∈ *failures* (*ts-process* (*traces P*)) **by** (*simp add*: *refusals-def*)
qed
qed

lemma *ts-process-traces-implies-d*:
ts-process (*traces P*) = *P* ⟹ *deterministic P*
by (*insert trace-set-traces* [*of P*], *drule ts-process-d*, *simp*)

lemma *d-equals-ts-process-traces*:
deterministic P = (*ts-process* (*traces P*) = *P*)
by (*rule iffI*, *erule d-implies-ts-process-traces*, *rule ts-process-traces-implies-d*)

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)
by (*rule d-ipurge-unwinding*, *rule ts-process-d*)

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.