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

April 20, 2020

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 2
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 $\forall a_1 \Rightarrow \ldots \Rightarrow \forall a_m \Rightarrow \forall a_{m+1} \Rightarrow \ldots \Rightarrow \forall a_n \Rightarrow \forall b$, the explanatory text may discuss of $F$ using attributes that would more exactly apply to a term of type $\forall a_{m+1} \Rightarrow \ldots \Rightarrow \forall a_n \Rightarrow \forall b$. 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 $[(\forall xs, \forall Y) \in failures \ ?P; \ ?X \subseteq \ ?Y] \Rightarrow (\forall xs, \ ?X) \in failures \ ?P$, so that the union of the elements of $A'$, which is equal to the union of the elements of $A$, is a refusal by hypothesis.

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

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

More remarkably, the formal differences with respect to Rushby’s original predicates are the following ones:
• The relations in the range of the domain-relation map hold between event lists rather than machine states.

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

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

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

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

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

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

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

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

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

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

definition weakly-future-consistent ::
\( 'a \) process \( \Rightarrow \) ('d \times 'd) set \( \Rightarrow \) ('a \Rightarrow 'd) \( \Rightarrow \) ('a, 'd) dom-rel-map \( \Rightarrow \) bool where
weakly-future-consistent \( P \) \( I \) \( D \) \( R \equiv \)
\( \forall u \in \text{range} \ D \cap (-I) \sim \text{range} \ D. \forall xs \ ys, (xs, ys) \in R \ u \longrightarrow \)
next-dom-events \( P \) \( D \) \( u \) \( xs \) = next-dom-events \( P \) \( D \) \( u \) \( ys \) ∧
ref-dom-events \( P \) \( D \) \( u \) \( xs \) = ref-dom-events \( P \) \( D \) \( u \) \( ys \)

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

\[ \text{lemma} \ process-rule-2-traces: \]
\[ (xs, X) \in \text{failures} \implies (xs \ @ \ [x], \ {\}) \in \text{failures} \lor (xs, \ \text{insert} \ x \ X) \in \text{failures} \]
\[ \text{proof (simp add: traces-def Domain-iff, erule exE, rule-tac x = {} in exI)} \]
\[ \text{qed (rule process-rule-2-failures)} \]

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

\[ \text{lemma} \ failures-traces: \]
\[ (xs, X) \in \text{failures} \longrightarrow xs \in \text{traces} \]
\[ \text{by (simp add: traces-def Domain-iff, rule exI)} \]

\[ \text{lemma} \ traces-failures: \]
\[ xs \in \text{traces} \longrightarrow (xs, \ {\}) \in \text{failures} \]
\[ \text{proof (simp add: traces-def Domain-iff, rule exI)} \]
\[ \text{qed (rule process-rule-3, simp)} \]

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

5
moreover \{ 
  assume \( C; D x = D x' \)
  have \(?thesis\) using True
  proof (rule disjE, erule-tac [2] bexE)
    assume \((u, D x') \in I\)
    hence \((a, D x) \in I\) using \(C\) by simp
    thus \(?thesis\) ..
  next
    fix \( v \)
    assume \((v, D x') \in I\)
    hence \((v, D x) \in I\) using \(C\) by simp
    moreover assume \( v \in \text{sinks } I D u xs \)
    hence \( v \in \text{sinks } I D u (xs @ [x']) \) by simp
    ultimately have \( \exists v \in \text{sinks } I D u (xs @ [x']), (v, D x) \in I \) ..
    thus \(?thesis\) ..
  qed
\}
moreover \{ 
  assume \( D x \in \text{sinks } I D u xs \)
  with \( A \) have \((u, D x) \in I \lor (\exists v \in \text{sinks } I D u xs. (v, D x) \in I) \) ..
  hence \(?thesis\)
  proof (rule disjE, erule-tac [2] bexE)
    assume \((u, D x) \in I\)
    thus \(?thesis\) ..
  next
    fix \( v \)
    assume \((v, D x) \in I\)
    moreover assume \( v \in \text{sinks } I D u xs \)
    hence \( v \in \text{sinks } I D u (xs @ [x']) \) by simp
    ultimately have \( \exists v \in \text{sinks } I D u (xs @ [x']), (v, D x) \in I \) ..
    thus \(?thesis\) ..
  qed
\}
ultimately show \(?thesis\) ..
next
  case False
  hence \( C; \text{sinks } I D u (xs @ [x']) = \text{sinks } I D u xs \) by simp
  hence \( D x \in \text{sinks } I D u xs \) using \( B \) by simp
  with \( A \) have \((u, D x) \in I \lor (\exists v \in \text{sinks } I D u xs. (v, D x) \in I) \) ..
  thus \(?thesis\) using \(C\) by simp
qed

lemma sinks-interference-eq:
\((u, D x) \in I \lor (\exists v \in \text{sinks } I D u xs. (v, D x) \in I)\) =
\((D x \in \text{sinks } I D u (xs @ [x']))\)
qed (erule contrapos-nn, rule sinks-interference)
In what follows, some lemmas concerning the constants defined above are proved.

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 \( A : \forall x. (\forall X \in A. (x, X) \in \text{failures } P) \rightarrow (x, \bigcup X \in A. X) \in \text{failures } P \)

shows \( \forall x. x \in \text{traces } P \)

**Proof**

fix \( x \)

have \( (\forall X \in \{\}. (x, X) \in \text{failures } P) \rightarrow (x, \bigcup X \in \{\}. X) \in \text{failures } P \)

using \( A \) by blast

moreover have \( (x, X) \in \text{failures } P \) by simp

ultimately have \( (x, \bigcup X \in \{\}. X) \in \text{failures } P \).

thus \( x \in \text{traces } P \) by (rule failures-traces)

qed

**Lemma** \( \text{traces-dom-events} : \)

assumes \( A : \forall x. A. (\forall X \in A. (x, X) \in \text{failures } P) \rightarrow (x, \bigcup X \in A. X) \in \text{failures } P \)

shows \( x \in \text{traces } P = (\text{next-dom-events } P D u x \cup \text{ref-dom-events } P D u x \neq \{\}) \)

is \( x = (\forall S \neq \{\}) \)

**Proof**

have \( \exists x. u = D x \) using \( A \) by (simp add: image-def)

then obtain \( x \) where \( B : u = D x \).

assume \( x \in \text{traces } P \)

hence \((x, \{\}) \in \text{failures } P \) by (rule traces-failures)

hence \((x \oplus [x], \{\}) \in \text{failures } P \lor (x, \{x\}) \in \text{failures } P \) by (rule process-rule-4)

moreover {

assume \((x \oplus [x], \{\}) \in \text{failures } P \)

hence \((x \oplus [x], \{\}) \in \text{traces } P \) by (rule failures-traces)

hence \( x \in \text{next-dom-events } P D u x \)

using \( B \) by (simp add: next-dom-events-def next-events-def)

hence \( x \in \forall S \).

}\)

moreover {

assume \((x, \{x\}) \in \text{failures } P \)
hence \( x \in \text{ref-dom-events} \ P \ D \ u \ \text{xs} \)
using \( B \) by (simp add: ref-dom-events-def refusals-def)

hence \( x \in ?S \).

\}
ultimately have \( x \in ?S \).

hence \( \exists x. \ x \in ?S \).

thus \( ?S \neq \{} \) by (subst ex-in-conv [symmetric])

next
assume \( ?S \neq \{} \)

hence \( \exists x. \ x \in ?S \) by (subst ex-in-conv)

then obtain \( x \) where \( x \in ?S \).

moreover {
assume \( x \in \text{next-dom-events} \ P \ D \ u \ \text{xs} \)

hence \( \text{xs} @ [x] \in \text{traces} \ P \) by (simp add: next-dom-events-def next-events-def)

hence \( \text{xs} \in \text{traces} \ P \) by (rule process-rule-2-traces)

}

moreover {
assume \( x \in \text{ref-dom-events} \ P \ D \ u \ \text{xs} \)

hence \( (\text{xs}, \{x\}) \in \text{failures} \ P \) by (simp add: ref-dom-events-def refusals-def)

hence \( \text{xs} \in \text{traces} \ P \) by (rule failures-traces)

}

ultimately show \( \text{xs} \in \text{traces} \ P \).

qed

lemma fc-traces:

assumes
\( A: \text{future-consistent} \ P \ D \ R \) and
\( B: \ u \in \text{range} \ D \) and
\( C: (\text{xs}, \text{ys}) \in R \ u \)

shows \( (\text{xs} \in \text{traces} \ P) = (\text{ys} \in \text{traces} \ P) \)

proof –

have \( \forall u \in \text{range} \ D. \forall \text{xs} \text{ ys}. \ (\text{xs}, \text{ys}) \in R \ u \longrightarrow \)
next-dom-events \( P \ D \ u \ \text{xs} = \text{next-dom-events} \ P \ D \ u \ \text{ys} \land \)
ref-dom-events \( P \ D \ u \ \text{xs} = \text{ref-dom-events} \ P \ D \ u \ \text{ys} \land \)
using \( A \) by (simp add: future-consistent-def)

hence \( \forall \text{xs} \ \text{ys}. \ (\text{xs}, \text{ys}) \in R \ u \longrightarrow \)
next-dom-events \( P \ D \ u \ \text{xs} = \text{next-dom-events} \ P \ D \ u \ \text{ys} \land \)
ref-dom-events \( P \ D \ u \ \text{xs} = \text{ref-dom-events} \ P \ D \ u \ \text{ys} \land \)
using \( B \).

hence \( (\text{xs}, \text{ys}) \in R \ u \longrightarrow \)
next-dom-events \( P \ D \ u \ \text{xs} = \text{next-dom-events} \ P \ D \ u \ \text{ys} \land \)
ref-dom-events \( P \ D \ u \ \text{xs} = \text{ref-dom-events} \ P \ D \ u \ \text{ys} \land \)
by blast

hence \( \text{next-dom-events} \ P \ D \ u \ \text{xs} = \text{next-dom-events} \ P \ D \ u \ \text{ys} \land \)
ref-dom-events \( P \ D \ u \ \text{xs} = \text{ref-dom-events} \ P \ D \ u \ \text{ys} \land \)
using \( C \).

hence \( \text{next-dom-events} \ P \ D \ u \ \text{xs} \cup \text{ref-dom-events} \ P \ D \ u \ \text{xs} \neq \{} \) =
(\text{next-dom-events} \ P \ D \ u \ \text{ys} \cup \text{ref-dom-events} \ P \ D \ u \ \text{ys} \neq \{} \)
by simp
moreover have \( xs \in \text{traces } P = \) 
\((\text{next-dom-events } P \ D \ u \ xs \cup \text{ref-dom-events } P \ D \ u \ xs \neq \{\})\) 
using \( B \) by (rule traces-dom-events) 

moreover have \( ys \in \text{traces } P = \) 
\((\text{next-dom-events } P \ D \ u \ ys \cup \text{ref-dom-events } P \ D \ u \ ys \neq \{\})\) 
using \( B \) by (rule traces-dom-events) 
ultimately show \(?thesis by simp\)

qed

lemma wfc-traces:
  assumes 
  \( A: \text{ weakly-future-consistent } P \ I \ D \ R \) and 
  \( B: u \in \text{range } D \cap (-I) \) " range \( D \) and 
  \( C: (xs, ys) \in R \ u \) 
  shows \((xs \in \text{traces } P) = (ys \in \text{traces } P)\) 
proof 
  have \( \forall u \in \text{range } D \cap (-I) \) " range \( D \). \( \forall xs \ ys. (xs, ys) \in R \ u \longrightarrow \) 
  \( \text{next-dom-events } P \ D \ u \ xs = \text{next-dom-events } P \ D \ u \ ys \land \) 
  \( \text{ref-dom-events } P \ D \ u \ xs = \text{ref-dom-events } P \ D \ u \ ys \) 
  using \( A \) by (simp add: weakly-future-consistent-def) 
  hence \( \forall xs \ ys. (xs, ys) \in R \ u \longrightarrow \) 
  \( \text{next-dom-events } P \ D \ u \ xs = \text{next-dom-events } P \ D \ u \ ys \land \) 
  \( \text{ref-dom-events } P \ D \ u \ xs = \text{ref-dom-events } P \ D \ u \ ys \) 
  using \( B \) .. 
  hence \( (xs, ys) \in R \ u \longrightarrow \) 
  \( \text{next-dom-events } P \ D \ u \ xs = \text{next-dom-events } P \ D \ u \ ys \land \) 
  \( \text{ref-dom-events } P \ D \ u \ xs = \text{ref-dom-events } P \ D \ u \ ys \) 
  by blast 
  hence \( \text{next-dom-events } P \ D \ u \ xs = \text{next-dom-events } P \ D \ u \ ys \land \) 
  \( \text{ref-dom-events } P \ D \ u \ xs = \text{ref-dom-events } P \ D \ u \ ys \) 
  using \( C \) .. 
  hence \( \text{next-dom-events } P \ D \ u \ xs \cup \text{ref-dom-events } P \ D \ u \ xs \neq \{\} = \) 
  \( (\text{next-dom-events } P \ D \ u \ ys \cup \text{ref-dom-events } P \ D \ u \ ys \neq \{\})\) 
  by simp 
  moreover have \( B': u \in \text{range } D \) using \( B \) .. 
  hence \( xs \in \text{traces } P = \) 
  \( (\text{next-dom-events } P \ D \ u \ xs \cup \text{ref-dom-events } P \ D \ u \ xs \neq \{\})\) 
  by (rule traces-dom-events) 
  moreover have \( ys \in \text{traces } P = \) 
  \( (\text{next-dom-events } P \ D \ u \ ys \cup \text{ref-dom-events } P \ D \ u \ ys \neq \{\})\) 
  using \( B' \) by (rule traces-dom-events) 
  ultimately show \(?thesis by simp\)
qed

lemma fc-implies-wfc:
  future-consistent \( P \ D \ R \Longrightarrow \text{ weakly-future-consistent } P \ I \ D \ R \) 
by (simp only: future-consistent-def weakly-future-consistent-def, blast)
Finally, the definition is given of an auxiliary function \textit{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.

\textbf{definition} \textit{singleton-set} :: 'a set ⇒ 'a set set
\textit{singleton-set} \textit{X} \equiv \{ \textit{Y}. \exists \textit{x} \in \textit{X}. \textit{Y} = \{ \textit{x} \} \}

\textbf{lemma} \textit{singleton-set-some}:
(∃ \textit{Y}. \textit{Y} ∈ \textit{singleton-set} \textit{X}) = (∃ \textit{x}. \textit{x} ∈ \textit{X})
\textbf{proof} (rule \textit{iffI}, \textit{simp-all add}: \textit{singleton-set-def}, erule_tac \textit{[]} \textit{exE}, erule bexE)
fix \textit{x}
assume \textit{x} ∈ \textit{X}
thus ∃ \textit{x}. \textit{x} ∈ \textit{X} ..
next
fix \textit{x}
assume \textit{A}: \textit{x} ∈ \textit{X}
have \{ \textit{x} \} = \{ \textit{x} \} ..
hence ∃ \textit{x′} ∈ \textit{X}. \{ \textit{x} \} = \{ \textit{x′} \} using \textit{A} ..
thus ∃ \textit{Y}. ∃ \textit{x′} ∈ \textit{X}. \textit{Y} = \{ \textit{x′} \} by (rule \textit{exI})
qed

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

1.2 Additional intransitive purge functions and their properties

Functions \textit{sinks-aux}, \textit{ipurge-tr-aux}, and \textit{ipurge-ref-aux}, defined here below, are auxiliary versions of functions \textit{sinks}, \textit{ipurge-tr}, and \textit{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
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 ::**

\[
\text{('d} \times \text{'d}) \text{ set } \Rightarrow \text{('}a \Rightarrow \text{'d}) \Rightarrow \text{'d set } \Rightarrow \text{'a list } \Rightarrow \text{'a list where}
\]

\[
sinks-aux \ - \ - \ U \ [ \ ] = U \ |
\]

\[
sinks-aux I D U (xs @ \ [x]) = (\text{if } \exists v \in sinks-aux I D U xs. (v, D x) \in I \then insert (D x) (sinks-aux I D U xs) \else sinks-aux I D U xs)
\]

**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 ::**

\[
\text{('d} \times \text{'d}) \text{ set } \Rightarrow \text{('}a \Rightarrow \text{'d}) \Rightarrow \text{'d set } \Rightarrow \text{'a list } \Rightarrow \text{'a list where}
\]

\[
ipurge-tr-aux \ - \ - \ - \ [ \ ] = [ \ ] |
\]

\[
ipurge-tr-aux I D U (xs @ \ [x]) = (\text{if } \exists v \in sinks-aux I D U xs. (v, D x) \in I \then ipurge-tr-aux I D U xs \else ipurge-tr-aux I D U xs @ \ [x])
\]

**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 ::**

\[
\text{('d} \times \text{'d}) \text{ set } \Rightarrow \text{('}a \Rightarrow \text{'d}) \Rightarrow \text{'d set } \Rightarrow \text{'a list } \Rightarrow \text{'a set where}
\]

\[
\{ x \in X. \forall v \in sinks-aux I D U xs. (v, D x) \notin I \}
\]

**definition unaffected-domains ::**

\[
\text{('d} \times \text{'d}) \text{ set } \Rightarrow \text{('}a \Rightarrow \text{'d}) \Rightarrow \text{'d set } \Rightarrow \text{'a list } \Rightarrow \text{'d set where}
\]

\[
\text{unaffected-domains I D U xs } \equiv \{ u \in \text{range D. } \forall v \in sinks-aux I D U xs. (v, u) \notin I \}
\]

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 e-sources and e-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.

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

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

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

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

Here below are some lemmas on functions sinks-aux, ipurge-tr-aux, ipurge-ref-aux, and unaffected-domains. As anticipated above, these lemmas essentially concern distributivity over list concatenation and expressions in terms of single domain functions in the degenerate case of a singleton set of domains.
lemma sinks-aux-subset:
\[ U \subseteq \text{sinks-aux I D U} \]
proof (induction \( xs \) rule: rev-induct, simp-all, rule impI)
qed (rule subset-insertI2)

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

lemma sinks-aux-single-event:
\[ \text{sinks-aux I D U} [x] = (\text{if } \exists v \in U. (v, D x) \in I \ \text{then insert} \ (D x) \ U \ \text{else U}) \]
proof –
have \( \text{sinks-aux I D U} [x] = \text{sinks-aux I D U} ([\ ] @ [x]) \) by simp
thus ?thesis by (simp only: sinks-aux.simps)
qed

lemma sinks-aux-cons:
\[ \text{sinks-aux I D U} (x \# \text{xs}) = (\text{if } \exists v \in U. (v, D x) \in I \ \text{then sinks-aux I D (insert} \ (D x) \ U) \ \text{xs} \ \text{else sinks-aux I D U} \ \text{xs}) \]
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 \( x' \) \( \text{xs} \)
assume \( A: \text{sinks-aux I D U} (x \# \text{xs}) = \text{sinks-aux I D (insert} \ (D x) \ U) \ \text{xs} \)
(is \( ?S = ?S' \))
show \( \text{sinks-aux I D U} (x \# \text{xs} @ [x']) = \text{sinks-aux I D (insert} \ (D x) \ U) (\text{xs} @ [x']) \)
proof (cases \( \exists v \in ?S. (v, D x') \in I \))
case True
hence \( \text{sinks-aux I D U} ((x \# \text{xs} @ [x']) = \text{insert} \ (D x') \ ?S \)
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 \( \text{sinks-aux I D (insert} \ (D x) \ U) (\text{xs} @ [x']) = \text{insert} \ (D x') \ ?S' \)
by simp
ultimately show ?thesis using \( A \) by simp
next
case False
hence \( \text{sinks-aux I D U} ((x \# \text{xs} @ [x']) = ?S \)
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 \( \text{sinks-aux I D (insert} \ (D x) \ U) (\text{xs} @ [x']) = ?S' \) by simp
ultimately show ?thesis using \( A \) by simp
qed
next
fix \( x' \) \( \text{xs} \)
assume \( A: \text{sinks-aux I D U} (x \# \text{xs}) = \text{sinks-aux I D U} \ \text{xs} \)
(is \( ?S = ?S' \))
show \( \text{sinks-aux } I D U \ (x \ # \ xs \ @ [x']) = \text{sinks-aux } I D U \ (xs \ @ [x']) \)

proof (cases \( \exists v \in S. (v, D x') \in I \))

---

next

---

qed

qed

lemma \( \text{ipurge-tr-aux-single-dom} \):

\( \text{ipurge-tr-aux } I D \ {u} \ \{u\} \ xs = \text{ipurge-tr } I D u \ xs \)

proof (induction \( xs \ \text{rule: rev-induct, simp})

---

qed

qed

lemma \( \text{ipurge-ref-aux-single-dom} \):

\( \text{ipurge-ref-aux } I D \ \{u\} \ xs X = \text{ipurge-ref } I D u \ xs X \)

by (simp add: \( \text{ipurge-ref-aux-def} \ \text{ipurge-ref-def} \ \text{sinks-aux-single-dom} \))

lemma \( \text{ipurge-ref-aux-all} \ [\text{rule-format}] \):
\[(\forall u \in U. \neg (\exists v \in D \cdot (X \cup \text{set } xs). (u, v) \in I)) \rightarrow ipurge-ref-aux I D U xs X = X\]

**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:
\(\neg (\exists v \in D \cdot (X \cup \text{set } xs). (u, v) \in I) \rightarrow ipurge-ref I D u xs X = X\)

**by** (subst ipurge-ref-aux-single-dom [symmetric], rule ipurge-ref-aux-all, simp)

**lemma** unaffected-domains-single-dom:
\(\{x \in X. D x \in \text{unaffected-domains } I D \{u\} xs\} = ipurge-ref I D u xs X\)

**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:
\(\text{sources } I D u xs = \text{sinks } (I^{-1}) D u (\text{rev } xs)\)

**by** (induction xs, simp-all)

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

**by** (induction xs, simp-all)

**lemma** sources-aux-subset:
\(U \subseteq \text{sources-aux } I D U xs\)

**by** (subst sources-sinks-aux, rule sinks-aux-subset)

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

**by** (induction xs, simp-all)

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

**by** (induction xs, simp-all)

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

**by** (induction xs, simp-all add: sources-aux-append)

**lemma** ipurge-tr-rev-aux-nil-I [rule-format]:
\(\text{ipurge-tr-rev-aux } I D U xs = [] \rightarrow (\forall u \in U. \neg (\exists v \in D \cdot \text{set } xs. (v, u) \in I))\)
by (induction xs rule: rev-induct, simp-all add: ipurge-tr-rev-aux-append)

lemma ipurge-tr-rev-aux-nil-2 [rule-format]:
$(\forall u \in U. \neg (\exists v \in D \setminus set xs. (v, u) \in I)) \rightarrow \text{ipurge-tr-rev-aux I D U xs} = []$
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 = []) = $(\forall u \in U. \neg (\exists v \in D \setminus set xs. (v, u) \in I))$
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 = [] \rightarrow \text{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 = [] \rightarrow 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 \rightarrow
$(\exists ys zs. xs = ys @ x # zs \land$
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] \land
$(\exists v \in sources-aux I D U zs. (D x, v) \in I))$
proof (induction xs, simp, rule impI)
fix $x'$ xs
assume A: ipurge-tr-rev-aux I D U xs = x # ws \rightarrow
$(\exists ys zs. xs = ys @ x # zs \land$
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] \land
$(\exists v \in sources-aux I D U zs. (D x, v) \in I))$
and B: ipurge-tr-rev-aux I D U (x' # xs) = x # ws
show $\exists ys zs. x' # xs = ys @ x # zs \land$
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] \land
$(\exists v \in sources-aux I D U zs. (D x, v) \in I)$
proof (cases $\exists v \in sources-aux I D U zs. (D x', v) \in I$)
case True
then have $x' = x$ using B by simp
with True have $x' # xs = x # zs \land$
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) [] = [] \land
$(\exists v \in sources-aux I D U zs. (D x, v) \in I)$
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 $\exists ys zs. xs = ys @ x # zs \land$
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] \land
proof

lemma ipurge-tr-rev-aux-last-1 [rule-format]:
ipurge-tr-rev-aux I D U xs = ws @ [x] \rightarrow (\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] \rightarrow (\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] \rightarrow 
(\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: \text{ipurge-tr-rev-aux} \, I \, D \, U \, \underline{xs} = \underline{ws} \, @ \, [x] \quad \rightarrow \\
(\exists \, y @ s . \, s = \underline{ys} \, @ \, x \, # \, s \, \land \, \text{ipurge-tr-rev-aux} \, I \, D \, U \, \underline{zs} = [] ) \quad \text{and} \\
B: \text{ipurge-tr-rev-aux} \, I \, D \, U \, (\underline{xs} \, @ \, [x']) = \underline{ws} \, @ \, [x] \\
prove \; (cases \; \exists \, v \, \in \, U . \, (D \, x', \, v) \, \in \, I ) \\
\text{case} \; \text{True} \\
\text{hence} \; \text{ipurge-tr-rev-aux} \, I \, D \, U \, (\underline{xs} \, @ \, [x']) = \\
\text{ipurge-tr-rev-aux} \, I \, D \, (\text{insert} \, (D \, x') \, U) \, \underline{xs} \, @ \, [x'] \\
\text{by} \; (\text{simp add: ipurge-tr-rev-aux-append}) \\
\text{hence} \; \underline{xs} \, @ \, [x'] = \underline{ys} \, @ \, x \, # \, zs \land \text{ipurge-tr-rev-aux} \, I \, D \, U \, [] = [] \\
\text{using} \; B \; \text{by simp} \\
\text{thus} \; ?\text{thesis by blast} \\
\text{next} \\
\text{case} \; \text{False} \\
\text{hence} \; \text{ipurge-tr-rev-aux} \, I \, D \, U \, (\underline{xs} \, @ \, [x']) = \text{ipurge-tr-rev-aux} \, I \, D \, U \, \underline{xs} \\
\text{by} \; (\text{simp add: ipurge-tr-rev-aux-append}) \\
\text{hence} \; \text{ipurge-tr-rev-aux} \, I \, D \, U \, (\underline{xs} \, @ \, [x']) = \underline{ys} \, @ \, x \, # \, zs \land \text{ipurge-tr-rev-aux} \, I \, D \, U \, [] = [] \\
\text{using} \; B \; \text{by simp} \\
\text{with} \; A \; \text{have} \; (\exists \, y @ s . \, s = \underline{ys} \, @ \, x \, # \, zs \land \text{ipurge-tr-rev-aux} \, I \, D \, U \, [] = [] ) \\
\text{then obtain} \; y @ s \, \text{and} \, zs \, \text{where} \\
\text{C:} \; \underline{xs} = y @ s \, @ \, x \, # \, zs \land \text{ipurge-tr-rev-aux} \, I \, D \, U \, zs = [] \\
\text{by blast} \\
\text{hence} \; \underline{xs} \, @ \, [x'] = y @ s \, @ \, x \, # \, zs \, @ \, [x'] \quad \text{by simp} \\
\text{moreover have} \\
\text{ipurge-tr-rev-aux} \, I \, D \, U \, (zs \, @ \, [x']) = \text{ipurge-tr-rev-aux} \, I \, D \, U \, zs \\
\text{using} \; False \; \text{by} \; (\text{simp add: ipurge-tr-rev-aux-append}) \\
\text{hence} \; \text{ipurge-tr-rev-aux} \, I \, D \, U \, (zs \, @ \, [x']) = [] \quad \text{using} \; C \; \text{by simp} \\
\text{ultimately have} \; \underline{xs} \, @ \, [x'] = y @ s \, @ \, x \, # \, zs \, @ \, [x'] \land \\
\text{ipurge-tr-rev-aux} \, I \, D \, U \, (zs \, @ \, [x']) = [] \\
\text{thus} \; ?\text{thesis by blast} \\
\text{qed} \\
\text{qed} \\
\text{lemma} \, \text{ipurge-tr-rev-aux-all} \quad [\text{rule-format}]: \\
(\forall \, v \, \in \, D \, ' \, \text{set} \, s . \, \exists \, u \, \in \, U . \, (v, \, u) \, \in \, I ) \; \rightarrow \; \text{ipurge-tr-rev-aux} \, I \, D \, U \, \underline{xs} = \underline{xs} \\
\text{proof} \; (\text{induction} \, \underline{xs} , \, \text{simp} , \, \text{rule impI} , \, \text{simp} , \, \text{erule conjE} ) \\
\text{fix} \, x \, \underline{xs} \\
\text{assume} \; (\exists \, u \, \in \, U . \, (D \, x, \, u) \, \in \, I ) \\
\text{then obtain} \; u \, \text{where} \; A: \; u \, \in \, U \, \text{and} \; B: \; (D \, x, \, u) \, \in \, I \\
\text{have} \; U \subseteq \text{sources-aux} \, I \, D \, U \, \underline{xs} \, \text{by} \; (\text{rule sources-aux-subset}) \\
\text{hence} \; u \, \in \, \text{sources-aux} \, I \, D \, U \, \underline{xs} \, \text{using} \; A \\
\text{with} \; B \, \text{show} \; (\exists \, u \, \in \, \text{sources-aux} \, I \, D \, U \, \underline{xs} . \, (D \, x, \, u) \, \in \, I ). \\
\text{qed} \\
\text{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 impl)
  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 # zs 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 (n + m) x xs = Some (the (offset n x xs) + m)
proof (induction xs arbitrary: k n, simp, rule implI)
fix w xs k n
assume
A: \( \forall n. \) 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 \( \ldots = \) Some (the (offset n x xs) + Suc 0) using A by (rule offset-addition)
also have \( \ldots = \) Some (Suc (the (offset n x xs))) by simp
finally show \(?thesis\).
qed

lemma ipurge-tr-rev-aux-first-offset [rule-format]:
xxs = ys @ x \# zs \land ipurge-tr-rev-aux I D (sources-aux I D U (x \# zs)) ys = [] \land
(\( \exists v \in \) sources-aux I D U zs. (D x, v) \( \in \) I) \( \longrightarrow \)
yxs = take (the (offset 0 x xs)) xxs
proof (induction xs arbitrary: ys, simp, rule implI, (erule conjE)+)
fix x' xs ys
assume
A: \( \forall yxs. \) xxs = ys @ x \# zs \land
ipurge-tr-rev-aux I D (sources-aux I D U (x \# zs)) ys = [] \land
(\( \exists v \in \) sources-aux I D U zs. (D x, v) \( \in \) I) \( \longrightarrow \)
yxs = take (the (offset 0 x xs)) xxs and
B: x' \# xxs = 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 yxs = take (the (offset 0 x (x' \# xxs))) (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
    $G$: $\neg (\exists v \in sources-aux I D (sources-aux I D U (x # zs)) ys', (D y, v) \in I)$
    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 $xs = ys' @ x # zs$
  with A have $H$: $y = \text{take (the (offset 0 x xs))} xs$ ..
  hence $I$: $x' = y$ using Cons and B by simp
  hence
    $J$: $\neg (\exists v \in sources-aux I D (sources-aux I D U zs) (ys' @ [x]), (D x', v) \in I)$
    using G by (simp add: sources-aux-append)
    have $x' \neq x$
    proof
      assume $x' = x$
      hence $\exists v \in sources-aux I D U zs. (D x', v) \in I$ using D by simp
      then obtain v where $K$: $v \in sources-aux I D U zs$ and $L$: $(D x', v) \in I$ ..
      have sources-aux I D U zs $\subseteq$
        sources-aux I D (sources-aux I D U zs) (ys' @ [x])
        by (rule sources-aux-subset)
      hence $v \in sources-aux I D (sources-aux I D U zs) (ys' @ [x])$ using K ..
      with L have
        $\exists v \in sources-aux I D (sources-aux I D U zs) (ys' @ [x]). (D x', v) \in I$ ..
      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 $\exists ys, zs = ys @ x @ zs$ using E by blast
    hence offset 0 x xs $\neq$ 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
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 \rightarrow
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 \rightarrow
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 \( \exists v \in \text{sources-aux I D V xs} \). (D x, v) \in 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 \( \exists vs ws. x # xs @ ys = vs @ x # ws \wedge \)
ipurge-tr-rev-aux I D (sources-aux I D U (x # ws)) vs = [] \wedge
(\( \exists v \in \text{sources-aux I D U ws}. (D x, v) \in I \))
by (rule ipurge-tr-rev-aux-first)
then obtain vs and ws where *: x # xs @ ys = vs @ x # ws \wedge
ipurge-tr-rev-aux I D (sources-aux I D U (x # ws)) vs = [] \wedge
(\( \exists v \in \text{sources-aux I D U ws}. (D x, v) \in I \))
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 \( \exists v \in \text{sources-aux I D U (xs @ ys)} . (D x, v) \in I \) 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 \( \neg (\exists v \in \text{sources-aux I D U (xs @ ys)} . (D x, v) \in I) \)

proof
assume \( \exists v \in \text{sources-aux I D U (xs @ ys)} . (D x, v) \in I \)
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 \( \exists vs ws. x # xs = vs @ x # ws \wedge \)
ipurge-tr-rev-aux I D (sources-aux I D V (x # ws)) vs = [] \wedge
(\( \exists v \in \text{sources-aux I D V ws}. (D x, v) \in I \))
by (rule ipurge-tr-rev-aux-first)
then obtain vs and ws where *: x # xs = vs @ x # ws \wedge
ipurge-tr-rev-aux I D (sources-aux I D V (x # ws)) vs = [] \wedge
(\( \exists v \in \text{sources-aux I D V ws}. (D x, v) \in I \))
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 \( \exists v \in \text{sources-aux } I D V xs. (D x, v) \in I \) 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

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:

\( xs \equiv \{ \text{ipurge-tr-rev } I D u xs, \text{rev (ipurge-tr } (I^{-1}) D u (\text{rev } xs)), \lambda y ys. D y \in \text{sinks } (I^{-1}) D u (\text{rev } (y \neq y))) \} \)

proof (induction xs, simp, simp only: rev.simps)

fix x xs

assume A: \( xs \equiv \{ \text{ipurge-tr-rev } I D u xs, \text{rev (ipurge-tr } (I^{-1}) D u (\text{rev } xs)), \lambda y ys. D y \in \text{sinks } (I^{-1}) D u (\text{rev } ys @ [y])\} \)

(is - \equiv \{ ?ys, ?zs, ?P\})

show \( x \neq xs \equiv \{ \text{ipurge-tr-rev } I D u (x \neq xs), \text{rev (ipurge-tr } (I^{-1}) D u (\text{rev xs } @ [x])), ?P\} \)

proof (cases ?P x xs, simp-all add: sources-sinks del: sinks.simps)

\text{case True}

thus \( x \neq xs \equiv \{ x \neq ?ys, ?zs, ?P\} \) using A by (cases ?zs, simp-all)

next

\text{case False}

thus \( x \neq xs \equiv \{ ?ys, x \neq ?zs, ?P\} \) using A by (cases ?ys, simp-all)

qed

lemma Interleaves-ipurge-tr-aux:

\( xs \equiv \{ \text{ipurge-tr-rev-aux } I D U xs, \text{rev (ipurge-tr-aux } (I^{-1}) D U (\text{rev } xs)), \lambda y ys. \exists v \in \text{sinks-aux } (I^{-1}) D U (\text{rev } ys). (D y, v) \in I\} \)

23
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)), 
\\ldots \}
  (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 (no-asmp-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

lemma ipurge-tr-aux-all:
(ipurge-tr-aux I D U xs = xs) = (\forall u \in U. \neg (\exists v \in D \setminus set xs. (u, v) \in I))

proof –
  have A: rev xs ≡ \{ipurge-tr-rev-aux (I⁻¹) D U (rev xs), 
  rev (ipurge-tr-aux (I⁻¹) D U (rev xs)), 
  \ldots \}
  (is : ≡ \{\ldots\})
  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), ?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 \forall u \in U. \neg (\exists v \in D \setminus set xs. (u, v) \in I)
  by (simp add: ipurge-tr-rev-aux-nil)
next
  assume \forall u \in U. \neg (\exists v \in D \setminus set xs. (u, v) \in 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), [], \lambda w ws. \neg \?P w ws\}
  by (subgoal asm) interleaves-swap
  moreover have rev xs ≪= \{rev xs, [], \lambda w ws. \neg \?P w ws\}
  by (rule Interleaves-interleaves-nil)
ultimately have \( \text{rev} \ (\text{ipurge-tr-aux} \ I \ D \ U \ xs) = \text{rev} \ xs \) by (rule interleaves-equal-fst)
thus \( \text{ipurge-tr-aux} \ I \ D \ U \ xs = \xs \) by simp
qed

lemma ipurge-tr-rev-aux-single-dom:
\[ \text{ipurge-tr-rev-aux} \ I \ D \ \{u\} \ xs = \text{ipurge-tr-rev} \ I \ D \ u \ xs \quad (\text{is } ?ys = ?ys') \]
proof
\begin{itemize}
  \item have \( xs \cong \{ ?ys, \text{rev} \ (\text{ipurge-tr-aux} \ (I^{-1}) \ D \ \{u\} \ (\text{rev} \ xs)) \}, \)
  \( \lambda y \ ys. \exists v \in \text{sinks-aux} \ (I^{-1}) \ D \ \{u\} \ (\text{rev} \ ys). \ (D y, v) \in I \) \)
  by (rule interleaves-ipurge-tr-aux)
  hence \( xs \cong \{ ?ys, \text{rev} \ (\text{ipurge-tr} \ (I^{-1}) \ D \ u \ (\text{rev} \ xs)) \), \)
  \( \lambda y \ ys. \ (u, D y) \in I^{-1} \lor \exists v \in \text{sinks} \ (I^{-1}) \ D \ u \ (\text{rev} \ ys). \ (v, D y) \in I^{-1} \}) \)
  by (simp add: ipurge-tr-aux-single-dom sinks-aux-single-dom)
  hence \( xs \cong \{ ?ys, \text{rev} \ (\text{ipurge-tr} \ (I^{-1}) \ D \ u \ (\text{rev} \ xs)) \), \)
  \( \lambda y \ ys. \ D y \in \text{sinks} \ (I^{-1}) \ D \ u \ (\text{rev} \ (y \# \ ys)) \)
  \( \text{is } \cdot \cong \{ -, \ ?zs, \ ?P \} \)
  by (simp only: sinks-interference-eq, simp)
moreover have \( xs \cong \{ ?ys', ?zs, ?P \} \) by (rule Interleaves-ipurge-tr)
ultimately show \( ?\text{thesis} \) by (rule Interleaves-equal-fst)
qed

lemma ipurge-tr-all:
\( (\text{ipurge-tr} \ I \ D \ u \ xs = \xs) = (\neg (\exists v \in D \cdot \text{set} \ xs. \ (u, v) \in I)) \)
by (subst ipurge-tr-aux-single-dom [symmetric], simp add: ipurge-tr-aux-all)

lemma ipurge-tr-rev-all:
\( \forall v \in D \cdot \text{set} \ xs. \ (v, u) \in I \Longrightarrow \text{ipurge-tr-rev} \ I \ D \ u \ xs = \xs \)
proof (subst ipurge-tr-rev-aux-single-dom [symmetric], rule ipurge-tr-rev-aux-all)
qed (simp (no-asn-simp))

1.3 A domain-relation map based on intransitive purge

In what follows, constant \( \text{rel-ipurge} \) 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 \( \text{ipurge-tr-rev} \ I \ D \ u \) are equal, viz. whose events affecting \( u \) are the same.

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

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

**Definition** $rel-ipurge ::$

'$a$ process $\Rightarrow (d \times d)$ set $\Rightarrow (a \Rightarrow d)$ dom-rel-map where

$rel-ipurge \ P \ I \ D \ u \equiv \{(xs, ys). \ xs \in \text{traces} \ P \land \ ys \in \text{traces} \ P \land \ ipurge-tr-rev \ I \ D \ u \ xs = ipurge-tr-rev \ I \ D \ u \ ys\}$

**Definition** $rel-ipurge-aux ::$

'$a$ process $\Rightarrow (d \times d)$ set $\Rightarrow (a \Rightarrow d)$ domset-rel-map where

$rel-ipurge-aux \ P \ I \ D \ U \equiv \{(xs, ys). \ xs \in \text{traces} \ P \land \ ys \in \text{traces} \ P \land \ ipurge-tr-rev-aux \ I \ D \ U \ xs = ipurge-tr-rev-aux \ I \ D \ U \ ys\}$

**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 equiv)

fix $u$

show refl-on (traces $P$) (rel-ipurge $P \ I \ D \ u$)

**Proof** (rule refl-onI, simp-all add: rel-ipurge-def)

qed (rule subsetI, simp add: split-paired-all)

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

$A$: $\forall u \in \text{range} \ D \cap (-I) \ " \text{range} \ D, \ \forall xs \ ys. \ (xs, ys) \in \text{rel-ipurge} \ P \ I \ D \ u \ \rightarrow \next-dom-events \ P \ D \ u \ xs = \next-dom-events \ P \ D \ u \ ys \land \ref-dom-events \ P \ D \ u \ xs = \ref-dom-events \ P \ D \ u \ ys\ and$

$B$: $u \in \text{range} \ D$ and

$C$: $(xs, ys) \in \text{rel-ipurge} \ P \ I \ D \ u$

**Show** next-dom-events $P \ D \ u \ xs = \next-dom-events \ P \ D \ u \ ys \land$

26
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 €. 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 Unwind-
ing 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) =
iproge-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 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 ⊆ 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 .
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 (zs @ 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 (zs @ 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 (zs @ ys @ zs @ [z]) =
ipurge-tr-rev-aux I D U (zs @ 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 (zs @ 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]) =

\[ \text{ipurge-tr-rev-aux } \{D\} U \ (xs @ ys @ zs) \ @ [z] \]

by \(\text{subst ipurge-tr-rev-aux-append, simp}\)
also have \ldots = \text{ipurge-tr-rev-aux } \{D\} U \ (xs @ \text{ipurge-tr-aux } \{D'\} set ys) zs) \ @ [z] \]
using \(E\) and \(F\) by simp
also have \ldots = \text{ipurge-tr-rev-aux } \{D\} U \ ((xs @ \text{ipurge-tr-aux } \{D'\} set ys) zs) \ @ [z] \]
using True by \(\text{subst ipurge-tr-rev-aux-append, simp}\)
finally show \(?\text{thesis by simp}\)

next

\begin{itemize}
  \item case False
    hence \text{ipurge-tr-rev-aux } \{D\} U \ ((xs @ ys @ zs) \ @ [z]) = \text{ipurge-tr-rev-aux } \{D\} U \ (xs @ ys @ zs)
    by \(\text{subst ipurge-tr-rev-aux-append, simp}\)
  \item also have \ldots = \text{ipurge-tr-rev-aux } \{D\} U \ (xs @ \text{ipurge-tr-aux } \{D'\} set ys) zs)
    using \(D\)
  \item also have \ldots = \text{ipurge-tr-rev-aux } \{D\} U \ ((xs @ \text{ipurge-tr-aux } \{D'\} set ys) zs) \ @ [z]
    using False by \(\text{subst ipurge-tr-rev-aux-append, simp}\)
\end{itemize}
finally show \(?\text{thesis by simp}\)
qed
qed

\begin{lemma}
\text{ipurge-tr-rev-ipurge-tr-aux-2 [rule-format]:}
\end{lemma}
\(U \subseteq \text{unaffected-domains } \{D\} \ (D' \ set ys) zs \rightarrow \)
\text{ipurge-tr-rev-aux } \{D\} U \ (xs @ zs) = \text{ipurge-tr-rev-aux } \{D\} U \ (xs @ ys @ \text{ipurge-tr-aux } \{D'\} set ys) zs)
\begin{proof}
(induction zs arbitrary: U rule: rev-induct, rule-tac [!] impI, simp)
fix \(U\)
assume \(A\): \(\forall U. U \subseteq \text{unaffected-domains } \{D\} \ (D' \ set ys) \]
have \(\forall u \in U. \forall v \in D' \ set ys. (v, u) \notin I\)
proof
fix \(u\)
assume \(u \in U\)
with \(A\) have \(u \in \text{unaffected-domains } \{D\} \ (D' \ set ys) \]
thus \(\forall v \in D' \ set ys. (v, u) \notin I\) by \(\text{simp add: unaffected-domains-def}\)
qed
hence \text{ipurge-tr-rev-aux } \{D\} U ys = [] by \(\text{simp add: ipurge-tr-rev-aux-nil}\)
hence \text{ipurge-tr-rev-aux } \{D\} U (xs @ ys) = \text{ipurge-tr-rev-aux } \{D\} U \ (xs @ ys)
by \(\text{simp add: ipurge-tr-rev-aux-append-nil}\)
thus \text{ipurge-tr-rev-aux } \{D\} U xs = \text{ipurge-tr-rev-aux } \{D\} U (xs @ ys) \]
next
\begin{itemize}
  \item fix \(z\) \(zs\) \(U\)
  \item let \(?U' = \text{insert} \ (D \ z) \ U\)
\end{itemize}
assume \(A\): \(\forall U. U \subseteq \text{unaffected-domains } \{D\} \ (D' \ set ys) zs \rightarrow \)
\text{ipurge-tr-rev-aux } \{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-asmsimp))
  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

31
qed

hence ipurge-tr-rev-aux \( I \ D \ U \) \((xs @ zs) @ [z]\) =
ipurge-tr-rev-aux \( I \ D \ U \) \((zs @ zs)\)
by (subst ipurge-tr-rev-aux-append, simp)
also have
\[ \ldots = \text{ipurge-tr-rev-aux} \( I \ D \ U \) \((xs @ ys @ \text{ipurge-tr-aux} \( I \ D \) \( D \ ' \ \text{set} \ ys \) zs)\)\]
using \( D \).
finally show ipurge-tr-rev-aux \( I \ D \ U \) \((zs @ zs @ [z])\) =
ipurge-tr-rev-aux \( I \ D \ U \) \((zs @ ys @ \text{ipurge-tr-aux} \( I \ D \) \( D \ ' \ \text{set} \ ys \) zs)\)
by simp

next

case False

note \( F = \text{this} \)

show ipurge-tr-rev-aux \( I \ D \ U \) \((zs @ zs @ [z])\) =
ipurge-tr-rev-aux \( I \ D \ U \) \((zs @ ys @ \text{ipurge-tr-aux} \( I \ D \) \( D \ ' \ \text{set} \ ys \) zs) @ [z])\)

proof (cases \( \exists u \in U. \ (D \ z, u) \in I \))

case True

hence ipurge-tr-rev-aux \( I \ D \ ?U' \) \((xs @ zs) @ [z]\) =
ipurge-tr-rev-aux \( I \ D \ ?U' \) \((zs @ zs) @ [z]\)
by (subst ipurge-tr-rev-aux-append, simp)
also have \( \ldots = \text{ipurge-tr-rev-aux} \( I \ D \ ?U' \) \((xs @ ys @ \text{ipurge-tr-aux} \( I \ D \) \( D \ ' \ \text{set} \ ys \) zs) @ [z])\)
using \( E \) and \( F \) by simp
also have \( \ldots = \text{ipurge-tr-rev-aux} \( I \ D \ U \) \((xs @ ys @ \text{ipurge-tr-aux} \( I \ D \) \( D \ ' \ \text{set} \ ys \) 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 \( I \ D \ U \) \((zs @ zs) @ [z]\) =
ipurge-tr-rev-aux \( I \ D \ U \) \((zs @ zs)\)
by (subst ipurge-tr-rev-aux-append, simp)
also have \( \ldots = \text{ipurge-tr-rev-aux} \( I \ D \ U \) \((xs @ ys @ \text{ipurge-tr-aux} \( I \ D \) \( D \ ' \ \text{set} \ ys \) zs)\)\)
using \( D \).
also have \( \ldots = \text{ipurge-tr-rev-aux} \( I \ D \ U \) \((xs @ ys @ \text{ipurge-tr-aux} \( I \ D \) \( D \ ' \ \text{set} \ ys \) zs) @ [z])\)
using False by (subst ipurge-tr-rev-aux-append, simp)
finally show \(?thesis\) by simp

qed

qed

lemma ipurge-tr-rev-ipurge-tr-1:

assumes \( A: u \in \text{unaffected-domains} \ I \ D \ \{D \ y\} \ zs \)

shows ipurge-tr-rev \( I \ D \ u \) \((xs @ y \neq zs)\) =

32
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 \in unaffected-domains I D {D y} zs
shows ipurge-tr-rev I D u (zs @ zs) =
ipurge-tr-rev I D u (xs @ y # ipurge-tr I D (D y) zs)
proof

have ipurge-tr-rev I D u (zs @ 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) \in failures P and
B: zs @ ipurge-tr I D (D y) ys \in traces P and
C: 3 y'. y' \in 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) \in failures P
proof

let ?A = singleton-set (ipurge-ref I D (D y) ys Y)

have (\exists X. X \in {?A}) \rightarrow
(\forall X \in {?A}. (xs @ ipurge-tr I D (D y) ys, X) \in failures P) \rightarrow
(xs @ ipurge-tr I D (D y) ys, \bigcup X \in {?A}. X) \in failures P
using RUC by (simp add: ref-union-closed-def)
moreover obtain y' where D: y' \in ipurge-ref I D (D y) ys Y using C ..
hence \exists X. X \in {?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) \rightarrow
(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, rule bexE, simp)

fix y'
have \( \forall u \in \text{range } D \cap (-I) \) \( \Rightarrow \) \( \text{range } D \).

\( \forall xs ys. \ (xs, ys) \in \text{rel-ipurge } P I D u \longrightarrow \)

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 \( E: y' \in \text{ipurge-ref } I D (D y') ys Y \)

hence \( (D y, D y') \notin I \) by (simp add: ipurge-ref-def)

hence \( D y' \in \text{range } D \cap (-I) \) \( \Rightarrow \) \( \text{range } D \)

by (simp add: Image-iff, rule extI)

ultimately have \( \forall xs ys. \ (xs, ys) \in \text{rel-ipurge } P I D (D y') \longrightarrow \)

ref-dom-events \( P D (D y') xs = \) ref-dom-events \( P D (D y') ys \) ..

hence

\( F: (xs @ y \neq ys, xs @ ipurge-tr I D (D y) ys) \in \text{rel-ipurge } P I D (D y') \longrightarrow \)

ref-dom-events \( P D (D y') (xs @ y \neq ys) = \)

ref-dom-events \( P D (D y') (xs @ ipurge-tr I D (D y) ys) \)

by blast

have \( y' \in \{ x \in Y. D x \in \text{unaffected-domains } I D \{D y\} ys \} \)

using \( E \) by (simp add: unaffected-domains-single-dom)

hence \( D y' \in \text{unaffected-domains } I D \{D y\} ys \) by simp

hence \( \text{ipurge-tr-rev } I D (D y') (xs @ y \neq ys) = \)

\( \text{ipurge-tr-rev } I D (D y') (xs @ ipurge-tr I D (D y) ys) \)

by (rule ipurge-tr-rev-ipurge-tr-1)

moreover have \( xs @ y \neq ys \in \text{traces } P \) using \( A \) by (rule failures-traces)

ultimately have

\( (xs @ y \neq ys, xs @ ipurge-tr I D (D y) ys) \in \text{rel-ipurge } P I D (D y') \)

using \( B \) by (simp add: rel-ipurge-def)

with \( F \) have ref-dom-events \( P D (D y') (xs @ y \neq ys) = \)

ref-dom-events \( P D (D y') (xs @ ipurge-tr I D (D y) ys) \) ..

moreover have \( y' \in \text{ref-dom-events } P D (D y') (xs @ y \neq ys) \)

proof (simp add: ref-dom-events-def refusals-def)

have \( \{ y' \} \subseteq Y \) using \( E \) by (simp add: ipurge-ref-def)

with \( A \) show \( (xs @ y \neq ys, \{ y' \}) \in \text{failures } P \) by (rule process-rule-3)

qed

ultimately have \( y' \in \text{ref-dom-events } P D (D y') \)

\( (xs @ ipurge-tr I D (D y) ys) \)

by simp

thus \( (xs @ ipurge-tr I D (D y) ys, \{ y' \}) \in \text{failures } P \)

by (simp add: ref-dom-events-def refusals-def)

qed

ultimately have \( (xs @ ipurge-tr I D (D y) ys, \bigcup X \in \ ?A. X) \in \text{failures } P \) ..

thus \( \text{thesis } \) by (simp only: singleton-set-union)

qed

lemma \( \text{iu-condition-imply-secure-aux-2}: \)

assumes

\( RUC: \) ref-union-closed \( P \) and

\( IU: \) weakly-future-consistent \( P I D \) (rel-ipurge \( P I D) \) and

\( A: (xs @ zs, Z) \in \text{failures } P \) and

\( B: \) \( \) \( zs @ y \neq \) \( \) ipurge-tr \( I D (D y) zs \in \text{traces } P \) and

\( C: \exists z. z' \in ipurge-ref I D (D y) zs Z \)

shows \( (xs @ y \neq \) \( \) ipurge-tr \( I D (D y) zs, ipurge-ref I D (D y) zs Z) \in \text{failures } P \)
proof –

let \( \exists A = \text{singleton-set} (\text{ipurge-ref I D} (D y) zs Z) \)

have \( \exists X. X \in \exists A \) \( \longrightarrow \)
\( \forall X \in \exists A. (xs \& y \# \text{ipurge-tr I D} (D y) zs, X) \in \text{failures P} \) \( \longrightarrow \)
\( (xs \& y \# \text{ipurge-tr I D} (D y) zs, \bigcup X \in \exists A. X) \in \text{failures P} \)

using RUC by (simp add: ref-union-closed-def)

moreover obtain \( z' \) where \( D: z' \in \text{ipurge-ref I D} (D y) zs Z \) using \( C \) ..

hence \( \exists X. X \in \exists A \) by (simp add: singleton-set-some, rule \text{exI})

ultimately have
\( \forall X \in \exists A. (xs \& y \# \text{ipurge-tr I D} (D y) zs, X) \in \text{failures P} \) \( \longrightarrow \)
\( (xs \& y \# \text{ipurge-tr I D} (D y) zs, \bigcup X \in \exists A. X) \in \text{failures P} \) ..

moreover have \( \forall X \in \exists A. (xs \& y \# \text{ipurge-tr I D} (D y) zs, X) \in \text{failures P} \)

proof (rule \text{ballI}, simp add: singleton-set-def, erule \text{bexE}, simp)

fix \( z' \)

have \( \forall u \in \text{range D} \cap (-I) \) " range D.
\( \forall zs ys. (xs, ys) \in \text{rel-ipurge P I D} u \) \( \longrightarrow \)
\( \text{ref-dom-events P D} \ u \ xs = \text{ref-dom-events P D} \ u \ ys \)

using \( IU \) by (simp add: weakly-future-consistent-def)

moreover assume \( E: z' \in \text{ipurge-ref I D} (D y) zs Z \)

hence \( (D y, D z') \notin I \) by (simp add: ipurge-ref-def)

hence \( D z' \in \text{range D} \cap (-I) \) " range D by (simp add: \text{Image-iff}, rule \text{exI})

ultimately have \( \forall zs ys. (xs, ys) \in \text{rel-ipurge P I D} (D z') \) \( \longrightarrow \)
\( \text{ref-dom-events P D} (D z') \ xs = \text{ref-dom-events P D} (D z') \ ys \ ..

hence
\( F: (xs \& zs, xs \& y \# \text{ipurge-tr I D} (D y) zs) \in \text{rel-ipurge P I D} (D z') \) \( \longrightarrow \)
\( \text{ref-dom-events P D} (D z') (xs \& zs) = \text{ref-dom-events P D} (D z') (xs \& y \# \text{ipurge-tr I D} (D y) zs) \)

by blast

have \( z' \in \{ x \in Z. D x \in \text{unaffected-domains I D} \{D y\} zs\} \)

using \( E \) by (simp add: unaffected-domains-single-dom)

hence \( D z' \in \text{unaffected-domains I D} \{D y\} \) zs by simp

hence \( \text{ipurge-tr-rev I D} (D z') (xs \& zs) = \text{ipurge-tr-rev I D} (D z') (xs \& y \# \text{ipurge-tr I D} (D y) zs) \)

by (rule \text{ipurge-tr-rev-ipurge-tr-2})

moreover have \( xs \& zs \in \text{traces P} \) using \( A \) by (rule failures-traces)

ultimately have
\( (xs \& zs, xs \& y \# \text{ipurge-tr I D} (D y) zs) \in \text{rel-ipurge P I D} (D z') \)

using \( B \) by (simp add: rel-ipurge-def)

with \( F \) have \( \text{ref-dom-events P D} (D z') (xs \& zs) = \text{ref-dom-events P D} (D z') (xs \& y \# \text{ipurge-tr I D} (D y) zs) \) ..

moreover have \( z' \in \text{ref-dom-events P D} (D z') (xs \& zs) \)

proof (simp add: ref-dom-events-def refusals-def)

have \( \{ z' \} \subseteq Z \) using \( E \) by (simp add: ipurge-ref-def)

with \( A \) show \( (xs \& zs, \{ z' \}) \in \text{failures P} \) by (rule \text{process-rule-3})

qed

ultimately have \( z' \in \text{ref-dom-events P D} (D z') \)
\( (xs \& y \# \text{ipurge-tr I D} (D y) zs) \)

by simp

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

lemma iu-condition-imply-secure-1 [rule-formal]:
assumes
\( \text{RUC: ref-union-closed P and} \)
\( \text{IU: weakly-future-consistent P I D (rel-ipurge P I D)} \)
shows \( (xs @ y \# ys, Y) \in \text{failures P} \)
proof (induction ys arbitrary: Y rule: rev-induct, rule-tac [!] impI)
fix Y
assume \( A: (xs @ [y], Y) \in \text{failures P} \)
show \( (xs @ \text{ipurge-tr I D (D y) []}, \text{ipurge-ref I D (D y) [] Y}) \in \text{failures P} \)
proof (cases \( \exists y': y' \in \text{ipurge-ref I D (D y) [] Y} \))
  case True
  have \( xs @ [y] \in \text{traces P using A by (rule failures-traces)} \)
  hence \( xs \in \text{traces P by (rule process-rule-2-traces)} \)
  hence \( xs @ \text{ipurge-tr I D (D y) [] } \in \text{traces P by simp} \)
  with \( \text{RUC and IU} \)
  ultimately show \(?thesis\) by simp
next
  case False
moreover have \( (xs, \{\}) \in \text{failures P using A by (rule process-rule-2)} \)
ultimately show \(?thesis\) by simp
qed

next
fix \( y' ys Y \)
assume
\( A: A'. (xs @ y \# ys, Y') \in \text{failures P} \)
\( (xs @ \text{ipurge-tr I D (D y) ys, ipurge-ref I D (D y) ys Y'}) \in \text{failures P} \) and
\( B: (xs @ y \# ys @ [y'], Y) \in \text{failures P} \)
have \( (xs @ y \# ys, \{\}) \in \text{failures P} \)
(\( \text{is - } \rightarrow (\sim, ?Y') \in - \))
using A .
moreover have \( (xs @ y \# ys) @ [y'], Y) \in \text{failures P using B by simp} \)
hence \( C: (xs @ y \# ys, \{\}) \in \text{failures P by (rule process-rule-2)} \)
ultimately have \( (xs @ \text{ipurge-tr I D (D y) ys, ?Y'}) \in \text{failures P} \) ..
moreover have \( \{\} \subseteq ?Y' .. \)
ultimately have \( D: (xs @ \text{ipurge-tr I D (D y) ys, \{\}) \in \text{failures P} \)
by (rule process-rule-3)
have \( E: xs @ \text{ipurge-tr I D (D y) (ys @ [y'])} \in \text{traces P} \)
proof (cases \( D y' \in \text{sinks I D (D y) (ys @ [y'])} \))
  case True
  hence \( (xs @ \text{ipurge-tr I D (D y) (ys @ [y'])}, \{\}) \in \text{failures P using D by simp} \)
thus \( \exists \)thesis by (rule failures-traces)

next

case False

have \( \forall u \in \text{range}\ D \cap (-I) \ " \ " \text{range}\ D. \)
\( \forall xs\ ys. (xs, ys) \in \text{rel-ipurge}\ P I D u \rightarrow \)
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 \((D y, D y') \notin I\)

using False by (simp add: sinks-interference-eq [symmetric] del: sinks.simps)

hence \( D y' \in \text{range}\ D \cap (-I) \ " \ " \text{range}\ D\) by (simp add: image-iff, rule exI)

ultimately have \( \forall xs\ ys. (xs, ys) \in \text{rel-ipurge}\ P I D (D y') \rightarrow \)
next-dom-events\ P D (D y') xs = next-dom-events\ P D (D y') ys ..

hence
\( F: (xs \@ y \notin ys, xs \@ \text{ipurge-tr} I D (D y') ys) \in \text{rel-ipurge}\ P I D (D y') \rightarrow \)
next-dom-events\ P D (D y') (xs \@ y \# ys) =
next-dom-events\ P D (D y') (xs \@ \text{ipurge-tr} I D (D y') ys)

by blast

have \( \forall v \in \text{insert}\ (D y) (\text{sinks}\ I D (D y') ys). (v, D y') \notin I\)

using False by (simp add: sinks-interference-eq [symmetric] del: sinks.simps)

hence \( \forall v \in \text{sinks-aux}\ I D \{D y\} ys. (v, D y') \notin I\)

by (simp add: unaffected-domains-def)

hence \text{ipurge-tr-rev}\ I D (D y') (xs \@ y \# ys) =
\text{ipurge-tr-rev} I D (D y') (xs \@ \text{ipurge-tr} I D (D y') ys)

by (rule ipurge-tr-rev-ipurge-tr-1)

moreover have \( xs \@ y \notin ys \in \text{traces}\ P\) using C by (rule failures-traces)

moreover have \( xs \@ \text{ipurge-tr} I D (D y') ys \in \text{traces}\ P\)

using D by (rule failures-traces)

ultimately have
\( (xs \@ y \notin ys, xs \@ \text{ipurge-tr} I D (D y') ys) \in \text{rel-ipurge}\ P I D (D y')\)

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 \@ \text{ipurge-tr} I D (D y') ys) ..

moreover have \( y' \in \text{next-dom-events}\ P D (D y') (xs \@ y \# ys)\)

proof (simp add: next-dom-events-def next-events-def)

qed (rule failures-traces \{OF \[B]\})

ultimately have \( y' \in \text{next-dom-events}\ P D (D y')\)
\( (xs \@ \text{ipurge-tr} I D (D y) ys)\)

by simp

hence \( xs \@ \text{ipurge-tr} I D (D y) ys \@ \[y'] \in \text{traces}\ P\)

by (simp add: next-dom-events-def next-events-def)

thus \( \exists \)thesis using False by simp

qed

show \( (xs \@ \text{ipurge-tr} I D (D y) (ys \@ \[y'])), \text{ipurge-ref}\ I D (D y) (ys \@ \[y'])\ Y\)

\in \text{failures}\ P\)

proof (cases \( \exists x. x \in \text{ipurge-ref} I D (D y) (ys \@ \[y'])\ Y\)

case True

with RUC and IU and B and E show \( \exists \)thesis by (rule iu-condition-implies-secure-aux-1)
next

  case False
  moreover have \((xs @ ipurge-tr I D (D y) (ys @ [y']), \{\}) \in \text{failures } P\)
  using \(E\) by (rule traces-failures)
  ultimately show \(\text{thesis by simp}\)
  qed
qed

lemma \textit{iu-condition-imply-secure-2} [rule-formal]:

  assumes
    \(\text{RUC: ref-union-closed } P\) and
    \(\text{IU: weakly-future-consistent } P I D\) and
    \(Y: \text{xs @ [y]} \in \text{traces } P\)
  shows \((xs @ zs, Z) \in \text{failures } P \rightarrow \)
    \((zs @ y # \text{ipurge-tr I D (D y) zs}, \text{ipurge-ref I D (D y) zs Z}) \in \text{failures } P\)

proof (induction zs arbitrary: Z rule: rev-induct, rule-tac [!] impI)

  fix \(Z\)

  assume \(A: (xs @ [], Z) \in \text{failures } P\)

  show \((xs @ y # \text{ipurge-tr I D (D y) []}, \text{ipurge-ref I D (D y) [] Z}) \in \text{failures } P\)

proof (cases \(\exists z'. z' \in \text{ipurge-ref I D (D y) [] Z}\)

  case True
  have \((xs @ y # \text{ipurge-tr I D (D y) []}) \in \text{traces } P\) using \(Y\) by simp
  with \(\text{RUC and IU and A show thesis}\)
  using \(\text{True by (rule iu-condition-imply-secure-aux-2)\) next\)

  case False

  moreover have \((xs @ [y], \{\}) \in \text{failures } P\) using \(E\) by (rule traces-failures)

  ultimately show \(\text{thesis by simp}\)
  qed

next

  fix \(z, zs, Z\)

  assume
    \(A: (xs @ zs, Z) \in \text{failures } P \rightarrow \)
    \((xs @ y # \text{ipurge-tr I D (D y) zs}, \text{ipurge-ref I D (D y) zs Z}) \in \text{failures } P\) and
    \(B: (xs @ zs @ [z], Z) \in \text{failures } P\)

  have \((xs @ zs, \{\}) \in \text{failures } P \rightarrow \)
    \((xs @ y # \text{ipurge-tr I D (D y) zs}, \text{ipurge-ref I D (D y) zs \{\}) \in \text{failures } P\)
    (is \(\rightarrow (-, ?Z') \in -\)

using \(A\).

  moreover have \((xs @ zs) @ [z], Z) \in \text{failures } P\) using \(B\) by simp

hence \(C: (xs @ zs, \{\}) \in \text{failures } P\) by (rule process-rule-2)

ultimately have \((xs @ y # \text{ipurge-tr I D (D y) zs, ?Z'}) \in \text{failures } P\)

moreover have \(\{\} \subseteq ?Z'\)

ultimately have \(D: (xs @ y # \text{ipurge-tr I D (D y) zs, \{\}) \in \text{failures } P\)

by (rule process-rule-3)

have \(E: \text{zs @ y # \text{ipurge-tr I D (D y) (zs @ [z])} \in \text{traces } P}\)

proof (cases \(D, z \in \text{sinks I D (D y) (zs @ [z])}\))

  case True

  hence \((xs @ y # \text{ipurge-tr I D (D y) (zs @ [z]), \{\}) \in \text{failures } P\)
using $D$ by simp
thus $?thesis$ by (rule failures-traces)
next
case $False$
have $\forall u \in \text{range } D \cap (-I)$ "" range $D$.
  \[
  \forall xs ys. (xs, ys) \in \text{rel-ipurge } P I D u \rightarrow \text{next-dom-events } P D u xs = \text{next-dom-events } P D u ys
  \]
using $IU$ by (simp add: weakly-future-consistent-def)
moreover have $(D y, D z) \notin I$
using $False$ by (simp add: sinks-interference-eq [symmetric] del: sinks.simps)
hence $D z \in \text{range } D \cap (-I)$ "" range $D$ by (simp add: Image-iff, rule exI)
ultimately have $\forall xs ys. (xs, ys) \in \text{rel-ipurge } P I D (D z) \rightarrow \text{next-dom-events } P D (D z) xs = \text{next-dom-events } P D (D z) ys ..$
hence
\[
F: (xs \in D z, zs @ y \# \text{ipurge-tr } I D (D y) zs) \in \text{rel-ipurge } P I D (D z) \rightarrow \text{next-dom-events } P D (D z) (zs @zs) = \text{next-dom-events } P D (D z) (zs @ y \# \text{ipurge-tr } I D (D y) zs)
\]
by blast
have $\forall v \in \text{insert } (D y) (\text{sinks } I D (D y) zs), (v, D z) \notin I$
using $False$ by (simp add: sinks-interference-eq [symmetric] del: sinks.simps)
hence $\forall v \in \text{sinks-aux } I D \{D y\} zs, (v, D z) \notin I$
by (simp add: sinks-aux-single-dom)
hence $D z \in \text{unaffected-domains } I D \{D y\} zs$
by (simp add: unaffected-domains-def)
hence $\text{ipurge-tr-rec } I D (D z) (zs @zs) = \text{ipurge-tr-rec } I D (D z) (xs @ y \# \text{ipurge-tr } I D (D y) zs)$
by (rule ipurge-tr-rec-ipurge-tr-2)
moreover have $xs @zs \in \text{traces } P$ using $C$ by (rule failures-traces)
moreover have $xs @ y \# \text{ipurge-tr } I D (D y) zs \in \text{traces } P$
using $D$ by (rule failures-traces)
ultimately have
\[
(xs @zs, xs @ y \# \text{ipurge-tr } I D (D y) zs) \in \text{rel-ipurge } P I D (D z)
\]
by (simp add: rel-ipurge-def)
with $F$ have $\text{next-dom-events } P D (D z) (xs @zs) =$
  \[
  \text{next-dom-events } P D (D z) (xs @ y \# \text{ipurge-tr } I D (D y) zs) ..
  \]
moreover have $z \in \text{next-dom-events } P D (D z) (xs @zs)$
proof (simp add: next-dom-events-def next-events-def)
qed (rule failures-traces [OF $B$])
ultimately have $z \in \text{next-dom-events } P D (D z)$
\[
(xs @ y \# \text{ipurge-tr } I D (D y) zs)
\]
by simp
hence $xs @ y \# \text{ipurge-tr } I D (D y) zs @ [z] \in \text{traces } P$
by (simp add: next-dom-events-def next-events-def)
thus $?thesis$ using $False$ by simp
qed
show $(xs @ y \# \text{ipurge-ref } I D (D y) (zs @ [z]),$
  \[
  \text{ipurge-ref } I D (D y) (zs @ [z]) Z
  \]
  $\in \text{failures } P$
proof (cases $\exists x. x \in \text{ipurge-ref } I D (D y) (zs @ [z]) Z$)
case True
with RUC and IU and B and E show \( \text{thesis by (rule iu-condition-imply-secure-aux-2)} \)
next
  case False
  moreover have \((xs @ y # \text{ipurge-tr } ID) \ (zs @ \{z\}, \{\}) \in \text{failures } P\)
  using E by (rule traces-failures)
  ultimately show \( \text{thesis by simp} \)
qed
qed

theorem iu-condition-imply-secure:
assumes
  \( RUC: \text{ref-union-closed } P \) and
  \( IU: \text{weakly-future-consistent } PID \) (rel-ipurge \( PID \))
shows secure \( PID \)
proof (simp add: secure-def futures-def, (rule allI)+, rule impI, erule conjE)
fix \( xs \ y \ ys \ Y \ zs \ Z \)
assume
  \( A: (xs @ y # ys, Y) \in \text{failures } P \) and
  \( B: (xs @ zs, Z) \in \text{failures } P \)
show \((xs @ \text{ipurge-tr } ID \ (D \ y) \ ys, \text{ipurge-ref } ID \ (D \ y) \ ys \ Y) \in \text{failures } P \)
  \((xs @ y # \text{ipurge-tr } ID \ (D \ y) \ zs, \text{ipurge-ref } ID \ (D \ y) \ zs \ Z) \in \text{failures } P \)
  \( (\text{is } \ ?P \wedge \ ?Q) \)
proof
  show \( \text{?P} \) using RUC and IU and A by (rule iu-condition-imply-secure-1)
next
  have \((xs @ [y]) @ zs, X) \in \text{failures } P \) using A by simp
  hence \((xs @ [y], \{\}) \in \text{failures } P \) by (rule process-rule-2-failures)
  hence \(xs @ [y] \in \text{traces } P \) by (rule failures-traces)
  with RUC and IU show \( \text{?Q} \) 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 \( S: \text{secure } PID \)
shows \((xs @ ys @ zs, X) \in \text{failures } P \) \( \rightarrow \)
  \( \text{ipurge-tr-rev-aux } ID \ (D \cdot (X \cup \text{set } zs)) \) \( ys = \ [] \) \( \rightarrow \)
  \((xs @ zs, X) \in \text{failures } P \)
proof (induction \( ys \) rule: rev-induct, simp-all, (rule impI)+)
fix \( y \ ys \)
assume \( *: \text{ipurge-tr-rev-aux } ID \ (D \cdot (X \cup \text{set } zs)) \) \( (ys @ [y]) = \ [] \)
then have \( A: \neg (\exists v \in D \cdot (X \cup \text{set } zs). \ (D \ y, v) \in I) \)
by (cases \( \exists v \in D \cdot (X \cup set\ zs) \). \((D y, v) \in I\),  
simp-all add: ipurge-tr-rev-aux-append)

with * have \( B:\ ipurge-tr-rev-aux I D (D y \cdot (X \cup set\ zs))\) \(ys = []\)  
by (simp add: ipurge-tr-rev-aux-append)

assume \((zs @ ys @ y \# zs, X) \in \text{failures} P\)
hence \((y \# zs, X) \in \text{failures} P (zs @ ys)\) by (simp add: futures-def)
hence \((ipurge-tr I D (D y)\ zs, ipurge-ref I D (D y)\ zs\ X)\)  
\(\in \text{failures} P (zs @ ys)\)

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 \( ipurge-ref I D (D y)\ zs = X\) using \( A\) by (rule ipurge-ref-all)
ultimately have \((zs, X) \in \text{failures} P (zs @ ys)\) by simp

hence \((zs @ ys @ zs, X) \in \text{failures} P\) by (simp add: futures-def)

assume \((zs @ ys @ zs, X) \in \text{failures} P \rightarrow \)  
\((ipurge-tr-rev-aux I D (D y \cdot (X \cup set\ zs))\) \(ys = []\) \(\rightarrow\) \((zs @ zs, X) \in \text{failures} P\)

hence \((ipurge-tr-rev-aux I D (D y \cdot (X \cup set\ zs))\) \(ys = []\) \(\rightarrow\) \((zs @ zs, X) \in \text{failures} P\)

using \( C\) ..
thus \((zs @ zs, X) \in \text{failures} P\) using \( B\) ..

qed

lemma secure-implies-failure-consistency [rule-format]:
assumes \( S:\ \text{secure} P I D\)
shows \((zs, ys) \in \text{rel-ipurge-aux} P I D (D y \cdot (X \cup set\ zs))\) \(\rightarrow\) \((zs @ zs, X) \in \text{failures} P\)

proof (induction \( ys \) arbitrary: zs zs rule: rev-induct,  
simp-all add: rel-ipurge-aux-def, (rule-tac \(!!\) impI)+, (crule-tac \(!!\) conjE)+)

fix \( xs\) \(zs\)
assume \((zs @ zs, X) \in \text{failures} P\)
hence \([[] @ zs @ zs, X) \in \text{failures} P\) by simp

moreover assume \((ipurge-tr-rev-aux I D (D y \cdot (X \cup set\ zs))\) \(xs = []\) \(\rightarrow\)

ultimately have \(([[] @ zs, X) \in \text{failures} P\)

using \( S\) by (rule-tac secure-implies-failure-consistency-aux)
thus \((zs, X) \in \text{failures} P\) by simp

next

fix \( y\) \(ys\) \(zs\)

assume  
A: \( \forall xs' z's', xs' \in \text{traces} P \land ys \in \text{traces} P \land\)  
\((ipurge-tr-rev-aux I D (D y \cdot (X \cup set\ zs'))\) \(xs' =\)  
\((ipurge-tr-rev-aux I D (D y \cdot (X \cup set\ zs'))\) \(ys \rightarrow\)  
\((xs' @ zs', X) \in \text{failures} P \rightarrow (ys @ zs', X) \in \text{failures} P\) and
B: \((zs @ zs, X) \in \text{failures} P\) and
C: \(zs \in \text{traces} P\) and
D: \(ys @ [y] \in \text{traces} P\) and
E: \((ipurge-tr-rev-aux I D (D y \cdot (X \cup set\ zs))\) \(xs =\)  
\((ipurge-tr-rev-aux I D (D y \cdot (X \cup set\ zs))\) \(ys @ [y]\)

show \((ys @ y \# zs, X) \in \text{failures} P\)

proof (cases \( \exists v \in D \cdot (X \cup set\ zs)\). \((D y, v) \in I)\)
case True

hence $F$: $\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ xs =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ (y \# \ zs))) \ ys \ @ [y]$

using $E$ by (simp add: $\text{ipurge-tr-rev-aux-append}$)

hence

$\exists \ vs \ ws. \ xs = vs @ y \# ws \wedge \text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ ws = []$

by (rule $\text{ipurge-tr-rev-aux-last-2}$)

then obtain $vs$ and $ws$ where

$G$: $xs = vs @ y \# ws \wedge \text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ ws = []$

by blast

hence $\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ xs =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ (vs @ [y]) @ ws$

by simp

hence $\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ xs =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ (vs @ [y]$

using $G$ by (simp only: $\text{ipurge-tr-rev-aux-append-nil}$)

moreover have $\exists v \in D \ (X \cup \text{set} \ zs). \ (D, y, v) \in I$

using $F$ by (rule $\text{ipurge-tr-rev-aux-last-1}$)

ultimately have $\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ xs =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ vs @ [y]$

by (simp add: $\text{ipurge-tr-rev-aux-append}$)

hence $\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ vs =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ ys$

using $F$ by simp

moreover have $vs @ y \# ws \in \text{traces} \ P$ using $C$ and $G$ by simp

hence $vs \in \text{traces} \ P$ by (rule process-rule-2-traces)

moreover have $ys \in \text{traces} \ P$ using $D$ by (rule process-rule-2-traces)

moreover have $vs \in \text{traces} \ P$ and $ys \in \text{traces} \ P$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ (y \# zs))) \ vs =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ (y \# zs))) \ ys \rightarrow$

$\{ (vs @ y \# zs, X) \in \text{failures} \ P \rightarrow \ (ys @ y \# zs, X) \in \text{failures} \ P \}$

using $A$.

ultimately have $H$: $(vs @ y \# zs, X) \in \text{failures} \ P \rightarrow$

$(ys @ y \# zs, X) \in \text{failures} \ P$

by simp

have $(\{ (vs @ [y]) @ ws @ zs, X) \in \text{failures} \ P$ using $B$ and $G$ by simp

moreover have $\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ ws =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ ys \rightarrow$

$\text{using} \ G$

ultimately have $(\{ (vs @ [y]) @ ws @ zs, X) \in \text{failures} \ P$

using $S$ by (rule-tac secure-implies-failure-consistency-aux)

thus $\vartheta$thesis using $H$ by simp

next

case False

hence $\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ xs =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ ys$

using $E$ by (simp add: $\text{ipurge-tr-rev-aux-append}$)

moreover have $ys \in \text{traces} \ P$ using $D$ by (rule process-rule-2-traces)

moreover have $xs \in \text{traces} \ P$ and $ys \in \text{traces} \ P$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ xs =$

$\text{ipurge-tr-rev-aux} \ I \ D \ (D \ (X \cup \text{set} \ zs)) \ ys \rightarrow$
\[(xs @ zs, X) \in \text{failures } P \implies (ys @ zs, X) \in \text{failures } P\]

using A.

ultimately have \((ys @ zs, X) \in \text{failures } P\) using B and C by simp

hence \((zs, X) \in \text{futures } P\) by (simp add: futures-def)

moreover have \(\exists Y. ([y], Y) \in \text{futures } P\)

using D by (simp add: traces-def Domain-iff futures-def)

then obtain Y where \(([y], Y) \in \text{futures } P\)

ultimately have \((y # \text{ipurge-tr } I D (D y) zs, \text{ipurge-ref } I D (D y) zs X) \in \text{futures } P\)

using S by (simp add: secure-def)

moreover have \(\text{ipurge-tr } I D (D y) zs = zs\)

using False by (simp add: ipurge-tr-all)

moreover have \(\text{ipurge-ref } I D (D y) zs X = X\)

using False by (rule ipurge-ref-all)

ultimately show \(?thesis\) by (simp add: futures-def)

qed

lemma secure-implies-trace-consistency:
secure P I D \(\implies (xs, ys) \in \text{rel-ipurge-aux } P I D (D \cdot \text{set } zs) \implies (xs @ zs) \in \text{traces } P \implies ys @ zs \in \text{traces } P\)

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 \(\implies (xs, ys) \in \text{rel-ipurge } P I D (D x) \implies x \in \text{next-events } P\)\(zs \implies x \in \text{next-events } P\)

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 \(\implies (xs, ys) \in \text{rel-ipurge-aux } P I D (D \cdot X) \implies (X \in \text{refusals } P)\)\(zs \implies X \in \text{refusals } P\)

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 \(\implies (xs, ys) \in \text{rel-ipurge } P I D (D x) \implies \{x\} \in \text{refusals } P\)\(zs \implies \{x\} \in \text{refusals } P\)

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 all)+, rule implI, rule conjI, rule_tac \[\] equalityI, rule_tac \[\] subsetI, simp-all, erule-tac \[\] \text{conjE})

fix xs ys x

assume \((xs, ys) \in \text{rel-ipurge } P I D (D x)\) and \(x \in \text{next-events } P\)\(zs\)

43
with $S$ show $x \in \text{next-events } P \text{ ys}$ by (rule secure-implies-next-event-consistency)

next
fix $xs \ ys \ x$

have $\forall u \in \text{range } D. \ \text{equiv } (\text{traces } P) \ (\text{rel-ipurge } P I D u)$
using view-partition-rel-ipurge by (simp add: view-partition-def)

hence $\text{sym } (\text{rel-ipurge } P I D (D x))$ by (simp add: equiv-def)

moreover assume $(xs, \ ys) \in \text{rel-ipurge } P I D (D x)$

ultimately have $(ys, \ xs) \in \text{rel-ipurge } P I D (D x)$ by (rule symE)

moreover assume $x \in \text{next-events } P \ ys$

ultimately show $x \in \text{next-events } P \ xs$
using $S$ by (rule-tac secure-implies-next-event-consistency)

next
fix $xs \ ys \ x$

assume $(xs, \ ys) \in \text{rel-ipurge } P I D (D x) \ \text{and} \ \{x\} \in \text{refusals } P \ xs$

with $S$ show $\{x\} \in \text{refusals } P \ ys$ by (rule secure-implies-ref-event-consistency)

next
fix $xs \ ys \ x$

have $\forall u \in \text{range } D. \ \text{equiv } (\text{traces } P) \ (\text{rel-ipurge } P I D u)$
using view-partition-rel-ipurge by (simp add: view-partition-def)

hence $\text{sym } (\text{rel-ipurge } P I D (D x))$ by (simp add: equiv-def)

moreover assume $(xs, \ ys) \in \text{rel-ipurge } P I D (D x)$

ultimately have $(ys, \ xs) \in \text{rel-ipurge } P I D (D x)$ by (rule symE)

moreover assume $\{x\} \in \text{refusals } P \ ys$

ultimately show $\{x\} \in \text{refusals } P \ xs$
using $S$ by (rule-tac secure-implies-ref-event-consistency)

qed

theorem ipurge-unwinding:

ref-union-closed $P \implies$ secure $P I D = \text{weakly-future-consistent } P I D \ (\text{rel-ipurge } P I D)$

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

44
trace set processes.

2.1 Deterministic processes

Here below are the definitions of predicates \(d\text{-future-consistent}\) and \(d\text{-weakly-future-consistent}\), which are variants of predicates \(\text{future-consistent}\) and \(\text{weakly-future-consistent}\) 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. \(d\text{-future-consistent}\) implies \(d\text{-weakly-future-consistent}\), and they are equivalent for domain-relation map \(\text{rel-ipurge}\). Finally, the predicates are shown to be equivalent to their general-purpose counterparts in the case of a deterministic process.

**definition** \(d\text{-future-consistent}::\)

\(\forall u \in \text{range } D. \forall xs ys. (xs, ys) \in R u \rightarrow (xs \in \text{traces } P) = (ys \in \text{traces } P) \land \) \(\text{next-dom-events } P D u xs = \text{next-dom-events } P D u ys\)

**definition** \(d\text{-weakly-future-consistent}::\)

\(\forall u \in \text{range } D \cap (-1) \cap \text{range } D. \forall xs ys. (xs, ys) \in R u \rightarrow (xs \in \text{traces } P) = (ys \in \text{traces } P) \land \) \(\text{next-dom-events } P D u xs = \text{next-dom-events } P D u ys\)

**lemma** \(\text{dfc-implies-dwfc}:\)

\(d\text{-future-consistent } P D R \implies d\text{-weakly-future-consistent } P I D R\)

**by** (simp only: \(d\text{-future-consistent-def} d\text{-weakly-future-consistent-def}, \text{ blast}\)

**lemma** \(\text{dfc-equals-dwfc-rel-ipurge}:\)

\(d\text{-future-consistent } P D (\text{rel-ipurge } P I D) = d\text{-weakly-future-consistent } P I D (\text{rel-ipurge } P I D)\)

**proof** (rule iffI, erule \(d\text{-future-consistent-def} d\text{-weakly-future-consistent-def}, \text{ rule ballI, (rule allI)+, rule impI}\)

**fix** \(u \) \(xs\) \(ys\)

**assume**

\(A: \forall u \in \text{range } D \cap (-1) \cap \text{range } D. \forall xs ys. (xs, ys) \in \text{rel-ipurge } P I D u \rightarrow (xs \in \text{traces } P) = (ys \in \text{traces } P) \land \) \(\text{next-dom-events } P D u xs = \text{next-dom-events } P D u ys\)

\(B: u \in \text{range } D\) and
\( C: (xs, ys) \in \text{rel-ipurge } P \ D \ u \)

show \((xs \in \text{traces } P) = (ys \in \text{traces } P) \land \\
next-dom-events \ P \ D \ u \ xs = next-dom-events \ P \ D \ u \ ys \)

proof (cases \( u \in \text{range } D \cap (\neg I) \) " range \( D \))

\begin{align*}
\text{case } & \text{True} \\
\text{with } & A \text{ have } \forall xs \ ys. (xs, ys) \in \text{rel-ipurge } P \ D \ u \longrightarrow \\
& (xs \in \text{traces } P) = (ys \in \text{traces } P) \land \\
& next-dom-events \ P \ D \ u \ xs = next-dom-events \ P \ D \ u \ ys \
\end{align*}

hence \((xs, ys) \in \text{rel-ipurge } P \ D \ u \longrightarrow \\
(xs \in \text{traces } P) = (ys \in \text{traces } P) \land \\
next-dom-events \ P \ D \ u \ xs = next-dom-events \ P \ D \ u \ ys \)

by blast

thus \(?thesis \) using \( C \) ..

next

\begin{align*}
\text{case } & \text{False} \\
\text{hence } & D: u \notin (\neg I) \) " range \( D \) using \( B \) by \( \text{simp} \)
\end{align*}

have ipurge-tr-rev \ P \ D \ u \ xs = ipurge-tr-rev \ P \ D \ u \ ys

using \( C \) by (simp add: rel-ipurge-def)

moreover have \( \forall zs. \) ipurge-tr-rev \ P \ D \ u \ zs = zs

proof (rule allI, rule ipurge-tr-rev-all, rule ballI, erule imageE, rule ccontr)

fix \( v \ x \)

assume \((v, u) \notin I \)

hence \((v, u) \in (\neg I) \) " range \( D \) by \( \text{simp} \)

moreover assume \( v = D \ x \)

hence \( v \in \text{range } D \) by \( \text{simp} \)

ultimately have \( u \in (\neg I) \) " range \( D \) ..

thus \( False \) using \( D \) by \( \text{contradiction} \)

qed

ultimately show \(?thesis \) by \( \text{simp} \)

qed

\text{lemma } d-fc-equals-dfc:

assumes \( A: \text{deterministic } P \)

shows \( \text{future-consistent } P \ D \ R = d\text{-future-consistent } P \ D \ R \)

proof (rule iffI, simp-all only: d-future-consistent-def, 
rule ballI, (rule allI)+, rule implI, rule conjI, rule fc-traces, assumption+, 
 simp-all add: future-consistent-def del: ball-simps)

assume \( B: \forall u \in \text{range } D. \forall xs \ ys. (xs, ys) \in R \ u \longrightarrow \\
(xs \in \text{traces } P) = (ys \in \text{traces } P) \land \\
next-dom-events \ P \ D \ u \ xs = next-dom-events \ P \ D \ u \ ys \)

show \( \forall u \in \text{range } D. \forall xs \ ys. (xs, ys) \in R \ u \longrightarrow \\
ref-dom-events \ P \ D \ u \ xs = ref-dom-events \ P \ D \ u \ ys \)

proof (rule ballI, (rule allI)+, rule implI, 
simp add: ref-dom-events-def set-eq-iff, rule allI)

fix \( u \ x \ z \)

assume \( u \in \text{range } D \)

with \( B \) have \( \forall xs \ ys. (xs, ys) \in R \ u \longrightarrow \\
(xs \in \text{traces } P) = (ys \in \text{traces } P) \land \\
\)
next-dom-events P D u xs = \text{next-dom-events P D u ys} ..

**hence** \((xs, ys) \in R u \rightarrow
(xs \in \text{traces P}) = (ys \in \text{traces P}) \wedge
\text{next-dom-events P D u xs} = \text{next-dom-events P D u ys}\)

by blast

moreover assume \((xs, ys) \in R u \)

ultimately have \(C: (xs \in \text{traces P}) = (ys \in \text{traces P}) \wedge
\text{next-dom-events P D u xs} = \text{next-dom-events P D u ys} ..\)

**show** \((u = D x \land \{x\} \in \text{refusals P xs}) = (u = D x \land \{x\} \in \text{refusals P ys})\)

**proof** (cases \(u = D x, \text{simp-all} \), \(\text{cases xs} \in \text{traces P}\))

assume \(D: u = D x \text{ and } E: xs \in \text{traces P}\)

have

\(A': \forall xs \in \text{traces P}, \forall X. X \in \text{refusals P xs} = (X \cap \text{next-events P xs} = \{\})\)

using \(A\) by (simp add: deterministic-def)

**hence** \(\forall X. X \in \text{refusals P xs} = (X \cap \text{next-events P xs} = \{\})\) using \(E\) ..

**hence** \(\{x\} \in \text{refusals P xs} = (\{x\} \cap \text{next-events P xs} = \{\})\) ..

moreover have \(ys \in \text{traces P using C and E by simp}\)

with \(A'\) have \(\forall X. X \in \text{refusals P ys} = (X \cap \text{next-events P ys} = \{\})\) ..

**hence** \(\{x\} \in \text{refusals P ys} = (\{x\} \cap \text{next-events P ys} = \{\})\) ..

moreover have \(\{x\} \cap \text{next-events P ys} = \{x\} \cap \text{next-events P ys}\)

**proof** (simp add: set-eq-iff, rule allI, rule iffI, erule-tac [[] conjE, simp-all)

assume \(x \in \text{next-events P xs}\)

**hence** \(x \in \text{next-dom-events P D u xs using D by (simp add: next-dom-events-def)\)

**hence** \(x \in \text{next-dom-events P D u ys using C by simp}\)

**thus** \(x \in \text{next-events P ys by (simp add: next-dom-events-def)\)

next

assume \(x \in \text{next-events P ys}\)

**hence** \(x \in \text{next-dom-events P D u ys using D by (simp add: next-dom-events-def)\)

**hence** \(x \in \text{next-dom-events P D u xs using C by simp}\)

**thus** \(x \in \text{next-events P xs by (simp add: next-dom-events-def)\)

qed

ultimately show \(\{(x) \in \text{refusals P xs} = (\{x\} \in \text{refusals P ys) by simp}\)

next

assume \(D: xs \notin \text{traces P}\)

**hence** \(\forall X. (xs, X) \notin \text{failures P by (simp add: traces-def Domain-iff)\)

**hence** \(\text{refusals P xs} = \{\} by (rule-tac equals0I, simp add: refusals-def)\)

moreover have \(ys \notin \text{traces P using C and D by simp}\)

**hence** \(\forall X. (ys, X) \notin \text{failures P by (simp add: traces-def Domain-iff)\)

**hence** \(\text{refusals P ys} = \{\} by (rule-tac equals0I, simp add: refusals-def)\)

ultimately show \((\{x\} \in \text{refusals P xs} = (\{x\} \in \text{refusals P ys) by simp}\)

qed

qed

qed

**lemma** \(d-wfc-equals-dwfc:\)

**assumes** \(A: \text{deterministic P}\)

**shows** weakly-future-consistent \(P I D R = \text{d-weakly-future-consistent P I D R}\)

**proof** (rule iffI, simp-all only: d-weakly-future-consistent-def, rule ballI, (rule allI)+, rule implI, rule conjI, rule wfc-traces, assumption+,
simp-all add: weakly-future-consistent-def del: ball-simps

assumes B: \(\forall u. (u \in D \cap (- I)) \quad \text{range} \quad D. \quad \forall x. \quad (x, y) \in R \quad u \rightarrow \quad (x \in \text{traces} \quad P) \quad = \quad (y \in \text{traces} \quad P) \quad \land \quad \text{next-dom-events} \quad D \quad u \quad x = \quad \text{next-dom-events} \quad D \quad u \quad y \quad s\)

shows \(\forall u. (u \in D \cap (- I)) \quad \text{range} \quad D. \quad \forall x. \quad (x, y) \in R \quad u \rightarrow \quad \text{ref-dom-events} \quad D \quad u \quad x = \quad \text{ref-dom-events} \quad D \quad u \quad y \quad s\)

proof (rule ballI, (rule allI)+, rule impI, simp (no-asn-simp) add: ref-dom-events-def set-eq-iff, rule allI)

fix \(u \quad x \quad y \quad s\)

assumes u: \(u \in \text{range} \quad D \cap (- I) \quad \text{range} \quad D\)

with B have \(\forall x. \quad y. \quad (x, y) \in R \quad u \rightarrow \quad (x \in \text{traces} \quad P) \quad = \quad (y \in \text{traces} \quad P) \quad \land \quad \text{next-dom-events} \quad D \quad u \quad x = \quad \text{next-dom-events} \quad D \quad u \quad y \quad s\)

hence \((x, y) \in R \quad u \rightarrow \quad (x \in \text{traces} \quad P) \quad = \quad (y \in \text{traces} \quad P) \quad \land \quad \text{next-dom-events} \quad D \quad u \quad x = \quad \text{next-dom-events} \quad D \quad u \quad y \quad s\)

by blast

moreover assume \((x, y) \in R \quad u\)

ultimately have C: \((x \in \text{traces} \quad P) \quad = \quad (y \in \text{traces} \quad P) \quad \land \quad \text{next-dom-events} \quad D \quad u \quad x = \quad \text{next-dom-events} \quad D \quad u \quad y \quad s\)

show \((u \quad = \quad D \quad x \quad \land \quad \{x\} \quad \in \quad \text{refusals} \quad P \quad x)\quad = \quad (u \quad = \quad D \quad x \quad \land \quad \{x\} \quad \in \quad \text{refusals} \quad P \quad y)\)

proof (cases u = D x, simp-all, cases xs \(\in\) traces P)

assume D: \(u \quad = \quad D \quad x\) and E: \(x \quad \in \quad \text{traces} \quad P\)

have A': \(\forall x. \quad (x \in \text{traces} \quad P. \quad \forall X. \quad \text{next-events} \quad P \quad x = \quad \{\}\) using A by (simp add: deterministic-def)

hence \(\forall X. \quad (x \in \text{traces} \quad P \quad x = \quad \{\}) \quad \land \quad \text{next-events} \quad P \quad x \quad y = \quad \{\}\) using E ..

hence \(\{x\} \quad \in \quad \text{refusals} \quad P \quad x = \quad \{\} \quad \land \quad \text{next-events} \quad P \quad y = \quad \{\}\) ..

moreover have \(ys \quad \in \quad \text{traces} \quad P\) using C and E by simp

with A' have \(\forall X. \quad (x \quad \in \quad \text{traces} \quad P \quad y = \quad \{\})\) ..

hence \(\{x\} \quad \in \quad \text{refusals} \quad P \quad y = \quad \{\} \quad \land \quad \text{next-events} \quad P \quad y = \quad \{\}\) ..

moreover have \(\{x\} \quad \in \quad \text{next-events} \quad P \quad x = \quad \{\} \quad \land \quad \text{next-events} \quad P \quad y = \quad \{\}\)

proof (simp add: set-eq-iff, rule allI, rule iffI,erule-tac [!] conjE, simp-all)

assume x: \(\in \quad \text{next-events} \quad P \quad x\)

hence x: \(\in \quad \text{next-dom-events} \quad D \quad u \quad x = \quad D \quad y\) using D by (simp add: next-dom-events-def)

hence x: \(\in \quad \text{next-dom-events} \quad D \quad u \quad y\) using C by simp

thus x: \(\in \quad \text{next-events} \quad P \quad y\) by (simp add: next-dom-events-def)

next assume x: \(\in \quad \text{next-events} \quad P \quad y\)

hence x: \(\in \quad \text{next-dom-events} \quad D \quad u \quad y\) using D by (simp add: next-dom-events-def)

hence x: \(\in \quad \text{next-dom-events} \quad D \quad u \quad x = \quad D \quad y\) using C by simp

thus x: \(\in \quad \text{next-events} \quad P \quad x\) by (simp add: next-dom-events-def)

qed

ultimately show \(\{x\} \quad \in \quad \text{refusals} \quad P \quad x = \quad \{\} \quad \in \quad \text{refusals} \quad P \quad y\) by simp

next assume D: \(x \quad \notin \quad \text{traces} \quad P\)

hence \(\forall X. \quad (x, X) \quad \notin \quad \text{failures} \quad P\) by (simp add: traces-def Domain-iff)

hence \(\text{refusals} \quad P \quad x = \quad \{\}\) by (rule-tac equals0I, simp add: refusals-def)

moreover have \(ys \quad \notin \quad \text{traces} \quad P\) using C and D by simp

48
hence \( \forall X. (ys, X) \notin \text{failures } P \) by (simp add: traces-def Domain-iff)
hence \( \text{refusals } P \ ys = \{\} \) by (rule-tac equals0I, simp add: refusals-def)
ultimately show \( \{x\} \in \text{refusals } P \ xs = \{\}\) 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 \( d\text{-weakly-future-consistent} \) in place of \( \text{weakly-future-consistent} \). 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 \( d\text{-implies-ruc} \):
  assumes \( A: \text{deterministic } P \)
  shows \( \text{ref-union-closed } P \)
proof (subst ref-union-closed-def, (rule allI)+, (rule impI)+, erule exE)
  fix \( xs \ A \ X \)
  have \( \forall xs \in \text{traces } P. \forall X. X \in \text{refusals } P \ xs = (X \cap \text{next-events } P \ xs = \{\}) \)
    using \( A \) by (simp add: deterministic-def)
  moreover assume \( B: \forall X \in A. (xs, X) \in \text{failures } P \) and \( X \in A \)
  hence \( (xs, X) \in \text{failures } P \) ..
  hence \( xs \in \text{traces } P \) by (rule failures-traces)
  ultimately have \( C: \forall X. X \in \text{refusals } P \ xs = (X \cap \text{next-events } P \ xs = \{}) \) ..
  have \( D: \forall X \in A. X \cap \text{next-events } P \ xs = \{\} \)
proof
  fix \( X \)
  assume \( X \in A \)
  with \( D \) have \( (xs, X) \in \text{failures } P \) ..
  hence \( X \in \text{refusals } P \ xs \) by (simp add: refusals-def)
  thus \( X \cap \text{next-events } P \ xs = \{\} \) using \( C \) by simp
  qed
  have \( (\bigcup X \in A. X) \in \text{refusals } P \ xs = ((\bigcup X \in A. X) \cap \text{next-events } P \ xs = \{}) \)
    using \( C \) ..
  hence \( E: (xs, \bigcup X \in A. X) \in \text{failures } P = \)
    \( (\bigcup X \in A. X) \cap \text{next-events } P \ xs = \{\}) \)
    by (simp add: refusals-def)
  show \( (xs, \bigcup X \in A. X) \in \text{failures } P \)
proof (rule ssubst [OF \( E \)], rule equals0I, erule IntE, erule UN-E)
  fix \( x \)
  assume \( X \in A \)
  with \( D \) have \( X \cap \text{next-events } P \ xs = \{\} \) ..
  moreover assume \( x \in X \) and \( x \in \text{next-events } P \ xs \)
  hence \( x \in X \cap \text{next-events } P \ xs \) ..
  hence \( \exists x. x \in X \cap \text{next-events } P \ xs \) ..
  hence \( X \cap \text{next-events } P \ xs \neq \{\} \) by (subst ex-in-conv [symmetric])
ultimately show \( \text{False by contradiction} \)

\[ \text{qed} \]

\[ \text{qed} \]

**Theorem** \( \text{d-ipurge-unwinding:} \)

- **Assumes** \( A: \text{deterministic } P \)
- **Shows** \( \text{secure } P I D = \text{d-weakly-future-consistent } P I D \) (rel-ipurge \( P I D \))

**Proof** (insert d-ufc-equals-defc \( \text{[of } P I D \text{ rel-ipurge } P I D, OF A], \text{erule subst} \))

\[ \text{qed (insert d-implies-ruc [OF A], rule ipurge-unwinding)} \]

### 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 \( \text{ts-process} \), converting the input set of lists into a process, and a predicate \( \text{trace-set} \), returning \( \text{True} \) 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 \( \text{ts-process } T \) where \( \text{trace-set } T \) matches \( \text{True} \).

**Definition** \( \text{ts-process :: 'a list set ⇒ 'a process where} \)

\( \text{ts-process } T \equiv \text{Abs-process} \) (\( \{(xs, X). xs \in T \land (\forall x \in X. xs \# [x] \notin T)\}, \})

**Definition** \( \text{trace-set :: 'a list set ⇒ bool where} \)

\( \text{trace-set } T \equiv [] \ fl T \land (\forall xs x. xs \# [x] \in T \longrightarrow xs \in T) \)

**Lemma** \( \text{ts-process-rep:} \)

- **Assumes** \( A: \text{trace-set } T \)
- **Shows** \( \text{Rep-process } (\text{ts-process } T) =\)
  \( \{(xs, X). xs \in T \land (\forall x \in X. xs \# [x] \notin T)\}, \})\)


\[ \text{show } [] \in T \text{ using } A \text{ by (simp add: trace-set-def)} \]

**Next**

- **Show** \( \forall xs. (\exists x. xs \# [x] \in T \land (\exists X. \forall x' \in X. xs \# [x, x'] \notin T)) \longrightarrow xs \in T \)

**Proof** (rule allI, rule impI, erule exE, erule conjE)
fix $xs \ x$

have $\forall \ y. xs \ y \in T \to xs \in T$ using $A$ by (simp add: trace-set-def)

hence $xs @ [x] \in T \to xs \in T$ by blast

moreover assume $xs @ [x] \in T$

ultimately show $xs \in T$ ..

qed

def

next

show $\forall \ x. xS \ x \in T \land (\exists \ Y. (\forall x \in Y. xs @ [x] \notin T) \land X \subseteq Y) \to
(\forall x \in X. xs @ [x] \notin T)$

proof ((rule allI)+, rule impI, (erule conjE, (erule exE)+, rule ballI)

fix $xs \ X \ Y$

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 $T \to$

failures (ts-process $T) = \{(xs, X). xs \in T \land (\forall x \in X. xs @ [x] \notin T)\}$

by (drule ts-process-rep, simp add: failures-def)

lemma ts-process-futures:

trace-set $T \to$

futures (ts-process $T) xs =
\{(ys, Y). xs @ ys \in T \land (\forall y \in Y. xs @ ys @ [y] \notin T)\}$

by (simp add: futures-def ts-process-failures)

lemma ts-process-traces:

trace-set $T \to$ 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 $T \to$ xs \in T \to

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 $T \to$ (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 \rightarrow \text{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 \( \forall x \in X. \text{xs @ [x]} \notin T \)
thus \( X \cap \{x. \text{xs @ [x]} \in T\} = {} \)
by (rule-tac equals0I, erule-tac IntE, simp)

next
fix xs X
assume A: \( X \cap \{x. \text{xs @ [x]} \in T\} = {} \)
show \( \forall x \in X. \text{xs @ [x]} / \in T \)
proof (rule ballI, rule notI)

fix x
assume x \in X and xs @ [x] \notin T

hence x \in X \cap \{x. \text{xs @ [x]} \in T\} by simp
moreover have x \notin X \cap \{x. \text{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
\[ \text{divergences P} \equiv \text{snd (Rep-process P)} \]

lemma d-divergences:
assumes A: \text{deterministic P}
shows \( \text{divergences P} = {} \)
proof (subst divergences-def, rule equals0I)

fix xs
have B: \text{Rep-process P} \in \text{process-set} (\text{is P' \in -}) by (rule Rep-process)

hence \( \forall xs. \exists x. \text{xs @ [x]} \in \text{snd P'} \rightarrow \text{xs @ [x]} \in \text{snd P'} \)
by (simp add: process-set-def process-prop-5-def)

hence \( \exists x. \text{xs @ [x]} \in \text{snd P'} \rightarrow \text{xs @ [x]} \in \text{snd P'} \)
then obtain x where \( \text{xs @ [x]} \in \text{snd P'} \) using A by (rule equals0D)

moreover have C: \( \text{xs @ [x]} \in \text{snd P'} \)
ultimately have D: \( \text{xs @ [x]} \in \text{snd P'} \)

have E: \( \forall x X. \text{xs @ [x]} \in \text{snd P'} \rightarrow (x, X) \in \text{fst P'} \)
using B by (simp add: process-set-def process-prop-6-def)

hence \( \text{xs @ [x]} \in \text{snd P'} \rightarrow (x, \{x\}) \in \text{fst P'} \) by blast

hence \( \text{xs @ [x]} \in \text{refusals P} \)
using C by (drule-tac mp, simp-all add: failures-def refusals-def)

moreover have \( \text{xs @ [x]} \in \text{snd P'} \rightarrow (\text{xs @ [x]}, \{\}) \in \text{fst P'} \)
using E by blast

hence \( \text{xs @ [x]}, \{\} \) \in failures P
using D by (drule-tac mp, simp-all add: failures-def)

hence P: \( \text{xs @ [x]} \in \text{traces P} \) by (rule failures-traces)

hence \( \{x\} \cap \text{next-events P} \text{xs} \neq \{\} \) by (simp add: next-events-def)
ultimately have G: \( \{\text{x} \in \text{refusals P} \text{xs} \neq \{\} \cap \text{next-events P} \text{xs} = \{\} \)
proof

lemma \(\text{d-implies-ts-process-traces}\):

deterministic \(P \implies\) ts-process (traces \(P\)) = \(P\)

proof (simp add: Rep-process-def deterministic-def)

assume \(A: \forall \mathbf{x}s \in \text{traces} \ P, \forall X. \ X \in \text{refusals} \ P \ \mathbf{x}s = (X \cap \text{next-events} \ P \ \mathbf{x}s = \{\})\)

assume \(B: \text{trace-set} (\text{traces} \ P)\)

hence \(C: \text{traces} (\text{ts-process} (\text{traces} \ P)) = \text{traces} \ P\) 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 \(\mathbf{x}s X\)

assume \(D: (\mathbf{x}s, X) \in \text{failures} (\text{ts-process} (\text{traces} \ P))\)

hence \(\mathbf{x}s \in \text{traces} (\text{ts-process} (\text{traces} \ P))\) by (rule failures-traces)

hence \(E: \mathbf{x}s \in \text{traces} \ P\) using \(C\) by simp

with \(B\) have \(\text{refusals} (\text{ts-process} (\text{traces} \ P)) \ \mathbf{x}s = \{X. \ \forall x \in X, \mathbf{x}s \oplus [x] \notin \text{traces} \ P\}\)

by (rule ts-process-refusals)

moreover have \(X \in \text{refusals} (\text{ts-process} (\text{traces} \ P)) \ \mathbf{x}s\)

using \(D\) by (simp add: refusals-def)

ultimately have \(\forall \mathbf{x}s \in X, \mathbf{x}s \oplus [x] \notin \text{traces} \ P\) by simp

hence \(X \cap \text{next-events} \ P \ \mathbf{x}s = \{\}\)

qed
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 = {}) ..
moreover have X ∈ refusals P xs using D by (simp add: refusals-def)
ultimately have P: 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} by simp using P 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

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 =>
secure \( (ts\text{-}process\ T)\ ID = \)
d-weakly-future-consistent \( (ts\text{-}process\ T)\ ID\ (rel\text{-}ipurge\ (ts\text{-}process\ T)\ ID)\)
by \( (\text{rule\ d\text{-}ipurge\text{-}unwinding},\ \text{rule\ ts\text{-}process\text{-}d})\)
end

References


