Noninterference Security in Communicating Sequential Processes

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

March 17, 2025

Abstract

An extension of classical noninterference security for deterministic state machines, as introduced by Goguen and Meseguer and elegantly formalized by Rushby, to nondeterministic systems should satisfy two fundamental requirements: it should be based on a mathematically precise theory of nondeterminism, and should be equivalent to (or at least not weaker than) the classical notion in the degenerate deterministic case.

This paper proposes a definition of noninterference security applying to Hoare's Communicating Sequential Processes (CSP) in the general case of a possibly intransitive noninterference policy, and proves the equivalence of this security property to classical noninterference security for processes representing deterministic state machines.

Furthermore, McCullough's generalized noninterference security is shown to be weaker than both the proposed notion of CSP noninterference security for a generic process, and classical noninterference security for processes representing deterministic state machines. This renders CSP noninterference security preferable as an extension of classical noninterference security to nondeterministic systems.

Contents

1 Noninterference in CSP

theory *CSPNoninterference*
imports *Main*
begin

An extension of classical noninterference security for deterministic state machines, as introduced by Goguen and Meseguer [1] and elegantly formalized by Rushby [8], to nondeterministic systems should satisfy two fundamental requirements: it should be based on a mathematically precise theory of nondeterminism, and should be equivalent to (or at least not weaker than) the classical notion in the degenerate deterministic case.

The purpose of this section is to formulate a definition of noninterference security that meet these requirements, applying to the concept of process as formalized by Hoare in his remarkable theory of Communicating Sequential Processes (CSP) [2]. The general case of a possibly intransitive noninterference policy will be considered.

Throughout this paper, the salient points of definitions and proofs are commented; for additional information see Isabelle documentation, particularly [7], [6], [5], and [3].

1.1 Processes

It is convenient to represent CSP processes by means of a type definition including a type variable, which stands for the process alphabet. Type *process* shall then be isomorphic to the subset of the product type of failures sets and divergences sets comprised of the pairs that satisfy the properties enunciated in [2], section 3.9. Such subset shall be shown to contain process *STOP*, which proves that it is nonempty.

Property *C5* is not considered as it is entailed by *C7*. Moreover, the formalization of properties *C2* and *C6* only takes into account event lists *t* containing a single item. Such formulation is equivalent to the original one, since the truth of *C2* and *C6* for a singleton list *t* immediately derives from that for a generic list, and conversely:

- the truth of *C2* and *C6* for a generic nonempty list *t* results from the repeated application of *C2* and *C6* for a singleton list;

- the truth of *C2* for t matching the empty list is implied by property *C3*;
- the truth of *C6* for t matching the empty list is a tautology.

The advantage of the proposed formulation is that it facilitates the task to prove that pairs of failures and divergences sets defined inductively indeed be processes, viz. be included in the set of pairs isomorphic to type *process*, since the introduction rules in such inductive definitions will typically construct process traces by appending one item at a time.

In what follows, the concept of process is formalized according to the previous considerations.

type-synonym *'a failure* = *'a list* × *'a set*

type-synonym *'a process-prod* = *'a failure set* × *'a list set*

definition *process-prop-1* :: *'a process-prod* ⇒ *bool* **where**
process-prop-1 P ≡ ([], {}) ∈ *fst P*

definition *process-prop-2* :: *'a process-prod* ⇒ *bool* **where**
process-prop-2 P ≡ ∀ *xs x X*. (*xs* @ [*x*], *X*) ∈ *fst P* ⟶ (*xs*, {}) ∈ *fst P*

definition *process-prop-3* :: *'a process-prod* ⇒ *bool* **where**
process-prop-3 P ≡ ∀ *xs X Y*. (*xs*, *Y*) ∈ *fst P* ∧ *X* ⊆ *Y* ⟶ (*xs*, *X*) ∈ *fst P*

definition *process-prop-4* :: *'a process-prod* ⇒ *bool* **where**
process-prop-4 P ≡ ∀ *xs x X*. (*xs*, *X*) ∈ *fst P* ⟶
(*xs* @ [*x*], {}) ∈ *fst P* ∨ (*xs*, *insert x X*) ∈ *fst P*

definition *process-prop-5* :: *'a process-prod* ⇒ *bool* **where**
process-prop-5 P ≡ ∀ *xs x*. *xs* ∈ *snd P* ⟶ *xs* @ [*x*] ∈ *snd P*

definition *process-prop-6* :: *'a process-prod* ⇒ *bool* **where**
process-prop-6 P ≡ ∀ *xs X*. *xs* ∈ *snd P* ⟶ (*xs*, *X*) ∈ *fst P*

definition *process-set* :: *'a process-prod set* **where**
process-set ≡ {*P*.
process-prop-1 P ∧
process-prop-2 P ∧
process-prop-3 P ∧
process-prop-4 P ∧
process-prop-5 P ∧
process-prop-6 P}

typedef *'a process* = *process-set* :: *'a process-prod set*
by (*rule-tac x* = ({(*xs*, *X*). *xs* = []}, {}) **in** *exI*, *simp add*:
process-set-def)

process-prop-1-def
process-prop-2-def
process-prop-3-def
process-prop-4-def
process-prop-5-def
process-prop-6-def)

Here below are the definitions of some functions acting on processes. Functions *failures*, *traces*, and *deterministic* match the homonymous notions defined in [2]. As for the other ones:

- *futures P xs* matches the failures set of process *P / xs*;
- *refusals P xs* matches the refusals set of process *P / xs*;
- *next-events P xs* matches the event set $(P \ / \ xs)^0$.

definition *failures* :: *'a process* ⇒ *'a failure set* **where**
failures P ≡ *fst* (*Rep-process P*)

definition *futures* :: *'a process* ⇒ *'a list* ⇒ *'a failure set* **where**
futures P xs ≡ {(*ys*, *Y*). (*xs* @ *ys*, *Y*) ∈ *failures P*}

definition *traces* :: *'a process* ⇒ *'a list set* **where**
traces P ≡ *Domain* (*failures P*)

definition *refusals* :: *'a process* ⇒ *'a list* ⇒ *'a set set* **where**
refusals P xs ≡ *failures P* `` {*xs*}

definition *next-events* :: *'a process* ⇒ *'a list* ⇒ *'a set* **where**
next-events P xs ≡ {*x*. *xs* @ [*x*] ∈ *traces P*}

definition *deterministic* :: *'a process* ⇒ *bool* **where**
deterministic P ≡
  ∀ *xs* ∈ *traces P*. ∀ *X*. *X* ∈ *refusals P xs* = (*X* ∩ *next-events P xs* = {})

In what follows, properties *process-prop-2* and *process-prop-3* of processes are put into the form of introduction rules, which will turn out to be useful in subsequent proofs. Particularly, the more general formulation of *process-prop-2* as given in [2] (section 3.9, property *C2*) is restored, and it is expressed in terms of both functions *failures* and *futures*.

lemma *process-rule-2*: (*xs* @ [*x*], *X*) ∈ *failures P* ⟹ (*xs*, {}) ∈ *failures P*
proof (*simp add*: *failures-def*)
  have *Rep-process P* ∈ *process-set* (**is** *?P'* ∈ -) **by** (*rule Rep-process*)

hence $\forall$ *xs x X*. (*xs* @ [*x*], *X*) $\in$ *fst ?P'* $\longrightarrow$ (*xs*, {}) $\in$ *fst ?P'*
by (*simp add*: *process-set-def process-prop-2-def*)
thus (*xs* @ [*x*], *X*) $\in$ *fst ?P'* $\Longrightarrow$ (*xs*, {}) $\in$ *fst ?P'* **by** *blast*
qed

lemma *process-rule-3*: (*xs*, *Y*) $\in$ *failures P* $\Longrightarrow$ *X* $\subseteq$ *Y* $\Longrightarrow$ (*xs*, *X*) $\in$ *failures P*
proof (*simp add*: *failures-def*)
have *Rep-process P* $\in$ *process-set* (**is** *?P'* $\in$ -) **by** (*rule Rep-process*)
hence $\forall$ *xs X Y*. (*xs*, *Y*) $\in$ *fst ?P'* $\wedge$ *X* $\subseteq$ *Y* $\longrightarrow$ (*xs*, *X*) $\in$ *fst ?P'*
by (*simp add*: *process-set-def process-prop-3-def*)
thus (*xs*, *Y*) $\in$ *fst ?P'* $\Longrightarrow$ *X* $\subseteq$ *Y* $\Longrightarrow$ (*xs*, *X*) $\in$ *fst ?P'* **by** *blast*
qed

lemma *process-rule-2-failures* [*rule-format*]:
(*xs* @ *xs'*, *X*) $\in$ *failures P* $\longrightarrow$ (*xs*, {}) $\in$ *failures P*
proof (*induction xs' arbitrary*: *X rule*: *rev-induct*, *rule-tac* [!] *impI*, *simp*)
fix *X*
assume (*xs*, *X*) $\in$ *failures P*
moreover have {} $\subseteq$ *X* **..**
ultimately show (*xs*, {}) $\in$ *failures P* **by** (*rule process-rule-3*)
next
fix *x xs' X*
assume $\bigwedge$*X*. (*xs* @ *xs'*, *X*) $\in$ *failures P* $\longrightarrow$ (*xs*, {}) $\in$ *failures P*
hence (*xs* @ *xs'*, {}) $\in$ *failures P* $\longrightarrow$ (*xs*, {}) $\in$ *failures P* **.**
moreover assume (*xs* @ *xs'* @ [*x*], *X*) $\in$ *failures P*
hence ((*xs* @ *xs'*) @ [*x*], *X*) $\in$ *failures P* **by** *simp*
hence (*xs* @ *xs'*, {}) $\in$ *failures P* **by** (*rule process-rule-2*)
ultimately show (*xs*, {}) $\in$ *failures P* **..**
qed

lemma *process-rule-2-futures*:
(*ys* @ *ys'*, *Y*) $\in$ *futures P xs* $\Longrightarrow$ (*ys*, {}) $\in$ *futures P xs*
by (*simp add*: *futures-def*, *simp only*: *append-assoc* [*symmetric*], *rule process-rule-2-failures*)

1.2 Noninterference

In the classical theory of noninterference, a deterministic state machine is considered to be secure just in case, for any trace of the machine and any action occurring next, the observable effect of the action, i.e. the produced output, is compatible with the assigned noninterference policy.

Thus, by analogy, it seems reasonable to regard a process as being noninterference-secure just in case, for any of its traces and any event occurring next, the observable effect of the event, i.e. the set of the possible futures of the process, is compatible with a given noninterference policy.

More precisely, let *sinks I D u xs* be the set of the security domains of the events within event list *xs* that may be affected by domain *u* according to interference relation *I*, where *D* is the mapping of events into their domains. Since the general case of a possibly intransitive relation *I* is considered, function *sinks* has to be defined recursively, similarly to what happens for function *sources* in [8]. However, contrariwise to function *sources*, function *sinks* takes into account the influence of the input domain on the input event list, so that the recursive decomposition of the latter has to be performed by item appending rather than prepending.

Furthermore, let *ipurge-tr I D u xs* be the sublist of event list *xs* obtained by recursively deleting the events that may be affected by domain *u* as detected via function *sinks*, and *ipurge-ref I D u xs X* be the subset of refusal *X* whose elements may not be affected by either *u* or any domain in *sinks I D u xs*.

Then, a process *P* is secure just in case, for each event list *xs* and each $(y \# ys, Y)$, $(zs, Z) \in$ *futures P xs*, both of the following conditions are satisfied:

- (*ipurge-tr I D* (*D y*) *ys*, *ipurge-ref I D* (*D y*) *ys Y*) $\in$ *futures P xs*.
  Otherwise, the absence of event *y* after *xs* would affect the possibility for pair (*ipurge-tr I D* (*D y*) *ys*, *ipurge-ref I D* (*D y*) *ys Y*) to occur as a future of *xs*, although its components, except for the deletion of *y*, are those of possible future $(y \# ys, Y)$ deprived of any event allowed to be affected by *y*.

- (*y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs Z*)
  $\in$ *futures P xs*.
  Otherwise, the presence of event *y* after *xs* would affect the possibility for pair (*y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs Z*) to occur as a future of *xs*, although its components, except for the addition of *y*, are those of possible future (zs, Z) deprived of any event allowed to be affected by *y*.

Observe that this definition of security, henceforth referred to as *CSP noninterference security*, does not rest on the supposition that noninterference policy *I* be reflexive, even though any policy of practical significance will be such.

Moreover, this simpler formulation is equivalent to the one obtained by restricting the range of event list *xs* to the traces of process *P*. In fact, for each *zs*, *Z*, $(zs, Z) \in$ *futures P xs* just in case $(xs @ zs, Z) \in$ *failures P*, which by virtue of rule *process-rule-2-failures* implies that *xs* is a trace of *P*. Therefore, formula $(zs, Z) \in$ *futures P xs* is invariably false in case *xs* is not a trace of *P*.

Here below are the formal counterparts of the definitions discussed so far.

function *sinks* :: $('d \times 'd)$ *set* $\Rightarrow$ $('a \Rightarrow 'd) \Rightarrow 'd \Rightarrow 'a$ *list* $\Rightarrow$ $'d$ *set* **where**
sinks - - - [] = {} |

sinks I D u (*xs* @ [*x*]) = (*if* (*u*, *D x*) ∈ *I* ∨ (∃ *v* ∈ *sinks I D u xs*. (*v*, *D x*) ∈ *I*)
  then insert (*D x*) (*sinks I D u xs*)
  else sinks I D u xs)
proof (*atomize-elim*, *simp-all add*: *split-paired-all*)
qed (*rule rev-cases*, *rule disjI1*, *assumption*, *simp*)
termination by *lexicographic-order*

function *ipurge-tr* :: (*'d* × *'d*) *set* ⇒ (*'a* ⇒ *'d*) ⇒ *'d* ⇒ *'a list* ⇒ *'a list* **where**
ipurge-tr - - - [] = [] |
ipurge-tr I D u (*xs* @ [*x*]) = (*if D x* ∈ *sinks I D u* (*xs* @ [*x*])
  then ipurge-tr I D u xs
  else ipurge-tr I D u xs @ [*x*])
proof (*atomize-elim*, *simp-all add*: *split-paired-all*)
qed (*rule rev-cases*, *rule disjI1*, *assumption*, *simp*)
termination by *lexicographic-order*

definition *ipurge-ref* ::
 (*'d* × *'d*) *set* ⇒ (*'a* ⇒ *'d*) ⇒ *'d* ⇒ *'a list* ⇒ *'a set* ⇒ *'a set* **where**
ipurge-ref I D u xs X ≡
  {*x* ∈ *X*. (*u*, *D x*) ∉ *I* ∧ (∀ *v* ∈ *sinks I D u xs*. (*v*, *D x*) ∉ *I*)}

definition *secure* :: *'a process* ⇒ (*'d* × *'d*) *set* ⇒ (*'a* ⇒ *'d*) ⇒ *bool* **where**
secure P I D ≡
  ∀ *xs y ys Y zs Z*. (*y* # *ys*, *Y*) ∈ *futures P xs* ∧ (*zs*, *Z*) ∈ *futures P xs* ⟶
  (*ipurge-tr I D* (*D y*) *ys*, *ipurge-ref I D* (*D y*) *ys Y*) ∈ *futures P xs* ∧
  (*y* # *ipurge-tr I D* (*D y*) *zs*, *ipurge-ref I D* (*D y*) *zs Z*) ∈ *futures P xs*

The continuation of this section is dedicated to the demonstration of some lemmas concerning functions *sinks*, *ipurge-tr*, and *ipurge-ref* which will turn out to be useful in subsequent proofs.

lemma *sinks-cons-same*:
  assumes *R*: *refl I*
  shows *sinks I D* (*D x*) (*x* # *xs*) = *insert* (*D x*) (*sinks I D* (*D x*) *xs*)
proof (*rule rev-induct*, *simp*)
  have *A*: [*x*] = [] @ [*x*] **by** *simp*
  have *sinks I D* (*D x*) [*x*] = (*if* (*D x*, *D x*) ∈ *I* ∨ (∃ *v* ∈ {}. (*v*, *D x*) ∈ *I*)
    then insert (*D x*) {}
    else {})
   by (*subst A*, *simp only*: *sinks.simps*)
  moreover have (*D x*, *D x*) ∈ *I* **using** *R* **by** (*simp add*: *refl-on-def*)
  ultimately show *sinks I D* (*D x*) [*x*] = {*D x*} **by** *simp*
next
  fix *x' xs*
  assume *A*: *sinks I D* (*D x*) (*x* # *xs*) = *insert* (*D x*) (*sinks I D* (*D x*) *xs*)
  show *sinks I D* (*D x*) (*x* # *xs* @ [*x'*]) =
    insert (*D x*) (*sinks I D* (*D x*) (*xs* @ [*x'*]))
  proof (*cases* (*D x*, *D x'*) ∈ *I* ∨ (∃ *v* ∈ *sinks I D* (*D x*) *xs*. (*v*, *D x'*) ∈ *I*),

simp-all (*no-asm-simp*))
case *True*
hence $(D\ x,\ D\ x') \in I \vee (\exists\, v \in \mathit{sinks}\ I\ D\ (D\ x)\ (x\ \#\ xs).\ (v,\ D\ x') \in I)$
using *A* **by** *simp*
hence *sinks I D* (*D x*) ((*x* # *xs*) @ [*x'*]) =
insert (*D x'*) (*sinks I D* (*D x*) (*x* # *xs*))
by (*simp only*: *sinks.simps if-True*)
thus *sinks I D* (*D x*) (*x* # *xs* @ [*x'*]) =
insert (*D x*) (*insert* (*D x'*) (*sinks I D* (*D x*) *xs*))
using *A* **by** (*simp add*: *insert-commute*)
next
case *False*
hence $\neg\ ((D\ x,\ D\ x') \in I \vee (\exists\, v \in \mathit{sinks}\ I\ D\ (D\ x)\ (x\ \#\ xs).\ (v,\ D\ x') \in I))$
using *A* **by** *simp*
hence *sinks I D* (*D x*) ((*x* # *xs*) @ [*x'*]) = *sinks I D* (*D x*) (*x* # *xs*)
by (*simp only*: *sinks.simps if-False*)
thus *sinks I D* (*D x*) (*x* # *xs* @ [*x'*]) = *insert* (*D x*) (*sinks I D* (*D x*) *xs*)
using *A* **by** *simp*
qed
qed

lemma *ipurge-tr-cons-same*:
assumes *R*: *refl I*
shows *ipurge-tr I D* (*D x*) (*x* # *xs*) = *ipurge-tr I D* (*D x*) *xs*
proof (*induction xs rule*: *rev-induct*, *simp*)
have *A*: [*x*] = [] @ [*x*] **by** *simp*
have *ipurge-tr I D* (*D x*) [*x*] = (*if D x* ∈ *sinks I D* (*D x*) ([] @ [*x*])
then []
else [] @ [*x*])
by (*subst A*, *simp only*: *ipurge-tr.simps*)
moreover have *sinks I D* (*D x*) [*x*] = {*D x*}
using *R* **by** (*simp add*: *sinks-cons-same*)
ultimately show *ipurge-tr I D* (*D x*) [*x*] = [] **by** *simp*
next
fix *x' xs*
assume *A*: *ipurge-tr I D* (*D x*) (*x* # *xs*) = *ipurge-tr I D* (*D x*) *xs*
show *ipurge-tr I D* (*D x*) (*x* # *xs* @ [*x'*]) = *ipurge-tr I D* (*D x*) (*xs* @ [*x'*])
proof (*cases D x'* ∈ *sinks I D* (*D x*) (*x* # *xs* @ [*x'*]))
assume *B*: *D x'* ∈ *sinks I D* (*D x*) (*x* # *xs* @ [*x'*])
hence *D x'* ∈ *sinks I D* (*D x*) ((*x* # *xs*) @ [*x'*]) **by** *simp*
hence *ipurge-tr I D* (*D x*) ((*x* # *xs*) @ [*x'*]) = *ipurge-tr I D* (*D x*) (*x* # *xs*)
by (*simp only*: *ipurge-tr.simps if-True*)
hence *C*: *ipurge-tr I D* (*D x*) (*x* # *xs* @ [*x'*]) = *ipurge-tr I D* (*D x*) *xs*
using *A* **by** *simp*
have *D x'* = *D x* ∨ *D x'* ∈ *sinks I D* (*D x*) (*xs* @ [*x'*])
using *R* **and** *B* **by** (*simp add*: *sinks-cons-same*)
moreover {
assume *D x'* = *D x*
hence (*D x*, *D x'*) ∈ *I* **using** *R* **by** (*simp add*: *refl-on-def*)

hence *ipurge-tr I D* (*D x*) (*xs* @ [*x'*]) = *ipurge-tr I D* (*D x*) *xs* **by** *simp*
}
moreover {
assume *D x'* ∈ *sinks I D* (*D x*) (*xs* @ [*x'*])
hence *ipurge-tr I D* (*D x*) (*xs* @ [*x'*]) = *ipurge-tr I D* (*D x*) *xs* **by** *simp*
}
ultimately have *D*: *ipurge-tr I D* (*D x*) (*xs* @ [*x'*]) = *ipurge-tr I D* (*D x*) *xs*
by *blast*
show *?thesis* **using** *C* **and** *D* **by** *simp*
next
assume *B*: *D x'* ∉ *sinks I D* (*D x*) (*x* # *xs* @ [*x'*])
hence *D x'* ∉ *sinks I D* (*D x*) ((*x* # *xs*) @ [*x'*]) **by** *simp*
hence *ipurge-tr I D* (*D x*) ((*x* # *xs*) @ [*x'*]) =
ipurge-tr I D (*D x*) (*x* # *xs*) @ [*x'*]
by (*simp only*: *ipurge-tr.simps if-False*)
hence *ipurge-tr I D* (*D x*) (*x* # *xs* @ [*x'*]) = *ipurge-tr I D* (*D x*) *xs* @ [*x'*]
using *A* **by** *simp*
moreover have ¬ (*D x'* = *D x* ∨ *D x'* ∈ *sinks I D* (*D x*) (*xs* @ [*x'*]))
using *R* **and** *B* **by** (*simp add*: *sinks-cons-same*)
hence *ipurge-tr I D* (*D x*) (*xs* @ [*x'*]) = *ipurge-tr I D* (*D x*) *xs* @ [*x'*]
by *simp*
ultimately show *?thesis* **by** *simp*
qed
qed

lemma *sinks-cons-nonint*:
assumes *A*: (*u*, *D x*) ∉ *I*
shows *sinks I D u* (*x* # *xs*) = *sinks I D u xs*
proof (*rule rev-induct*, *simp*)
have *sinks I D u* [*x*] = *sinks I D u* ([] @ [*x*]) **by** *simp*
hence *sinks I D u* [*x*] = (*if* (*u*, *D x*) ∈ *I* ∨ (∃ *v* ∈ {}. (*v*, *D x*) ∈ *I*)
then insert (*D x*) {}
else {})
by (*simp only*: *sinks.simps*)
thus *sinks I D u* [*x*] = {} **using** *A* **by** *simp*
next
fix *xs x'*
assume *B*: *sinks I D u* (*x* # *xs*) = *sinks I D u xs* (**is** *?d'* = *?d*)
have *x* # *xs* @ [*x'*] = (*x* # *xs*) @ [*x'*] **by** *simp*
hence *C*: *sinks I D u* (*x* # *xs* @ [*x'*]) =
(*if* (*u*, *D x'*) ∈ *I* ∨ (∃ *v* ∈ *?d'*. (*v*, *D x'*) ∈ *I*)
then insert (*D x'*) *?d'*
else ?d')
by (*simp only*: *sinks.simps*)
show *sinks I D u* (*x* # *xs* @ [*x'*]) = *sinks I D u* (*xs* @ [*x'*])
proof (*cases* (*u*, *D x'*) ∈ *I* ∨ (∃ *v* ∈ *?d*. (*v*, *D x'*) ∈ *I*))
case *True*
with *B* **and** *C* **have** *sinks I D u* (*x* # *xs* @ [*x'*]) = *insert* (*D x'*) *?d*
by *simp*

with *True* **show** *?thesis* **by** *simp*
next
case *False*
with *B* **and** *C* **have** *sinks I D u* (*x* # *xs* @ [*x'*]) = *?d* **by** *simp*
with *False* **show** *?thesis* **by** *simp*
qed
qed

lemma *sinks-empty* [*rule-format*]:
sinks I D u xs = {} ⟶ *ipurge-tr I D u xs* = *xs*
proof (*rule rev-induct*, *simp*, *rule impI*)
fix *x xs*
assume *A*: *sinks I D u* (*xs* @ [*x*]) = {}
moreover have *sinks I D u xs* ⊆ *sinks I D u* (*xs* @ [*x*])
by (*simp add*: *subset-insertI*)
ultimately have *sinks I D u xs* = {} **by** *simp*
moreover assume *sinks I D u xs* = {} ⟶ *ipurge-tr I D u xs* = *xs*
ultimately have *ipurge-tr I D u xs* = *xs* **by** (*rule rev-mp*)
thus *ipurge-tr I D u* (*xs* @ [*x*]) = *xs* @ [*x*] **using** *A* **by** *simp*
qed

lemma *ipurge-ref-eq*:
assumes *A*: *D x* ∈ *sinks I D u* (*xs* @ [*x*])
shows *ipurge-ref I D u* (*xs* @ [*x*]) *X* =
ipurge-ref I D u xs {*x'* ∈ *X*. (*D x*, *D x'*) ∉ *I*}
proof (*rule equalityI*, *rule-tac* [!] *subsetI*, *simp-all add*: *ipurge-ref-def del*: *sinks.simps*,
(*erule conjE*)+, (*erule-tac* [*2*] *conjE*)+)
fix *y*
assume *B*: ∀ *v* ∈ *sinks I D u* (*xs* @ [*x*]). (*v*, *D y*) ∉ *I*
show (*D x*, *D y*) ∉ *I* ∧ (∀ *v* ∈ *sinks I D u xs*. (*v*, *D y*) ∉ *I*)
proof (*rule conjI*, *rule-tac* [*2*] *ballI*)
show (*D x*, *D y*) ∉ *I* **using** *B* **and** *A* **..**
next
fix *v*
assume *v* ∈ *sinks I D u xs*
hence *v* ∈ *sinks I D u* (*xs* @ [*x*]) **by** *simp*
with *B* **show** (*v*, *D y*) ∉ *I* **..**
qed
next
fix *y*
assume
B: (*D x*, *D y*) ∉ *I* **and**
C: ∀ *v* ∈ *sinks I D u xs*. (*v*, *D y*) ∉ *I*
show ∀ *v* ∈ *sinks I D u* (*xs* @ [*x*]). (*v*, *D y*) ∉ *I*
proof (*rule ballI*, *cases* (*u*, *D x*) ∈ *I* ∨ (∃ *v* ∈ *sinks I D u xs*. (*v*, *D x*) ∈ *I*))
fix *v*
case *True*
moreover assume *v* ∈ *sinks I D u* (*xs* @ [*x*])
ultimately have *v* = *D x* ∨ *v* ∈ *sinks I D u xs* **by** *simp*

```
    moreover {
      assume v = D x
      with B have (v, D y) ∉ I by simp
    }
    moreover {
      assume v ∈ sinks I D u xs
      with C have (v, D y) ∉ I ..
    }
    ultimately show (v, D y) ∉ I by blast
  next
    fix v
    case False
    moreover assume v ∈ sinks I D u (xs @ [x])
    ultimately have v ∈ sinks I D u xs by simp
    with C show (v, D y) ∉ I ..
  qed
qed

end
```

2 CSP noninterference vs. classical noninterference

```
theory ClassicalNoninterference
imports CSPNoninterference
begin
```

The purpose of this section is to prove the equivalence of CSP noninterference security as defined previously to the classical notion of noninterference security as formulated in [8] in the case of processes representing deterministic state machines, henceforth briefly referred to as *classical processes*.

For clarity, all the constants and fact names defined in this section, with the possible exception of main theorems, contain prefix *c-*.

2.1 Classical noninterference

Here below are the formalizations of the functions *sources* and *ipurge* defined in [8], as well as of the classical notion of noninterference security as stated ibid. for a deterministic state machine in the general case of a possibly intransitive noninterference policy.

Observe that the function *run* used in *R3* is formalized as function *foldl step*, where *step* is the state transition function of the machine.

```
primrec c-sources :: ('d × 'd) set ⇒ ('a ⇒ 'd) ⇒ 'd ⇒ 'a list ⇒ 'd set where
```

c-sources - - u [] = {*u*} |
c-sources I D u (*x* # *xs*) = (*if* ∃ *v* ∈ *c-sources I D u xs*. (*D x*, *v*) ∈ *I*
  then insert (*D x*) (*c-sources I D u xs*)
  else c-sources I D u xs)

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

definition *c-secure* ::
 ('*s* ⇒ '*a* ⇒ '*s*) ⇒ ('*s* ⇒ '*a* ⇒ '*o*) ⇒ '*s* ⇒ ('*d* × '*d*) *set* ⇒ ('*a* ⇒ '*d*) ⇒ *bool*
where
c-secure step out s_0 *I D* ≡
 ∀ *x xs*. *out* (*foldl step* s_0 *xs*) *x* = *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*) *xs*)) *x*

In addition, the definitions are given of variants of functions *c-sources* and *c-ipurge* accepting in input a set of security domains rather than a single domain, and then some lemmas concerning them are demonstrated. These definitions and lemmas will turn out to be useful in subsequent proofs.

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

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

lemma *c-sources-aux-singleton-1*: *c-sources-aux I D* {*u*} *xs* = *c-sources I D u xs*
by (*induction xs*, *simp-all*)

lemma *c-ipurge-aux-singleton*: *c-ipurge-aux I D* {*u*} *xs* = *c-ipurge I D u xs*
by (*induction xs*, *simp-all add*: *c-sources-aux-singleton-1*)

lemma *c-sources-aux-singleton-2*:
 D x ∈ *c-sources-aux I D U* [*x*] = (*D x* ∈ *U* ∨ (∃ *v* ∈ *U*. (*D x*, *v*) ∈ *I*))
by *simp*

lemma *c-sources-aux-append*:
 c-sources-aux I D U (*xs* @ [*x*]) = (*if D x* ∈ *c-sources-aux I D U* [*x*]

then c-sources-aux I D (insert (D x) U) xs
else c-sources-aux I D U xs)
by (*induction xs*, *simp-all add*: *insert-absorb*)

lemma *c-ipurge-aux-append*:
c-ipurge-aux I D U (xs @ [x]) = (if D x ∈ c-sources-aux I D U [x]
then c-ipurge-aux I D (insert (D x) U) xs @ [x]
else c-ipurge-aux I D U xs)
by (*induction xs*, *simp-all add*: *c-sources-aux-append*)

In what follows, a few useful lemmas are proven about functions *c-sources*, *c-ipurge* and their relationships with functions *sinks*, *ipurge-tr*.

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

lemma *c-sources-append-1*:
c-sources I D (D x) (xs @ [x]) = c-sources I D (D x) xs
by (*induction xs*, *simp-all*)

lemma *c-ipurge-append-1*:
c-ipurge I D (D x) (xs @ [x]) = c-ipurge I D (D x) xs @ [x]
by (*induction xs*, *simp-all add*: *c-sources-append-1*)

lemma *c-sources-append-2*:
(D x, u) ∉ I ⟹ c-sources I D u (xs @ [x]) = c-sources I D u xs
by (*induction xs*, *simp-all*)

lemma *c-ipurge-append-2*:
refl I ⟹ (D x, u) ∉ I ⟹ c-ipurge I D u (xs @ [x]) = c-ipurge I D u xs
proof (*induction xs*, *simp-all add*: *refl-on-def c-sources-append-2*)
qed (*rule notI*, *simp*)

lemma *c-sources-mono*:
assumes *A*: *c-sources I D u ys ⊆ c-sources I D u zs*
shows *c-sources I D u (x # ys) ⊆ c-sources I D u (x # zs)*
proof (*cases ∃ v ∈ c-sources I D u ys. (D x, v) ∈ I*)
assume *B*: *∃ v ∈ c-sources I D u ys. (D x, v) ∈ I*
then obtain *v* **where** *C*: *v ∈ c-sources I D u ys* **and** *D*: *(D x, v) ∈ I* **..**
from *A* **and** *C* **have** *v ∈ c-sources I D u zs* **..**
with *D* **have** *E*: *∃ v ∈ c-sources I D u zs. (D x, v) ∈ I* **..**
have *insert (D x) (c-sources I D u ys) ⊆ insert (D x) (c-sources I D u zs)*
using *A* **by** (*rule insert-mono*)
moreover have *c-sources I D u (x # ys) = insert (D x) (c-sources I D u ys)*
using *B* **by** *simp*
moreover have *c-sources I D u (x # zs) = insert (D x) (c-sources I D u zs)*
using *E* **by** *simp*
ultimately show *c-sources I D u (x # ys) ⊆ c-sources I D u (x # zs)* **by** *simp*

next
  assume ¬ (∃ *v* ∈ *c-sources I D u ys*. (*D x*, *v*) ∈ *I*)
  hence *c-sources I D u* (*x* # *ys*) = *c-sources I D u ys* **by** *simp*
  hence *c-sources I D u* (*x* # *ys*) ⊆ *c-sources I D u zs* **using** *A* **by** *simp*
  moreover have *c-sources I D u zs* ⊆ *c-sources I D u* (*x* # *zs*)
   by (*simp add*: *subset-insertI*)
  ultimately show *c-sources I D u* (*x* # *ys*) ⊆ *c-sources I D u* (*x* # *zs*) **by** *simp*
qed

lemma *c-sources-sinks* [*rule-format*]:
  D x ∉ *c-sources I D u* (*x* # *xs*) ⟶ *sinks I D* (*D x*) (*c-ipurge I D u xs*) = {}
proof (*induction xs*, *simp*, *rule impI*)
  fix *x′ xs*
  assume *A*: *D x* ∉ *c-sources I D u* (*x* # *xs*) ⟶
    sinks I D (*D x*) (*c-ipurge I D u xs*) = {}
  assume *B*: *D x* ∉ *c-sources I D u* (*x* # *x′* # *xs*)
  have *c-sources I D u xs* ⊆ *c-sources I D u* (*x′* # *xs*)
   by (*simp add*: *subset-insertI*)
  hence *c-sources I D u* (*x* # *xs*) ⊆ *c-sources I D u* (*x* # *x′* # *xs*)
   by (*rule c-sources-mono*)
  hence *D x* ∉ *c-sources I D u* (*x* # *xs*) **using** *B* **by** (*rule contra-subsetD*)
  with *A* **have** *C*: *sinks I D* (*D x*) (*c-ipurge I D u xs*) = {} **..**
  show *sinks I D* (*D x*) (*c-ipurge I D u* (*x′* # *xs*)) = {}
  proof (*cases D x′* ∈ *c-sources I D u* (*x′* # *xs*),
   simp-all only: *c-ipurge.simps if-True if-False*)
    assume *D*: *D x′* ∈ *c-sources I D u* (*x′* # *xs*)
    have (*D x*, *D x′*) ∉ *I*
    proof
      assume (*D x*, *D x′*) ∈ *I*
      hence ∃ *v* ∈ *c-sources I D u* (*x′* # *xs*). (*D x*, *v*) ∈ *I* **using** *D* **..**
      hence *D x* ∈ *c-sources I D u* (*x* # *x′* # *xs*) **by** *simp*
      thus *False* **using** *B* **by** *contradiction*
    qed
    thus *sinks I D* (*D x*) (*x′* # *c-ipurge I D u xs*) = {}
     using *C* **by** (*simp add*: *sinks-cons-nonint*)
  next
    show *sinks I D* (*D x*) (*c-ipurge I D u xs*) = {} **using** *C* **.**
  qed
qed

lemmas *c-ipurge-tr-ipurge* = *c-sources-sinks* [*THEN sinks-empty*]

lemma *c-ipurge-aux-ipurge-tr* [*rule-format*]:
  assumes *R*: *refl I*
  shows ¬ (∃ *v* ∈ *sinks I D u ys*. ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I*) ⟶
    c-ipurge-aux I D U (*xs* @ *ipurge-tr I D u ys*) = *c-ipurge-aux I D U* (*xs* @ *ys*)
proof (*induction ys arbitrary*: *U rule*: *rev-induct*, *simp*, *rule impI*)
  fix *y ys U*
  assume

A: ⋀*U*. ¬ (∃ *v* ∈ *sinks I D u ys*. ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I*) ⟶
c-ipurge-aux I D U (*xs* @ *ipurge-tr I D u ys*) =
c-ipurge-aux I D U (*xs* @ *ys*) **and**
B: ¬ (∃ *v* ∈ *sinks I D u* (*ys* @ [*y*]). ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I*)
have *C*: ¬ (∃ *v* ∈ *sinks I D u ys*. ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I*)
proof (*rule notI*, (*erule bexE*)+)
fix *v w*
assume (*v*, *w*) ∈ *I* **and** *w* ∈ *U*
hence ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I* **..**
moreover assume *v* ∈ *sinks I D u ys*
hence *v* ∈ *sinks I D u* (*ys* @ [*y*]) **by** *simp*
ultimately have ∃ *v* ∈ *sinks I D u* (*ys* @ [*y*]). ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I* **..**
thus *False* **using** *B* **by** *contradiction*
qed
show *c-ipurge-aux I D U* (*xs* @ *ipurge-tr I D u* (*ys* @ [*y*])) =
c-ipurge-aux I D U (*xs* @ *ys* @ [*y*])
proof (*cases D y* ∈ *c-sources-aux I D U* [*y*],
case-tac [!] *D y* ∈ *sinks I D u* (*ys* @ [*y*]),
simp-all (*no-asm-simp*) *only*: *ipurge-tr.simps append-assoc* [*symmetric*]
c-ipurge-aux-append append-same-eq if-True if-False)
assume *D*: *D y* ∈ *sinks I D u* (*ys* @ [*y*])
assume *D y* ∈ *c-sources-aux I D U* [*y*]
hence *D y* ∈ *U* ∨ (∃ *w* ∈ *U*. (*D y*, *w*) ∈ *I*)
by (*simp only*: *c-sources-aux-singleton-2*)
moreover **{**
have (*D y*, *D y*) ∈ *I* **using** *R* **by** (*simp add*: *refl-on-def*)
moreover assume *D y* ∈ *U*
ultimately have ∃ *w* ∈ *U*. (*D y*, *w*) ∈ *I* **..**
hence ∃ *v* ∈ *sinks I D u* (*ys* @ [*y*]). ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I* **using** *D* **..**
}
moreover **{**
assume ∃ *w* ∈ *U*. (*D y*, *w*) ∈ *I*
hence ∃ *v* ∈ *sinks I D u* (*ys* @ [*y*]). ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I* **using** *D* **..**
}
ultimately have ∃ *v* ∈ *sinks I D u* (*ys* @ [*y*]). ∃ *w* ∈ *U*. (*v*, *w*) ∈ *I* **by** *blast*
thus *c-ipurge-aux I D U* (*xs* @ *ipurge-tr I D u ys*) =
c-ipurge-aux I D (*insert* (*D y*) *U*) (*xs* @ *ys*) @ [*y*]
using *B* **by** *contradiction*
next
assume *D*: *D y* ∉ *sinks I D u* (*ys* @ [*y*])
have ¬ (∃ *v* ∈ *sinks I D u ys*. ∃ *w* ∈ *insert* (*D y*) *U*. (*v*, *w*) ∈ *I*) ⟶
c-ipurge-aux I D (*insert* (*D y*) *U*) (*xs* @ *ipurge-tr I D u ys*) =
c-ipurge-aux I D (*insert* (*D y*) *U*) (*xs* @ *ys*)
using *A* **.**
moreover have ¬ (∃ *v* ∈ *sinks I D u ys*. ∃ *w* ∈ *insert* (*D y*) *U*. (*v*, *w*) ∈ *I*)
proof (*rule notI*, (*erule bexE*)+, *simp*, *erule disjE*, *simp*)
fix *v*
assume (*v*, *D y*) ∈ *I* **and** *v* ∈ *sinks I D u ys*
hence ∃ *v* ∈ *sinks I D u ys*. (*v*, *D y*) ∈ *I* **..**

```
    hence D y ∈ sinks I D u (ys @ [y]) by simp
    thus False using D by contradiction
  next
    fix v w
    assume (v, w) ∈ I and w ∈ U
    hence ∃ w ∈ U. (v, w) ∈ I ..
    moreover assume v ∈ sinks I D u ys
    ultimately have ∃ v ∈ sinks I D u ys. ∃ w ∈ U. (v, w) ∈ I ..
    thus False using C by contradiction
  qed
  ultimately show c-ipurge-aux I D (insert (D y) U) (xs @ ipurge-tr I D u ys)
    = c-ipurge-aux I D (insert (D y) U) (xs @ ys) ..
 next
  have ¬ (∃ v ∈ sinks I D u ys. ∃ w ∈ U. (v, w) ∈ I) ⟶
    c-ipurge-aux I D U (xs @ ipurge-tr I D u ys) = c-ipurge-aux I D U (xs @ ys)
   using A .
  thus c-ipurge-aux I D U (xs @ ipurge-tr I D u ys) =
    c-ipurge-aux I D U (xs @ ys)
   using C ..
 next
  have ¬ (∃ v ∈ sinks I D u ys. ∃ w ∈ U. (v, w) ∈ I) ⟶
    c-ipurge-aux I D U (xs @ ipurge-tr I D u ys) = c-ipurge-aux I D U (xs @ ys)
   using A .
  thus c-ipurge-aux I D U (xs @ ipurge-tr I D u ys) =
    c-ipurge-aux I D U (xs @ ys)
   using C ..
 qed
qed

lemma c-ipurge-ipurge-tr:
 assumes R: refl I and D: ¬ (∃ v ∈ sinks I D u ys. (v, u') ∈ I)
 shows c-ipurge I D u' (xs @ ipurge-tr I D u ys) = c-ipurge I D u' (xs @ ys)
proof −
 have ¬ (∃ v ∈ sinks I D u ys. ∃ w ∈ {u'}. (v, w) ∈ I) using D by simp
 with R have c-ipurge-aux I D {u'} (xs @ ipurge-tr I D u ys) =
  c-ipurge-aux I D {u'} (xs @ ys)
  by (rule c-ipurge-aux-ipurge-tr)
 thus ?thesis by (simp add: c-ipurge-aux-singleton)
qed
```

2.2 Classical processes

The deterministic state machines used as model of computation in the classical theory of noninterference security, as expounded in [8], have the property that each action produces an output. Hence, it is natural to take as alphabet of a classical process the universe of the pairs (x, p), where x is an action and p an output. For any state s, such an event (x, p) may occur just in case p matches the output produced by x in s.

Therefore, a trace of a classical process can be defined as an event list *xps* such that for each item (x, p), p is equal to the output produced by x in the state resulting from the previous actions in *xps*. Furthermore, for each trace *xps*, the refusals set associated to *xps* is comprised of any set of pairs (x, p) such that p is different from the output produced by x in the state resulting from the actions in *xps*.

In accordance with the previous considerations, an inductive definition is formulated here below for the failures set *c-failures step out* s_0 corresponding to the deterministic state machine with state transition function *step*, output function *out*, and initial state s_0. Then, the classical process *c-process step out* s_0 representing this machine is defined as the process having *c-failures step out* s_0 as failures set and the empty set as divergences set.

inductive-set *c-failures* ::
 $('s \Rightarrow 'a \Rightarrow 's) \Rightarrow ('s \Rightarrow 'a \Rightarrow 'o) \Rightarrow 's \Rightarrow ('a \times 'o)$ *failure set*
for *step* :: $'s \Rightarrow 'a \Rightarrow 's$ **and** *out* :: $'s \Rightarrow 'a \Rightarrow 'o$ **and** s_0 :: $'s$ **where**
R0: ([], $\{(x, p).\ p \neq out\ s_0\ x\}$) $\in$ *c-failures step out* s_0 |
R1: ⟦(*xps*, -) $\in$ *c-failures step out* s_0; s = *foldl step* s_0 (*map fst xps*)⟧ $\Longrightarrow$
  (*xps* @ [(x, *out s x*)], $\{(y, p).\ p \neq out\ (step\ s\ x)\ y\}$) $\in$ *c-failures step out* s_0 |
R2: ⟦(*xps*, Y) $\in$ *c-failures step out* s_0; $X \subseteq Y$⟧ $\Longrightarrow$
  (*xps*, X) $\in$ *c-failures step out* s_0

definition *c-process* ::
 $('s \Rightarrow 'a \Rightarrow 's) \Rightarrow ('s \Rightarrow 'a \Rightarrow 'o) \Rightarrow 's \Rightarrow ('a \times 'o)$ *process* **where**
c-process step out s_0 $\equiv$ *Abs-process* (*c-failures step out* s_0, {})

In what follows, the fact that classical processes are indeed processes is proven as a theorem.

lemma *c-process-prop-1* [*simp*]: *process-prop-1* (*c-failures step out* s_0, {})
proof (*simp add*: *process-prop-1-def*)
 have ([], $\{(x, p).\ p \neq out\ s_0\ x\}$) $\in$ *c-failures step out* s_0 **by** (*rule R0*)
 moreover have $\{\} \subseteq \{(x, p).\ p \neq out\ s_0\ x\}$ **..**
 ultimately show ([], {}) $\in$ *c-failures step out* s_0 **by** (*rule R2*)
qed

lemma *c-process-prop-2* [*simp*]: *process-prop-2* (*c-failures step out* s_0, {})
proof (*simp only*: *process-prop-2-def fst-conv*, (*rule allI*)+, *rule impI*)
 fix *xps xp X*
 assume (*xps* @ [*xp*], X) $\in$ *c-failures step out* s_0
 hence (*butlast* (*xps* @ [*xp*]), {}) $\in$ *c-failures step out* s_0
 proof (*rule c-failures.induct*
  [**where** $P = \lambda xps\ X.$ (*butlast xps*, {}) $\in$ *c-failures step out* s_0], *simp-all*)
  have ([], $\{(x, p).\ p \neq out\ s_0\ x\}$) $\in$ *c-failures step out* s_0 **by** (*rule R0*)
  moreover have $\{\} \subseteq \{(x, p).\ p \neq out\ s_0\ x\}$ **..**
  ultimately show ([], {}) $\in$ *c-failures step out* s_0 **by** (*rule R2*)
 next

fix *xps'* *X'*
assume (*xps'*, *X'*) ∈ *c-failures step out* s_0
moreover have {} ⊆ *X'* **..**
ultimately show (*xps'*, {}) ∈ *c-failures step out* s_0 **by** (*rule R2*)
qed
thus (*xps*, {}) ∈ *c-failures step out* s_0 **by** *simp*
qed

lemma *c-process-prop-3* [*simp*]: *process-prop-3* (*c-failures step out* s_0, {})
by (*simp only*: *process-prop-3-def fst-conv*, (*rule allI*)+, *rule impI*, *erule conjE*, *rule R2*)

lemma *c-process-prop-4* [*simp*]: *process-prop-4* (*c-failures step out* s_0, {})
proof (*simp only*: *process-prop-4-def fst-conv*, (*rule allI*)+, *rule impI*)
fix *xps xp X*
assume (*xps*, *X*) ∈ *c-failures step out* s_0
thus (*xps* @ [*xp*], {}) ∈ *c-failures step out* s_0 ∨
(*xps*, *insert xp X*) ∈ *c-failures step out* s_0
proof (*case-tac xp*, *rule c-failures.induct*)
fix *x p*
assume *A*: *xp* = (*x*, *p*)
have *B*: ([], {(*x*, *p*). *p* ≠ *out* s_0 *x*}) ∈ *c-failures step out* s_0
(**is** (-, *?X*) ∈ -) **by** (*rule R0*)
show ([] @ [*xp*], {}) ∈ *c-failures step out* s_0 ∨ ([], *insert xp ?X*)
∈ *c-failures step out* s_0
proof (*cases p* = *out* s_0 *x*)
assume *C*: *p* = *out* s_0 *x*
have s_0 = *foldl step* s_0 (*map fst* []) **by** *simp*
with *B* **have** ([] @ [(*x*, *out* s_0 *x*)], {(*y*, *p*). *p* ≠ *out* (*step* s_0 *x*) *y*})
∈ *c-failures step out* s_0
(**is** (-, *?Y*) ∈ -) **by** (*rule R1*)
hence ([] @ [*xp*], *?Y*) ∈ *c-failures step out* s_0 **using** *A* **and** *C* **by** *simp*
moreover have {} ⊆ *?Y* **..**
ultimately have ([] @ [*xp*], {}) ∈ *c-failures step out* s_0 **by** (*rule R2*)
thus *?thesis* **..**
next
assume *p* ≠ *out* s_0 *x*
hence *xp* ∈ *?X* **using** *A* **by** *simp*
hence *insert xp ?X* = *?X* **by** (*rule insert-absorb*)
hence ([], *insert xp ?X*) ∈ *c-failures step out* s_0 **using** *B* **by** *simp*
thus *?thesis* **..**
qed
next
fix *x p xps' X' s x'*
let *?s* = *step s x'*
assume *A*: *xp* = (*x*, *p*)
assume (*xps'*, *X'*) ∈ *c-failures step out* s_0 **and**
S: *s* = *foldl step* s_0 (*map fst xps'*)
hence *B*: (*xps'* @ [(*x'*, *out s x'*)], {(*y*, *p*). *p* ≠ *out ?s y*})

$\in$ *c-failures step out* s_0
(**is** (*?xps*, *?X*) $\in$ -) **by** (*rule R1*)
show (*?xps* @ [*xp*], {}) $\in$ *c-failures step out* s_0 $\vee$ (*?xps*, *insert xp ?X*)
$\in$ *c-failures step out* s_0
proof (*cases p* = *out ?s x*)
assume *C*: *p* = *out ?s x*
have *?s* = *foldl step* s_0 (*map fst ?xps*) **using** *S* **by** *simp*
with *B* **have** (*?xps* @ [(*x*, *out ?s x*)], {(*y*, *p*). *p* $\neq$ *out* (*step ?s x*) *y*})
$\in$ *c-failures step out* s_0
(**is** (-, *?Y*) $\in$ -) **by** (*rule R1*)
hence (*?xps* @ [*xp*], *?Y*) $\in$ *c-failures step out* s_0 **using** *A* **and** *C* **by** *simp*
moreover have {} $\subseteq$ *?Y* **..**
ultimately have (*?xps* @ [*xp*], {}) $\in$ *c-failures step out* s_0 **by** (*rule R2*)
thus *?thesis* **..**
next
assume *p* $\neq$ *out ?s x*
hence *xp* $\in$ *?X* **using** *A* **by** *simp*
hence *insert xp ?X* = *?X* **by** (*rule insert-absorb*)
hence (*?xps*, *insert xp ?X*) $\in$ *c-failures step out* s_0 **using** *B* **by** *simp*
thus *?thesis* **..**
qed
next
fix *xps'* *X'* *Y*
assume
(*xps'* @ [*xp*], {}) $\in$ *c-failures step out* s_0 $\vee$
(*xps'*, *insert xp Y*) $\in$ *c-failures step out* s_0 (**is** *?A* $\vee$ *?B*) **and**
X' $\subseteq$ *Y*
show (*xps'* @ [*xp*], {}) $\in$ *c-failures step out* s_0 $\vee$ (*xps'*, *insert xp X'*)
$\in$ *c-failures step out* s_0
using ‹*?A* $\vee$ *?B*›
proof (*rule disjE*)
assume *?A*
thus *?thesis* **..**
next
assume *?B*
moreover have *insert xp X'* $\subseteq$ *insert xp Y* **using** ‹*X'* $\subseteq$ *Y*›
by (*rule insert-mono*)
ultimately have (*xps'*, *insert xp X'*) $\in$ *c-failures step out* s_0 **by** (*rule R2*)
thus *?thesis* **..**
qed
qed
qed

lemma *c-process-prop-5* [*simp*]: *process-prop-5* (*F*, {})
by (*simp add*: *process-prop-5-def*)

lemma *c-process-prop-6* [*simp*]: *process-prop-6* (*F*, {})
by (*simp add*: *process-prop-6-def*)

theorem *c-process-process*: $(c\text{-}failures\ step\ out\ s_0,\ \{\}) \in process\text{-}set$
by (*simp add*: *process-set-def*)

The continuation of this section is dedicated to the proof of a few lemmas on the properties of classical processes, particularly on the application to them of the generic functions acting on processes defined previously, and culminates in the theorem stating that classical processes are deterministic. Since they are intended to be a representation of deterministic state machines as processes, this result provides an essential confirmation of the correctness of such correspondence.

lemma *c-failures-last* [*rule-format*]:
$(xps,\ X) \in c\text{-}failures\ step\ out\ s_0 \Longrightarrow xps \neq [] \longrightarrow$
$snd\ (last\ xps) = out\ (foldl\ step\ s_0\ (butlast\ (map\ fst\ xps)))\ (last\ (map\ fst\ xps))$
by (*erule c-failures.induct*, *simp-all*)

lemma *c-failures-ref*:
$(xps,\ X) \in c\text{-}failures\ step\ out\ s_0 \Longrightarrow$
$X \subseteq \{(x,\ p).\ p \neq out\ (foldl\ step\ s_0\ (map\ fst\ xps))\ x\}$
by (*erule c-failures.induct*, *simp-all*)

lemma *c-failures-failures*: $failures\ (c\text{-}process\ step\ out\ s_0) = c\text{-}failures\ step\ out\ s_0$
by (*simp add*: *failures-def c-process-def c-process-process Abs-process-inverse*)

lemma *c-futures-failures*:
$(yps,\ Y) \in futures\ (c\text{-}process\ step\ out\ s_0)\ xps =$
$((xps\ @\ yps,\ Y) \in c\text{-}failures\ step\ out\ s_0)$
by (*simp add*: *futures-def failures-def c-process-def c-process-process Abs-process-inverse*)

lemma *c-traces*:
$xps \in traces\ (c\text{-}process\ step\ out\ s_0) = (\exists X.\ (xps,\ X) \in c\text{-}failures\ step\ out\ s_0)$
by (*simp add*: *traces-def failures-def Domain-iff c-process-def c-process-process Abs-process-inverse*)

lemma *c-refusals*:
$X \in refusals\ (c\text{-}process\ step\ out\ s_0)\ xps = ((xps,\ X) \in c\text{-}failures\ step\ out\ s_0)$
by (*simp add*: *refusals-def c-failures-failures*)

lemma *c-next-events*:
$xp \in next\text{-}events\ (c\text{-}process\ step\ out\ s_0)\ xps =$
$(\exists X.\ (xps\ @\ [xp],\ X) \in c\text{-}failures\ step\ out\ s_0)$
by (*simp add*: *next-events-def c-traces*)

lemma *c-traces-failures*:
$xps \in traces\ (c\text{-}process\ step\ out\ s_0) \Longrightarrow$
$(xps,\ \{(x,\ p).\ p \neq out\ (foldl\ step\ s_0\ (map\ fst\ xps))\ x\}) \in c\text{-}failures\ step\ out\ s_0$
proof (*simp add*: *c-traces*, *erule exE*, *rule rev-cases* [*of xps*],

simp-all add: *R0 split-paired-all*)
fix *yps y p Y*
assume *A*: $(yps\ @\ [(y,\ p)],\ Y) \in$ *c-failures step out* s_0
let $?s = foldl\ step\ s_0\ (map\ fst\ yps)$
let $?ys' = map\ fst\ (yps\ @\ [(y,\ p)])$
have $(yps\ @\ [(y,\ p)],\ Y) \in failures\ (c\text{-}process\ step\ out\ s_0)$
using *A* **by** (*simp add*: *c-failures-failures*)
hence $(yps,\ \{\}) \in failures\ (c\text{-}process\ step\ out\ s_0)$ **by** (*rule process-rule-2*)
hence $(yps,\ \{\}) \in c\text{-}failures\ step\ out\ s_0$ **by** (*simp add*: *c-failures-failures*)
moreover have $?s = foldl\ step\ s_0\ (map\ fst\ yps)$ **by** *simp*
ultimately have $(yps\ @\ [(y,\ out\ ?s\ y)],\ \{(x,\ p).\ p \neq out\ (step\ ?s\ y)\ x\})$
$\in c\text{-}failures\ step\ out\ s_0$
by (*rule R1*)
moreover have $yps\ @\ [(y,\ p)] \neq []$ **by** *simp*
with *A* **have** $snd\ (last\ (yps\ @\ [(y,\ p)])) =$
$out\ (foldl\ step\ s_0\ (butlast\ ?ys'))\ (last\ ?ys')$
by (*rule c-failures-last*)
hence $p = out\ ?s\ y$ **by** *simp*
ultimately show $(yps\ @\ [(y,\ p)],\ \{(x,\ p).\ p \neq out\ (step\ ?s\ y)\ x\})$
$\in c\text{-}failures\ step\ out\ s_0$
by *simp*
qed

theorem *c-process-deterministic*: *deterministic* (*c-process step out* s_0)
proof (*simp add*: *deterministic-def c-refusals c-next-events set-eq-iff*, *rule ballI*, *rule allI*)
fix *xps X*
assume *T*: $xps \in traces\ (c\text{-}process\ step\ out\ s_0)$
let $?s = foldl\ step\ s_0\ (map\ fst\ xps)$
show $(xps,\ X) \in c\text{-}failures\ step\ out\ s_0 =$
$(\forall x\ p.\ (x,\ p) \in X \longrightarrow (\forall X.\ (xps\ @\ [(x,\ p)],\ X) \notin c\text{-}failures\ step\ out\ s_0))$
(**is** $?P = ?Q$)
proof (*rule iffI*, (*rule allI*)+, *rule impI*, *rule allI*, *rule notI*)
fix *x p Y*
let $?xs' = map\ fst\ (xps\ @\ [(x,\ p)])$
assume *?P*
hence $X \subseteq \{(x,\ p).\ p \neq out\ ?s\ x\}$ (**is** - $\subseteq ?X'$) **by** (*rule c-failures-ref*)
moreover assume $(x,\ p) \in X$
ultimately have $(x,\ p) \in ?X'$ **..**
hence *A*: $p \neq out\ ?s\ x$ **by** *simp*
assume $(xps\ @\ [(x,\ p)],\ Y) \in c\text{-}failures\ step\ out\ s_0$
moreover have $xps\ @\ [(x,\ p)] \neq []$ **by** *simp*
ultimately have $snd\ (last\ (xps\ @\ [(x,\ p)])) =$
$out\ (foldl\ step\ s_0\ (butlast\ ?xs'))\ (last\ ?xs')$
by (*rule c-failures-last*)
hence $p = out\ ?s\ x$ **by** *simp*
thus *False* **using** *A* **by** *contradiction*
next
assume *?Q*

have *A*: (*xps*, {(*x*, *p*). $p \neq$ *out ?s x*}) $\in$ *c-failures step out* s_0
using *T* **by** (*rule c-traces-failures*)
moreover have $X \subseteq$ {(*x*, *p*). $p \neq$ *out ?s x*}
proof (*rule subsetI*, *simp add*: *split-paired-all*, *rule notI*)
fix *x p*
assume (*x*, *p*) $\in X$ **and** $p =$ *out ?s x*
hence (*xps* @ [(*x*, *out ?s x*)], {(*y*, *p*). $p \neq$ *out* (*step ?s x*) *y*})
$\notin$ *c-failures step out* s_0
using ‹*?Q*› **by** *simp*
moreover have *?s* = *foldl step* s_0 (*map fst xps*) **by** *simp*
with *A* **have** (*xps* @ [(*x*, *out ?s x*)], {(*y*, *p*). $p \neq$ *out* (*step ?s x*) *y*})
$\in$ *c-failures step out* s_0
by (*rule R1*)
ultimately show *False* **by** *contradiction*
qed
ultimately show *?P* **by** (*rule R2*)
qed
qed

2.3 Traces in classical processes

Here below is the definition of function *c-tr*, where *c-tr step out s xs* is the trace of classical process *c-process step out s* corresponding to the trace *xs* of the associated deterministic state machine. Moreover, some useful lemmas are proven about this function.

function *c-tr* :: ('*s* ⇒ '*a* ⇒ '*s*) ⇒ ('*s* ⇒ '*a* ⇒ '*o*) ⇒ '*s* ⇒ '*a list* ⇒ ('*a* × '*o*) *list*
where
c-tr - - - [] = [] |
c-tr step out s (*xs* @ [*x*]) = *c-tr step out s xs* @ [(*x*, *out* (*foldl step s xs*) *x*)]
proof (*atomize-elim*, *simp-all add*: *split-paired-all*)
qed (*rule rev-cases*, *rule disjI1*, *assumption*, *simp*)
termination by *lexicographic-order*

lemma *c-tr-length*: *length* (*c-tr step out s xs*) = *length xs*
by (*rule rev-induct*, *simp-all*)

lemma *c-tr-map*: *map fst* (*c-tr step out s xs*) = *xs*
by (*rule rev-induct*, *simp-all*)

lemma *c-tr-singleton*: *c-tr step out s* [*x*] = [(*x*, *out s x*)]
proof −
have *c-tr step out s* [*x*] = *c-tr step out s* ([] @ [*x*]) **by** *simp*
also have ... = *c-tr step out s* [] @ [(*x*, *out* (*foldl step s* []) *x*)]
by (*rule c-tr.simps*(*2*))
also have ... = [(*x*, *out s x*)] **by** *simp*
finally show *?thesis* **.**
qed

lemma *c-tr-append*:
c-tr step out s (*xs* @ *ys*) = *c-tr step out s xs* @ *c-tr step out* (*foldl step s xs*) *ys*
proof (*rule-tac xs* = *ys* **in** *rev-induct*, *simp*, *subst append-assoc* [*symmetric*])
qed (*simp del*: *append-assoc*)

lemma *c-tr-hd-tl*:
assumes *A*: *xs* ≠ []
shows *c-tr step out s xs* =
(*hd xs*, *out s* (*hd xs*)) # *c-tr step out* (*step s* (*hd xs*)) (*tl xs*)
proof −
let *?s* = *foldl step s* [*hd xs*]
have *c-tr step out s* ([*hd xs*] @ *tl xs*) =
c-tr step out s [*hd xs*] @ *c-tr step out ?s* (*tl xs*)
by (*rule c-tr-append*)
moreover have [*hd xs*] @ *tl xs* = *xs* **using** *A* **by** *simp*
ultimately have *c-tr step out s xs* =
c-tr step out s [*hd xs*] @ *c-tr step out ?s* (*tl xs*)
by *simp*
moreover have *c-tr step out s* [*hd xs*] = [(*hd xs*, *out s* (*hd xs*))]
by (*simp add*: *c-tr-singleton*)
ultimately show *?thesis* **by** *simp*
qed

lemma *c-failures-tr*:
(*xps*, *X*) ∈ *c-failures step out* s_0 ⟹ *xps* = *c-tr step out* s_0 (*map fst xps*)
by (*erule c-failures.induct*, *simp-all*)

lemma *c-futures-tr*:
assumes *A*: (*yps*, *Y*) ∈ *futures* (*c-process step out* s_0) *xps*
shows *yps* = *c-tr step out* (*foldl step* s_0 (*map fst xps*)) (*map fst yps*)
proof −
have *B*: (*xps* @ *yps*, *Y*) ∈ *c-failures step out* s_0
using *A* **by** (*simp add*: *c-futures-failures*)
hence *xps* @ *yps* = *c-tr step out* s_0 (*map fst* (*xps* @ *yps*))
by (*rule c-failures-tr*)
hence *xps* @ *yps* = *c-tr step out* s_0 (*map fst xps*) @
c-tr step out (*foldl step* s_0 (*map fst xps*)) (*map fst yps*)
by (*simp add*: *c-tr-append*)
moreover have (*xps* @ *yps*, *Y*) ∈ *failures* (*c-process step out* s_0)
using *B* **by** (*simp add*: *c-failures-failures*)
hence (*xps*, {}) ∈ *failures* (*c-process step out* s_0)
by (*rule process-rule-2-failures*)
hence (*xps*, {}) ∈ *c-failures step out* s_0 **by** (*simp add*: *c-failures-failures*)
hence *xps* = *c-tr step out* s_0 (*map fst xps*) **by** (*rule c-failures-tr*)
ultimately show *?thesis* **by** *simp*
qed

lemma *c-tr-failures*:

($c\text{-}tr\ step\ out\ s_0\ xs$, $\{(x,\ p).\ p \neq out\ (foldl\ step\ s_0\ xs)\ x\}$)
$\in$ *c-failures step out* s_0
proof (*rule rev-induct*, *simp-all*, *rule R0*)
fix *x xs*
let *?s* = *foldl step* s_0 (*map fst* (*c-tr step out* s_0 *xs*))
assume (*c-tr step out* s_0 *xs*, $\{(x,\ p).\ p \neq out\ (foldl\ step\ s_0\ xs)\ x\}$)
$\in$ *c-failures step out* s_0
moreover have *?s* = *foldl step* s_0 (*map fst* (*c-tr step out* s_0 *xs*)) **by** *simp*
ultimately have (*c-tr step out* s_0 *xs* @ [(*x*, *out ?s x*)],
$\{(y,\ p).\ p \neq out\ (step\ ?s\ x)\ y\}$) $\in$ *c-failures step out* s_0
by (*rule R1*)
moreover have *?s* = *foldl step* s_0 *xs* **by** (*simp add*: *c-tr-map*)
ultimately show (*c-tr step out* s_0 *xs* @ [(*x*, *out* (*foldl step* s_0 *xs*) *x*)],
$\{(y,\ p).\ p \neq out\ (step\ (foldl\ step\ s_0\ xs)\ x)\ y\}$) $\in$ *c-failures step out* s_0 **by** *simp*
qed

lemma *c-tr-futures*:
(*c-tr step out* (*foldl step* s_0 *xs*) *ys*,
$\{(x,\ p).\ p \neq out\ (foldl\ step\ (foldl\ step\ s_0\ xs)\ ys)\ x\}$)
$\in$ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *xs*)
proof (*simp add*: *c-futures-failures*)
have (*c-tr step out* s_0 (*xs* @ *ys*), $\{(x,\ p).\ p \neq out\ (foldl\ step\ s_0\ (xs\ @\ ys))\ x\}$)
$\in$ *c-failures step out* s_0
by (*rule c-tr-failures*)
moreover have *c-tr step out* s_0 (*xs* @ *ys*) =
c-tr step out s_0 *xs* @ *c-tr step out* (*foldl step* s_0 *xs*) *ys*
by (*rule c-tr-append*)
ultimately show (*c-tr step out* s_0 *xs* @ *c-tr step out* (*foldl step* s_0 *xs*) *ys*,
$\{(x,\ p).\ p \neq out\ (foldl\ step\ (foldl\ step\ s_0\ xs)\ ys)\ x\}$)
$\in$ *c-failures step out* s_0
by *simp*
qed

2.4 Noninterference in classical processes

Given a mapping D of the actions of a deterministic state machine into their security domains, it is natural to map each event $(x,\ p)$ of the corresponding classical process into the domain $D\ x$ of action x.

Such mapping of events into domains, formalized as function *c-dom* D in the continuation, ensures that the same noninterference policy applying to a deterministic state machine be applicable to the associated classical process as well. This is the simplest, and thus preferable way to construct a policy for the process such as to be isomorphic to the one assigned for the machine, as required in order to prove the equivalence of CSP noninterference security to the classical notion in the case of classical processes.

In what follows, function *c-dom* will be used in the proof of some useful lemmas concerning the application of functions *sinks*, *ipurge-tr*, *c-sources*, *c-ipurge* from noninterference theory to the traces of classical processes, constructed by means of function *c-tr*.

definition *c-dom* :: $('a \Rightarrow 'd) \Rightarrow ('a \times 'o) \Rightarrow 'd$ **where**
c-dom D xp ≡ *D* (*fst xp*)

lemma *c-dom-sources*:
c-sources I (*c-dom D*) *u xps* = *c-sources I D u* (*map fst xps*)
by (*induction xps*, *simp-all add*: *c-dom-def*)

lemma *c-dom-sinks*: *sinks I* (*c-dom D*) *u xps* = *sinks I D u* (*map fst xps*)
by (*induction xps rule*: *rev-induct*, *simp-all add*: *c-dom-def*)

lemma *c-tr-sources*:
c-sources I (*c-dom D*) *u* (*c-tr step out s xs*) = *c-sources I D u xs*
by (*simp add*: *c-dom-sources c-tr-map*)

lemma *c-tr-sinks*: *sinks I* (*c-dom D*) *u* (*c-tr step out s xs*) = *sinks I D u xs*
by (*simp add*: *c-dom-sinks c-tr-map*)

lemma *c-tr-ipurge*:
c-ipurge I (*c-dom D*) *u* (*c-tr step out s* (*c-ipurge I D u xs*)) =
c-tr step out s (*c-ipurge I D u xs*)
proof (*induction xs arbitrary*: *s*, *simp*)
fix *x xs s*
assume *A*: ⋀*s*. *c-ipurge I* (*c-dom D*) *u* (*c-tr step out s* (*c-ipurge I D u xs*)) =
c-tr step out s (*c-ipurge I D u xs*)
show *c-ipurge I* (*c-dom D*) *u* (*c-tr step out s* (*c-ipurge I D u* (*x* # *xs*))) =
c-tr step out s (*c-ipurge I D u* (*x* # *xs*))
proof (*cases D x* ∈ *c-sources I D u* (*x* # *xs*), *simp-all del*: *c-sources.simps*)
have *B*: *c-tr step out s* (*x* # *c-ipurge I D u xs*) =
(*x*, *out s x*) # *c-tr step out* (*step s x*) (*c-ipurge I D u xs*)
by (*simp add*: *c-tr-hd-tl*)
assume *C*: *D x* ∈ *c-sources I D u* (*x* # *xs*)
hence *D x* ∈ *c-sources I D u* (*c-ipurge I D u* (*x* # *xs*))
by (*subst c-sources-ipurge*)
hence *D x* ∈ *c-sources I* (*c-dom D*) *u* (*c-tr step out s* (*x* # *c-ipurge I D u xs*))
using *C* **by** (*simp add*: *c-tr-sources*)
hence *c-dom D* (*x*, *out s x*) ∈ *c-sources I* (*c-dom D*) *u*
((*x*, *out s x*) # *c-tr step out* (*step s x*) (*c-ipurge I D u xs*))
using *B* **by** (*simp add*: *c-dom-def*)
hence *c-ipurge I* (*c-dom D*) *u* (*c-tr step out s* (*x* # *c-ipurge I D u xs*)) =
(*x*, *out s x*) # *c-ipurge I* (*c-dom D*) *u*
(*c-tr step out* (*step s x*) (*c-ipurge I D u xs*))
using *B* **by** *simp*
moreover have *c-ipurge I* (*c-dom D*) *u*
(*c-tr step out* (*step s x*) (*c-ipurge I D u xs*)) =
c-tr step out (*step s x*) (*c-ipurge I D u xs*)
using *A* **.**
ultimately show *c-ipurge I* (*c-dom D*) *u*

(*c-tr step out s* (*x* # *c-ipurge I D u xs*)) =
c-tr step out s (*x* # *c-ipurge I D u xs*)
using *B* **by** *simp*
next
show *c-ipurge I* (*c-dom D*) *u* (*c-tr step out s* (*c-ipurge I D u xs*)) =
c-tr step out s (*c-ipurge I D u xs*)
using *A* **.**
qed
qed

lemma *c-tr-ipurge-tr-1* [*rule-format*]:
(∀ *n* ∈ {..<*length xs*}. *D* (*xs* ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) *xs*) ⟶
out (*foldl step s* (*ipurge-tr I D u* (*take n xs*))) (*xs* ! *n*) =
out (*foldl step s* (*take n xs*)) (*xs* ! *n*)) ⟶
ipurge-tr I (*c-dom D*) *u* (*c-tr step out s xs*) = *c-tr step out s* (*ipurge-tr I D u xs*)
proof (*induction xs rule*: *rev-induct*, *simp*, *rule impI*)
fix *x xs*
assume (∀ *n* ∈ {..<*length xs*}.
D (*xs* ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) *xs*) ⟶
out (*foldl step s* (*ipurge-tr I D u* (*take n xs*))) (*xs* ! *n*) =
out (*foldl step s* (*take n xs*)) (*xs* ! *n*)) ⟶
ipurge-tr I (*c-dom D*) *u* (*c-tr step out s xs*) =
c-tr step out s (*ipurge-tr I D u xs*)
moreover assume *A*: ∀ *n* ∈ {..<*length* (*xs* @ [*x*])}.
D ((*xs* @ [*x*]) ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) (*xs* @ [*x*])) ⟶
out (*foldl step s* (*ipurge-tr I D u* (*take n* (*xs* @ [*x*])))) ((*xs* @ [*x*]) ! *n*) =
out (*foldl step s* (*take n* (*xs* @ [*x*]))) ((*xs* @ [*x*]) ! *n*)
have ∀ *n* ∈ {..<*length xs*}.
D (*xs* ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) *xs*) ⟶
out (*foldl step s* (*ipurge-tr I D u* (*take n xs*))) (*xs* ! *n*) =
out (*foldl step s* (*take n xs*)) (*xs* ! *n*)
proof (*rule ballI*, *rule impI*)
fix *n*
assume *B*: *n* ∈ {..<*length xs*}
hence *n* ∈ {..<*length* (*xs* @ [*x*])} **by** *simp*
with *A* **have** *D* ((*xs* @ [*x*]) ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) (*xs* @ [*x*])) ⟶
out (*foldl step s* (*ipurge-tr I D u* (*take n* (*xs* @ [*x*])))) ((*xs* @ [*x*]) ! *n*) =
out (*foldl step s* (*take n* (*xs* @ [*x*]))) ((*xs* @ [*x*]) ! *n*) **..**
hence *D* (*xs* ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) *xs*) ⟶
out (*foldl step s* (*ipurge-tr I D u* (*take n xs*))) (*xs* ! *n*) =
out (*foldl step s* (*take n xs*)) (*xs* ! *n*)
using *B* **by** (*simp add*: *nth-append*)
moreover assume *D* (*xs* ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) *xs*)
ultimately show *out* (*foldl step s* (*ipurge-tr I D u* (*take n xs*))) (*xs* ! *n*) =
out (*foldl step s* (*take n xs*)) (*xs* ! *n*) **..**
qed
ultimately have *C*: *ipurge-tr I* (*c-dom D*) *u* (*c-tr step out s xs*) =
c-tr step out s (*ipurge-tr I D u xs*) **..**
show *ipurge-tr I* (*c-dom D*) *u* (*c-tr step out s* (*xs* @ [*x*])) =

c-tr step out s (*ipurge-tr I D u* (*xs* @ [*x*]))
proof (*cases D x* ∈ *sinks I D u* (*xs* @ [*x*]))
case *True*
then have *D x* ∈ *sinks I* (*c-dom D*) *u*
(*c-tr step out s* (*xs* @ [*x*]))
by (*subst c-tr-sinks*)
hence *c-dom D* (*x*, *out* (*foldl step s xs*) *x*)
∈ *sinks I* (*c-dom D*) *u* (*c-tr step out s xs* @ [(*x*, *out* (*foldl step s xs*) *x*)])
by (*simp add*: *c-dom-def*)
with *True* **show** *?thesis* **using** *C* **by** *simp*
next
case *False*
then have *D x* ∉ *sinks I* (*c-dom D*) *u*
(*c-tr step out s* (*xs* @ [*x*]))
by (*subst c-tr-sinks*)
hence *c-dom D* (*x*, *out* (*foldl step s xs*) *x*)
∉ *sinks I* (*c-dom D*) *u* (*c-tr step out s xs* @ [(*x*, *out* (*foldl step s xs*) *x*)])
by (*simp add*: *c-dom-def*)
with *False* **show** *?thesis*
proof (*simp add*: *C*)
have *length xs* ∈ {..<*length* (*xs* @ [*x*])} **by** *simp*
with *A* **have** *D* ((*xs* @ [*x*]) ! *length xs*)
∉ *sinks I D u* (*take* (*Suc* (*length xs*)) (*xs* @ [*x*])) ⟶
out (*foldl step s* (*ipurge-tr I D u* (*take* (*length xs*) (*xs* @ [*x*]))))
((*xs* @ [*x*]) ! (*length xs*)) =
out (*foldl step s* (*take* (*length xs*) (*xs* @ [*x*]))) ((*xs* @ [*x*]) ! (*length xs*)) **..**
hence *D x* ∉ *sinks I D u* (*xs* @ [*x*]) ⟶
out (*foldl step s* (*ipurge-tr I D u xs*)) *x* = *out* (*foldl step s xs*) *x*
by *simp*
thus *out* (*foldl step s xs*) *x* = *out* (*foldl step s* (*ipurge-tr I D u xs*)) *x*
using *False* **by** *simp*
qed
qed
qed

lemma *c-tr-ipurge-tr-2* [*rule-format*]:
assumes *A*: ∀ *n* ∈ {..*length ys*}. ∃ *Y*.
(*ipurge-tr I* (*c-dom D*) *u* (*c-tr step out* (*foldl step* s_0 *xs*) (*take n ys*)), *Y*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *xs*)
shows *n* ∈ {..<*length ys*} ⟶ *D* (*ys* ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) *ys*) ⟶
out (*foldl step* (*foldl step* s_0 *xs*) (*ipurge-tr I D u* (*take n ys*))) (*ys* ! *n*) =
out (*foldl step* (*foldl step* s_0 *xs*) (*take n ys*)) (*ys* ! *n*)
proof (*rule nat-less-induct*, (*rule impI*)+)
fix *n*
let *?s* = *foldl step* s_0 *xs*
let *?yp* = (*ys* ! *n*, *out* (*foldl step ?s* (*take n ys*)) (*ys* ! *n*))
assume
B: ∀ *m* < *n*. *m* ∈ {..<*length ys*} ⟶
D (*ys* ! *m*) ∉ *sinks I D u* (*take* (*Suc m*) *ys*) ⟶

out (foldl step ?s (ipurge-tr I D u (take m ys))) (ys ! m) =
out (foldl step ?s (take m ys)) (ys ! m) **and**
C: n ∈ {..<length ys} **and**
D: D (ys ! n) ∉ sinks I D u (take (Suc n) ys)
have n < length ys **using** C **by** simp
hence E: take (Suc n) ys = take n ys @ [ys ! n]
by (rule take-Suc-conv-app-nth)
moreover have Suc n ∈ {..length ys} **using** C **by** simp
with A **have** ∃ Y.
(ipurge-tr I (c-dom D) u (c-tr step out ?s (take (Suc n) ys)), Y)
∈ futures (c-process step out s_0) (c-tr step out s_0 xs) **..**
then obtain Y **where**
(ipurge-tr I (c-dom D) u (c-tr step out ?s (take (Suc n) ys)), Y)
∈ futures (c-process step out s_0) (c-tr step out s_0 xs) **..**
ultimately have
(ipurge-tr I (c-dom D) u (c-tr step out ?s (take n ys) @ [?yp]), Y)
∈ futures (c-process step out s_0) (c-tr step out s_0 xs)
by simp
moreover have c-dom D ?yp
∉ sinks I (c-dom D) u (c-tr step out ?s (take (Suc n) ys))
using D **by** (simp add: c-dom-def c-tr-sinks)
hence c-dom D ?yp ∉ sinks I (c-dom D) u
(c-tr step out ?s (take n ys) @ [?yp])
using E **by** simp
ultimately have
(ipurge-tr I (c-dom D) u (c-tr step out ?s (take n ys)) @ [?yp], Y)
∈ futures (c-process step out s_0) (c-tr step out s_0 xs)
by simp
moreover have ipurge-tr I (c-dom D) u (c-tr step out ?s (take n ys)) =
c-tr step out ?s (ipurge-tr I D u (take n ys))
proof (rule c-tr-ipurge-tr-1, simp, erule conjE)
fix m
have m < n ⟶ m ∈ {..<length ys} ⟶
D (ys ! m) ∉ sinks I D u (take (Suc m) ys) ⟶
out (foldl step ?s (ipurge-tr I D u (take m ys))) (ys ! m) =
out (foldl step ?s (take m ys)) (ys ! m) **using** B **..**
moreover assume m < n
ultimately have m ∈ {..<length ys} ⟶
D (ys ! m) ∉ sinks I D u (take (Suc m) ys) ⟶
out (foldl step ?s (ipurge-tr I D u (take m ys))) (ys ! m) =
out (foldl step ?s (take m ys)) (ys ! m) **..**
moreover assume m < length ys
hence m ∈ {..<length ys} **by** simp
ultimately have D (ys ! m) ∉ sinks I D u (take (Suc m) ys) ⟶
out (foldl step ?s (ipurge-tr I D u (take m ys))) (ys ! m) =
out (foldl step ?s (take m ys)) (ys ! m) **..**
moreover assume D (ys ! m) ∉ sinks I D u (take (Suc m) ys)
ultimately show out (foldl step ?s (ipurge-tr I D u (take m ys))) (ys ! m) =
out (foldl step ?s (take m ys)) (ys ! m) **..**

qed
ultimately have (*c-tr step out ?s* (*ipurge-tr I D u* (*take n ys*)) @ [*?yp*], *Y*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *xs*)
by *simp*
hence (*c-tr step out* s_0 (*xs* @ *ipurge-tr I D u* (*take n ys*)) @ [*?yp*], *Y*)
∈ *c-failures step out* s_0
(**is** (*?xps*, -) ∈ -) **by** (*simp add*: *c-futures-failures c-tr-append*)
moreover have *?xps* ≠ [] **by** *simp*
ultimately have *snd* (*last ?xps*) =
out (*foldl step* s_0 (*butlast* (*map fst ?xps*))) (*last* (*map fst ?xps*))
by (*rule c-failures-last*)
thus *out* (*foldl step ?s* (*ipurge-tr I D u* (*take n ys*))) (*ys* ! *n*) =
out (*foldl step ?s* (*take n ys*)) (*ys* ! *n*)
by (*simp add*: *c-tr-map butlast-append*)
qed

lemma *c-tr-ipurge-tr* [*rule-format*]:
assumes *A*: ∀ *n* ∈ {..*length ys*}. ∃ *Y*.
(*ipurge-tr I* (*c-dom D*) *u* (*c-tr step out* (*foldl step* s_0 *xs*) (*take n ys*)), *Y*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *xs*)
shows *ipurge-tr I* (*c-dom D*) *u* (*c-tr step out* (*foldl step* s_0 *xs*) *ys*) =
c-tr step out (*foldl step* s_0 *xs*) (*ipurge-tr I D u ys*)
proof (*rule c-tr-ipurge-tr-1*)
fix *n*
have ⋀*n*. *n* ∈ {..*length ys*} ⟹ ∃ *Y*.
(*ipurge-tr I* (*c-dom D*) *u* (*c-tr step out* (*foldl step* s_0 *xs*) (*take n ys*)), *Y*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *xs*)
using *A* **..**
moreover assume
n ∈ {..<*length ys*} **and**
D (*ys* ! *n*) ∉ *sinks I D u* (*take* (*Suc n*) *ys*)
ultimately show
out (*foldl step* (*foldl step* s_0 *xs*) (*ipurge-tr I D u* (*take n ys*))) (*ys* ! *n*) =
out (*foldl step* (*foldl step* s_0 *xs*) (*take n ys*)) (*ys* ! *n*)
by (*rule c-tr-ipurge-tr-2*)
qed

2.5 Equivalence between security properties

The remainder of this section is dedicated to the proof of the equivalence between the CSP noninterference security of a classical process and the classical noninterference security of the corresponding deterministic state machine.

In some detail, it will be proven that CSP noninterference security alone is a sufficient condition for classical noninterference security, whereas the latter security property entails the former for any reflexive noninterference policy. Therefore, the security properties under consideration turn out to be equivalent if the enforced noninterference policy is reflexive, which is the case for any policy of practical significance.

```
lemma secure-implies-c-secure-aux:
  assumes S: secure (c-process step out s0) I (c-dom D)
  shows out (foldl step (foldl step s0 xs) ys) x =
    out (foldl step (foldl step s0 xs) (c-ipurge I D (D x) ys)) x
proof (induction ys arbitrary: xs, simp)
  fix y ys xs
  assume ⋀xs. out (foldl step (foldl step s0 xs) ys) x =
    out (foldl step (foldl step s0 xs) (c-ipurge I D (D x) ys)) x
  hence A: out (foldl step (foldl step s0 (xs @ [y])) ys) x =
    out (foldl step (foldl step s0 (xs @ [y])) (c-ipurge I D (D x) ys)) x .
  show out (foldl step (foldl step s0 xs) (y # ys)) x =
    out (foldl step (foldl step s0 xs) (c-ipurge I D (D x) (y # ys))) x
  proof (cases D y ∈ c-sources I D (D x) (y # ys))
    assume D y ∈ c-sources I D (D x) (y # ys)
    thus ?thesis using A by simp
  next
    let ?s = foldl step s0 xs
    let ?yp = (y, out ?s y)
    have (c-tr step out ?s [y], {(x′, p). p ≠ out (foldl step ?s [y]) x′})
      ∈ futures (c-process step out s0) (c-tr step out s0 xs) (is (-, ?Y) ∈ -)
     by (rule c-tr-futures)
    hence ([?yp], ?Y) ∈ futures (c-process step out s0) (c-tr step out s0 xs)
     by (simp add: c-tr-hd-tl)
    moreover have (c-tr step out ?s (c-ipurge I D (D x) (ys @ [x])),
      {(x′, p). p ≠ out (foldl step ?s (c-ipurge I D (D x) (ys @ [x]))) x′})
      ∈ futures (c-process step out s0) (c-tr step out s0 xs) (is (-, ?Z) ∈ -)
     by (rule c-tr-futures)
    ultimately have (?yp # ipurge-tr I (c-dom D) (c-dom D ?yp)
      (c-tr step out ?s (c-ipurge I D (D x) (ys @ [x]))),
      ipurge-ref I (c-dom D) (c-dom D ?yp)
      (c-tr step out ?s (c-ipurge I D (D x) (ys @ [x]))) ?Z)
      ∈ futures (c-process step out s0) (c-tr step out s0 xs)
     (is (-, ?X) ∈ -) using S by (simp add: secure-def)
    hence C: (?yp # ipurge-tr I (c-dom D) (c-dom D ?yp)
      (c-ipurge I (c-dom D) (D x)
      (c-tr step out ?s (c-ipurge I D (D x) (ys @ [x])))), ?X)
      ∈ futures (c-process step out s0) (c-tr step out s0 xs)
     by (simp add: c-tr-ipurge)
    assume D: D y ∉ c-sources I D (D x) (y # ys)
    hence D y ∉ c-sources I D (D x) ((y # ys) @ [x])
     by (subst c-sources-append-1)
    hence D y ∉ c-sources I D (D x) (y # ys @ [x]) by simp
    moreover have c-sources I D (D x) (y # ys @ [x]) =
      c-sources I D (D x) (y # c-ipurge I D (D x) (ys @ [x]))
     by (simp add: c-sources-ipurge)
    ultimately have D y ∉ c-sources I D (D x)
      (y # c-ipurge I D (D x) (ys @ [x]))
```

```
      by simp
    moreover have map fst (?yp # c-tr step out ?s
      (c-ipurge I D (D x) (ys @ [x]))) =
      y # c-ipurge I D (D x) (ys @ [x])
     by (simp add: c-tr-map)
    hence c-sources I D (D x) (y # c-ipurge I D (D x) (ys @ [x])) =
      c-sources I (c-dom D) (D x)
      (?yp # c-tr step out ?s (c-ipurge I D (D x) (ys @ [x])))
     by (subst c-dom-sources, simp)
    ultimately have c-dom D ?yp ∉ c-sources I (c-dom D) (D x)
      (?yp # c-tr step out ?s (c-ipurge I D (D x) (ys @ [x])))
     by (simp add: c-dom-def)
    hence ipurge-tr I (c-dom D) (c-dom D ?yp) (c-ipurge I (c-dom D) (D x)
      (c-tr step out ?s (c-ipurge I D (D x) (ys @ [x])))) =
      c-ipurge I (c-dom D) (D x) (c-tr step out ?s (c-ipurge I D (D x) (ys @ [x])))
     by (rule c-ipurge-tr-ipurge)
    hence (?yp # c-tr step out ?s (c-ipurge I D (D x) (ys @ [x])), ?X)
      ∈ futures (c-process step out s0) (c-tr step out s0 xs)
     using C by (simp add: c-tr-ipurge)
    hence (c-tr step out s0 xs @ ?yp #
      c-tr step out ?s (c-ipurge I D (D x) ys @ [x]), ?X)
      ∈ c-failures step out s0
     (is (?xps, -) ∈ -) by (simp add: c-futures-failures c-ipurge-append-1)
    moreover have ?xps ≠ [] by simp
    ultimately have snd (last ?xps) =
      out (foldl step s0 (butlast (map fst ?xps))) (last (map fst ?xps))
     by (rule c-failures-last)
    hence snd (last ?xps) =
      out (foldl step (foldl step s0 (xs @ [y])) (c-ipurge I D (D x) ys)) x
     by (simp add: c-tr-map butlast-append)
    moreover have snd (last ?xps) =
      out (foldl step (foldl step s0 xs) (c-ipurge I D (D x) (y # ys))) x
     using D by simp
    ultimately show ?thesis using A by simp
  qed
qed

theorem secure-implies-c-secure:
  assumes S: secure (c-process step out s0) I (c-dom D)
  shows c-secure step out s0 I D
proof (simp add: c-secure-def, (rule allI)+)
  fix x xs
  have out (foldl step (foldl step s0 []) xs) x =
    out (foldl step (foldl step s0 []) (c-ipurge I D (D x) xs)) x
   using S by (rule secure-implies-c-secure-aux)
  thus out (foldl step s0 xs) x = out (foldl step s0 (c-ipurge I D (D x) xs)) x
   by simp
qed
```

lemma *c-secure-futures-1*:
  assumes *R*: *refl I* **and** *S*: *c-secure step out* s_0 *I D*
  shows (*yps* @ [*yp*], *Y*) ∈ *futures* (*c-process step out* s_0) *xps* ⟹
    (*yps*, {*x* ∈ *Y*. (*c-dom D yp*, *c-dom D x*) ∉ *I*})
    ∈ *futures* (*c-process step out* s_0) *xps*
proof (*simp add*: *c-futures-failures*)
  let *?zs* = *map fst* (*xps* @ *yps*)
  let *?y* = *fst yp*
  assume *A*: (*xps* @ *yps* @ [*yp*], *Y*) ∈ *c-failures step out* s_0
  hence ((*xps* @ *yps*) @ [*yp*], *Y*) ∈ *failures* (*c-process step out* s_0)
   by (*simp add*: *c-failures-failures*)
  hence (*xps* @ *yps*, {}) ∈ *failures* (*c-process step out* s_0)
   by (*rule process-rule-2-failures*)
  hence (*xps* @ *yps*, {}) ∈ *c-failures step out* s_0 **by** (*simp add*: *c-failures-failures*)
  hence *B*: *xps* @ *yps* = *c-tr step out* s_0 *?zs* **by** (*rule c-failures-tr*)
  have *Y* ⊆ {(*x*, *p*). *p* ≠ *out* (*foldl step* s_0 (*map fst* (*xps* @ *yps* @ [*yp*]))) *x*}
   using *A* **by** (*rule c-failures-ref*)
  hence *C*: *Y* ⊆ {(*x*, *p*). *p* ≠ *out* (*foldl step* s_0 (*?zs* @ [*?y*])) *x*}
   (**is** - ⊆ *?Y′*) **by** *simp*
  have (*xps* @ *yps*, {(*x*, *p*). *p* ≠ *out* (*foldl step* s_0 *?zs*) *x*}) ∈ *c-failures step out* s_0
   (**is** (-, *?X′*) ∈ -) **by** (*subst B*, *rule c-tr-failures*)
  moreover have {*x* ∈ *Y*. (*c-dom D yp*, *c-dom D x*) ∉ *I*} ⊆ *?X′* (**is** *?X* ⊆ -)
  proof (*rule subsetI*, *simp add*: *split-paired-all c-dom-def del*: *map-append*,
   erule conjE)
    fix *x p*
    assume (*x*, *p*) ∈ *Y*
    with *C* **have** (*x*, *p*) ∈ *?Y′* **..**
    hence *p* ≠ *out* (*foldl step* s_0 (*?zs* @ [*?y*])) *x* **by** *simp*
    moreover have *out* (*foldl step* s_0 (*?zs* @ [*?y*])) *x* =
      out (*foldl step* s_0 (*c-ipurge I D* (*D x*) (*?zs* @ [*?y*]))) *x*
     using *S* **by** (*simp add*: *c-secure-def*)
    ultimately have *p* ≠ *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*) (*?zs* @ [*?y*]))) *x*
     by *simp*
    moreover assume (*D ?y*, *D x*) ∉ *I*
    with *R* **have** *c-ipurge I D* (*D x*) (*?zs* @ [*?y*]) = *c-ipurge I D* (*D x*) *?zs*
     by (*rule c-ipurge-append-2*)
    ultimately have *p* ≠ *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*) *?zs*)) *x* **by** *simp*
    moreover have *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*) *?zs*)) *x* =
      out (*foldl step* s_0 *?zs*) *x*
     using *S* **by** (*simp add*: *c-secure-def*)
    ultimately show *p* ≠ *out* (*foldl step* s_0 *?zs*) *x* **by** *simp*
  qed
  ultimately show (*xps* @ *yps*, *?X*) ∈ *c-failures step out* s_0 **by** (*rule R2*)
qed

lemma *c-secure-implies-secure-aux-1* [*rule-format*]:
  assumes
    R: *refl I* **and**
    S: *c-secure step out* s_0 *I D*

shows $(yp \# yps, Y) \in$ *futures* (*c-process step out* s_0) *xps* $\longrightarrow$
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *yps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *yps Y*)
$\in$ *futures* (*c-process step out* s_0) *xps*
proof (*induction yps arbitrary*: *Y rule*: *length-induct*, *rule impI*)
fix *yps Y*
assume
A: $\forall$ *yps'*. *length yps'* $<$ *length yps* $\longrightarrow$
($\forall$ *Y'*. (*yp* # *yps'*, *Y'*) $\in$ *futures* (*c-process step out* s_0) *xps* $\longrightarrow$
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *yps'*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *yps' Y'*)
$\in$ *futures* (*c-process step out* s_0) *xps*) **and**
B: (*yp* # *yps*, *Y*) $\in$ *futures* (*c-process step out* s_0) *xps*
show (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *yps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *yps Y*)
$\in$ *futures* (*c-process step out* s_0) *xps*
proof (*cases yps*, *simp add*: *ipurge-ref-def*)
case *Nil*
hence ([] @ [*yp*], *Y*) $\in$ *futures* (*c-process step out* s_0) *xps* **using** *B* **by** *simp*
with *R* **and** *S* **show** ([], {*x* $\in$ *Y*. (*c-dom D yp*, *c-dom D x*) $\notin$ *I*})
$\in$ *futures* (*c-process step out* s_0) *xps*
by (*rule c-secure-futures-1*)
next
case *Cons*
have $\exists$ *wps wp*. *yps* = *wps* @ [*wp*]
by (*rule rev-cases* [*of yps*], *simp-all add*: *Cons*)
then obtain *wps* **and** *wp* **where** *C*: *yps* = *wps* @ [*wp*] **by** *blast*
have *B'*: ((*yp* # *wps*) @ [*wp*], *Y*) $\in$ *futures* (*c-process step out* s_0) *xps*
using *B* **and** *C* **by** *simp*
show *?thesis*
proof (*simp only*: *C*,
cases c-dom D wp $\in$ *sinks I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]))
let *?Y'* = {*x* $\in$ *Y*. (*c-dom D wp*, *c-dom D x*) $\notin$ *I*}
have *length wps* $<$ *length yps* $\longrightarrow$
($\forall$ *Y'*. (*yp* # *wps*, *Y'*) $\in$ *futures* (*c-process step out* s_0) *xps* $\longrightarrow$
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *wps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps Y'*)
$\in$ *futures* (*c-process step out* s_0) *xps*)
using *A* **..**
moreover have *length wps* $<$ *length yps* **using** *C* **by** *simp*
ultimately have $\forall$ *Y'*.
(*yp* # *wps*, *Y'*) $\in$ *futures* (*c-process step out* s_0) *xps* $\longrightarrow$
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *wps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps Y'*)
$\in$ *futures* (*c-process step out* s_0) *xps* **..**
hence (*yp* # *wps*, *?Y'*) $\in$ *futures* (*c-process step out* s_0) *xps* $\longrightarrow$
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *wps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps ?Y'*)
$\in$ *futures* (*c-process step out* s_0) *xps* **..**

moreover have (*yp* # *wps*, *?Y'*) ∈ *futures* (*c-process step out* s_0) *xps*
using *R* **and** *S* **and** *B'* **by** (*rule c-secure-futures-1*)
ultimately have (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *wps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps ?Y'*)
∈ *futures* (*c-process step out* s_0) *xps* **..**
moreover assume
D: *c-dom D wp* ∈ *sinks I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*])
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
ipurge-tr I (*c-dom D*) (*c-dom D yp*) *wps*
by *simp*
moreover have *ipurge-ref I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) *Y* =
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps ?Y'*
using *D* **by** (*rule ipurge-ref-eq*)
ultimately show (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]),
ipurge-ref I (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) *Y*)
∈ *futures* (*c-process step out* s_0) *xps*
by *simp*
next
let *?xs* = *map fst xps*
let *?y* = *fst yp*
let *?ws* = *map fst wps*
let *?w* = *fst wp*
let *?s* = *foldl step* s_0 *?xs*
have (*xps* @ *yp* # *wps* @ [*wp*], *Y*) ∈ *failures* (*c-process step out* s_0)
using *B'* **by** (*simp add*: *c-futures-failures c-failures-failures*)
hence (*xps*, {}) ∈ *failures* (*c-process step out* s_0)
by (*rule process-rule-2-failures*)
hence (*xps*, {}) ∈ *c-failures step out* s_0
by (*simp add*: *c-failures-failures*)
hence *X*: *xps* = *c-tr step out* s_0 *?xs* **by** (*rule c-failures-tr*)
have *W*: (*yp* # *wps*, {}) ∈ *futures* (*c-process step out* s_0) *xps*
using *B'* **by** (*rule process-rule-2-futures*)
hence *yp* # *wps* = *c-tr step out ?s* (*map fst* (*yp* # *wps*))
by (*rule c-futures-tr*)
hence *W'*: *yp* # *wps* = *c-tr step out ?s* (*?y* # *?ws*) **by** *simp*
assume *D*: *c-dom D wp* ∉ *sinks I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*])
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
ipurge-tr I (*c-dom D*) (*c-dom D yp*) (*yp* # *wps*) @ [*wp*]
using *R* **by** (*simp add*: *ipurge-tr-cons-same*)
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
ipurge-tr I (*c-dom D*) (*c-dom D yp*) (*c-tr step out ?s* (*?y* # *?ws*)) @ [*wp*]
using *W'* **by** *simp*
also have ... =
c-tr step out ?s (*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*)) @ [*wp*]
proof (*simp*, *rule c-tr-ipurge-tr*)
fix *n*
show ∃ *W*. (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*)
(*c-tr step out ?s* (*take n* (*?y* # *?ws*))), *W*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *?xs*)

proof (*cases n*, *simp-all add*: *c-tr-hd-tl*)
have (*c-tr step out ?s* [], {(*x*, *p*). *p* ≠ *out* (*foldl step ?s* []) *x*})
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *?xs*)
by (*rule c-tr-futures*)
hence ([], {(*x*, *p*). *p* ≠ *out ?s x*})
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *?xs*)
by *simp*
thus ∃ *W*. ([], *W*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *?xs*) **..**
next
case (*Suc m*)
let *?wps′* = *c-tr step out* (*step ?s ?y*) (*take m ?ws*)
have *length ?wps′* < *length yps* ⟶
(∀ *Y′*. (*yp* # *?wps′*, *Y′*) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps′*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *?wps′ Y′*)
∈ *futures* (*c-process step out* s_0) *xps*)
using *A* **..**
moreover have *length ?wps′* < *length yps*
using *C* **by** (*simp add*: *c-tr-length*)
ultimately have ∀ *Y′*.
(*yp* # *?wps′*, *Y′*) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps′*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *?wps′ Y′*)
∈ *futures* (*c-process step out* s_0) *xps* **..**
hence (*yp* # *?wps′*, {}) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps′*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *?wps′* {})
∈ *futures* (*c-process step out* s_0) *xps*
(**is** - ⟶ (-, *?W′*) ∈ -) **..**
moreover have *E*: *yp* # *wps* = (*?y*, *out ?s ?y*) #
c-tr step out (*step ?s ?y*) (*take m ?ws* @ *drop m ?ws*)
using *W′* **by** (*simp add*: *c-tr-hd-tl*)
hence *F*: *yp* = (*?y*, *out ?s ?y*) **by** *simp*
hence *yp* # *wps* = *yp* # *?wps′* @
c-tr step out (*foldl step* (*step ?s ?y*) (*take m ?ws*)) (*drop m ?ws*)
using *E* **by** (*simp only*: *c-tr-append*)
hence ((*yp* # *?wps′*) @
c-tr step out (*foldl step* (*step ?s ?y*) (*take m ?ws*)) (*drop m ?ws*), {})
∈ *futures* (*c-process step out* s_0) *xps*
using *W* **by** *simp*
hence (*yp* # *?wps′*, {}) ∈ *futures* (*c-process step out* s_0) *xps*
by (*rule process-rule-2-futures*)
ultimately have (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps′*, *?W′*)
∈ *futures* (*c-process step out* s_0) *xps* **..**
moreover have *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps′* =
ipurge-tr I (*c-dom D*) (*c-dom D yp*) ((*?y*, *out ?s ?y*) # *?wps′*)
using *R* **and** *F* **by** (*simp add*: *ipurge-tr-cons-same*)
ultimately have

(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) ((*?y*, *out ?s ?y*) # *?wps'*), *?W'*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *?xs*)
using *X* **by** *simp*
thus ∃ *W*.
(*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) ((*?y*, *out ?s ?y*) # *?wps'*), *W*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *?xs*)
by (*rule-tac x* = *?W'* **in** *exI*)
qed
qed
finally have *E*: *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s (*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*)) @ [*wp*] **.**
have (*xps* @ *yp* # *wps* @ [*wp*], *Y*) ∈ *c-failures step out* s_0
(**is** (*?xps'*, -) ∈ -) **using** *B'* **by** (*simp add*: *c-futures-failures*)
moreover have *?xps'* ≠ [] **by** *simp*
ultimately have *snd* (*last ?xps'*) =
out (*foldl step* s_0 (*butlast* (*map fst ?xps'*))) (*last* (*map fst ?xps'*))
by (*rule c-failures-last*)
hence *snd wp* = *out* (*foldl step* s_0 (*?xs* @ *?y* # *?ws*)) *?w*
by (*simp add*: *butlast-append*)
hence *snd wp* =
out (*foldl step* s_0 (*c-ipurge I D* (*D ?w*) (*?xs* @ *?y* # *?ws*))) *?w*
using *S* **by** (*simp add*: *c-secure-def*)
moreover have *F*: *D ?w* ∉ *sinks I D* (*c-dom D yp*) (*?ws* @ [*?w*])
using *D* **by** (*simp only*: *c-dom-sinks*, *simp add*: *c-dom-def*)
have ¬ (∃ *v* ∈ *sinks I D* (*c-dom D yp*) (*?y* # *?ws*). (*v*, *D ?w*) ∈ *I*)
proof (*rule notI*, *simp add*: *c-dom-def sinks-cons-same R*, *erule disjE*)
assume (*D ?y*, *D ?w*) ∈ *I*
hence *D ?w* ∈ *sinks I D* (*c-dom D yp*) (*?ws* @ [*?w*])
by (*simp add*: *c-dom-def*)
thus *False* **using** *F* **by** *contradiction*
next
assume ∃ *v* ∈ *sinks I D* (*D ?y*) *?ws*. (*v*, *D ?w*) ∈ *I*
hence *D ?w* ∈ *sinks I D* (*c-dom D yp*) (*?ws* @ [*?w*])
by (*simp add*: *c-dom-def*)
thus *False* **using** *F* **by** *contradiction*
qed
ultimately have *snd wp* = *out* (*foldl step* s_0
(*c-ipurge I D* (*D ?w*) (*?xs* @ *ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*)))) *?w*
using *R* **by** (*simp add*: *c-ipurge-ipurge-tr*)
hence *snd wp* =
out (*foldl step* s_0 (*?xs* @ *ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*))) *?w*
using *S* **by** (*simp add*: *c-secure-def*)
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s (*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*)) @
[(*?w*, *out* (*foldl step ?s* (*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*))) *?w*)]
using *E* **by** (*cases wp*, *simp*)
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s (*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*)) @
c-tr step out (*foldl step ?s* (*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*))) [*?w*]

by (*simp add*: *c-tr-singleton*)
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s (*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*) @ [*?w*])
by (*simp add*: *c-tr-append*)
moreover have
(*c-tr step out ?s* (*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*) @ [*?w*]),
{(*x*, *p*). *p* ≠ *out* (*foldl step ?s*
(*ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*) @ [*?w*])) *x*})
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *?xs*)
(**is** (-, *?Y′*) ∈ -) **by** (*rule c-tr-futures*)
ultimately have
(*xps* @ *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]), *?Y′*)
∈ *c-failures step out* s_0
using *X* **by** (*simp add*: *c-futures-failures*)
moreover have
ipurge-ref I (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) *Y* ⊆ *?Y′*
proof (*rule subsetI*, *simp add*: *split-paired-all ipurge-ref-def c-dom-def*
del: *sinks.simps*, (*erule conjE*)+)
fix *x p*
assume
G: ∀ *v* ∈ *sinks I* (*c-dom D*) (*D ?y*) (*wps* @ [*wp*]). (*v*, *D x*) ∉ *I* **and**
H: (*D ?y*, *D x*) ∉ *I*
have (*xps* @ *yp* # *wps* @ [*wp*], *Y*) ∈ *c-failures step out* s_0
using *B′* **by** (*simp add*: *c-futures-failures*)
hence *Y* ⊆ {(*x′*, *p′*). *p′* ≠
out (*foldl step* s_0 (*map fst* (*xps* @ *yp* # *wps* @ [*wp*]))) *x′*}
by (*rule c-failures-ref*)
hence *Y* ⊆ {(*x′*, *p′*). *p′* ≠
out (*foldl step* s_0 (*?xs* @ *?y* # *?ws* @ [*?w*])) *x′*}
by *simp*
moreover assume (*x*, *p*) ∈ *Y*
ultimately have (*x*, *p*) ∈ {(*x′*, *p′*). *p′* ≠
out (*foldl step* s_0 (*?xs* @ *?y* # *?ws* @ [*?w*])) *x′*} **..**
hence *p* ≠ *out* (*foldl step* s_0
(*c-ipurge I D* (*D x*) (*?xs* @ *?y* # *?ws* @ [*?w*]))) *x*
using *S* **by** (*simp add*: *c-secure-def*)
moreover have
¬ (∃ *v* ∈ *sinks I D* (*D ?y*) (*?y* # *?ws* @ [*?w*]). (*v*, *D x*) ∈ *I*)
proof
assume ∃ *v* ∈ *sinks I D* (*D ?y*) (*?y* # *?ws* @ [*?w*]). (*v*, *D x*) ∈ *I*
then obtain *v* **where**
A: *v* ∈ *sinks I D* (*D ?y*) (*?y* # *?ws* @ [*?w*]) **and**
B: (*v*, *D x*) ∈ *I* **..**
have *v* = *D ?y* ∨ *v* ∈ *sinks I D* (*D ?y*) (*?ws* @ [*?w*])
using *R* **and** *A* **by** (*simp add*: *sinks-cons-same*)
moreover {
assume *v* = *D ?y*
hence (*D ?y*, *D x*) ∈ *I* **using** *B* **by** *simp*
hence *False* **using** *H* **by** *contradiction*

}
moreover {
assume $v \in$ *sinks I D* (*D ?y*) (*?ws* @ [*?w*])
hence $v \in$ *sinks I* (*c-dom D*) (*D ?y*) (*wps* @ [*wp*])
by (*simp only*: *c-dom-sinks*, *simp*)
with *G* **have** $(v, D\ x) \notin I$ **..**
hence *False* **using** *B* **by** *contradiction*
}
ultimately show *False* **by** *blast*
qed
ultimately have $p \neq$ *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*)
(*?xs* @ *ipurge-tr I D* (*D ?y*) (*?y* # *?ws* @ [*?w*])))) *x*
using *R* **by** (*simp add*: *c-ipurge-ipurge-tr*)
hence $p \neq$ *out* (*foldl step* s_0 (*?xs* @ *ipurge-tr I D* (*D ?y*) (*?ws* @ [*?w*]))) *x*
using *R* **and** *S* **by** (*simp add*: *c-secure-def ipurge-tr-cons-same*)
hence $p \neq$ *out* (*foldl step* s_0 (*?xs* @ *ipurge-tr I D* (*D ?y*) *?ws* @ [*?w*])) *x*
using *F* **by** (*simp add*: *c-dom-def*)
thus $p \neq$ *out* (*step* (*foldl step ?s*
(*ipurge-tr I D* (*D ?y*) (*?y* # *?ws*))) *?w*) *x*
using *R* **by** (*simp add*: *ipurge-tr-cons-same*)
qed
ultimately have (*xps* @ *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]),
ipurge-ref I (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) *Y*)
$\in$ *c-failures step out* s_0
by (*rule R2*)
thus (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]),
ipurge-ref I (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) *Y*)
$\in$ *futures* (*c-process step out* s_0) *xps*
by (*simp add*: *c-futures-failures*)
qed
qed
qed

lemma *c-secure-futures-2*:
assumes *R*: *refl I* **and** *S*: *c-secure step out* s_0 *I D*
shows (*yps* @ [*yp*], *A*) $\in$ *futures* (*c-process step out* s_0) *xps* $\Longrightarrow$
(*yps*, *Y*) $\in$ *futures* (*c-process step out* s_0) *xps* $\Longrightarrow$
(*yps* @ [*yp*], $\{x \in Y.$ (*c-dom D yp*, *c-dom D x*) $\notin I\}$)
$\in$ *futures* (*c-process step out* s_0) *xps*
proof (*simp add*: *c-futures-failures*)
let *?zs* = *map fst* (*xps* @ *yps*)
let *?y* = *fst yp*
assume (*xps* @ *yps* @ [*yp*], *A*) $\in$ *c-failures step out* s_0
hence *xps* @ *yps* @ [*yp*] = *c-tr step out* s_0 (*map fst* (*xps* @ *yps* @ [*yp*]))
by (*rule c-failures-tr*)
hence *A*: *xps* @ *yps* @ [*yp*] = *c-tr step out* s_0 (*?zs* @ [*?y*]) **by** *simp*
assume (*xps* @ *yps*, *Y*) $\in$ *c-failures step out* s_0
hence *B*: $Y \subseteq \{(x, p).\ p \neq$ *out* (*foldl step* s_0 *?zs*) $x\}$
(**is** - $\subseteq$ *?Y'*) **by** (*rule c-failures-ref*)

have (*xps* @ *yps* @ [*yp*], {(*x*, *p*). *p* ≠ *out* (*foldl step* s_0 (*?zs* @ [*?y*])) *x*})
∈ *c-failures step out* s_0
(**is** (-, *?X′*) ∈ -) **by** (*subst A*, *rule c-tr-failures*)
moreover have {*x* ∈ *Y*. (*c-dom D yp*, *c-dom D x*) ∉ *I*} ⊆ *?X′* (**is** *?X* ⊆ -)
proof (*rule subsetI*, *simp add*: *split-paired-all c-dom-def*
del: *map-append foldl-append*, *erule conjE*)
fix *x p*
assume (*x*, *p*) ∈ *Y*
with *B* **have** (*x*, *p*) ∈ *?Y′* **..**
hence *p* ≠ *out* (*foldl step* s_0 *?zs*) *x* **by** *simp*
moreover have *out* (*foldl step* s_0 *?zs*) *x* =
out (*foldl step* s_0 (*c-ipurge I D* (*D x*) *?zs*)) *x*
using *S* **by** (*simp add*: *c-secure-def*)
ultimately have *p* ≠ *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*) *?zs*)) *x* **by** *simp*
moreover assume (*D ?y*, *D x*) ∉ *I*
with *R* **have** *c-ipurge I D* (*D x*) (*?zs* @ [*?y*]) = *c-ipurge I D* (*D x*) *?zs*
by (*rule c-ipurge-append-2*)
ultimately have *p* ≠ *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*) (*?zs* @ [*?y*]))) *x*
by *simp*
moreover have *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*) (*?zs* @ [*?y*]))) *x* =
out (*foldl step* s_0 (*?zs* @ [*?y*])) *x*
using *S* **by** (*simp add*: *c-secure-def*)
ultimately show *p* ≠ *out* (*foldl step* s_0 (*?zs* @ [*?y*])) *x* **by** *simp*
qed
ultimately show (*xps* @ *yps* @ [*yp*], *?X*) ∈ *c-failures step out* s_0 **by** (*rule R2*)
qed

lemma *c-secure-ipurge-tr*:
assumes *R*: *refl I* **and** *S*: *c-secure step out* s_0 *I D*
shows *ipurge-tr I* (*c-dom D*) (*D x*) (*c-tr step out* (*step* (*foldl step* s_0 *xs*) *x*) *ys*)
= *ipurge-tr I* (*c-dom D*) (*D x*) (*c-tr step out* (*foldl step* s_0 *xs*) *ys*)
proof (*induction ys rule*: *rev-induct*, *simp*, *simp only*: *c-tr.simps*)
let *?s* = *foldl step* s_0 *xs*
fix *ys y*
assume *A*: *ipurge-tr I* (*c-dom D*) (*D x*) (*c-tr step out* (*step ?s x*) *ys*) =
ipurge-tr I (*c-dom D*) (*D x*) (*c-tr step out ?s ys*)
show *ipurge-tr I* (*c-dom D*) (*D x*) (*c-tr step out* (*step ?s x*) *ys* @
[(*y*, *out* (*foldl step* (*step ?s x*) *ys*) *y*)]) =
ipurge-tr I (*c-dom D*) (*D x*)
(*c-tr step out ?s ys* @ [(*y*, *out* (*foldl step ?s ys*) *y*)])
(**is** - (- @ [*?yp′*]) = - (- @ [*?yp*]))
proof (*cases D y* ∈ *sinks I D* (*D x*) (*ys* @ [*y*]))
assume *D*: *D y* ∈ *sinks I D* (*D x*) (*ys* @ [*y*])
hence *c-dom D ?yp′* ∈ *sinks I* (*c-dom D*) (*D x*)
(*c-tr step out* (*step ?s x*) *ys* @ [*?yp′*])
using *D* **by** (*simp only*: *c-dom-sinks*, *simp add*: *c-dom-def c-tr-map*)
hence *ipurge-tr I* (*c-dom D*) (*D x*) (*c-tr step out* (*step ?s x*) *ys* @ [*?yp′*]) =
ipurge-tr I (*c-dom D*) (*D x*) (*c-tr step out* (*step ?s x*) *ys*)
by *simp*

moreover have *c-dom D ?yp* ∈ *sinks I* (*c-dom D*) (*D x*)
(*c-tr step out ?s ys* @ [*?yp*])
using *D* **by** (*simp only*: *c-dom-sinks*, *simp add*: *c-dom-def c-tr-map*)
hence *ipurge-tr I* (*c-dom D*) (*D x*) (*c-tr step out ?s ys* @ [*?yp*]) =
ipurge-tr I (*c-dom D*) (*D x*) (*c-tr step out ?s ys*)
by *simp*
ultimately show *?thesis* **using** *A* **by** *simp*
next
assume *D*: *D y* ∉ *sinks I D* (*D x*) (*ys* @ [*y*])
hence *c-dom D ?yp'* ∉ *sinks I* (*c-dom D*) (*D x*)
(*c-tr step out* (*step ?s x*) *ys* @ [*?yp'*])
using *D* **by** (*simp only*: *c-dom-sinks*, *simp add*: *c-dom-def c-tr-map*)
hence *ipurge-tr I* (*c-dom D*) (*D x*) (*c-tr step out* (*step ?s x*) *ys* @ [*?yp'*]) =
ipurge-tr I (*c-dom D*) (*D x*) (*c-tr step out* (*step ?s x*) *ys*) @ [*?yp'*]
by *simp*
moreover have *c-dom D ?yp* ∉ *sinks I* (*c-dom D*) (*D x*)
(*c-tr step out ?s ys* @ [*?yp*])
using *D* **by** (*simp only*: *c-dom-sinks*, *simp add*: *c-dom-def c-tr-map*)
hence *ipurge-tr I* (*c-dom D*) (*D x*) (*c-tr step out ?s ys* @ [*?yp*]) =
ipurge-tr I (*c-dom D*) (*D x*) (*c-tr step out ?s ys*) @ [*?yp*]
by *simp*
ultimately show *?thesis*
proof (*simp add*: *A*)
have *B*: ¬ (∃ *v* ∈ *sinks I D* (*D x*) *ys*. (*v*, *D y*) ∈ *I*)
proof
assume ∃ *v* ∈ *sinks I D* (*D x*) *ys*. (*v*, *D y*) ∈ *I*
hence *D y* ∈ *sinks I D* (*D x*) (*ys* @ [*y*]) **by** *simp*
thus *False* **using** *D* **by** *contradiction*
qed
have *C*: ¬ (∃ *v* ∈ *sinks I D* (*D x*) (*x* # *ys*). (*v*, *D y*) ∈ *I*)
proof (*rule notI*, *simp add*: *sinks-cons-same R B*)
assume (*D x*, *D y*) ∈ *I*
hence *D y* ∈ *sinks I D* (*D x*) (*ys* @ [*y*]) **by** *simp*
thus *False* **using** *D* **by** *contradiction*
qed
have *out* (*foldl step* (*step ?s x*) *ys*) *y* = *out* (*foldl step* s_0 (*xs* @ *x* # *ys*)) *y*
by *simp*
also have . . . = *out* (*foldl step* s_0 (*c-ipurge I D* (*D y*) (*xs* @ *x* # *ys*))) *y*
using *S* **by** (*simp add*: *c-secure-def*)
also have . . . = *out* (*foldl step* s_0 (*c-ipurge I D* (*D y*)
(*xs* @ *ipurge-tr I D* (*D x*) (*x* # *ys*)))) *y*
using *R* **and** *C* **by** (*simp add*: *c-ipurge-ipurge-tr*)
also have . . . = *out* (*foldl step* s_0 (*c-ipurge I D* (*D y*)
(*xs* @ *ipurge-tr I D* (*D x*) *ys*))) *y*
using *R* **by** (*simp add*: *ipurge-tr-cons-same*)
also have . . . = *out* (*foldl step* s_0 (*c-ipurge I D* (*D y*) (*xs* @ *ys*))) *y*
using *R* **and** *B* **by** (*simp add*: *c-ipurge-ipurge-tr*)
also have . . . = *out* (*foldl step* s_0 (*xs* @ *ys*)) *y*
using *S* **by** (*simp add*: *c-secure-def*)

also have ... = *out* (*foldl step ?s ys*) *y* **by** *simp*
finally show *out* (*foldl step* (*step ?s x*) *ys*) *y* = *out* (*foldl step ?s ys*) *y* **.**
qed
qed
qed

lemma *c-secure-implies-secure-aux-2* [*rule-format*]:
assumes
R: *refl I* **and**
S: *c-secure step out* s_0 *I D* **and**
Y: (*yp* # *yps*, *Y*) ∈ *futures* (*c-process step out* s_0) *xps*
shows (*zps*, *Z*) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *zps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *zps Z*)
∈ *futures* (*c-process step out* s_0) *xps*
proof (*induction zps arbitrary*: *Z rule*: *length-induct*, *rule impI*)
fix *zps Z*
assume
A: ∀ *zps'*. *length zps'* < *length zps* ⟶
(∀ *Z'*. (*zps'*, *Z'*) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *zps'*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *zps' Z'*)
∈ *futures* (*c-process step out* s_0) *xps*) **and**
B: (*zps*, *Z*) ∈ *futures* (*c-process step out* s_0) *xps*
show (*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *zps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *zps Z*)
∈ *futures* (*c-process step out* s_0) *xps*
proof (*cases zps*, *simp add*: *ipurge-ref-def*)
case *Nil*
hence *C*: ([], *Z*) ∈ *futures* (*c-process step out* s_0) *xps* **using** *B* **by** *simp*
have (([] @ [*yp*]) @ *yps*, *Y*) ∈ *futures* (*c-process step out* s_0) *xps*
using *Y* **by** *simp*
hence ([] @ [*yp*], {}) ∈ *futures* (*c-process step out* s_0) *xps*
by (*rule process-rule-2-futures*)
with *R* **and** *S* **have** ([] @ [*yp*], {*x* ∈ *Z*. (*c-dom D yp*, *c-dom D x*) ∉ *I*})
∈ *futures* (*c-process step out* s_0) *xps*
using *C* **by** (*rule c-secure-futures-2*)
thus ([*yp*], {*x* ∈ *Z*. (*c-dom D yp*, *c-dom D x*) ∉ *I*})
∈ *futures* (*c-process step out* s_0) *xps*
by *simp*
next
case *Cons*
have ∃ *wps wp*. *zps* = *wps* @ [*wp*]
by (*rule rev-cases* [*of zps*], *simp-all add*: *Cons*)
then obtain *wps* **and** *wp* **where** *C*: *zps* = *wps* @ [*wp*] **by** *blast*
have *B'*: (*wps* @ [*wp*], *Z*) ∈ *futures* (*c-process step out* s_0) *xps*
using *B* **and** *C* **by** *simp*
show *?thesis*
proof (*simp only*: *C*,

cases c-dom D wp ∈ *sinks I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]))
let *?Z′* = {*x* ∈ *Z*. (*c-dom D wp*, *c-dom D x*) ∉ *I*}
have *length wps* < *length zps* ⟶
(∀ *Z′*. (*wps*, *Z′*) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *wps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps Z′*)
∈ *futures* (*c-process step out* s_0) *xps*)
using *A* **..**
moreover have *length wps* < *length zps* **using** *C* **by** *simp*
ultimately have ∀ *Z′*. (*wps*, *Z′*) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *wps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps Z′*)
∈ *futures* (*c-process step out* s_0) *xps* **..**
hence (*wps*, *?Z′*) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *wps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps ?Z′*)
∈ *futures* (*c-process step out* s_0) *xps* **..**
moreover have (*wps*, *?Z′*) ∈ *futures* (*c-process step out* s_0) *xps*
using *R* **and** *S* **and** *B′* **by** (*rule c-secure-futures-1*)
ultimately have (*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *wps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps ?Z′*)
∈ *futures* (*c-process step out* s_0) *xps* **..**
moreover assume
D: *c-dom D wp* ∈ *sinks I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*])
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
ipurge-tr I (*c-dom D*) (*c-dom D yp*) *wps*
by *simp*
moreover have *ipurge-ref I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) *Z* =
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *wps ?Z′*
using *D* **by** (*rule ipurge-ref-eq*)
ultimately show (*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]),
ipurge-ref I (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) *Z*)
∈ *futures* (*c-process step out* s_0) *xps*
by *simp*
next
let *?xs* = *map fst xps*
let *?y* = *fst yp*
let *?ws* = *map fst wps*
let *?w* = *fst wp*
let *?s* = *foldl step* s_0 *?xs*
let *?s′* = *foldl step* s_0 (*?xs* @ [*?y*])
have ((*xps* @ [*yp*]) @ *yps*, *Y*) ∈ *failures* (*c-process step out* s_0)
using *Y* **by** (*simp add*: *c-futures-failures c-failures-failures*)
hence (*xps* @ [*yp*], {}) ∈ *failures* (*c-process step out* s_0)
by (*rule process-rule-2-failures*)
hence (*xps* @ [*yp*], {}) ∈ *c-failures step out* s_0
by (*simp add*: *c-failures-failures*)
hence *xps* @ [*yp*] = *c-tr step out* s_0 (*map fst* (*xps* @ [*yp*]))
by (*rule c-failures-tr*)

hence XY: xps @ $[yp]$ = $c\text{-}tr\ step\ out\ s_0$ ($?xs$ @ $[?y]$) **by** *simp*
hence X: xps = $c\text{-}tr\ step\ out\ s_0\ ?xs$ **by** *simp*
have ($[yp]$ @ yps, Y) ∈ *futures* (*c-process step out* s_0) *xps*
 using Y **by** *simp*
hence ($[yp]$, {}) ∈ *futures* (*c-process step out* s_0) *xps*
 by (*rule process-rule-2-futures*)
hence $[yp]$ = *c-tr step out ?s* (*map fst* $[yp]$) **by** (*rule c-futures-tr*)
hence Y': $[yp]$ = *c-tr step out ?s* ($[?y]$) **by** *simp*
have W: (*wps*, {}) ∈ *futures* (*c-process step out* s_0) *xps*
 using B' **by** (*rule process-rule-2-futures*)
hence W': *wps* = *c-tr step out* (*foldl step* s_0 *?xs*) *?ws* **by** (*rule c-futures-tr*)
assume D: *c-dom D wp* ∉ *sinks I* (*c-dom D*) (*c-dom D yp*) (*wps* @ $[wp]$)
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ $[wp]$) =
 ipurge-tr I (*c-dom D*) (*c-dom D yp*) *wps* @ $[wp]$
 by *simp*
hence *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ $[wp]$) =
 ipurge-tr I (*c-dom D*) (*c-dom D yp*)
 (*c-tr step out* (*foldl step* s_0 *?xs*) *?ws*) @ $[wp]$
 using W' **by** *simp*
also have ... =
 ipurge-tr I (*c-dom D*) (*c-dom D yp*) (*c-tr step out ?s' ?ws*) @ $[wp]$
 using R **and** S **by** (*simp add*: *c-secure-ipurge-tr c-dom-def*)
also have ... = *c-tr step out ?s'* (*ipurge-tr I D* (*c-dom D yp*) *?ws*) @ $[wp]$
proof (*simp del*: *foldl-append*, *rule c-tr-ipurge-tr*)
 fix n
 let *?wps'* = *c-tr step out ?s* (*take n ?ws*)
 have *length ?wps'* < *length zps* ⟶
  (∀ Z'. (*?wps'*, Z') ∈ *futures* (*c-process step out* s_0) *xps* ⟶
  (*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps'*,
  ipurge-ref I (*c-dom D*) (*c-dom D yp*) *?wps'* Z')
  ∈ *futures* (*c-process step out* s_0) *xps*)
  using A **..**
 moreover have *length ?wps'* < *length zps*
  using C **by** (*simp add*: *c-tr-length*)
 ultimately have ∀ Z'.
  (*?wps'*, Z') ∈ *futures* (*c-process step out* s_0) *xps* ⟶
  (*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps'*,
  ipurge-ref I (*c-dom D*) (*c-dom D yp*) *?wps'* Z')
  ∈ *futures* (*c-process step out* s_0) *xps* **..**
 hence (*?wps'*, {}) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
  (*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps'*,
  ipurge-ref I (*c-dom D*) (*c-dom D yp*) *?wps'* {})
  ∈ *futures* (*c-process step out* s_0) *xps*
  (**is** - ⟶ (-, *?W'*) ∈ -) **..**
 moreover have *wps* = *c-tr step out ?s* (*take n ?ws* @ *drop n ?ws*)
  using W' **by** *simp*
 hence *wps* = *?wps'* @
  c-tr step out (*foldl step ?s* (*take n ?ws*)) (*drop n ?ws*)
  by (*simp only*: *c-tr-append*)

hence (*?wps′* @ *c-tr step out* (*foldl step ?s* (*take n ?ws*)) (*drop n ?ws*), {})
∈ *futures* (*c-process step out* s_0) *xps*
using *W* **by** *simp*
hence (*?wps′*, {}) ∈ *futures* (*c-process step out* s_0) *xps*
by (*rule process-rule-2-futures*)
ultimately have (*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps′*, *?W′*)
∈ *futures* (*c-process step out* s_0) *xps* **..**
hence (*c-tr step out* s_0 (*?xs* @ [*?y*]) @
ipurge-tr I (*c-dom D*) (*c-dom D yp*) *?wps′*, *?W′*)
∈ *c-failures step out* s_0
using *XY* **by** (*simp add*: *c-futures-failures*)
hence (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *?wps′*, *?W′*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 (*?xs* @ [*?y*]))
by (*simp add*: *c-futures-failures*)
hence (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*)
(*c-tr step out ?s′* (*take n ?ws*)), *?W′*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 (*?xs* @ [*?y*]))
using *R* **and** *S* **by** (*simp add*: *c-dom-def c-secure-ipurge-tr*)
thus ∃ *W*. (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*)
(*c-tr step out ?s′* (*take n ?ws*)), *W*)
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 (*?xs* @ [*?y*]))
by (*rule-tac x* = *?W′* **in** *exI*)
qed
finally have *E*: *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s′ (*ipurge-tr I D* (*c-dom D yp*) *?ws*) @ [*wp*] **.**
have (*xps* @ *wps* @ [*wp*], *Z*) ∈ *c-failures step out* s_0
(**is** (*?xps′*, -) ∈ -) **using** *B′* **by** (*simp add*: *c-futures-failures*)
moreover have *?xps′* ≠ [] **by** *simp*
ultimately have *snd* (*last ?xps′*) =
out (*foldl step* s_0 (*butlast* (*map fst ?xps′*))) (*last* (*map fst ?xps′*))
by (*rule c-failures-last*)
hence *snd wp* = *out* (*foldl step* s_0 (*?xs* @ *?ws*)) *?w*
by (*simp add*: *butlast-append*)
hence *snd wp* = *out* (*foldl step* s_0 (*c-ipurge I D* (*D ?w*) (*?xs* @ *?ws*))) *?w*
using *S* **by** (*simp add*: *c-secure-def*)
moreover have *F*: *D ?w* ∉ *sinks I D* (*c-dom D yp*) (*?ws* @ [*?w*])
using *D* **by** (*simp only*: *c-dom-sinks*, *simp add*: *c-dom-def*)
have *G*: ¬ (∃ *v* ∈ *sinks I D* (*c-dom D yp*) *?ws*. (*v*, *D ?w*) ∈ *I*)
proof
assume ∃ *v* ∈ *sinks I D* (*c-dom D yp*) *?ws*. (*v*, *D ?w*) ∈ *I*
hence *D ?w* ∈ *sinks I D* (*c-dom D yp*) (*?ws* @ [*?w*]) **by** *simp*
thus *False* **using** *F* **by** *contradiction*
qed
ultimately have *snd wp* = *out* (*foldl step* s_0
(*c-ipurge I D* (*D ?w*) (*?xs* @ *ipurge-tr I D* (*c-dom D yp*) *?ws*))) *?w*
using *R* **by** (*simp add*: *c-ipurge-ipurge-tr*)
hence *snd wp* = *out* (*foldl step* s_0
(*c-ipurge I D* (*D ?w*) (*?xs* @ *ipurge-tr I D* (*c-dom D yp*) (*?y* # *?ws*)))) *?w*
using *R* **by** (*simp add*: *c-dom-def ipurge-tr-cons-same*)

moreover have
¬ (∃ *v* ∈ *sinks I D* (*c-dom D yp*) (*?y* # *?ws*). (*v*, *D ?w*) ∈ *I*)
proof (*rule notI*, *simp add*: *sinks-cons-same c-dom-def R G* [*simplified*])
assume (*D ?y*, *D ?w*) ∈ *I*
hence *D ?w* ∈ *sinks I D* (*c-dom D yp*) (*?ws* @ [*?w*])
by (*simp add*: *c-dom-def*)
thus *False* **using** *F* **by** *contradiction*
qed
ultimately have *snd wp* =
out (*foldl step* s_0 (*c-ipurge I D* (*D ?w*) (*?xs* @ [*?y*] @ *?ws*))) *?w*
using *R* **by** (*simp add*: *c-ipurge-ipurge-tr*)
moreover have *c-ipurge I D* (*D ?w*) ((*?xs* @ [*?y*]) @
ipurge-tr I D (*c-dom D yp*) *?ws*) =
c-ipurge I D (*D ?w*) ((*?xs* @ [*?y*]) @ *?ws*)
using *R* **and** *G* **by** (*rule c-ipurge-ipurge-tr*)
ultimately have *snd wp* = *out* (*foldl step* s_0
(*c-ipurge I D* (*D ?w*) (*?xs* @ [*?y*] @ *ipurge-tr I D* (*c-dom D yp*) *?ws*))) *?w*
by *simp*
hence *snd wp* =
out (*foldl step* s_0 (*?xs* @ [*?y*] @ *ipurge-tr I D* (*c-dom D yp*) *?ws*)) *?w*
using *S* **by** (*simp add*: *c-secure-def*)
hence *yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s ([*?y*]) @
c-tr step out ?s' (*ipurge-tr I D* (*c-dom D yp*) *?ws*) @
[(*?w*, *out* (*foldl step ?s'* (*ipurge-tr I D* (*c-dom D yp*) *?ws*)) *?w*)]
using *Y'* **and** *E* **by** (*cases wp*, *simp*)
hence *yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s ([*?y*]) @
c-tr step out ?s' (*ipurge-tr I D* (*c-dom D yp*) *?ws*) @
c-tr step out (*foldl step ?s'* (*ipurge-tr I D* (*c-dom D yp*) *?ws*)) [*?w*]
by (*simp add*: *c-tr-singleton*)
hence *yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s ([*?y*]) @
c-tr step out (*foldl step ?s* [*?y*]) (*ipurge-tr I D* (*c-dom D yp*) *?ws* @ [*?w*])
by (*simp add*: *c-tr-append*)
hence *yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) =
c-tr step out ?s ([*?y*] @ *ipurge-tr I D* (*c-dom D yp*) *?ws* @ [*?w*])
by (*simp only*: *c-tr-append*)
moreover have
(*c-tr step out ?s* (*?y* # *ipurge-tr I D* (*c-dom D yp*) *?ws* @ [*?w*]),
{(*x*, *p*). *p* ≠ *out* (*foldl step ?s*
(*?y* # *ipurge-tr I D* (*c-dom D yp*) *?ws* @ [*?w*])) *x*})
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *?xs*)
(**is** (-, *?Z'*) ∈ -) **by** (*rule c-tr-futures*)
ultimately have
(*xps* @ *yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]), *?Z'*)
∈ *c-failures step out* s_0
using *X* **by** (*simp add*: *c-futures-failures*)
moreover have

ipurge-ref I (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*]) *Z* ⊆ *?Z′*
proof (*rule subsetI*, *simp add*: *split-paired-all ipurge-ref-def c-dom-def del*: *sinks.simps foldl.simps*, (*erule conjE*)+)
fix *x p*
assume
H: ∀ *v* ∈ *sinks I* (*c-dom D*) (*D ?y*) (*wps* @ [*wp*]). (*v*, *D x*) ∉ *I* **and**
I: (*D ?y*, *D x*) ∉ *I*
have (*xps* @ *wps* @ [*wp*], *Z*) ∈ *c-failures step out* s_0
using *B′* **by** (*simp add*: *c-futures-failures*)
hence *Z* ⊆ {(*x′*, *p′*). *p′* ≠
out (*foldl step* s_0 (*map fst* (*xps* @ *wps* @ [*wp*]))) *x′*}
by (*rule c-failures-ref*)
hence *Z* ⊆ {(*x′*, *p′*). *p′* ≠
out (*foldl step* s_0 (*?xs* @ *?ws* @ [*?w*])) *x′*}
by *simp*
moreover assume (*x*, *p*) ∈ *Z*
ultimately have (*x*, *p*) ∈ {(*x′*, *p′*). *p′* ≠
out (*foldl step* s_0 (*?xs* @ *?ws* @ [*?w*])) *x′*} **..**
hence *p* ≠ *out* (*foldl step* s_0
(*c-ipurge I D* (*D x*) (*?xs* @ *?ws* @ [*?w*]))) *x*
using *S* **by** (*simp add*: *c-secure-def*)
moreover have
J: ¬ (∃ *v* ∈ *sinks I D* (*D ?y*) (*?ws* @ [*?w*]). (*v*, *D x*) ∈ *I*)
proof (*rule notI*,
cases (*D ?y*, *D ?w*) ∈ *I* ∨ (∃ *v* ∈ *sinks I D* (*D ?y*) *?ws*. (*v*, *D ?w*) ∈ *I*),
simp-all only: *sinks.simps if-True if-False*)
case *True*
hence *c-dom D wp* ∈ *sinks I* (*c-dom D*) (*c-dom D yp*) (*wps* @ [*wp*])
by (*simp only*: *c-dom-sinks*, *simp add*: *c-dom-def*)
thus *False* **using** *D* **by** *contradiction*
next
assume ∃ *v* ∈ *sinks I D* (*D ?y*) *?ws*. (*v*, *D x*) ∈ *I*
then obtain *v* **where**
A: *v* ∈ *sinks I D* (*D ?y*) *?ws* **and**
B: (*v*, *D x*) ∈ *I* **..**
have *v* ∈ *sinks I* (*c-dom D*) (*D ?y*) (*wps* @ [*wp*])
using *A* **by** (*simp add*: *c-dom-sinks*)
with *H* **have** (*v*, *D x*) ∉ *I* **..**
thus *False* **using** *B* **by** *contradiction*
qed
ultimately have *p* ≠ *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*)
(*?xs* @ *ipurge-tr I D* (*D ?y*) (*?ws* @ [*?w*])))) *x*
using *R* **by** (*simp add*: *c-ipurge-ipurge-tr del*: *ipurge-tr.simps*)
hence *p* ≠ *out* (*foldl step* s_0 (*c-ipurge I D* (*D x*)
(*?xs* @ *ipurge-tr I D* (*D ?y*) (*?y* # *?ws* @ [*?w*])))) *x*
using *R* **by** (*simp add*: *ipurge-tr-cons-same*)
moreover have
¬ (∃ *v* ∈ *sinks I D* (*D ?y*) (*?y* # *?ws* @ [*?w*]). (*v*, *D x*) ∈ *I*)
proof

```
        assume ∃ v ∈ sinks I D (D ?y) (?y # ?ws @ [?w]). (v, D x) ∈ I
        then obtain v where
          A: v ∈ sinks I D (D ?y) (?y # ?ws @ [?w]) and
          B: (v, D x) ∈ I ..
        have v = D ?y ∨ v ∈ sinks I D (D ?y) (?ws @ [?w])
         using R and A by (simp add: sinks-cons-same)
        moreover {
          assume v = D ?y
          hence (D ?y, D x) ∈ I using B by simp
          hence False using I by contradiction
        }
        moreover {
          assume v ∈ sinks I D (D ?y) (?ws @ [?w])
          with B have ∃ v ∈ sinks I D (D ?y) (?ws @ [?w]). (v, D x) ∈ I ..
          hence False using J by contradiction
        }
        ultimately show False by blast
      qed
      ultimately have p ≠ out (foldl step s0 (c-ipurge I D (D x)
        (?xs @ [?y] @ ?ws @ [?w]))) x
       using R by (simp add: c-ipurge-ipurge-tr del: ipurge-tr.simps)
      moreover have c-ipurge I D (D x) ((?xs @ [?y]) @
        ipurge-tr I D (D ?y) (?ws @ [?w])) =
        c-ipurge I D (D x) ((?xs @ [?y]) @ ?ws @ [?w])
       using R and J by (rule c-ipurge-ipurge-tr)
      ultimately have p ≠ out (foldl step s0 (c-ipurge I D (D x)
        (?xs @ ?y # ipurge-tr I D (D ?y) (?ws @ [?w])))) x
       by simp
      hence p ≠ out (foldl step s0
        (?xs @ ?y # ipurge-tr I D (D ?y) (?ws @ [?w]))) x
       using S by (simp add: c-secure-def)
      thus p ≠ out (foldl step ?s
        (?y # ipurge-tr I D (D ?y) ?ws @ [?w])) x
       using F by (simp add: c-dom-def)
    qed
    ultimately have
      (xps @ yp # ipurge-tr I (c-dom D) (c-dom D yp) (wps @ [wp]),
      ipurge-ref I (c-dom D) (c-dom D yp) (wps @ [wp]) Z)
      ∈ c-failures step out s0
     by (rule R2)
    thus (yp # ipurge-tr I (c-dom D) (c-dom D yp) (wps @ [wp]),
      ipurge-ref I (c-dom D) (c-dom D yp) (wps @ [wp]) Z)
      ∈ futures (c-process step out s0) xps
     by (simp add: c-futures-failures)
  qed
 qed
qed

theorem c-secure-implies-secure:
```

assumes R: *refl I* **and** S: *c-secure step out* s_0 *I D*
shows *secure* (*c-process step out* s_0) *I* (*c-dom D*)
proof (*simp only*: *secure-def*, (*rule allI*)+, *rule impI*, *erule conjE*)
fix *xps yp yps Y zps Z*
assume
Y: (*yp* # *yps*, Y) ∈ *futures* (*c-process step out* s_0) *xps* **and**
Z: (*zps*, Z) ∈ *futures* (*c-process step out* s_0) *xps*
show (*ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *yps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *yps Y*)
∈ *futures* (*c-process step out* s_0) *xps* ∧
(*yp* # *ipurge-tr I* (*c-dom D*) (*c-dom D yp*) *zps*,
ipurge-ref I (*c-dom D*) (*c-dom D yp*) *zps Z*)
∈ *futures* (*c-process step out* s_0) *xps*
(**is** *?P* ∧ *?Q*)
proof
show *?P* **using** R **and** S **and** Y
by (*rule c-secure-implies-secure-aux-1*)
next
show *?Q* **using** R **and** S **and** Y **and** Z
by (*rule c-secure-implies-secure-aux-2*)
qed
qed

theorem *secure-equals-c-secure*:
refl I ⟹ *secure* (*c-process step out* s_0) *I* (*c-dom D*) = *c-secure step out* s_0 *I D*
by (*rule iffI*, *rule secure-implies-c-secure*, *assumption*, *rule c-secure-implies-secure*)

end

3 CSP noninterference vs. generalized noninterference

theory *GeneralizedNoninterference*
imports *ClassicalNoninterference*
begin

The purpose of this section is to compare CSP noninterference security as defined previously with McCullough's notion of generalized noninterference security as formulated in [4]. It will be shown that this security property is weaker than both CSP noninterference security for a generic process, and classical noninterference security for classical processes, viz. it is a necessary but not sufficient condition for them. This renders CSP noninterference security preferable as an extension of classical noninterference security to nondeterministic systems.

For clarity, all the constants and fact names defined in this section, with the possible exception of datatype constructors and main theorems, contain prefix *g-*.

3.1 Generalized noninterference

The original formulation of generalized noninterference security as contained in [4] focuses on systems whose events, split in inputs and outputs, are mapped into either of two security levels, *high* and *low*. Such a system is said to be secure just in case, for any trace *xs* and any high-level input *x*, the set of the *possible low-level futures* of *xs*, i.e. of the sequences of low-level events that may succeed *xs* in the traces of the system, is equal to the set of the possible low-level futures of *xs* @ [*x*].

This definition requires the following corrections:

- Variable *x* must range over all high-level events rather than over high-level inputs alone, since high-level outputs must not be allowed to affect low-level futures as well.

- For any *x*, the range of trace *xs* must be restricted to the traces of the system that may be succeeded by *x*, viz. trace *xs* must be such that event list *xs* @ [*x*] be itself a trace.
  Otherwise, a system that admits both high-level and low-level events in its alphabet but never accepts any high-level event, always accepting any low-level one instead, would turn out not to be secure, which is paradoxical since *high* can by no means affect *low* in a system never engaging in high-level events. The cause of the paradox is that, for each trace *xs* and each high-level event *x* of such a system, the set of the possible low-level futures of *xs* matches the Kleene closure of the set of low-level events, whereas the set of the possible low-level futures of *xs* @ [*x*] matches the empty set as *xs* @ [*x*] is not a trace.

Observe that the latter correction renders it unnecessary to explicitly assume that event list *xs* be a trace of the system, as this follows from the assumption that *xs* @ [*x*] be such.

Here below is a formal definition of the notion of generalized noninterference security for processes, amended in accordance with the previous considerations.

datatype *g-level* = *High* | *Low*

definition *g-secure* :: *'a process* ⇒ (*'a* ⇒ *g-level*) ⇒ *bool* **where**
g-secure P L ≡ ∀ *xs x*. *xs* @ [*x*] ∈ *traces P* ∧ *L x* = *High* ⟶
  {*ys'*. ∃ *ys*. *xs* @ *ys* ∈ *traces P* ∧ *ys'* = [*y*←*ys*. *L y* = *Low*]} =
  {*ys'*. ∃ *ys*. *xs* @ *x* # *ys* ∈ *traces P* ∧ *ys'* = [*y*←*ys*. *L y* = *Low*]}

It is possible to prove that a weaker sufficient (as well as necessary, as obvious) condition for generalized noninterference security is that the set of the possible low-level futures of trace *xs* be included in the set of the possible low-level futures of trace *xs* @ [*x*], because the latter is always included in the former.

In what follows, such security property is defined formally and its sufficiency for generalized noninterference security to hold is demonstrated in the form of an introduction rule, which will turn out to be useful in subsequent proofs.

definition *g-secure-suff* :: *'a process* ⇒ (*'a* ⇒ *g-level*) ⇒ *bool* **where**
g-secure-suff P L ≡ ∀ *xs x*. *xs* @ [*x*] ∈ *traces P* ∧ *L x* = *High* ⟶
{*ys'*. ∃ *ys*. *xs* @ *ys* ∈ *traces P* ∧ *ys'* = [*y*←*ys*. *L y* = *Low*]} ⊆
{*ys'*. ∃ *ys*. *xs* @ *x* # *ys* ∈ *traces P* ∧ *ys'* = [*y*←*ys*. *L y* = *Low*]}

lemma *g-secure-suff-implies-g-secure*:
assumes *S*: *g-secure-suff P L*
shows *g-secure P L*
proof (*simp add*: *g-secure-def*, (*rule allI*)+, *rule impI*, *erule conjE*)
fix *xs x*
assume
A: *xs* @ [*x*] ∈ *traces P* **and**
B: *L x* = *High*
show {*ys'*. ∃ *ys*. *xs* @ *ys* ∈ *traces P* ∧ *ys'* = [*y*←*ys* . *L y* = *Low*]} =
{*ys'*. ∃ *ys*. *xs* @ *x* # *ys* ∈ *traces P* ∧ *ys'* = [*y*←*ys* . *L y* = *Low*]}
(**is** {*ys'*. ∃ *ys*. *?Q ys ys'*} = {*ys'*. ∃ *ys*. *?Q' ys ys'*})
proof (*rule equalityI*, *rule-tac* [*2*] *subsetI*, *simp-all*, *erule-tac* [*2*] *exE*, *erule-tac* [*2*] *conjE*)
show {*ys'*. ∃ *ys*. *?Q ys ys'*} ⊆ {*ys'*. ∃ *ys*. *?Q' ys ys'*}
using *S* **and** *A* **and** *B* **by** (*simp add*: *g-secure-suff-def*)
next
fix *ys ys'*
assume *xs* @ *x* # *ys* ∈ *traces P*
moreover assume *ys'* = [*y*←*ys*. *L y* = *Low*]
hence *ys'* = [*y*←*x* # *ys*. *L y* = *Low*] **using** *B* **by** *simp*
ultimately have *?Q* (*x* # *ys*) *ys'* **..**
thus ∃ *ys*. *?Q ys ys'* **..**
qed
qed

3.2 Comparison between security properties

In the continuation, it will be proven that CSP noninterference security is a sufficient condition for generalized noninterference security for any process whose events are mapped into either security domain *High* or *Low*, under the policy that *High* may not affect *Low*.

Particularly, this is the case for any such classical process. This fact, along with the equivalence between CSP noninterference security and classical noninterference security for classical processes, is used to additionally prove that the classical noninterference security of a deterministic state machine is a sufficient condition for the generalized noninterference security of the corresponding classical process under the aforesaid policy.

```
definition g-I :: (g-level × g-level) set where
g-I ≡ {(High, High), (Low, Low), (Low, High)}

lemma g-I-refl: refl g-I
proof (simp add: refl-on-def, rule allI)
  fix x
  show (x, x) ∈ g-I by (cases x, simp-all add: g-I-def)
qed

lemma g-sinks: sinks g-I L High xs ⊆ {High}
proof (induction xs rule: rev-induct, simp)
  fix x xs
  assume A: sinks g-I L High xs ⊆ {High}
  show sinks g-I L High (xs @ [x]) ⊆ {High}
  proof (cases L x)
    assume L x = High
    thus ?thesis using A by simp
  next
    assume B: L x = Low
    have ¬ ((High, L x) ∈ g-I ∨ (∃ v ∈ sinks g-I L High xs. (v, L x) ∈ g-I))
    proof (rule notI, simp add: B, erule disjE)
      assume (High, Low) ∈ g-I
      moreover have (High, Low) ∉ g-I by (simp add: g-I-def)
      ultimately show False by contradiction
    next
      assume ∃ v ∈ sinks g-I L High xs. (v, Low) ∈ g-I
      then obtain v where C: v ∈ sinks g-I L High xs and D: (v, Low) ∈ g-I ..
      have v ∈ {High} using A and C ..
      hence (High, Low) ∈ g-I using D by simp
      moreover have (High, Low) ∉ g-I by (simp add: g-I-def)
      ultimately show False by contradiction
    qed
    thus ?thesis using A by simp
  qed
qed

lemma g-ipurge-tr: ipurge-tr g-I L High xs = [x←xs. L x = Low]
proof (induction xs rule: rev-induct, simp)
  fix x xs
  assume A: ipurge-tr g-I L High xs = [x'←xs. L x' = Low]
  show ipurge-tr g-I L High (xs @ [x]) = [x'←xs @ [x]. L x' = Low]
  proof (cases L x)
```

assume *B*: *L x* = *High*
hence *ipurge-tr g-I L High* (*xs* @ [*x*]) = *ipurge-tr g-I L High xs*
by (*simp add*: *g-I-def*)
moreover have [*x*′←*xs* @ [*x*]. *L x*′ = *Low*] = [*x*′←*xs*. *L x*′ = *Low*]
using *B* **by** *simp*
ultimately show *?thesis* **using** *A* **by** *simp*
next
assume *B*: *L x* = *Low*
have *L x* ∉ *sinks g-I L High* (*xs* @ [*x*])
proof (*rule notI*, *simp only*: *B*)
have *sinks g-I L High* (*xs* @ [*x*]) ⊆ {*High*} **by** (*rule g-sinks*)
moreover assume *Low* ∈ *sinks g-I L High* (*xs* @ [*x*])
ultimately have *Low* ∈ {*High*} **..**
thus *False* **by** *simp*
qed
hence *ipurge-tr g-I L High* (*xs* @ [*x*]) = *ipurge-tr g-I L High xs* @ [*x*]
by *simp*
moreover have [*x*′←*xs* @ [*x*]. *L x*′ = *Low*] = [*x*′←*xs*. *L x*′ = *Low*] @ [*x*]
using *B* **by** *simp*
ultimately show *?thesis* **using** *A* **by** *simp*
qed
qed

theorem *secure-implies-g-secure*:
assumes *S*: *secure P g-I L*
shows *g-secure P L*
proof (*rule g-secure-suff-implies-g-secure*, *simp add*: *g-secure-suff-def*, (*rule allI*)+, *rule impI*, *rule subsetI*, *simp*, *erule exE*, (*erule conjE*)+)
fix *xs x ys ys*′
assume *xs* @ [*x*] ∈ *traces P*
hence ∃ *X*. ([*x*], *X*) ∈ *futures P xs*
by (*simp add*: *traces-def Domain-iff futures-def*)
then obtain *X* **where** ([*x*], *X*) ∈ *futures P xs* **..**
moreover assume *xs* @ *ys* ∈ *traces P*
hence ∃ *Y*. (*ys*, *Y*) ∈ *futures P xs*
by (*simp add*: *traces-def Domain-iff futures-def*)
then obtain *Y* **where** (*ys*, *Y*) ∈ *futures P xs* **..**
ultimately have (*x* # *ipurge-tr g-I L* (*L x*) *ys*,
ipurge-ref g-I L (*L x*) *ys Y*) ∈ *futures P xs*
(**is** (-, *?Y*′) ∈ *futures P xs*) **using** *S* **by** (*simp add*: *secure-def*)
moreover assume *L x* = *High* **and** *A*: *ys*′ = [*y*←*ys*. *L y* = *Low*]
ultimately have (*x* # *ys*′, *?Y*′) ∈ *futures P xs* **by** (*simp add*: *g-ipurge-tr*)
hence ∃ *Y*′. (*x* # *ys*′, *Y*′) ∈ *futures P xs* **..**
hence *xs* @ *x* # *ys*′ ∈ *traces P*
by (*simp add*: *traces-def Domain-iff futures-def*)
moreover have *ys*′ = [*y*←*ys*′. *L y* = *Low*] **using** *A* **by** *simp*
ultimately have *xs* @ *x* # *ys*′ ∈ *traces P* ∧ *ys*′ = [*y*←*ys*′. *L y* = *Low*] **..**
thus ∃ *ys*. *xs* @ *x* # *ys* ∈ *traces P* ∧ *ys*′ = [*y*←*ys*. *L y* = *Low*] **..**
qed

theorem *c-secure-implies-g-secure*:
c-secure step out s_0 *g-I L* $\Longrightarrow$ *g-secure* (*c-process step out* s_0) (*c-dom L*)
by (*rule secure-implies-g-secure*, *rule c-secure-implies-secure*, *rule g-I-refl*)

Since the definition of generalized noninterference security does not impose any explicit requirement on process refusals, intuition suggests that this security property is likely to be generally weaker than CSP noninterference security for nondeterministic processes, which are such that even a complete specification of their traces leaves underdetermined their refusals. This is not the case for deterministic processes, so the aforesaid security properties might in principle be equivalent as regards such processes.

However, a counterexample proving the contrary is provided by a deterministic state machine resembling systems A and B described in [4], section 3.1. This machine is proven not to be classical noninterference-secure, whereas the corresponding classical process turns out to be generalized noninterference-secure, which proves that the generalized noninterference security of a classical process is not a sufficient condition for the classical noninterference security of the associated deterministic state machine.

This result, along with the equivalence between CSP noninterference security and classical noninterference security for classical processes, is then used to demonstrate that the generalized noninterference security of the aforesaid classical process does not entail its CSP noninterference security, which proves that generalized noninterference security is actually not a sufficient condition for CSP noninterference security even in the case of deterministic processes.

The remainder of this section is dedicated to the construction of such counterexample.

datatype *g-state* = *Even* | *Odd*

datatype *g-action* = *Any* | *Count*

primrec *g-step* :: *g-state* ⇒ *g-action* ⇒ *g-state* **where**
g-step s Any = (*case s of Even* ⇒ *Odd* | *Odd* ⇒ *Even*) |
g-step s Count = *s*

primrec *g-out* :: *g-state* ⇒ *g-action* ⇒ *g-state option* **where**
g-out - Any = *None* |
g-out s Count = *Some s*

primrec *g-D* :: *g-action* ⇒ *g-level* **where**
g-D Any = *High* |
g-D Count = *Low*

definition *g*-s_0 :: *g-state* **where**
g-s_0 ≡ *Even*

lemma *g-secure-counterexample*:
g-secure (*c-process g-step g-out g*-s_0) (*c-dom g-D*)
proof (*rule g-secure-suff-implies-g-secure*, *simp add*: *g-secure-suff-def*, (*rule allI*)+,
rule impI, *rule subsetI*, *simp*, *erule exE*, (*erule conjE*)+)
fix *xps x p yps yps'*
assume *xps* @ [(*x*, *p*)] ∈ *traces* (*c-process g-step g-out g*-s_0)
hence ∃ *X*. (*xps* @ [(*x*, *p*)], *X*) ∈ *c-failures g-step g-out g*-s_0
by (*simp add*: *c-traces*)
then obtain *X* **where** (*xps* @ [(*x*, *p*)], *X*) ∈ *c-failures g-step g-out g*-s_0 **..**
hence *xps* @ [(*x*, *p*)] = *c-tr g-step g-out g*-s_0 (*map fst* (*xps* @ [(*x*, *p*)]))
by (*rule c-failures-tr*)
moreover assume *c-dom g-D* (*x*, *p*) = *High*
hence *x* = *Any* **by** (*cases x*, *simp-all add*: *c-dom-def*)
ultimately have *xps* @ [(*x*, *p*)] = *c-tr g-step g-out g*-s_0 (*map fst xps* @ [*Any*])
(**is** - = - (*?xs* @ -)) **by** *simp*
moreover assume *xps* @ *yps* ∈ *traces* (*c-process g-step g-out g*-s_0)
hence ∃ *Y*. (*xps* @ *yps*, *Y*) ∈ *c-failures g-step g-out g*-s_0
by (*simp add*: *c-traces*)
then obtain *Y* **where** (*xps* @ *yps*, *Y*) ∈ *c-failures g-step g-out g*-s_0 **..**
hence (*yps*, *Y*) ∈ *futures* (*c-process g-step g-out g*-s_0) *xps*
by (*simp add*: *c-futures-failures*)
hence *yps* = *c-tr g-step g-out* (*foldl g-step g*-s_0 *?xs*) (*map fst yps*)
(**is** - = *c-tr* - - - *?ys*) **by** (*rule c-futures-tr*)
hence *yps* =
c-tr g-step g-out (*foldl g-step* (*foldl g-step g*-s_0 (*?xs* @ [*Any*])) [*Any*]) *?ys*
(**is** - = *c-tr* - - (*foldl* - *?s* -) -) **by** (*cases foldl g-step g*-s_0 *?xs*, *simp-all*)
hence *c-tr g-step g-out ?s* [*Any*] @ *yps* = *c-tr g-step g-out ?s* ([*Any*] @ *?ys*)
(**is** *?yp* @ - = -) **by** (*simp only*: *c-tr-append*)
moreover have (*c-tr g-step g-out ?s* ([*Any*] @ *?ys*),
{(*x*, *p*). *p* ≠ *g-out* (*foldl g-step ?s* ([*Any*] @ *?ys*)) *x*})
∈ *futures* (*c-process g-step g-out g*-s_0) (*c-tr g-step g-out g*-s_0 (*?xs* @ [*Any*]))
(**is** (-, *?Y'*) ∈ -) **by** (*rule c-tr-futures*)
ultimately have (*?yp* @ *yps*, *?Y'*)
∈ *futures* (*c-process g-step g-out g*-s_0) (*xps* @ [(*x*, *p*)])
by *simp*
hence (*xps* @ (*x*, *p*) # *?yp* @ *yps*, *?Y'*) ∈ *c-failures g-step g-out g*-s_0
by (*simp add*: *c-futures-failures*)
hence ∃ *Y'*. (*xps* @ (*x*, *p*) # *?yp* @ *yps*, *Y'*) ∈ *c-failures g-step g-out g*-s_0 **..**
hence *xps* @ (*x*, *p*) # *?yp* @ *yps* ∈ *traces* (*c-process g-step g-out g*-s_0)
(**is** *?P* (*?yp* @ *yps*)) **by** (*simp add*: *c-traces*)
moreover assume *yps'* = [*yp*←*yps*. *c-dom g-D yp* = *Low*]
hence *yps'* = [*yp*←*?yp* @ *yps*. *c-dom g-D yp* = *Low*]
(**is** *?Q* (*?yp* @ *yps*)) **by** (*simp add*: *c-tr-singleton c-dom-def*)
ultimately have *?P* (*?yp* @ *yps*) ∧ *?Q* (*?yp* @ *yps*) **..**
thus ∃ *yps*. *?P yps* ∧ *?Q yps* **..**
qed

lemma *not-c-secure-counterexample*:
¬ *c-secure g-step g-out g-*s_0 *g-I g-D*
proof (*simp add*: *c-secure-def*)
  have *g-out* (*foldl g-step g-*s_0 [*Any*]) *Count* = *Some Odd*
  (**is** *?f Count* [*Any*] = -) **by** (*simp add*: *g-*s_0*-def*)
  moreover have
    g-out (*foldl g-step g-*s_0 (*c-ipurge g-I g-D* (*g-D Count*) [*Any*])) *Count* =
    Some Even
  (**is** *?g Count* [*Any*] = -) **by** (*simp add*: *g-I-def g-*s_0*-def*)
  ultimately have *?f Count* [*Any*] ≠ *?g Count* [*Any*] **by** *simp*
  thus ∃ *x xs*. *?f x xs* ≠ *?g x xs* **by** *blast*
qed

theorem *not-g-secure-implies-c-secure*:
¬ (*g-secure* (*c-process g-step g-out g-*s_0) (*c-dom g-D*) ⟶
c-secure g-step g-out g-s_0 *g-I g-D*)
proof (*simp*, *rule conjI*, *rule g-secure-counterexample*)
qed (*rule not-c-secure-counterexample*)

theorem *not-g-secure-implies-secure*:
¬ (*g-secure* (*c-process g-step g-out g-*s_0) (*c-dom g-D*) ⟶
secure (*c-process g-step g-out g-*s_0) *g-I* (*c-dom g-D*))
proof (*simp*, *rule conjI*, *rule g-secure-counterexample*)
qed (*rule notI*, *drule secure-implies-c-secure*, *erule contrapos-pp*,
rule not-c-secure-counterexample)

end

References

[1] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.

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

[3] A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. http://isabelle.in.tum.de/website-Isabelle2013-2/dist/Isabelle2013-2/doc/functions.pdf.

[4] D. McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Conference on Security and Privacy*, 1988.

[5] T. Nipkow. *A Tutorial Introduction to Structured Isar Proofs*. http://isabelle.in.tum.de/website-Isabelle2011/dist/Isabelle2011/doc/isar-overview.pdf.

[6] T. Nipkow. *Programming and Proving in Isabelle/HOL*, Dec. 2013. http://isabelle.in.tum.de/website-Isabelle2013-2/dist/Isabelle2013-2/doc/prog-prove.pdf.

[7] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Dec. 2013. http://isabelle.in.tum.de/website-Isabelle2013-2/dist/Isabelle2013-2/doc/tutorial.pdf.

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