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

September 19, 2015

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
1.1 Propaedeutic definitions and lemmas .......................... 3
1.2 Additional intransitive purge functions and their properties . 10
1.3 A domain-relation map based on intransitive purge ............ 25
1.4 The Ipurge Unwinding Theorem: proof of condition sufficiency 27
1.5 The Ipurge Unwinding Theorem: proof of condition necessity 40

2 The Ipurge Unwinding Theorem for deterministic and trace set processes 44
2.1 Deterministic processes ........................................... 45
2.2 Trace set processes ................................................. 50

1 The Ipurge Unwinding Theorem in its general form

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

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

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

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

The contents of this paper are based on those of [6]. The salient points of definitions and proofs are commented; for additional information, cf. Isabelle documentation, particularly [5], [4], [3], and [2].
For the sake of brevity, given a function \( F \) of type \( 'a_1 \Rightarrow \ldots \Rightarrow 'a_m \Rightarrow 'a_{m+1} \Rightarrow \ldots \Rightarrow 'a_n \Rightarrow 'b \), the explanatory text may discuss of \( F \) using attributes that would more exactly apply to a term of type \( 'a_{m+1} \Rightarrow \ldots \Rightarrow 'a_n \Rightarrow 'b \). In this case, it shall be understood that strictly speaking, such attributes apply to a term matching pattern \( F \ a_1 \ldots a_m \).

### 1.1 Propaedeutic definitions and lemmas

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

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

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

definition ref-union-closed :: 'a process ⇒ bool where
  ref-union-closed \( P \) ≡
  \( \forall \) xs A. (\( \exists \) X. X ∈ A) →→ (\( \forall \) X ∈ A. (xs, X) ∈ failures \( P \)) →→
  (xs, \( \bigcup \) X ∈ A. X) ∈ failures \( P \)

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

definition next-dom-events ::
  'a process ⇒ ('a ⇒ 'd) ⇒ 'd ⇒ 'a list ⇒ 'a set where
  next-dom-events \( P \) \( D \) \( u \) \( xs \) ≡ \{ x. u = D x \land x ∈ next-events \( P \) \( xs \)\}

definition ref-dom-events ::
  'a process ⇒ ('a ⇒ 'd) ⇒ 'd ⇒ 'a list ⇒ 'a set where
  ref-dom-events \( P \) \( D \) \( u \) \( xs \) ≡ \{ x. u = D x \land \{ x \} ∈ refusals \( P \) \( xs \)\}

definition future-consistent ::
  'a process ⇒ ('a ⇒ 'd) ⇒ ('a, 'd) dom-rel-map ⇒ bool where
  future-consistent \( P \) \( D \) \( R \) ≡
  \( \forall \) u ∈ range \( D \). \( \forall \) xs ys. (xs, ys) ∈ \( R \) \( u \) →→
  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 ⇒ ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ ('a, 'd) dom-rel-map ⇒ bool where
Here below are some lemmas propaedeutic for the proof of the Ipurge Unwinding Theorem, just involving constants defined in [6].

**lemma** process-rule-2-traces:
\[ \text{xs @ xs' \in traces P \implies xs \in traces P} \]

**proof** (simp add: traces-def Domain-iff, erule exE, rule-tac x = {} in exI)

**qed** (rule process-rule-2-failures)

**lemma** process-rule-4 [rule-format]:
\[ (xs, X) \in failures P \implies (xs @ [x], \{\}) \in failures P \lor (xs, \text{insert } x X) \in failures P \]

**proof** (simp add: failures-def)

have \( \text{Rep-process } P \in \text{process-set } (\text{is } ?P' \in -) \) by (rule Rep-process)

hence \( \forall xs X. (xs, X) \in \text{fst } ?P' \to (xs @ [x], \{\}) \in \text{fst } ?P' \lor (xs, \text{insert } x X) \in \text{fst } ?P' \)

by (simp add: process-set-def process-prop-4-def)

thus \( (xs, X) \in \text{fst } ?P' \to (xs @ [x], \{\}) \in \text{fst } ?P' \lor (xs, \text{insert } x X) \in \text{fst } ?P' \)

by blast

**qed**

**lemma** failures-traces:
\[ (xs, X) \in failures P \implies xs \in traces P \]

by (simp add: traces-def Domain-iff, rule exI)

**lemma** traces-failures:
\[ xs \in traces P \implies (xs, \{\}) \in failures P \]

**proof** (simp add: traces-def Domain-iff, erule exE)

**qed** (erule process-rule-3, simp)

**lemma** sinks-interference [rule-format]:
\[ D x \in \text{sinks } I D u xs \implies (u, D x) \in I \lor (\exists v \in \text{sinks } I D u xs. (v, D x) \in I) \]

**proof** (induction xs rule: rev-induct, simp, rule impl)

fix \( x' \) \( \text{xs} \)

assume

\( A: D x \in \text{sinks } I D u xs \implies (u, D x) \in I \lor (\exists v \in \text{sinks } I D u xs. (v, D x) \in I) \) and
\( B: D x \in \text{sinks } I D u (xs @ [x']) \)

show \( (u, D x) \in I \lor (\exists v \in \text{sinks } I D u (xs @ [x']). (v, D x) \in I) \)

**proof** (cases \( (u, D x') \in I \lor (\exists v \in \text{sinks } I D u xs. (v, D x') \in I) \))

**case** True

hence \( D x = D x' \lor D x \in \text{sinks } I D u xs \) using \( B \) by simp
moreover {  
  assume $C: D x = D x'$  
  have $\text{?thesis}$ using True  
  proof (rule disjE, erule-tac [2] bexE)  
    assume $(u, D x') \in I$  
    hence $(u, D x) \in I$ using $C$ by simp  
  thus $\text{?thesis}$ ..  
  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 $\text{?thesis}$ ..  
  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 $\text{?thesis}$  
  proof (rule disjE, erule-tac [2] bexE)  
    assume $(u, D x) \in I$  
    thus $\text{?thesis}$ ..  
  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 $\text{?thesis}$ ..  
  qed  
}  
ultimately show $\text{?thesis}$ ..  
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 $\text{?thesis}$ using $C$ by simp  
  qed  
  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 proven.

In the definition of predicate \texttt{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.

\begin{verbatim}
lemma assumes A: \( \forall xs A. (\forall X \in A. (xs, X) \in \text{failures P}) \rightarrow (xs, \bigcup X \in A. X) \in \text{failures P} \)
shows \( \forall xs. xs \in \text{traces P} \)
proof
fix xs
have \( (\forall X \in \{\}. (xs, X) \in \text{failures P}) \rightarrow (xs, \bigcup X \in \{\}. X) \in \text{failures P} \)
using A by blast
moreover have \( \forall X \in \{\}. (xs, X) \in \text{failures P} \) by simp
ultimately have \( (xs, \bigcup X \in \{\}. X) \in \text{failures P} \)
thus \( xs \in \text{traces P} \) by (rule failures-traces)
qed

lemma \texttt{traces-dom-events}: assumes A: \( u \in \text{range D} \)
shows \( xs \in \text{traces P} = (\text{next-dom-events P D u xs} \cup \text{ref-dom-events P D u xs} \neq \{\}) \)
(is \(-= (\{S \neq \{\})) \)
proof
have \( \exists x. u = D x \) using A by (simp add: image-def)
then obtain x where B: \( u = D x \)
assume \( xs \in \text{traces P} \)
hence \( (xs, \{\}) \in \text{failures P} \) by (rule traces-failures)
hence \( (xs @ [x], \{\}) \in \text{failures P} \lor (xs, \{x\}) \in \text{failures P} \) by (rule process-rule-4)
moreover {
assume \( (xs @ [x], \{\}) \in \text{failures P} \)
hence \( xs @ [x] \in \text{traces P} \) by (rule failures-traces)
hence \( x \in \text{next-dom-events P D u xs} \)
using B by (simp add: next-dom-events-def next-events-def)
hence \( x \in S \)
}
moreover {
assume \( (xs, \{x\}) \in \text{failures P} \)
}\end{verbatim}
hence $x \in \text{ref-dom-events } P \ D \ u \ xs$
using $B$ by (simp add: ref-dom-events-def refusals-def)
hence $x \in ?S$ ..
}\ultimately have $x \in ?S$ ..
\hence $\exists x. \ x \in ?S$ ..
\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 $x \in \text{next-dom-events } P \ D \ u \ xs$
\hence $xs @ [x] \in \text{traces } P$ by (simp add: next-dom-events-def next-events-def)
\hence $xs \in \text{traces } P$ by (rule process-rule-2-traces)
}
moreover {
assume $x \in \text{ref-dom-events } P \ D \ u \ xs$
\hence $(xs, \{x\}) \in \text{failures } P$ by (simp add: ref-dom-events-def refusals-def)
\hence $xs \in \text{traces } P$ by (rule failures-traces)
}
ultimately show $xs \in \text{traces } P$ ..
qed

\textbf{lemma fc-traces:}
\textbf{assumes}
$A$: future-consistent $P \ D \ R$ and
$B$: $u \in \text{range } D$ and
$C$: $(xs, \ ys) \in R \ u$
\textbf{shows} $(xs \in \text{traces } P) = (ys \in \text{traces } P)$
\textbf{proof} –
\have $\forall u \in \text{range } D. \ \forall xs ys. \ (xs, \ ys) \in R \ u \longrightarrow$
next-dom-events $P \ D \ u \ xs = \text{next-dom-events } P \ D \ u \ ys \land$
\hence $\text{ref-dom-events } P \ D \ u \ xs = \text{ref-dom-events } P \ D \ u \ ys$
\using $A$ by (simp add: future-consistent-def)
\hence $\forall xs ys. \ (xs, \ ys) \in R \ u \longrightarrow$
next-dom-events $P \ D \ u \ xs = \text{next-dom-events } P \ D \ u \ ys \land$
\hence $\text{ref-dom-events } P \ D \ u \ xs = \text{ref-dom-events } P \ D \ u \ ys$
\using $B$ ..
\hence $(xs, \ ys) \in R \ u \longrightarrow$
\hence $\text{next-dom-events } P \ D \ u \ xs = \text{next-dom-events } P \ D \ u \ ys \land$
\hence $\text{ref-dom-events } P \ D \ u \ xs = \text{ref-dom-events } P \ D \ u \ ys$
\by blast
\hence $\text{next-dom-events } P \ D \ u \ xs = \text{next-dom-events } P \ D \ u \ ys \land$
\hence $\text{ref-dom-events } P \ D \ u \ xs = \text{ref-dom-events } P \ D \ u \ ys$
\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 $xs \in \text{traces } P =$
\hspace{1em} ($\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\} )$
using $B$ by (rule traces-dom-events)
moreover have $ys \in \text{traces } P =$
\hspace{1em} ($\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\} )$
using $B$ by (rule traces-dom-events)
ultimately show $\text{thesis by simp}$
qed

lemma wfc-traces:
assumes
\hspace{1em} $A$: weakly-future-consistent $P \ I \ D \ R$
\hspace{1em} $B$: $u \in \text{range } D \cap (-I) \cap \text{range } D$
\hspace{1em} $C$: $(xs, ys) \in R \ u$
shows $(xs \in \text{traces } P) = (ys \in \text{traces } P)$
proof
\hspace{1em} have $\forall u \in \text{range } D \cap (-I) \cap \text{range } D \ \Rightarrow$
\hspace{2em} $\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\} $
\hspace{1em} using $A$ by (simp add: weakly-future-consistent-def)
\hspace{1em} hence $\forall xs ys. (xs, ys) \in \text{range } D \ \Rightarrow$
\hspace{2em} $\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\} $
\hspace{1em} using $B$ ..
\hspace{1em} hence $(xs, ys) \in \text{range } D \ \Rightarrow$
\hspace{2em} $\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\} $
\hspace{1em} by blast
\hspace{1em} hence $\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\}$
\hspace{1em} using $C$ ..
\hspace{1em} hence $(xs, ys) \in \text{range } D \ \Rightarrow$
\hspace{2em} $\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\} $
\hspace{1em} by simp
moreover have $B^\prime$: $u \in \text{range } D$ using $B$ ..
\hspace{1em} hence $xs \in \text{traces } P =$
\hspace{2em} ($\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\} )$
\hspace{1em} by (rule traces-dom-events)
moreover have $ys \in \text{traces } P =$
\hspace{2em} ($\text{next-dom-events } P \cup \text{ref-dom-events } P \neq \{\} )$
\hspace{1em} using $B^\prime$ by (rule traces-dom-events)
ultimately show $\text{thesis by simp}$
qed

lemma fc-implies-wfc:
\hspace{1em} $\text{future-consistent } P \ D \ R \Rightarrow \text{weakly-future-consistent } P \ I \ D \ R$
\hspace{1em} 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} singleton-set :: 'a set \Rightarrow 'a set set
\begin{align*}
singleton-set X & \equiv \{ Y. \exists x \in X. Y = \{x\} \}
\end{align*}

\textbf{lemma} singleton-set-some:
\begin{align*}
(\exists Y. Y \in \text{singleton-set } X) & = (\exists x. x \in X)
\end{align*}
\textbf{proof} (rule iffI, simp-all add: singleton-set-def, erule_tac \[!] exE, erule bexE)
\begin{align*}
\text{fix } x & \quad \text{assume } x \in X \\
\text{thus } \exists x. x \in X & \quad \ldots
\end{align*}
\textbf{next}
\begin{align*}
\text{fix } x & \quad \text{assume } A: x \in X \\
\text{have } \{x\} & = \{x\} \ldots \\
\text{hence } \exists x'. x \in X. \{x\} = \{x'\} & \quad \text{using } A \ldots \\
\text{thus } \exists Y. \exists x' \in X. Y = \{x'\} & \quad \text{by (rule exI)}
\end{align*}
\textbf{qed}

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

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

```latex
\begin{align*}
\text{function } & \text{sinks-aux} :: \\
\quad ('d \times 'd) \text{ set } \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd \text{ set } \Rightarrow 'a \text{ list } \Rightarrow 'd \text{ set} \text{ where} \\
\quad \text{sinks-aux} - - U [] &= U \mid \\
\quad \text{sinks-aux} I D U (xs @ [x]) &= (\text{if } \exists v \in \text{sinks-aux} I D U xs. (v, D x) \in I \\
\quad \text{then } \text{sinks-aux} I D U xs \\
\quad \text{else } \text{sinks-aux} I D U xs) \\
\text{proof } & (\text{atomize-elim, simp-all add: split-paired-all}) \\
\text{qed } & (\text{rule rev-cases, rule disjI1, assumption, simp})
\end{align*}
```

**termination by lexicographic-order**

```latex
\begin{align*}
\text{function } & \text{ipurge-tr-aux} :: \\
\quad ('d \times 'd) \text{ set } \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd \text{ set } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list} \text{ where} \\
\quad \text{ipurge-tr-aux} - - [] &= [] \\
\quad \text{ipurge-tr-aux} I D U (xs @ [x]) &= (\text{if } \exists v \in \text{sinks-aux} I D U xs. (v, D x) \in I \\
\quad \text{then } \text{ipurge-tr-aux} I D U xs \\
\quad \text{else } \text{ipurge-tr-aux} I D U xs @ [x]) \\
\text{proof } & (\text{atomize-elim, simp-all add: split-paired-all}) \\
\text{qed } & (\text{rule rev-cases, rule disjI1, assumption, simp})
\end{align*}
```

**termination by lexicographic-order**

```latex
\begin{align*}
\text{definition } & \text{ipurge-ref-aux} :: \\
\quad ('d \times 'd) \text{ set } \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd \text{ set } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list} \text{ where} \\
\quad \text{ipurge-ref-aux} I D U xs X \equiv \\
\quad \{ x \in X. \forall v \in \text{sinks-aux} I D U xs. (v, D x) \notin I \}
\end{align*}
```

```latex
\begin{align*}
\text{definition } & \text{unaffected-domains} :: \\
\quad ('d \times 'd) \text{ set } \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd \text{ set } \Rightarrow 'a \text{ list } \Rightarrow 'd \text{ set} \text{ where} \\
\quad \text{unaffected-domains} I D U xs \equiv \\
\quad \{ u \in \text{range D}. \forall v \in \text{sinks-aux} I D U xs. (v, u) \notin I \}
\end{align*}
```

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

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

- `sources` outputs the set of the domains of the events in `xs` allowed to affect `u`;
- `ipurge-tr-rev` outputs the sublist of `xs` obtained by recursively deleting the events not allowed to affect `u`, as detected via function `sources`.
In other words, these functions follow Rushby’s ones sources and ipurge [8], formalized in [6] as c-sources and c-ipurge. The only difference consists of dropping the implicit supposition that the noninterference policy be reflexive, as done in the definition of CPS noninterference security [6]. This goal is achieved by defining the output of function sources, when it is applied to the empty list, as being the empty set rather than the singleton comprised of the input domain.

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

\begin{verbatim}
primrec sources :: ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ 'd ⇒ 'a list ⇒ 'd set where
  sources - - - [] = {} |
  sources I D u (x # xs) = (if (D x, u) ∈ I ∨ (∃v ∈ sources I D u xs. (D x, v) ∈ I)
    then insert (D x) (sources I D u xs)
    else sources I D u xs)

primrec ipurge-tr-rev :: ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ 'd ⇒ 'a list ⇒ 'a list where
  ipurge-tr-rev - - - [] = [] |
  ipurge-tr-rev I D u (x # xs) = (if D x ∈ sources I D u (x # xs)
    then x # ipurge-tr-rev I D u xs
    else ipurge-tr-rev I D u xs)

primrec sources-aux :: ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ 'd ⇒ 'a list ⇒ 'd set where
  sources-aux - - U [] = U |
  sources-aux I D U (x # xs) = (if ∃v ∈ sources-aux I D U xs. (D x, v) ∈ I
    then insert (D x) (sources-aux I D U xs)
    else sources-aux I D U xs)

primrec ipurge-tr-rev-aux :: ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ 'd ⇒ 'a list ⇒ 'a list where
  ipurge-tr-rev-aux - - - [] = [] |
  ipurge-tr-rev-aux I D U (x # xs) = (if ∃v ∈ sources-aux I D U xs. (D x, v) ∈ I
    then x # ipurge-tr-rev-aux I D U xs
    else ipurge-tr-rev-aux I D U xs)
\end{verbatim}

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.

12
lemma sinks-aux-subset:
  \( U \subseteq \text{sinks-aux } I D U \) \( \mathcal{X} \)
proof (induction \( \mathcal{X} \) rule: rev-induct, simp-all, rule impI)
qed (rule subset-insertI2)

lemma sinks-aux-single-dom:
  \( \text{sinks-aux } I D \{ u \} \mathcal{X} = \text{insert } u (\text{sinks } I D u \mathcal{X}) \)
by (induction \( \mathcal{X} \) 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:
  \( \text{sinks-aux } I D U \{ v \} \mathcal{X} = (\text{if } \exists v \in U. (v, D x) \in I \text{ then insert } (D x) U \mathcal{X} \text{ else sinks-aux } I D U \mathcal{X}) \)
proof (induction \( \mathcal{X} \) rule: rev-induct, simp-all add: sinks-aux.single-event del: sinks-aux.simps(2))
  fix \( x' \) \( \mathcal{X} \)
  assume \( A: \text{sinks-aux } I D U [x' \mathcal{X}] = \text{sinks-aux } I D U \mathcal{X} \) (is \( \mathcal{S} = \mathcal{S}' \))
  show \( \text{sinks-aux } I D U [x' \mathcal{X}] = \text{insert } (D x') \mathcal{S} \)
  proof (cases \( \exists v \in \mathcal{S}. (v, D x') \in I \))
    case True
    hence \( \text{sinks-aux } I D U [x' \mathcal{X}] = \text{insert } (D x') \mathcal{S} \) by (simp only: sinks-aux.simps, simp)
    moreover have \( \exists v \in \mathcal{S}' . (v, D x') \in I \) using \( A \) and \( \text{True} \) by simp
    hence \( \text{sinks-aux } I D U [x' \mathcal{X}] = \text{insert } (D x') \mathcal{S}' \) by simp
    ultimately show \( ?\)thesis using \( A \) by simp
  next
    case False
    hence \( \text{sinks-aux } I D U [x' \mathcal{X}] = \mathcal{S} \) by (simp only: sinks-aux.simps, simp)
    moreover have \( \neg (\exists v \in \mathcal{S}' . (v, D x') \in I) \) using \( A \) and \( \text{False} \) by simp
    hence \( \text{sinks-aux } I D U [x' \mathcal{X}] = \mathcal{S}' \) by simp
    ultimately show \( ?\)thesis using \( A \) by simp
  qed
next
  fix \( x' \) \( \mathcal{X} \)
  assume \( A: \text{sinks-aux } I D U [x' \mathcal{X}] = \text{sinks-aux } I D U \mathcal{X} \) (is \( \mathcal{S} = \mathcal{S}' \))
show sinks-aux I D U (x # xs @ [x']) = sinks-aux I D U (xs @ [x'])
proof (cases ∃ v ∈ ?S. (v, D x') ∈ I)
case True
  hence sinks-aux I D U ((x # xs) @ [x']) = insert (D x') ?S
      by (simp only: sinks-aux.simps, simp)
moreover have ∃ v ∈ ?S'. (v, D x') ∈ I using A and True by simp
  hence sinks-aux I D U (xs @ [x']) = insert (D x') ?S' by simp
ultimately show ?thesis using A by simp
next
case False
  hence sinks-aux I D U ((x # xs) @ [x']) = ?S
      by (simp only: sinks-aux.simps, simp)
moreover have ¬ (∃ v ∈ ?S'. (v, D x') ∈ I) using A and False by simp
  hence sinks-aux I D U (xs @ [x']) = ?S' by simp
ultimately show ?thesis using A by simp
qed
qed

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

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

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

\[(\forall u \in U. \neg (\exists v \in D \cdot (X \cup set \, xs). (u, v) \in I)) \rightarrow \]

\[ipurge-ref-aux \, I \, D \, U \, xs \, X = X\]

**proof** (induction \(xs\), simp-all add: \(ipurge-ref-aux-def\) \(sinks-aux-cons\))

**qed** (rule \(\text{impI}\), rule \(\text{equalityI}\), rule-tac ![subsetI], simp-all)

**lemma** \(ipurge-ref-all\):
\[\neg (\exists v \in D \cdot (X \cup set \, xs). (u, v) \in I) \rightarrow ipurge-ref \, I \, D \, u \, xs \, X = X\]

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

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

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

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

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

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

**lemma** \(sources-aux-subset\):
\[U \subseteq sources-aux \, I \, D \, U \, xs\]

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

**lemma** \(sources-aux-append\):
\[sources-aux \, I \, D \, U \, (xs @ ys) = sources-aux \, I \, D \, (sources-aux \, I \, D \, U \, ys) \, xs\]

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

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

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

**lemma** \(ipurge-tr-rev-aux-append\):
\[ipurge-tr-rev-aux \, I \, D \, U \, (xs @ ys) = ipurge-tr-rev-aux \, I \, D \, (sources-aux \, I \, D \, U \, ys) \, xs @ ipurge-tr-rev-aux \, I \, D \, U \, ys\]

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

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

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

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

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

lemma ipurge-tr-rev-aux-append-nil-1 [rule-format]:
ipurge-tr-rev-aux I D U ys = [] → ipurge-tr-rev-aux I D U (xs @ ys) = ipurge-tr-rev-aux I D U xs
by (induction xs, simp-all add: ipurge-tr-rev-aux-nil-sources sources-aux-append-nil)

lemma ipurge-tr-rev-aux-first [rule-format]:
ipurge-tr-rev-aux I D U xs = x # ws →
(∃ ys zs. xs = ys @ x # zs ∧
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] ∧
(∃ v ∈ sources-aux I D U zs. (D x, v) ∈ I))
proof (induction xs, simp, rule impI)
fix x’ xs
assume
A: ipurge-tr-rev-aux I D U xs = x # ws →
(∃ ys zs. xs = ys @ x # zs ∧
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] ∧
(∃ v ∈ sources-aux I D U zs. (D x, v) ∈ I)) and
B: ipurge-tr-rev-aux I D U (x’ # xs) = x # ws
show ∃ ys zs. x’ # xs = ys @ x # zs ∧
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] ∧
(∃ v ∈ sources-aux I D U zs. (D x, v) ∈ I)
proof (cases ∃ v ∈ sources-aux I D U xs. (D x’, v) ∈ I)
case True
moreover from this have x’ = x using B by simp
ultimately have x’ # xs = x # xs ∧
ipurge-tr-rev-aux I D (sources-aux I D U (x # xs)) [] = [] ∧
(∃ v ∈ sources-aux I D U xs. (D x, v) ∈ I)
by simp
thus thesis by blast
next
case False
hence ipurge-tr-rev-aux I D U xs = x # ws using B by simp
with A have ∃ ys zs. xs = ys @ x # zs ∧
ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] ∧
(∃ v ∈ sources-aux I D U zs. (D x, v) ∈ I) ..

then obtain ys and zs where xs = ys @ x # zs ∧
  ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys = [] ∧
(∃ v ∈ sources-aux I D U zs. (D x, v) ∈ I)
by blast

moreover from this have

¬ (∃ v ∈ sources-aux I D (sources-aux I D U (x # zs)) ys. (D x', v) ∈ I)
using False by (simp add: sources-aux-append)

hence ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) (x' # ys) =
  ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) ys
by simp

ultimately have x' # xs = (x' # ys) @ x # zs ∧
  ipurge-tr-rev-aux I D (sources-aux I D U (x # zs)) (x' # ys) = [] ∧
(∃ v ∈ sources-aux I D U zs. (D x, v) ∈ I)
by (simp del: sources-aux.simps)
thus ?thesis by blast
qed

lemma ipurge-tr-rev-aux-last-1 [rule-format]:
ipurge-tr-rev-aux I D U xs = ws @ [x] −→ (∃ v ∈ U. (D x, v) ∈ I)
proof (induction xs rule: rev-induct, simp, rule impI)
  fix xs x'
  assume
    A: ipurge-tr-rev-aux I D U xs = ws @ [x] −→ (∃ v ∈ U. (D x, v) ∈ I) and
    B: (exists-ul: sources-aux.Append) xs @ [x'] = ws @ [x]
  show ∃ v ∈ U. (D x, v) ∈ I
  proof (cases ∃ v ∈ U. (D x', v) ∈ I)
    case True
    hence ipurge-tr-rev-aux I D U (xs @ [x']) =
      ipurge-tr-rev-aux I D (insert (D' x) U) xs @ [x']
    by (simp add: ipurge-tr-rev-aux-append)
    hence 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

lemma ipurge-tr-rev-aux-last-2 [rule-format]:
ipurge-tr-rev-aux I D U xs = ws @ [x] −→ (∃ ys zs. xs = ys @ x # zs ∧ ipurge-tr-rev-aux I D U zs = [])
proof (induction xs rule: rev-induct, simp, rule impI)
  fix xs x'
  assume
A: \text{ipurge-tr-rev-aux} I D U xzs = ws @ [x] \rightarrow
(\exists ys zs. xzs = ys @ x \# zs \land \text{ipurge-tr-rev-aux} I D U zs = []) \land
B: \text{ipurge-tr-rev-aux} I D U (xs @ [x']) = ws @ [x]

\textbf{show} \exists ys zs. xzs @ [x'] = ys @ x \# zs \land \text{ipurge-tr-rev-aux} I D U zs = []

\textbf{proof} (cases \exists v \in U. (D x', v) \in I)

\textbf{case} True
\textbf{hence} \text{ipurge-tr-rev-aux} I D U (xs @ [x']) = 
\text{ipurge-tr-rev-aux} I D U (\text{insert} (D x') U) xs @ [x']
\textbf{by} (simp add: ipurge-tr-rev-aux-append)

\textbf{hence} xs @ [x'] = ys @ x \# zs \land \text{ipurge-tr-rev-aux} I D U zs = []
\textbf{using} B \textbf{by} simp

\textbf{thus} \textbf{?thesis} \textbf{by} blast

\textbf{next}

\textbf{case} False
\textbf{hence} \text{ipurge-tr-rev-aux} I D U (xs @ [x']) = \text{ipurge-tr-rev-aux} I D U xs
\textbf{by} (simp add: ipurge-tr-rev-aux-append)

\textbf{hence} \text{ipurge-tr-rev-aux} I D U xs = ws @ [x] \textbf{using} B \textbf{by} simp

\textbf{with} A \textbf{have} \exists ys zs. xzs = ys @ x \# zs \land \text{ipurge-tr-rev-aux} I D U zs = []
\textbf{then obtain} ys and zs \textbf{where}

C: xzs = ys @ x \# zs \land \text{ipurge-tr-rev-aux} I D U zs = []
\textbf{by} blast

\textbf{hence} xs @ [x'] = ys @ x \# zs @ [x'] \textbf{by} simp

\textbf{moreover have}
\text{ipurge-tr-rev-aux} I D U (zs @ [x']) = \text{ipurge-tr-rev-aux} I D U zs
\textbf{using} False \textbf{by} (simp add: ipurge-tr-rev-aux-append)

\textbf{hence} \text{ipurge-tr-rev-aux} I D U (zs @ [x']) = [] \textbf{using} C \textbf{by} simp

\textbf{ultimately have} xs @ [x'] = ys @ x \# zs @ [x'] \land
\text{ipurge-tr-rev-aux} I D U (zs @ [x']) = []

\textbf{thus} \textbf{?thesis} \textbf{by} blast

\textbf{qed}

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

\textbf{proof} (induction xs, simp, rule impI, simp, erule conjE)

\textbf{fix} x xs
\textbf{assume} \exists u \in U. (D x, u) \in I
\textbf{then obtain} u \textbf{where} A: u \in U \textbf{and} B: (D x, u) \in I
\textbf{have} U \subseteq \text{sources-aux} I D U xs \textbf{by} (rule sources-aux-subset)
\textbf{hence} u \in \text{sources-aux} I D U xs \textbf{using} A
\textbf{with} B \textbf{show} \exists u \in \text{sources-aux} I D U xs. (D x, u) \in I

\textbf{qed}

Here below, further properties of the functions defined above are investigated thanks to the introduction of function \textit{offset}, which searches a list for a given item and returns the offset of its first occurrence, if any, from the first item of the list.
primrec offset :: nat ⇒ 'a ⇒ 'a list ⇒ nat option where
offset - [] = None |
offset n x (y # ys) = (if y = x then Some n else offset (Suc n) x ys)

lemma offset-not-none-1 [rule-format]:
offset k x xs ≠ None → (∃ ys zs. xs = ys @ x # zs)
proof (induction xs arbitrary: k, simp, rule impI)
  fix w xs k
  assume A: ∀ k. offset k x xs ≠ None → (∃ ys zs. xs = ys @ x # zs) and
  B: offset k x (w # xs) ≠ None
  show ∃ ys zs. w # xs = ys @ x # zs
  proof (cases w = x, simp)
    case True
    hence x # xs = [] @ x # xs by simp
    thus ∃ ys zs. x # xs = ys @ x # zs by blast
  next
    case False
    hence offset k x (w # xs) = offset (Suc k) x xs by simp
    hence offset (Suc k) x xs ≠ None using B by simp
    moreover have offset (Suc k) x xs ≠ None → (∃ ys zs. xs = ys @ x # zs)
      using A .
    ultimately have ∃ ys zs. xs = ys @ x # zs by simp
    then obtain ys and zs where xs = ys @ x # zs by blast
    hence w # xs = (w # ys) @ x # zs by simp
    thus ∃ ys zs. w # xs = ys @ x # zs by blast
  qed
qed

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

lemma offset-not-none:
(offset k x xs ≠ None) = (∃ ys zs. xs = ys @ x # zs)
by (rule ifI, erule offset-not-none-1, (erule exE)+, rule offset-not-none-2)

**lemma** offset-addition [rule-format]:
offset \( k \times xs \neq None \rightarrow offset (n + m) \times xs = Some (the (offset n \times xs) + m) \)

**proof** (induction \( xs \) arbitrary; \( k \), \( n \), simp, rule \( \text{impl} \))
\[
\begin{align*}
\text{fix } w &= z \times k \\
\text{assume } A: & \forall n. offset k \times xs \neq None \rightarrow offset (n + m) \times xs = Some (the (offset n \times xs) + m) \text{ and } \\
\text{B: } & offset k \times (w \# zs) \neq None \\
\text{show } & offset (n + m) \times (w \# xs) = Some (the (offset n \times (w \# xs)) + m) \\
\text{proof } & (cases w = x, simp-all) \\
\text{case } & False \\
\text{hence } & offset k \times (w \# xs) = offset (Suc k) \times xs \text{ by simp} \\
\text{hence } & offset (Suc k) \times xs \neq None \text{ using } B \text{ by simp} \\
\text{moreover have } & offset (Suc k) \times xs \neq None \rightarrow offset (Suc n + m) \times xs = Some (the (offset (Suc n) \times xs) + m) \\
\text{using } & A. \\
\text{ultimately show } & offset (Suc (n + m)) \times xs = Some (the (offset (Suc n) \times xs) + m) \\
& \text{by simp} \\
\text{qed} \\
\text{qed}
\end{align*}
\]

**lemma** offset-suc:
\[
\begin{align*}
\text{assumes } A: & offset k \times xs \neq None \\
\text{shows } & offset (Suc n) \times xs = Some (Suc (the (offset n \times xs))) \\
\text{proof } & \text{have } offset (Suc n) \times xs = offset (n + Suc 0) \times xs \text{ by simp} \\
\text{also have } & \ldots = Some (the (offset n \times xs) + Suc 0) \text{ using } A \text{ by (rule offset-addition)} \\
\text{also have } & \ldots = Some (Suc (the (offset n \times xs))) \text{ by simp} \\
\text{finally show } & ?thesis . \\
\text{qed}
\end{align*}
\]

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

**proof** (induction \( xs \) arbitrary; \( ys \), simp, \( \text{impl} \), (erule \( \text{conjE} \)+))
\[
\begin{align*}
\text{fix } & x' \times ys \\
\text{assume } A: & \forall ys. xs = ys @ x \# zs \land \\
\text{ipurge-tr-rev-aux } I \times D \times (\text{sources-aux } I \times D \times U \times (x \# zs)) \text{ ys } = [] \land \\
(\exists v \in \text{sources-aux } I \times D \times U \times zs. (D x, v) \in I) \rightarrow \\
y = \text{take } (the \times \text{offset } 0 \times xs) \times zs \text{ and } \\
B: & x' \# zs = ys @ x \# zs \text{ and } \\
C: & \text{ipurge-tr-rev-aux } I \times D \times (\text{sources-aux } I \times D \times U \times (x \# zs)) \text{ ys } = [] \land \\
D: & \exists v \in \text{sources-aux } I \times D \times U \times zs. (D x, v) \in I \\
\text{show } & ys = \text{take } (the \times \text{offset } 0 \times (x' \# zs)) \times (x' \# xs)
\end{align*}
\]
proof (cases ys)
case Nil
  moreover from this have \( x' = x \) using B by simp
  ultimately show thesis by simp
next
case (Cons y zs)
hence E: \( xs = ys' @ x \# z s \) using B by simp
moreover have 
  \begin{align*}
  F: \text{ipurge-tr-rev-aux } & I D (\text{sources-aux } I D U (x \# z s)) \ (y \# ys') = [] \\
  \text{using } & \text{Cons and } C \text{ by simp}
  \end{align*}
  hence 
  \begin{align*}
  G: \neg (\exists v \in \text{sources-aux } I D (\text{sources-aux } I D U (x \# z s))) \ ys', \ (D y, v) \in I \\
  \text{by } \text{(rule-tac notI, simp)}
  \end{align*}
  hence ipurge-tr-rev-aux I D (sources-aux I D U (x \# z s)) ys' = [] by simp
ultimately have \( xs = ys' @ x \# z s \) ∧ 
ipurge-tr-rev-aux I D (sources-aux I D U (x \# z s)) ys' = [] ∧ 
(\exists v \in \text{sources-aux } I D U z s. (D x, v) \in I)
  using D by blast
with A have H: \( ys' = \text{take } (\text{the } (\text{offset 0 } x xs)) \) zs ..
have I: \( x' = y \) using Cons and B by simp
hence 
  \begin{align*}
  J: \neg (\exists v \in \text{sources-aux } I D (\text{sources-aux } I D U z s)) \ (ys' @ [x]). \ (D x', v) \in I \\
  \text{by simp add: sources-aux-append}
  \end{align*}
have \( x' \neq x \)
proof 
assume \( x' = x \)
hence \( \exists v \in \text{sources-aux } I D U z s. (D x', v) \in I \) using D by simp
then obtain v where K: \( v \in \text{sources-aux } I D U z s \) and L: \( (D x', v) \in I ..
have sources-aux I D U z s ⊆ 
  \begin{align*}
  \text{sources-aux } I D (\text{sources-aux } I D U z s) (ys' @ [x])
  \end{align*}
  \text{by (rule sources-aux-subset)}
hence \( v \in \text{sources-aux } I D (\text{sources-aux } I D U z s) (ys' @ [x]) \) using K ..
with L have
\( \exists v \in \text{sources-aux } I D (\text{sources-aux } I D U z s) (ys' @ [x]) \) (ys' @ [x]). (D x', v) \in I ..
thus False using J by contradiction
qed
hence offset 0 x (x' # z s) = offset (Suc 0) x xs by simp
also have \( \) = Some (Suc (the (offset 0 x xs)))
proof –
have \( \exists y s \), \( xs = ys @ x \# z s \) 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' # z s))) (x' # z s) = 
x' # take (the (offset 0 x xs)) zs
by simp
thus thesis using Cons and H and I by simp
qed
qed

lemma ipurge-tr-rev-aux-append-nil-2 [rule-format]:
ipurge-tr-rev-aux I D U (xs @ ys) = ipurge-tr-rev-aux I D V xs \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 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 \land
  ipurge-tr-rev-aux I D (sources-aux I D U (x # ws)) vs = [] \land
  (\exists v \in 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 \land
  ipurge-tr-rev-aux I D (sources-aux I D U (x # ws)) vs = [] \land
  (\exists v \in sources-aux I D U ws. (D x, v) \in I)
  by blast
  moreover from this 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
  ultimately have \exists v \in 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 sources-aux I D U (xs @ ys). (D x, v) \in I)
  proof
  assume \exists v \in 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 \land
  ipurge-tr-rev-aux I D (sources-aux I D V (x # ws)) vs = [] \land
  (\exists v \in 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 \land
  ipurge-tr-rev-aux I D (sources-aux I D V (x # ws)) vs = [] \land
  (\exists v \in sources-aux I D V ws. (D x, v) \in I)
  by blast

  22
moreover from this have \( vs = \) take (the (offset 0 x (x \# xs))) (x \# xs) by (rule ipurge-tr-rev-aux-first-offset)
hence \( vs = \) \([],\) by simp
ultimately 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 \( \text{ipurge-tr-rev-aux } I D U (xs @ ys) = \)
\( \text{ipurge-tr-rev-aux } I D V xs \)
using \( B \) by simp
with \( A \) show \( \text{thesis }.. \)

qed

lemma ipurge-tr-rev-aux-append-nil:
\( \text{(ipurge-tr-rev-aux } I D U (xs @ ys) = \text{ipurge-tr-rev-aux } I D U xs) = \)
\( \text{ipurge-tr-rev-aux } I D U ys = [] \)
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 \( \text{ipurge-tr} \) and \( \text{ipurge-tr-rev} \), as well as those output by \( \text{ipurge-tr-aux} \) and \( \text{ipurge-tr-rev-aux} \), satisfy predicate \( \text{Interleaves} \) (cf. [7]), in correspondence with suitable input predicates expressed in terms of functions \( \text{sinks} \) and \( \text{sinks-aux} \), respectively. Then, some lemmas on the aforesaid functions are demonstrated without induction, using previous lemmas along with the properties of predicate \( \text{Interleaves} \).

lemma Interleaves-ipurge-tr:
\( xs \equiv \{ \text{ipurge-tr-rev } I D u xs, \text{rev } (\text{ipurge-tr } (I^{-1}) D u (\text{rev } xs)), \)
\( \lambda y ys. D y \in \text{sinks } (I^{-1}) D u (\text{rev } (y \# y)) \} \)
proof (induction \( xs \), simp, simp only: rev.simps)
fix \( x xs \)
assume \( A; xs \equiv \{ \text{ipurge-tr-rev } I D u xs, \text{rev } (\text{ipurge-tr } (I^{-1}) D u (\text{rev } xs)), \)
\( \lambda y ys. D y \in \text{sinks } (I^{-1}) D u (\text{rev } (y \# y)) \} \)
\( (\text{id } \equiv \{ ?ys, ?zs, ?P \}) \)
show \( x \# xs \equiv \{ \text{ipurge-tr-rev } I D u (x \# xs), \text{rev } (\text{ipurge-tr } (I^{-1}) D u (\text{rev } xs @ [x])), ?P \} \)
proof (cases \( ?P x xs \), simp-all add: sources-sinks del: sinks.simps)
case True
thus \( x \# xs \equiv \{ x \# ?ys, ?zs, ?P \} \) using \( A \) by (cases \( ?zs, \) simp-all)
next
case False
thus \( x \# xs \equiv \{ ?ys, x \# ?zs, ?P \} \) using \( A \) by (cases \( ?ys, \) simp-all)
qed

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

\textbf{proof} \ (\text{induction} \ xs, \ \text{simp, simp only: rev.simps})

\textbf{fix} \ x \ xs

\textbf{assume} \ A: \ xs \equiv \ { \text{ipurge-tr-rev-aux} \ I \ D \ U \ xs, \\
\text{rev} \ (\text{ipurge-tr-aux} \ (I^{-1}) \ D \ U \ (\text{rev} \ xs)), \\
\lambda y \ ys. \exists v \in \text{sinks-aux} \ ((I^{-1})^{-1}) \ D \ U \ (\text{rev} \ ys). \ (D \ y, v) \in I} \\
(is : \equiv \ { \{ys, zs, ?P\})}

\textbf{show} \ x \ # \ xs \equiv \n
{ \text{ipurge-tr-rev-aux} \ I \ D \ U \ (x \ # \ xs), \\
\text{rev} \ (\text{ipurge-tr-aux} \ (I^{-1}) \ D \ U \ (\text{rev} \ xs @ [x])), \ ?P} \\
\textbf{proof} \ (\text{cases} \ ?P \ x \ xs, \ \text{simp-all (no-asrn-simp) add: sources-sinks-aux})

\textbf{case} \ True

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

\textbf{next}

\textbf{case} \ False

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

\textbf{qed}

\textbf{qed}

\textbf{lemma} \ \text{ipurge-tr-aux-all}:

\[(\text{ipurge-tr-aux} \ I \ D \ U \ xs = xs) = (\forall u \in U. \ \neg (\exists v \in D \ D ^' \ set \ xs. \ (u, v) \in I))\]

\textbf{proof}

\textbf{have} \ A: \ \text{rev} \ xs \equiv \ { \text{ipurge-tr-rev-aux} \ (I^{-1}) \ D \ U \ (\text{rev} \ xs), \\
\text{rev} \ (\text{ipurge-tr-aux} \ ((I^{-1})^{-1}) \ D \ U \ (\text{rev} \ (\text{rev} \ xs))), \\
\lambda y \ ys. \exists v \in \text{sinks-aux} \ ((I^{-1})^{-1}) \ D \ U \ (\text{rev} \ ys). \ (D \ y, v) \in (I^{-1})} \\
(is : \equiv \ { \{\_ \, \_ \, ?P\})}

\textbf{by} \ (\text{rule Interleaves-ipurge-tr-aux})

\textbf{show} \ ?thesis

\textbf{proof}

\textbf{assume} \ \text{ipurge-tr-aux} \ I \ D \ U \ xs = xs

\textbf{hence} \ \text{rev} \ xs \equiv \ { \text{ipurge-tr-rev-aux} \ (I^{-1}) \ D \ U \ (\text{rev} \ xs), \ \text{rev} \ xs, \ ?P} \\
\textbf{using} \ A \ \textbf{by} \ \text{simp}

\textbf{hence} \ \text{rev} \ xs \equiv \ { \text{ipurge-tr-rev-aux} \ (I^{-1}) \ D \ U \ (\text{rev} \ xs), \ \text{rev} \ xs, \ ?P} \\
\textbf{by} \ (\text{rule Interleaves-interleaves})

\textbf{moreover have} \ \text{rev} \ xs \equiv \ { [], \ \text{rev} \ xs, \ ?P} \ \textbf{by} \ (\text{rule interleaves-nil-all})

\textbf{ultimately have} \ \text{ipurge-tr-rev-aux} \ (I^{-1}) \ D \ U \ (\text{rev} \ xs) = [] \\
\textbf{by} \ (\text{rule interleaves-equal-fst})

\textbf{thus} \ \forall u \in U. \ \neg (\exists v \in D \ D ^' \ set \ xs. \ (u, v) \in I) \\
\textbf{by} \ (\text{simp add: ipurge-tr-rev-aux-nil})

\textbf{next}

\textbf{assume} \ \forall u \in U. \ \neg (\exists v \in D \ D ^' \ set \ xs. \ (u, v) \in I)

\textbf{hence} \ \text{ipurge-tr-rev-aux} \ (I^{-1}) \ D \ U \ (\text{rev} \ xs) = [] \\
\textbf{by} \ (\text{simp add: ipurge-tr-rev-aux-nil})

\textbf{hence} \ \text{rev} \ xs \equiv \ { [], \ \text{rev} \ (\text{ipurge-tr-aux} \ I \ D \ U \ xs), \ ?P} \ \textbf{using} \ A \ \textbf{by} \ \text{simp}

\textbf{hence} \ \text{rev} \ xs \equiv \ { [], \ \text{rev} \ (\text{ipurge-tr-aux} \ I \ D \ U \ xs), \ ?P} \\
\textbf{by} \ (\text{rule Interleaves-interleaves})

\textbf{hence} \ \text{rev} \ xs \equiv \ { \text{rev} \ (\text{ipurge-tr-aux} \ I \ D \ U \ xs), \ [], \ \lambda w \ ws. \ \neg \ ?P \ w \ ws} \\
\textbf{by} \ (\text{subst (asm)} \ \text{interleaves-swap})

\textbf{moreover have} \ \text{rev} \ xs \equiv \ { \text{rev} \ xs, [], \ \lambda w \ ws. \ \neg ?P \ w \ ws}
ultimately have \( \text{rev (ipurge-tr-aux I D U xs)} = \text{rev xs} \)
by (rule interleaves-equal-fst)
thus \( \text{ipurge-tr-aux I D U xs} = \text{xs} \) by simp
qed

lemma ipurge-tr-rev-aux-single-dom:
\( \text{ipurge-tr-rev-aux I D \{u\} xs} = \text{ipurge-tr-rev I D u xs} (\text{is ?ys = ?ys'}) \)
proof
have \( \text{xs} \cong \{ ?ys, \text{rev (ipurge-tr-aux (I^-1) D \{u\} (rev xs))}, \}
\begin{align*}
\lambda y ys. \exists v \in \text{ sinks-aux (I^-1) D \{u\} (rev ys). (D y, v) \in I} 
\end{align*}
by (rule Interleaves-ipurge-tr-aux)

hence \( \text{xs} \cong \{ ?ys, \text{rev (ipurge-tr (I^-1) D u (rev xs))}, \}
\begin{align*}
\lambda y ys. (u, D y) \in I^-1 \lor (\exists v \in \text{ sinks (I^-1) D u (rev ys). (v, D y) \in I^-1}) \}
\end{align*}
by (simp add: ipurge-tr-aux-single-dom sinks-aux-single-dom)

hence \( \text{xs} \cong \{ ?ys, \text{rev (ipurge-tr (I^-1) D u (rev xs))}, \}
\begin{align*}
\lambda y ys. D y \in \text{ sinks (I^-1) D u (rev (y # ys))} 
\end{align*}
(is - \cong \{ - , ?zs, ?P} )
by (simp only: sinks-interference-eq, simp)

moreover have \( \text{xs} \cong \{ ?ys', ?zs, ?P} \) by (rule Interleaves-ipurge-tr)
ultimately show ?thesis by (rule Interleaves-equal-fst)
qed

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

lemma ipurge-tr-rev-all:
\( \forall v \in D \setminus \text{ set xs. (v, u) \in I} \implies \text{ipurge-tr-rev I D u xs = xs} \)
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 re-
placing \( \text{ipurge-tr-rev} \) with \( \text{ipurge-tr-rev-aux} \), so as to exploit the distribu-
tivity 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 do-
 mains, 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 \( \text{ipurge-tr-rev} \ D \ u \) matches the identity function, so that two traces are correlated by the image of \( \text{rel-ipurge} \) under \( u \) just in case they are equal.

definition rel-ipurge ::
'\text{a process} \Rightarrow ('d \times 'd) \text{ set} \Rightarrow ('a \Rightarrow 'd) \Rightarrow ('a, 'd) \text{ dom-rel-map}
where
\( \text{rel-ipurge} \ P \ I \ D \ u \equiv \{ (xs, ys). \ xs \in \text{traces} \ P \ \land \ ys \in \text{traces} \ P \ \land \ \text{ipurge-tr-rev} \ I \ D \ u \ xs = \text{ipurge-tr-rev} \ I \ D \ u \ ys \} \)

definition rel-ipurge-aux ::
'\text{a process} \Rightarrow ('d \times 'd) \text{ set} \Rightarrow ('a \Rightarrow 'd) \Rightarrow ('a, 'd) \text{ domset-rel-map}
where
\( \text{rel-ipurge-aux} \ P \ I \ D \ U \equiv \{ (xs, ys). \ xs \in \text{traces} \ P \ \land \ ys \in \text{traces} \ P \ \land \ \text{ipurge-tr-rev-aux} \ I \ D \ U \ xs = \text{ipurge-tr-rev-aux} \ I \ D \ U \ ys \} \)

lemma rel-ipurge-aux-single-dom:
\( \text{rel-ipurge-aux} \ P \ I \ D \ \{ u \} = \text{rel-ipurge} \ P \ I \ D \ u \)
by \( \text{simp add: rel-ipurge-def rel-ipurge-aux-def ipurge-tr-rev-aux-single-dom} \)

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

lemma fc-equals-wfc-rel-ipurge:
\( \text{future-consistent} \ P \ D \ (\text{rel-ipurge} \ P \ I \ D) = \)
\( \text{weakly-future-consistent} \ P \ I \ D \ (\text{rel-ipurge} \ P \ I \ D) \)
proof \( \text{rule iffI, erule fc-implies-wfc,}
\text{simp only: future-consistent-def weakly-future-consistent-def,}
\text{rule ballI, (rule allI)+, rule_implf} \)
  fix \( u \ x s \ y s \)
  assume
  A: \( \forall u \in \text{range} \ D \ \land \ (-1) \quad \text{"range} \ D. \ \forall x s \ y s. \ (xs, ys) \in \text{rel-ipurge} \ P \ I \ D \ u \ \rightarrow \)
  \( \text{next-dom-events} \ P \ D \ u \ x s = \text{next-dom-events} \ P \ D \ u \ y s \ \land \)
  \( \text{ref-dom-events} \ P \ D \ u \ x s = \text{ref-dom-events} \ P \ D \ u \ y s \quad \text{and} \)
  B: \( u \in \text{range} \ D \quad \text{and} \)

26
C: \((xs, ys) \in \text{rel-ipurge } P \ I \ D \ u\)

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

ref-dom-events \(P\ D\ u\ xs = \text{ref-dom-events } P\ D\ u\ ys\)

**proof** (cases \(u \in \text{range } D \cap (-I) \cup \text{range } D\))

**case** True

with \(A\) have \(\forall xs\ ys. (xs, ys) \in \text{rel-ipurge } P \ I \ D \ u \rightarrow\)

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

ref-dom-events \(P\ D\ u\ xs = \text{ref-dom-events } P\ D\ u\ ys\) ..

**hence** \((xs, ys) \in \text{rel-ipurge } P \ I \ D \ u \rightarrow\)

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

ref-dom-events \(P\ D\ u\ xs = \text{ref-dom-events } P\ D\ u\ ys\)

by blast

thus \(?\text{thesis using } C\). ..

next case False

**hence** \(D: u \notin (-I) \cup \text{range } D\) using \(B\) by simp

have ipurge-tr-rev \(I\ D\ u\ xs = \text{ipurge-tr-rev } I\ D\ u\ ys\)

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

moreover have \(\forall zs. \text{ipurge-tr-rev } I\ D\ u\ zs = zs\)

**proof** \((\text{rule allI, rule ipurge-tr-rev-all, rule ballI, erule imageE, rule ccontr})\)

fix \(v\ x\)

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

**hence** \((v, u) \in -I\) by simp

moreover assume \(v = D\ x\)

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

ultimately have \(u \in (-I) \cup \text{range } D\) ..

thus \(False\) using \(D\) by contradiction

qed

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

qed

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

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

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

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

**Lemma** ipurge-tr-rev-ipurge-tr-aux-1 [rule-format]:
\[
U \subseteq \text{unaffected-domains } I \ 1 \ (D \setminus \text{set } ys) \ zs \longrightarrow \\
\text{ipurge-tr-rev-aux } I \ 1 \ U \ (xs @ ys @ zs) = \\
\text{ipurge-tr-rev-aux } I \ 1 \ U \ (xs @ \text{ipurge-tr-aux } I \ 1 \ (D \setminus \text{set } ys) \ zs)
\]

**Proof** (induction zs arbitrary; \( U \) rule: rev-induct, rule-tac \([!]\) impI, simp)

1. \[\text{fix } U\]
2. \[\text{assume } A: U \subseteq \text{unaffected-domains } I \ 1 \ (D \setminus \text{set } ys) \ ][
3. \[\text{have } \forall u \in U. \forall v \in D \setminus \text{set } ys. (v, u) \notin I\]
4. **Proof**
   1. \[\text{fix } u\]
   2. \[\text{assume } u \in U\]
   3. \[\text{with } A \text{ have } u \in \text{unaffected-domains } I \ 1 \ (D \setminus \text{set } ys) \][
   4. \[\text{thus } \forall v \in D \setminus \text{set } ys. (v, u) \notin I \text{ by (simp add: unaffected-domains-def)}\]
5. **QED**
6. \[\text{have } D: \text{ipurge-tr-rev-aux } I \ 1 \ U \ (xs @ ys @ zs) = \\
\text{ipurge-tr-rev-aux } I \ 1 \ U \ (xs @ \text{ipurge-tr-aux } I \ 1 \ (D \setminus \text{set } ys) \ zs)
\]

**Proof**

1. \[\text{have } U \subseteq \text{unaffected-domains } I \ 1 \ (D \setminus \text{set } ys) \ zs \longrightarrow \\
\text{ipurge-tr-rev-aux } I \ 1 \ U \ (xs @ ys @ zs) = \\
\text{ipurge-tr-rev-aux } I \ 1 \ U \ (xs @ \text{ipurge-tr-aux } I \ 1 \ (D \setminus \text{set } ys) \ zs)\]

28
using $A$.
thus $\textit{thesis using } C$..

**QED**

have $E: \neg (\exists v \in \text{sinks-aux } I \ D (D \ ' \ set \ ys) \ zs. \ (v, \ D \ z) \in I) \right\rightarrow$

ipurge-tr-rev-aux $I \ D \ ?U' (xs \ @ \ ys \ @ \ zs) =$

ipurge-tr-rev-aux $I \ D \ ?U' (xs \ @ \ ipurge-tr-aux \ I \ D (D \ ' \ set \ ys) \ zs) (is \ ?P \right\rightarrow \ ?Q)$

**proof**

assume $?P$

have $?U' \subseteq \text{unaffected-domains } I \ D (D \ ' \ set \ ys) \ zs \right\rightarrow$

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 $?U' \subseteq \text{unaffected-domains } I \ D (D \ ' \ set \ ys) \ zs$

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 $\exists v \in \text{sinks-aux } I \ D (D \ ' \ set \ ys) \ zs. \ (v, \ D \ z) \in I$,

simp-all (no-asm-simp))

case $True$

have $\neg (\exists u \in U. \ (D \ z, \ u) \in I)$

**proof**

assume $\exists u \in U. \ (D \ z, \ u) \in I$

then obtain $u$ where $F$: $u \in U$ and $G$: $(D \ z, \ u) \in I$..

have $D \ z \in \text{sinks-aux } I \ D (D \ ' \ set \ ys) \ (zs \ @ \ [z])$ using $True$ by simp

with $G$ have $\exists v \in \text{sinks-aux } I \ D (D \ ' \ set \ ys) \ (zs \ @ \ [z]), \ (v, \ u) \in I$..

moreover have $u \in \text{unaffected-domains } I \ D (D \ ' \ set \ ys) \ (zs \ @ \ [z])$

using $B$ and $F$..

hence $\neg (\exists v \in \text{sinks-aux } I \ D (D \ ' \ set \ ys) \ (zs \ @ \ [z]), \ (v, \ u) \in I)$

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 $\ldots = \text{ipurge-tr-rev-aux } I \ D \ U$

$(xs \ @ \ ipurge-tr-aux \ I \ D (D \ ' \ set \ ys) \ zs)$

using $D$.

finally show ipurge-tr-rev-aux $I \ D \ U (xs \ @ \ ys \ @ \ zs \ @ \ [z]) =$

ipurge-tr-rev-aux $I \ D \ U (xs \ @ \ ipurge-tr-aux \ I \ D (D \ ' \ set \ ys) \ zs)$

by simp

next

case $False$

note $F = \textit{this}$

show ipurge-tr-rev-aux $I \ D \ U (xs \ @ \ ys \ @ \ zs \ @ \ [z]) =$

ipurge-tr-rev-aux $I \ D \ U (xs \ @ \ ipurge-tr-aux \ I \ D (D \ ' \ set \ ys) \ zs \ @ \ [z])$

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

29
case True
  hence ipurge-tr-rev-aux I D U ((xs @ ys @ zs) @ [z]) =
      ipurge-tr-rev-aux I D ?U' (xs @ ys @ zs) @ [z]
  by (subst ipurge-tr-rev-aux-append, simp)
also have ... =
      ipurge-tr-rev-aux I D ?U' (xs @ ipurge-tr-aux I D (D ' set ys) zs) @ [z]
using E and F by simp
also have ... =
      ipurge-tr-rev-aux I D U ((xs @ ipurge-tr-aux I D (D ' set ys) zs) @ [z])
using True by (subst ipurge-tr-rev-aux-append, simp)
finally show ?thesis by simp
next
  case False
  hence ipurge-tr-rev-aux I D U ((xs @ ys @ zs) @ [z]) =
      ipurge-tr-rev-aux I D U (xs @ ys @ zs)
  by (subst ipurge-tr-rev-aux-append, simp)
also have ... =
      ipurge-tr-rev-aux I D U (xs @ ipurge-tr-aux I D (D ' set ys) zs)
using D .
also have ... =
      ipurge-tr-rev-aux I D U ((xs @ ipurge-tr-aux I D (D ' set ys) zs) @ [z])
using False by (subst ipurge-tr-rev-aux-append, simp)
finally show ?thesis by simp
qed

lemma ipurge-tr-rev-ipurge-tr-aux-2 [rule-format]:
  U ⊆ unaffected-domains I D (D ' set ys) zs →
      ipurge-tr-rev-aux I D U (xs @ zs) =
      ipurge-tr-rev-aux I D U (xs @ ys @ ipurge-tr-aux I D (D ' set ys) zs)
proof (induction zs arbitrary: U rule: rev-induct, rule-tac [| impI, simp])
  fix U
  assume A: U ⊆ unaffected-domains I D (D ' set ys) []
  have ∀ u ∈ U. ∀ v ∈ D ' set ys. (v, u) ∉ I
  proof
    fix u
    assume u ∈ U
    with A have u ∈ unaffected-domains I D (D ' set ys) [] .
    thus ∀ v ∈ D ' set ys. (v, u) ∉ I by (simp add: unaffected-domains-def)
  qed
  hence ipurge-tr-rev-aux I D U ys = [] by (simp add: ipurge-tr-rev-aux-nil)
  hence ipurge-tr-rev-aux I D U (xs @ ys) = ipurge-tr-rev-aux I D U xs
  by (simp add: ipurge-tr-rev-aux-append-nil)
  thus ipurge-tr-rev-aux I D U xs = ipurge-tr-rev-aux I D U (xs @ ys) .
next
  fix z zs U
  let ?U' = insert (D z) U
  assume
A: $\forall U. U \subseteq \text{unaffected-domains} I D (D \cdot \text{set} \ ys) \Rightarrow$
\quad \text{ipurge-tr-rev-aux} I D U (zs @ zs) = \text{ipurge-tr-rev-aux} I D U (zs @ ys @ \text{ipurge-tr-aux} I D (D \cdot \text{set} \ ys) \ zs)$ and
B: $U \subseteq \text{unaffected-domains} I D (D \cdot \text{set} \ ys) (zs @ [z])$
have C: $U \subseteq \text{unaffected-domains} I D (D \cdot \text{set} \ ys) \ zs$
proof
\quad \text{fix} \ u
\quad \text{assume} \ u \in U
\quad \text{with} \ B \ \text{have} \ u \in \text{unaffected-domains} I D (D \cdot \text{set} \ ys) (zs @ [z]) ..
\quad \text{thus} \ u \in \text{unaffected-domains} I D (D \cdot \text{set} \ ys) \ zs
\quad \text{by} \ (\text{simp add: unaffected-domains-def})
qed
have D: \text{ipurge-tr-rev-aux} I D U (zs @ zs) = \text{ipurge-tr-rev-aux} I D U (zs @ ys @ \text{ipurge-tr-aux} I D (D \cdot \text{set} \ ys) \ zs)
proof
\quad \text{have} \ U \subseteq \text{unaffected-domains} I D (D \cdot \text{set} \ ys) \ zs \Rightarrow
\quad \text{ipurge-tr-rev-aux} I D U (zs @ zs) = \text{ipurge-tr-rev-aux} I D U (zs @ ys @ \text{ipurge-tr-aux} I D (D \cdot \text{set} \ ys) \ zs)
\quad \text{using} \ A .
\quad \text{thus} \ \text {?thesis} \ \text{using} \ C ..
qed
have E: $\neg (\exists v \in \text{sinks-aux} I D (D \cdot \text{set} \ ys) \ zs. (v, D z \in I) \Rightarrow$
\quad \text{ipurge-tr-rev-aux} I D \ ?U' (xs @ zs) = \text{ipurge-tr-rev-aux} I D \ ?U' (zs @ ys @ \text{ipurge-tr-aux} I D (D \cdot \text{set} \ ys) \ zs)
\quad \text{(is} \ ?P \ \Rightarrow \ ?Q)\text{)}
proof
\quad \text{assume} \ ?P
\quad \text{have} \ ?U' \subseteq \text{unaffected-domains} I D (D \cdot \text{set} \ ys) \ zs \Rightarrow
\quad \text{ipurge-tr-rev-aux} I D ?U' (xs @ zs) = \text{ipurge-tr-rev-aux} I D ?U' (xs @ ys @ \text{ipurge-tr-aux} I D (D \cdot \text{set} \ ys) \ zs)
\quad \text{using} \ A .
\quad \text{moreover have} \ ?U' \subseteq \text{unaffected-domains} I D (D \cdot \text{set} \ ys) \ zs
\quad \text{by} \ (\text{simp add: C, simp add: unaffected-domains-def} \ (?P, \text{simplified}))
\quad \text{ultimately show} \ ?Q ..
qed
show \text{ipurge-tr-rev-aux} I D U (xs @ zs @ [z]) = \text{ipurge-tr-rev-aux} I D U (zs @ ys @ \text{ipurge-tr-aux} I D (D \cdot \text{set} \ ys) (zs @ [z]))
proof \text{(cases} \exists v \in \text{sinks-aux} I D (D \cdot \text{set} \ ys) \ zs. (v, D z \in I, simp-all (no-asym-simp))
\quad \text{case} True
\quad \text{have} \ \neg (\exists u \in U. (D z, u) \in I)
proof
\quad \text{assume} \ \exists u \in U. (D z, u) \in I
\quad \text{then obtain} \ u \ \text{where} \ F: u \in U \ \text{and} \ G: (D z, u) \in I ..
\quad \text{have} \ D z \in \text{sinks-aux} I D (D \cdot \text{set} \ ys) (zs @ [z]) \ \text{using} \ True \ \text{by} \ \text{simp}
\quad \text{with} \ G \ \text{have} \ \exists v \in \text{sinks-aux} I D (D \cdot \text{set} \ ys) (zs @ [z]). (v, u) \in I ..
\quad \text{moreover have} \ u \in \text{unaffected-domains} I D (D \cdot \text{set} \ ys) (zs @ [z])
\quad \text{using} \ B \ \text{and} \ F ..
\quad \text{hence} \ \neg (\exists v \in \text{sinks-aux} I D (D \cdot \text{set} \ ys) (zs @ [z]). (v, u) \in I)
by (simp add: unaffected-domains-def)  
ultimately show False by contradiction  
qed 

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

next  

next  

next  

next  

qed 

qed  

next  

next  

next  

next  

next  

next 

next 

qed  

qed  

qed  

qed 

lemma ipurge-tr-rev-ipurge-tr-1:
assumes $A$: $u \in \text{unaffected-domains I D \{D y\}} \ zs$

shows $\text{ipurge-tr-rev I D u (xs @ y # zs) =}$
$\text{ipurge-tr-rev I D u (xs @ ipurge-tr I D (D y) zs)}$

proof –

have $\text{ipurge-tr-rev I D u (xs @ y # zs) =}$
$\text{ipurge-tr-rev-aux I D \{u\} (xs @ [y] @ zs)}$
by (simp add: $\text{ipurge-tr-rev-aux-single-dom}$)

also have $\ldots = \text{ipurge-tr-rev-aux I D \{u\}}$
$(xs @ \text{ipurge-tr-aux I D (D \set [y]) zs})$
by (rule $\text{ipurge-tr-rev-ipurge-tr-aux-1}$, simp add: $A$)

also have $\ldots = \text{ipurge-tr-rev I D u (xs @ ipurge-tr I D (D y) zs)}$

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

finally show $?\text{thesis}$.

qed

lemma $\text{ipurge-tr-rev-ipurge-tr-2}$:

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

shows $\text{ipurge-tr-rev I D u (zs @ zs) =}$
$\text{ipurge-tr-rev I D u (xs @ y # ipurge-tr I D (D y) zs)}$

proof –

have $\text{ipurge-tr-rev I D u (xs @ zs) = ipurge-tr-rev-aux I D \{u\} (zs @ zs)}$
by (simp add: $\text{ipurge-tr-rev-aux-single-dom}$)

also have $\ldots = \text{ipurge-tr-rev-aux I D \{u\}} (xs @ [y] @ \text{ipurge-tr-aux I D (D \set [y]) zs})$
by (rule $\text{ipurge-tr-rev-ipurge-tr-aux-2}$, simp add: $A$)

also have $\ldots = \text{ipurge-tr-rev I D u (xs @ y # ipurge-tr I D (D y) zs)}$
by (simp add: $\text{ipurge-tr-aux-single-dom}$ $\text{ipurge-tr-rev-aux-single-dom}$)

finally show $?\text{thesis}$.

qed

lemma $\text{iu-condition-imply-secure-aux-1}$:

assumes

$\text{RUC: ref-union-closed P}$ and

$\text{IU: weakly-future-consistent P I D (rel-ipurge P I D)}$ and

$A$: $(xs @ y # ys, Y) \in \text{failures P}$ and

$B$: $xs @ \text{ipurge-tr I D (D y) ys \in traces P}$ and

$C$: $\exists y'. y' \in \text{ipurge-ref I D (D y) ys Y}$

shows $(xs @ \text{ipurge-tr I D (D y) ys}, \text{ipurge-ref I D (D y) ys Y}) \in \text{failures P}$

proof –

let $?A = \text{singleton-set (ipurge-ref I D (D y) ys Y)}$

have $(\exists X. X \in ?A) \longrightarrow$
$(\forall X \in ?A. (xs @ \text{ipurge-tr I D (D y) ys, X}) \in \text{failures P}) \longrightarrow$
$(xs @ \text{ipurge-tr I D (D y) ys, } \bigcup X \in ?A. X) \in \text{failures P}$
using $\text{RUC}$ by (simp add: ref-union-closed-def)

moreover obtain $y'$ where $D$: $y' \in \text{ipurge-ref I D (D y) ys Y}$ using $C$ ..

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

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

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

33
proof (rule ballI, simp add: singleton-set-def, erule bexE, simp)

fix y'

have \( \forall u \in \text{range } D \cap (-1)^\circ \text{ range } D \).
\( \forall x s y s. \ (x, s, y) \in \text{rel-ipurge } P \cap D u \longrightarrow \)
\( \text{ref-dom-events } P \cap D u x s = \text{ref-dom-events } P \cap D u y s \)
using IU by (simp add: weakly-future-consistent-def)

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

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

hence D y' \in \text{range } D \cap (-1)^\circ \text{ range } D by (simp add: Image-iff, rule exI)

ultimately have \( \forall x s y s. \ (x, s, y) \in \text{rel-ipurge } P \cap D (D y') \longrightarrow \)
\( \text{ref-dom-events } P \cap D (D y') x s = \text{ref-dom-events } P \cap D (D y') y s .. \)

hence
\( F: \ (x s @ y = y s, \ x s @ \text{ipurge-tr } I \cap D (D y) y s) \in \text{rel-ipurge } P \cap D (D y') \longrightarrow \)
\( \text{ref-dom-events } P \cap D (D y') (x s @ y = y s) = \)
\( \text{ref-dom-events } P \cap D (D y') (x s @ \text{ipurge-tr } I \cap D (D y) y s) \)
by blast

have y' \in \{ x \in Y. \ D x \in \text{unaffected-domains } I \cap D (D y) y s \}
using E by (simp add: unaffected-domains-single-dom)

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

hence \text{ipurge-tr-rev } I \cap D (D y') (x s @ y = y s) =
\text{ipurge-tr-rev } I \cap D (D y') (x s @ \text{ipurge-tr } I \cap D (D y) y s)
by (rule ipurge-tr-rev-ipurge-tr-I)

moreover have x s @ y = y s \in \text{traces } P using A by (rule failures-traces)

ultimately have
\( (x s @ y = y s, x s @ \text{ipurge-tr } I \cap D (D y) y s) \in \text{rel-ipurge } P \cap D (D y') \)
using B by (simp add: rel-ipurge-def)

with F have \( \text{ref-dom-events } P \cap D (D y') (x s @ y = y s) = \)
\( \text{ref-dom-events } P \cap D (D y') (x s @ \text{ipurge-tr } I \cap D (D y) y s) .. \)

moreover have y' \in \text{ref-dom-events } P \cap D (D y') (x s @ y = y s)
proof (simp add: ref-dom-events-def refusals-def)

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

with A show (x s @ y = y s, \{ y' \}) \in \text{failures } P by (rule process-rule-3)

qed

ultimately have y' \in \text{ref-dom-events } P \cap D (D y')
\( (x s @ \text{ipurge-tr } I \cap D (D y) y s) \)
by simp

thus (x s @ \text{ipurge-tr } I \cap D (D y) y s, \{ y' \}) \in \text{failures } P
by (simp add: ref-dom-events-def refusals-def)

qed

ultimately have \( (x s @ \text{ipurge-tr } I \cap D (D y) x s) \in \text{traces } P .. \)

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

qed

lemma iu-condition-imply-secure-aux-2:

assumes
\( \text{RUC: ref-union-closed } P \text{ and} \)
\( \text{IU: weakly-future-consistent } P \cap D (\text{rel-ipurge } P \cap D) \text{ and} \)
\( A: (x s @ z s, Z) \in \text{failures } P \text{ and} \)
\( B: x s @ y = \text{ipurge-tr } I \cap D (D y) x s \in \text{traces } P \text{ and} \)
C: $\exists z'. z' \in \text{ipurge-ref I D} \ (D \ y) \ zs \ Z$

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

proof –

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

have $(\exists X, X \in ?A) \rightarrow$

$(\forall X \in ?A. (xs @ y \# \text{ipurge-tr I D} \ (D \ y) \ zs, X) \in \text{failures P}) \rightarrow$

$(xs @ y \# \text{ipurge-tr I D} \ (D \ y) \ zs, \bigcup X \in ?A. X) \in \text{failures P}$

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 $\exists X. X \in ?A$ by (simp add: singleton-set-some, rule exI)

ultimately have

$(\forall X \in ?A. (xs @ y \# \text{ipurge-tr I D} \ (D \ y) \ zs, X) \in \text{failures P}) \rightarrow$

$(xs @ y \# \text{ipurge-tr I D} \ (D \ y) \ zs, \bigcup X \in ?A. X) \in \text{failures P}$ ..

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

proof (rule ballI, simp add: singleton-set-def, erule bexE, simp)

fix $z'$. 

have $\forall u \in \text{range D} \cap (-I)" \text{ range D}$. 

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

$\text{ref-dom-events P D} u \ xs = \text{ref-dom-events P D} u \ ys$

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

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

$\text{ref-dom-events P D} (D z') \ xs = \text{ref-dom-events P D} (D z') \ ys$ ..

hence

$F: (xs @ zs, zs @ y \# \text{ipurge-tr I D} \ (D \ y) \ zs) \in \text{rel-ipurge P I D} (D z') \rightarrow$

$\text{ref-dom-events P D} (D z') (zs @ zs) = \text{ref-dom-events P D} (D z') (xs @ y \# \text{ipurge-tr I D} \ (D \ y) \ zs)$

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 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 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
\(RUC\): ref-union-closed \(P\) and
IU\(\): weakly-future-consistent \(P I D\) (rel-ipurge \(P I D\))
shows \((xs @ y \# ys, Y) \in \text{failures P} \longrightarrow\)
\((xs @ \text{ipurge-tr I D (D y) ys, ipurge-ref I D (D y) ys Y)} \in \text{failures P}\)
proof (induction ys arbitrary: Y rule: rev-induct, rule-tac \(!!\) \text{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 \(\text{RUC and IU and A show ?thesis}\)
  using True by (rule iu-condition-imply-secure-aux-1)
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
\(A\): \((ys @ y \# ys, Y') \in \text{failures P} \longrightarrow\)
\((xs @ \text{ipurge-tr I D (D y)} ys, \text{ipurge-ref I D (D y)} ys Y') \in \text{failures P}\)
\(B\): \((xs @ y \# ys @ [y'], Y) \in \text{failures P}\)
have \((xs @ y \# ys, \{\}) \in \text{failures P} \longrightarrow\)
\((xs @ \text{ipurge-tr I D (D y)} ys, \text{ipurge-ref I D (D y)} ys \{\}) \in \text{failures P}\)
\((\text{is - \(\rightarrow\)} (-, \?Y')) \in -\)
using \(A\).
moreover have \(((xs @ y \# ys) @ [y'], Y) \in \text{failures P using B by simp}\)
hence \((xs @ y \# ys, \{\}) \in \text{failures P by (rule process-rule-2)}\)
ultimately have \((xs @ \text{ipurge-tr I D (D y)} ys, \?Y') \in \text{failures P ..}\)
moreover have \(\{\} \subseteq \?Y'..\)
ultimately have \((xs @ \text{ipurge-tr I D (D y)} ys, \{\}) \in \text{failures P}\)
by (rule process-rule-3)
have \(E\): \((xs @ \text{ipurge-tr I D (D y)} (ys @ [y']), Y) \in \text{traces P}\)
proof (cases D y' \in sinks I D (D y) (ys @ [y']))
case True
  hence \( (xs \@ ipurge-tr ID (D y) (ys \@ [y']), \{\}) ) \in \text{failures P using } D \text{ by simp} \)
  thus \( \text{thesis by (rule failures-traces)} \)
next
case False
  have \( \forall u \in \text{range } D \cap (-I) \quad \text{"range } D. \quad \forall xs ys. (xs, ys) \in \text{rel-ipurge } P I D u \longrightarrow \quad \text{next-dom-events } P D u xs = \text{next-dom-events } P D u ys \)
    using \( IU \text{ by (simp add: weakly-future-consistent-def)} \)
  moreover have \( (D y, D y') \notin I \)
    using \( \text{False by (simp add: sinks-interference-eq [symmetric] del: sinks.simps)} \)
  hence \( D y' \in \text{range } D \cap (-I) \quad \text{"range } D \)
    by \( (\text{simp add: Image-iff, rule exI)} \)
  ultimately have \( \forall xs ys. (xs, ys) \in \text{rel-ipurge } P I D (D y') \longrightarrow \quad \text{next-dom-events } P D (D y') xs = \text{next-dom-events } P D (D y') ys \)
  hence \( F; (xs \@ y \# ys, xs \@ ipurge-tr ID (D y) ys) \in \text{rel-ipurge } P I D (D y') \longrightarrow \quad \text{next-dom-events } P D (D y') (xs \@ y \# ys) = \quad \text{next-dom-events } P D (D y') (xs \@ ipurge-tr ID (D y) ys) \)
    by \( \text{blast} \)
  have \( \forall v \in \text{insert } (D y) (\text{sinks } ID (D y) ys). (v, D y') \notin I \)
    using \( \text{False by (simp add: sinks-interference-eq [symmetric] del: sinks.simps)} \)
  hence \( D y' \in \text{unaffected-domains } ID \{D y\} ys \)
    by \( (\text{simp add: unaffected-domains-def)} \)
  hence \( \text{ipurge-tr-rev ID (D y') (xs \@ y \# ys)} = \quad \text{ipurge-tr-rev ID (D y') (xs \@ ipurge-tr ID (D y) ys)} \)
    by \( (\text{rule ipurge-tr-rev-ipurge-tr-1)} \)
  moreover have \( (xs \@ y \# ys) \in \text{traces P using } C \text{ by (rule failures-traces)} \)
  moreover have \( (xs \@ ipurge-tr ID (D y) ys) \in \text{traces P} \)
    using \( D \text{ by (rule failures-traces)} \)
  ultimately have \( (xs \@ y \# ys, xs \@ ipurge-tr ID (D y) ys) \in \text{rel-ipurge } P I D (D y') \)
    by \( (\text{simp add: rel-ipurge-def)} \)
  with \( F \) have \( \text{next-dom-events } P D (D y') (xs \@ y \# ys) = \quad \text{next-dom-events } P D (D y') (xs \@ ipurge-tr ID (D y) ys) \)
    moreover have \( (xs \@ y \# ys) \in \text{next-dom-events } P D (D y') (xs \@ y \# ys) \)
    proof \( (\text{simp add: next-dom-events-def next-events-def)} \)
    qed \( (\text{rule failures-traces [OF B]} \)
  ultimately have \( (xs \@ ipurge-tr ID (D y) ys) \)
    by \( \text{simp} \)
  hence \( (xs \@ ipurge-tr ID (D y) ys @ [y']) \in \text{traces P} \)
    by \( (\text{simp add: next-dom-events-def next-events-def)} \)
  thus \( \text{thesis using } \text{False by simp} \)
  qed
show \( (xs \@ ipurge-tr ID (D y) (ys @ [y']), \text{ipurge-ref ID (D y) (ys @ [y']) } Y) \)
    \( \in \text{failures P} \)
proof \( (\text{cases } \exists x. x \in \text{ipurge-ref ID (D y) (ys @ [y']) } Y) \)

37
case True
with RUC and IU and B and E show ?thesis by (rule iu-condition-imply-secure-aux-1)
next
case False
moreover have \((xs @ @ ipurge-tr I D (D y) (ys @ @ [y']), \{\}) \in failures P\)
using E by (rule traces-failures)
ultimately show ?thesis by simp
qed

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

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

next

case False
have \(\forall u \in \text{range} D \cap (-I) \quad \text{"range} D.\)
\(\forall xs ys. (xs, ys) \in \text{rel-ipurge} P I D u \longrightarrow\)
\(\text{next-dom-events} P D u xs = \text{next-dom-events} P D u ys\)
using \(IU\) by \(\text{(simp add: weakly-future-consistent-def)}\)
moreover have \((D y, D z) \notin I\)
using \(False\) by \(\text{(simp add: sinks-interference-eq [symmetric] del: sinks.simps)}\)
hence \(D z \in \text{range} D \cap (-I) \quad \text{"range} D\) by \(\text{(simp add: Image-iff, rule exI)}\)
ultimately have \(\forall xs ys. (xs, ys) \in \text{rel-ipurge} P I D (D z) \longrightarrow\)
\(\text{next-dom-events} P D (D z) xs = \text{next-dom-events} P D (D z) ys\) \quad ..

hence
\(F: (xs @ zs, xs @ y # \text{ipurge-tr} I D (D y) zs) \in \text{rel-ipurge} P I D (D z) \longrightarrow\)
\(\text{next-dom-events} P D (D z) (xs @ zs) =\)
\(\text{next-dom-events} P D (D z) (xs @ y # \text{ipurge-tr} I D (D y) zs)\)
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 \(\text{(simp add: sinks-interference-eq [symmetric] del: sinks.simps)}\)
hence \(\forall v \in \text{sinks-aux} I D \{D y\} zs. (v, D z) \notin I\)
by \(\text{(simp add: sinks-aux-single-dom)}\)
hence \(D z \in \text{unaffected-domains} I D \{D y\} zs\)
by \(\text{(simp add: unaffected-domains-def)}\)
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 \(\text{(rule ipurge-tr-rev-ipurge-tr-2)}\)
moreover have \(xs @ zs \in \text{traces} P\) using \(C\) by \(\text{(rule failures-traces)}\)
moreover have \(xs @ y # \text{ipurge-tr} I D (D y) zs \in \text{traces} P\)
using \(D\) by \(\text{(rule failures-traces)}\)
ultimately have
\((xs @ zs, xs @ y # \text{ipurge-tr} I D (D y) zs) \in \text{rel-ipurge} P I D (D z)\)
by \(\text{(simp add: rel-ipurge-def)}\)
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 \(\text{(simp add: next-dom-events-def next-events-def)}\)
qed \(\text{(rule failures-traces \{OF B\})}\)
ultimately have \(z \in \text{next-dom-events} P D (D z)\)
\((xs @ y # \text{ipurge-tr} I D (D y) zs)\)
by \(\text{simp}\)
hence \(xs @ y # \text{ipurge-tr} I D (D y) zs @ [z] \in \text{traces} P\)
by \(\text{(simp add: next-dom-events-def next-events-def)}\)
thus \(?thesis\) using \(False\) by \(\text{simp}\)
qed

show \((xs @ y # \text{ipurge-tr} I D (D y) (zs @ [z]),\)
\(\text{ipurge-ref} I D (D y) (zs @ [z]) Z)\)
\[ \exists x. x \in \text{ipurge-ref I D (D y) (z @ [z]) Z} \]

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

- case True

  with RUC and IU and B and E show \(?thesis by (rule iu-condition-imply-secure-aux-2)\)

next

- case False

moreover have \((xs @ y \# \text{ipurge-tr I D (D y) (zs @ [z]), \{\}]) \in \text{failures P}\)

using E by (rule traces-failures)

ultimately show \(?thesis by simp\)

qed

qed

theorem iu-condition-imply-secure:
assumes
\( RUC: \text{ref-union-closed P and} \)
\( IU: \text{weakly-future-consistent P I D (rel-ipurge P I D)} \)
shows secure P I D
proof (simp add: secure-def futures-def, (rule allI)+, rule impI, erule conjE)
fix \(xs, y, ys, Y, zs, Z\)
assume
\( A: (xs @ y \# ys, Y) \in \text{failures P and} \)
\( B: (xs @ zs, Z) \in \text{failures P} \)
show \((xs @ \text{ipurge-tr I D (D y) ys, ipurge-ref I D (D y) ys Y}) \in \text{failures P} \land \)
\((xs @ y \# \text{ipurge-tr I D (D y) zs, ipurge-ref I D (D y) zs Z}) \in \text{failures P} \)
(is \( ?P \land ?Q \))
proof
show \(?P using RUC and IU and A by (rule iu-condition-imply-secure-1)\)
next
have \(((xs @ [y]) @ ys, Y) \in \text{failures P using A by simp}\)

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 \(?Q using B by (rule iu-condition-imply-secure-2)\)

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 \(S: \text{secure P I D}\)
shows \((xs @ ys @ zs, X) \in \text{failures P} \rightarrow \)
\( \text{ipurge-tr-rev-aux I D (D' (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 ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs)) (ys @ [y]) = []
moreover from this have A: (∃ y ∈ D ⊕ (X ∪ set zs). (D y, v) ∈ I)
by (cases: y ∈ D ⊕ (X ∪ set zs). (D y, v) ∈ I,
  simp-all add: ipurge-tr-rev-aux-append)
ultimately have B: ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs)) ys = []
by (simp add: ipurge-tr-rev-aux-append)
assume (zs @ ys @ y ≠ zs, X) ∈ failures P
hence (y ≠ zs, X) ∈ failures P (zs @ ys) by (simp add: futures-def)
ultimately have C: (zs @ ys @ zs, X) ∈ futures P (zs @ ys)
by simp
hence ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs)) ys = [] ⟹
  (zs @ zs, X) ∈ failures P
ultimately have D: (zs @ zs, X) ∈ failures P (zs @ ys)
by simp
hence ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs)) ys = [] ⟹
  (zs @ zs, X) ∈ failures P
using C ..
thus (zs @ zs, X) ∈ futures P using B ..
qed

lemma secure-implies-failure-consistency [rule-format]:
assumes S: secure P I D
shows (zs, ys) ∈ rel-ipurge-aux P I D (D ⊕ (X ∪ set zs)) ⟹
  (zs @ zs, X) ∈ failures P ⟹ (ys @ zs, X) ∈ failures P
proof (induction ys arbitrary: zs zs rule: rev-induct,
  simp-all add: rel-ipurge-aux-def, (rule-tac [] impI)+, (erule-tac []) conjE)+)
fix xs zs
assume (zs @ zs, X) ∈ failures P
hence ([] @ zs @ zs, X) ∈ failures P by simp
moreover assume ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs)) xs = []
ultimately have ([] @ zs, X) ∈ failures P
using S by (rule-tac secure-implies-failure-consistency-aux)
thus (zs, X) ∈ futures P by simp
next
fix y ys zs xs
assume A: ∃ xs’ zs’, xs’ ∈ traces P ∧ ys ∈ traces P ∧
  ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs’)) xs’ =
  ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs’)) ys ⟹
  (xs’ @ zs, X) ∈ failures P ⟹ (ys @ zs’, X) ∈ failures P and
B: (xs @ zs, X) ∈ failures P and
C: zs ∈ traces P and
D: ys @ [y] ∈ traces P and
E: ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs)) xs =
  ipurge-tr-rev-aux I D (D ⊕ (X ∪ set zs)) (ys @ [y])
show \((ys @ y \neq zs, X) \in \text{failures \(P\)}\)

proof (cases \(\exists v \in D^i (X \cup \text{set} \(zs\)) \cdot (D, v) \in I\))

  case True

    hence \(F\) \(\vdash \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(zs\))) \cdot xs = \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(y \neq zs\))) \cdot ys \ \& [y]\)\)

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

    hence

\(\exists vs \ \text{and} \ ws \ \text{where} \)

\(G\) \(\vdash vs = vs \ @ \ y \neq ws \ \& \ \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(zs\))) \cdot ws = []\)\)

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

then obtain \(us\) and \(ws\) where

\(G\) \(\vdash us = vs \ @ \ y \neq ws \ \& \ \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(zs\))) \cdot ws = []\)\)

by blast

hence \(\text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(zs\))) \cdot xs = \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(y \neq zs\))) \cdot vs \ @ [y]\)\)

by simp

hence \(\text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(zs\))) \cdot xs = \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(y \neq zs\))) \cdot vs \ @ [y]\)\)

by simp

moreover have \(\exists v \in D^i (X \cup \text{set} \(zs\)) \cdot (D, v) \in I\)

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

ultimately have \(\text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(zs\))) \cdot xs = \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(y \neq zs\))) \cdot vs \ @ [y]\)\)

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

hence \(\text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(y \neq zs\))) \cdot vs = \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(y \neq zs\))) \cdot ys\)\)

using \(F\) by simp

moreover have \(vs \ @ \ y \neq ws \in \text{traces} \(P\)\)

using \(C\) and \(G\) by simp

hence \(vs \in \text{traces} \(P\)\)

by (rule \(\text{process-rule-2-traces}\))

moreover have \(ys \in \text{traces} \(P\)\)

using \(D\) by (rule \(\text{process-rule-2-traces}\))

moreover have \(vs \in \text{traces} \(P\) \ \& \ ys \in \text{traces} \(P\) \ \& \)

\(\text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(y \neq zs\))) \cdot vs = \text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(y \neq zs\))) \cdot ys \rightarrow\)

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

using \(A\).

ultimately have \(H\) \(\vdash (vs @ y \neq zs, X) \in \text{failures} \(P\) \rightarrow (ys @ y \neq 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} \((D \cdot (X \cup \text{set} \(zs\))) \cdot ws = []\)\)

using \(G\)

ultimately have \((\text{ipurge-tr-rev-aux} \((D \cdot (X \cup \text{set} \(zs\))) \cdot ws = [])\)\)

using \(S\) by (rule \(\text{rule-tac secure-implies-failure-consistency-aux}\))

thus \(\text{thesis}\) using \(H\) by simp

next

  case False

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

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

  moreover have \(ys \in \text{traces} \(P\)\)

  using \(D\) by (rule \(\text{process-rule-2-traces}\))

  moreover have \(xs \in \text{traces} \(P\) \ \& \ ys \in \text{traces} \(P\) \ \&\)
ipurge-tr-rev-aux I D (D : (X ∪ set zs)) xs = 
ipurge-tr-rev-aux I D (D : (X ∪ set zs)) ys → (xs @ zs, X) ∈ failures P → (ys @ zs, X) ∈ failures P
using A.
ultimately have (ys @ zs, X) ∈ failures P using B and C by simp
hence (zs, X) ∈ futures P ys by (simp add: futures-def)
moreover have ∃ Y. ([y], Y) ∈ futures P ys
using D by (simp add: traces-def Domain-iff futures-def)
then obtain Y where ([y], Y) ∈ futures P ys ..
ultimately have (y # ipurge-tr I D (D y) zs, ipurge-ref I D (D y) zs X) ∈ futures P ys
using S by (simp add: secure-def)
moreover have ipurge-tr I D (D y) zs = zs
using False by (simp add: ipurge-tr-all)
moreover have ipurge-ref I D (D y) zs X = X
using False by (rule ipurge-ref-all)
ultimately show ?thesis by (simp add: futures-def)
qed
lemma secure-implies-trace-consistency:
secure P I D ⇒ (zs, ys) ∈ rel-ipurge-aux P I D (D : set zs) ⇒
xs @ zs ∈ traces P ⇒ ys @ zs ∈ traces P
proof (simp add: traces-def Domain-iff, rule-tac x = {} in exI, rule secure-implies-failure-consistency, simp-all)
qed (erule exE, erule process-rule-3, simp)
lemma secure-implies-next-event-consistency:
secure P I D ⇒ (zs, ys) ∈ rel-ipurge P I D (D x) ⇒
x ∈ next-events P zs ⇒ x ∈ next-events P ys
by (simp add: next-events-def, rule secure-implies-trace-consistency, simp-all add: rel-ipurge-aux-single-dom)
lemma secure-implies-refusal-consistency:
secure P I D ⇒ (zs, ys) ∈ rel-ipurge-aux P I D (D X) ⇒
X ∈ refusals P zs ⇒ X ∈ refusals P ys
by (simp add: refusals-def, subst append-Nil2 [symmetric], rule secure-implies-failure-consistency, simp-all)
lemma secure-implies-ref-event-consistency:
secure P I D ⇒ (zs, ys) ∈ rel-ipurge P I D (D x) ⇒
[x] ∈ refusals P xs ⇒ [x] ∈ refusals P ys
by (rule secure-implies-refusal-consistency, simp-all add: rel-ipurge-aux-single-dom)
theorem secure-implies-in-condition:
assumes S: secure P I D
shows future-consistent P D (rel-ipurge P I D)
proof (simp add: future-consistent-def next-dom-events-def ref-dom-events-def, (rule allI)+, rule impl, rule conjI, rule-tac [!] equality1, rule-tac [!] subset1,
simp-all, erule-tac (! conjE)

fix xs ys x
assume (xs, ys) ∈ rel-ipurge P I D (D x) and x ∈ next-events P xs
with S show x ∈ next-events P ys by (rule secure-implies-next-event-consistency)

next
fix xs ys x
have ∀ u ∈ range D. equiv (traces P) (rel-ipurge P I D u)
  using view-partition-rel-ipurge by (simp add: view-partition-def)
hence sym (rel-ipurge P I D (D x)) by (simp add: equiv-def)
moreover assume (xs, ys) ∈ rel-ipurge P I D (D x)
ultimately have (ys, xs) ∈ rel-ipurge P I D (D x) by (rule symE)
moreover assume x ∈ next-events P ys
ultimately show x ∈ next-events P xs
  using S by (rule-tac secure-implies-next-event-consistency)

next
fix xs ys x
assume (xs, ys) ∈ rel-ipurge P I D (D x) and {x} ∈ refusals P xs
with S show {x} ∈ refusals P ys by (rule secure-implies-ref-event-consistency)

next
fix xs ys x
have ∀ u ∈ range D. equiv (traces P) (rel-ipurge P I D u)
  using view-partition-rel-ipurge by (simp add: view-partition-def)
hence sym (rel-ipurge P I D (D x)) by (simp add: equiv-def)
moreover assume (xs, ys) ∈ rel-ipurge P I D (D x)
ultimately have (ys, xs) ∈ rel-ipurge P I D (D x) by (rule symE)
moreover assume {x} ∈ refusals P ys
ultimately show {x} ∈ refusals P xs
  using S by (rule-tac secure-implies-ref-event-consistency)

qed

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

end

2 The Ipurge Unwinding Theorem for deterministic and trace set processes

theory DeterministicProcesses
imports IpurgeUnwinding
begin

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

2.1 Deterministic processes

Here below are the definitions of predicates \(\text{d-future-consistent}\) and
\(\text{d-weakly-future-consistent}\), 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 con-
nection as that of their general-purpose counterparts, viz. \(\text{d-future-consistent}\)
implies \(\text{d-weakly-future-consistent}\), 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 \(\text{d-future-consistent} \) ::
\(\forall u \in \text{range D}. \forall xs, ys. (xs, ys) \in R u \rightarrow \)
\((xs \in \text{traces P}) = (ys \in \text{traces P}) \land \)
\(\text{next-dom-events P D u xs} = \text{next-dom-events P D u ys} \)

definition \(\text{d-weakly-future-consistent} \) ::
\(\forall u \in \text{range D} \cap (-I)'' \text{range D}. \forall xs, ys. (xs, ys) \in R u \rightarrow \)
\((xs \in \text{traces P}) = (ys \in \text{traces P}) \land \)
\(\text{next-dom-events P D u xs} = \text{next-dom-events P D u ys} \)

lemma \(\text{dfc-implies-dwfc} \):
\(\text{d-future-consistent P D R} \implies \text{d-weakly-future-consistent P I D R} \)
by \((\text{simp only: d-future-consistent-def d-weakly-future-consistent-def}, \text{blast})\)

lemma \(\text{dfc-equals-dwfc-rel-ipurge} \):
\(\text{d-future-consistent P D (rel-ipurge P I D)} = \)
\(\text{d-weakly-future-consistent P I D (rel-ipurge P I D)} \)
proof \((\text{rule iffI, erule dfc-implies-dwfc,}
\text{simp only: d-future-consistent-def d-weakly-future-consistent-def,}
\text{rule ballI, (rule allI)+, rule impI})\)
fix \(u \, xs \, ys\)
assume
A: \( \forall u \in \text{range } D \cap (\neg I) \Rightarrow \text{range } D. \forall xs ys. (xs, ys) \in \text{rel-ipurge } P I D u \rightarrow (xs \in \text{traces } P) = (ys \in \text{traces } P) \land \text{next-dom-events } P D u xs = \text{next-dom-events } P D u ys \) and

B: \( u \in \text{range } D \) and

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

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

\[ \text{proof } (\text{cases } u \in \text{range } D \cap (\neg I) \Rightarrow \text{range } D) \]

case True with A have \( \forall xs ys. (xs, ys) \in \text{rel-ipurge } P I D u \rightarrow (xs \in \text{traces } P) = (ys \in \text{traces } P) \land \text{next-dom-events } P D u xs = \text{next-dom-events } P D u ys \) ..

\[ \text{by blast} \]

thus \(?thesis using C .. \)

next case False hence D: \( u \notin (\neg I) \Rightarrow \text{range } D \) using B by simp

have \( \text{ipurge-tr-rev } I D u xs = \text{ipurge-tr-rev } I D u ys \)

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

moreover have \( \forall zs. \text{ipurge-tr-rev } I D u zs = zs \)

\[ \text{proof } (\text{rule allI, rule ipurge-tr-rev-all, rule ballI, erule imageE, rule ccontr}) \]

fix v x

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

hence \((v, u) \in I \) by simp

moreover assume \( v = D x \)

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

ultimately have \( u \in (\neg I) \Rightarrow \text{range } D .. \)

thus False using D by contradiction

qed

ultimately show \(?thesis by simp \)

qed

qed

lemma d-fc-equals-dfc:

assumes A: deterministic \( P \)

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

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

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

\[ \text{proof } (\text{rule ballI, (rule allI)+, rule implI, simp add: ref-dom-events-def set-eq-iff, rule allI}) \]
lemma d-wfc-equals-dwfc:

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

shows weakly-future-consistent $P$ I $D$ $R$ = d-weakly-future-consistent-def $P$ 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)

assume $B$: $\forall u \in \text{range } D \cap (-I) ^{\prime} \text{range } D$. $\forall xs ys. (xs, ys) \in R u \longrightarrow (xs \in \text{traces } P) = (ys \in \text{traces } P) \land$

next-dom-events $P$ $D$ $u$ $xs = next-dom-events P D u ys$

show $\forall u \in \text{range } D \cap (-I) ^{\prime} \text{range } D$. $\forall xs ys. (xs, ys) \in R u \longrightarrow$

ref-dom-events $P D u xs = ref-dom-events P D u ys$

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

fix $u$ $xs$ $ys$ $x$

assume $u \in \text{range } D \cap (-I) ^{\prime} \text{range } D$

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 $P D u xs = next-dom-events P D u ys$

hence $\forall xs, ys \in R u \longrightarrow (xs \in \text{traces } P) = (ys \in \text{traces } P) \land$

next-dom-events $P D u xs = next-dom-events P D u ys$

by blast

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

ultimately have $C$: $(xs \in \text{traces } P) = (ys \in \text{traces } P) \land$

next-dom-events $P D u xs = next-dom-events P D u ys$

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

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

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

have $A ^{\prime}$: $\forall xs \in \text{traces } P$. $\forall X$.

$X \in \text{refusals } P xs = (X \cap \text{next-events } P xs = \{\})$

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

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

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 $A ^{\prime}$ 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, curule-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 $D$: $xs \notin traces \ P$

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

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

moreover have $ys \notin traces \ P$ using $C$ and $D$ by simp

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

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

ultimately show $\{x\} \in refusals \ P \ xs = \{\}$ by simp

definitions

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

lemma $d$-implies-ruc:

assumes $A$: deterministic $P$

shows ref-union-closed $P$

proof (subst ref-union-closed-def, (rule allI)+, (rule impI)+, erule exE)

fix $xs \ A \ X$

have $\forall zs \in traces \ P. \ \forall X. X \in refusals \ P \ zs = (X \cap next-events \ P \ zs = \{})$

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

moreover assume $B$: $\forall X \in A. (xs, X) \in failures \ P$ and $X \in A$

definitions

hence $(xs, X) \in failures \ P$ ..

hence $zs \in traces \ P$ by (rule failures-traces)

ultimately have $C$: $\forall X. X \in refusals \ P \ xs = (X \cap next-events \ P \ xs = \{})$ ..

have $D$: $\forall X \in A. X \cap next-events \ P \ xs = \{}$

proof

fix $X$

assume $X \in A$

with $B$ have $(xs, X) \in failures \ P$ ..

hence $X \in refusals \ P \ xs$ by (simp add: refusals-def)

thus $X \cap next-events \ P \ zs = \{}$ using $C$ by simp

definitions

have $(\bigcup X \in A \ . \ X) \in refusals \ P \ zs = ((\bigcup X \in A \ . \ X) \cap next-events \ P \ zs = \{})$

using $C$ ..

hence $E$: $(xs, \bigcup X \in A \ . \ X) \in failures \ P =

((\bigcup X \in A \ . \ X) \cap next-events \ P \ zs = \{})$

by (simp add: refusals-def)

show $(xs, \bigcup X \in A \ . \ X) \in failures \ P$

proof (rule ssubst [OF $E$], rule equals0I, erule IntE, erule UN-E)

fix $x \ X$

assume $X \in A$

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

theorem \( d\text{-ipurge-unwinding} \):
assumes \( A: \text{deterministic } P \)
shows \( \text{secure } P \, I \, D = d\text{-weakly-future-consistent } P \, I \, D \) (rel-ipurge \( P \, I \, D \))
proof (insert \( d\text{-ufc-equals-dwfc} \) [of \( P \, I \, D \) rel-ipurge \( P \, I \, D \), erule subst])
qed (insert \( d\text{-implies-ruc} \) [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
\( \text{trace-set } T \equiv [\sim] \in T \land (\forall xs x. xs @ [x] \in T \rightarrow xs \in T) \)

lemma \( \text{ts-process-rep} \):
assumes \( A: \text{trace-set } T \)
shows \( \text{Rep-process} (\text{ts-process } T) = (\{(xs, X). xs \in T \land (\forall x \in X. xs @ [x] \notin T)\}, \{\}) \)
proof (subst \( \text{ts-process-def} \), rule \( \text{Abs-process-inverse} \), simp add: process-set-def, (subst conj-assoc [symmetric]), (rule conjI), simp-all add: process-prop-1-def process-prop-2-def process-prop-3-def process-prop-4-def process-prop-5-def process-prop-6-def)
show \[ \emptyset \in T \text{ using } A \text{ by (simp add: trace-set-def)} \]

next
show \( \forall xs. (\exists x. xs @ [x] \in T \land (\exists X. \forall x' \in X. xs @ [x, x'] \notin T)) \rightarrow xs \in T \)
proof (rule allI, rule impI, erule exE, erule conjE)
  fix xs x
  have \( \forall xs x. xs @ [x] \in T \rightarrow xs \in T \text{ using } A \text{ by (simp add: trace-set-def)} \)
  hence \( xs @ [x] \in T \rightarrow xs \in T \) by blast
  moreover assume \( xs @ [x] \in T \)
  ultimately show \( xs \in T \) ..
qed

next
show \( \forall xs X. xs \in T \land (\exists Y. (\forall x \in Y. xs @ [x] \notin T) \land X \subseteq Y) \rightarrow \)
\( (\forall x \in X. xs @ [x] \notin T) \)
proof ((rule allI)+, rule impI, (erule conjE, (erule exE)+, rule ballI)
  fix xs x X Y
  assume \( \forall x \in Y. xs @ [x] \notin T \)
  moreover assume \( X \subseteq Y \text{ and } x \in X \)
  hence \( x \in Y \) ..
  ultimately show \( xs @ [x] \notin T \) ..
qed

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

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

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

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

**lemma** ts-process-d:
*trace-set T \implies deterministic (ts-process T)*

**proof** (frule ts-process-traces, simp add: deterministic-def, rule ballI, 
drule ts-process-refusals, assumption, simp add: next-events-def, 
rule allI, rule iffI)

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

next
fix xs X
assume A: \( X \cap \{x. \, xs \, @ \, [x] \in T\} = \{\} \)
show \( \forall x \in X. \, xs \, @ \, [x] \notin T \)
proof (rule ballI, rule notI)
fix x
assume x \in X and xs \, @ \, [x] \in T
hence x \in X \cap \{x. \, xs \, @ \, [x] \in T\} by simp
moreover have x \notin X \cap \{x. \, xs \, @ \, [x] \in T\} using A by (rule equals0D)
ultimately show False by contradiction
qed

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

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

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

hence $\{x\} \cap \text{next-events } P \, xs \neq \{\}$ by $(\text{simp add: next-events-def})$

ultimately have $G$: $\{x\} \in \text{refusals } P \, xs = (\{x\} \cap \text{next-events } P \, xs = \{\})$

by simp

have $\forall \, zs \in \text{traces } P \land \forall \, X \, . \, \, X \in \text{refusals } P \, xs = (X \cap \text{next-events } P \, xs = \{\})$

using $A$ by $(\text{simp add: deterministic-def})$

moreover have $xs \in \text{traces } P \, \text{using } F$ by $(\text{rule process-rule-2-traces})$

ultimately have $\forall \, X \, . \, X \in \text{refusals } P \, xs = (X \cap \text{next-events } P \, xs = \{\})$ ..

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

thus $\text{False using } G$ by contradiction

qed

lemma trace-set-traces:

trace-set $\text{(traces } P)$


have $\text{Rep-process } P \in \text{process-set (is } ?P' \in -)$ by $(\text{rule Rep-process})$

hence $\exists X. (\{\}, X) \in \text{fst } ?P'$ by $(\text{simp add: process-set-def process-prop-1-def})$

thus $\exists X. (\{\}, X) \in \text{fst } ?P'$ ..

next

fix $xs \, x \, X$

have $\text{Rep-process } P \in \text{process-set (is } ?P' \in -)$ by $(\text{rule Rep-process})$

hence $\forall xs \, x \, X. (xs @ [x], X) \in \text{fst } ?P' \rightarrow (xs, \{\}) \in \text{fst } ?P'$

by $(\text{simp add: process-set-def process-prop-2-def})$

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

moreover assume $(xs @ [x], X) \in \text{fst } ?P'$

ultimately have $(xs, \{\}) \in \text{fst } ?P'$ ..

thus $\exists X. (xs, X) \in \text{fst } ?P'$ ..

qed

lemma d-implies-ts-process-traces:

deterministic $P \Longrightarrow \text{ts-process (traces } P) = P$


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

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

hence $C$: $\text{traces (ts-process (traces } P)) = \text{traces } P$ by $(\text{rule ts-process-traces})$

show failures $(\text{ts-process (traces } P)) = \text{failures } P$

proof $(\text{rule equality1, rule-tac [!] subsetI, simp-all only: split-paired-all})$

fix $xs \, x$

assume $D$: $(xs, X) \in \text{failures (ts-process (traces } P))$

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

hence $E$: $xs \in \text{traces } P$ using $C$ by simp

with $B$ have $\text{failures (ts-process (traces } P)) \, x = \{X. \forall x \in X. \, x @ [x] \notin \text{traces } P\}$

by $(\text{rule ts-process-refusals})$

53
moreover have \( X \in \text{refusals} \left( \text{ts-process} \left( \text{traces} \ P \right) \right) \) \( xs \)
using \( D \) by \( \text{simp add: refusals-def} \)
ultimately have \( \forall x \in X, \ xs @ [x] \notin \text{traces} \ P \) by \( \text{simp} \)
hence \( X \cap \text{next-events} \ P \ xs = \{\} \)
by \( \text{rule-tac equals0I}, \text{erule-tac IntE}, \text{simp add: next-events-def} \)
moreover have \( \forall X. \ (X \in \text{refusals} \ P \ xs) = (X \cap \text{next-events} \ P \ xs = \{\}) \)
using \( A \) and \( E \) ..
hence \( (X \in \text{refusals} \ P \ xs) = (X \cap \text{next-events} \ P \ xs = \{\}) \) ..
ultimately have \( X \in \text{refusals} \ P \ xs \) by \( \text{simp} \)
thus \( (xs, X) \in \text{failures} \ P \) by \( \text{simp add: refusals-def} \)

next
fix \( xs \) \( X \)
assume \( D \colon (xs, X) \in \text{failures} \ P \)
hence \( E \colon xs \in \text{traces} \ P \) by \( \text{rule failures-traces} \)
with \( A \) have \( \forall X. \ (X \in \text{refusals} \ P \ xs) = (X \cap \text{next-events} \ P \ xs = \{\}) \) ..
hence \( (X \in \text{refusals} \ P \ xs) = (X \cap \text{next-events} \ P \ xs = \{\}) \) ..
moreover have \( X \in \text{refusals} \ P \ xs \) using \( D \) by \( \text{simp add: refusals-def} \)
ultimately have \( F \colon X \cap \{x. \ xs @ [x] \in \text{traces} \ P\} = \{\} \)
by \( \text{simp add: next-events-def} \)
have \( \forall x \in X, \ xs @ [x] \notin \text{traces} \ P \)
proof \( \text{rule ballI}, \text{rule notI} \)
fix \( x \)
assume \( x \in X \) and \( xs @ [x] \in \text{traces} \ P \)
hence \( x \in X \cap \{x. \ xs @ [x] \in \text{traces} \ P\} \) by \( \text{simp} \)
moreover have \( x \notin X \cap \{x. \ xs @ [x] \in \text{traces} \ P\} \) using \( F \) by \( \text{rule equals0D} \)
ultimately show \( \text{False} \) by \( \text{contradiction} \)
qed
moreover have \( \text{refusals} \left( \text{ts-process} \left( \text{traces} \ P \right) \right) \) \( xs = \{X. \ \forall x \in X. \ xs @ [x] \notin \text{traces} \ P\} \)
using \( B \) and \( E \) by \( \text{rule ts-process-refusals} \)
ultimately have \( X \in \text{refusals} \left( \text{ts-process} \left( \text{traces} \ P \right) \right) \) \( xs \) by \( \text{simp} \)
thus \( (xs, X) \in \text{failures} \left( \text{ts-process} \left( \text{traces} \ P \right) \right) \) by \( \text{simp add: refusals-def} \)
qed

lemma \( \text{ts-process-traces-implies-d} \):
\( \text{ts-process} \left( \text{traces} \ P \right) = P \Rightarrow \text{deterministic} \ P \)
by \( \text{insert trace-set-traces [of} P\text{], drule ts-process-d, simp} \)

lemma \( \text{d-equals-ts-process-traces} \):
\( \text{deterministic} \ P = (\text{ts-process} \left( \text{traces} \ P \right) = P) \)
by \( \text{rule iffI}, \text{erule d-implies-ts-process-traces, rule ts-process-traces-implies-d} \)

Finally, a variant of the Ipurge Unwinding Theorem applying to trace set processes is derived from the variant for deterministic processes. Particularly, the assumption that the process be deterministic is replaced by the assumption that it be a trace set process, since the former property is entailed by the latter (cf. above).
theorem ts-ipurge-unwinding:
trace-set T \implies
secure (ts-process T) I D =
d-weakly-future-consistent (ts-process T) I D (rel-ipurge (ts-process T) I D)
by (rule d-ipurge-unwinding, rule ts-process-d)
end

References


