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

December 14, 2021

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*
⟨*proof*⟩

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 $(P \mathbin{/} xs)^0$.

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*⟩

lemma *process-rule-3*: (*xs*, *Y*) ∈ *failures P* ⟹ *X* ⊆ *Y* ⟹ (*xs*, *X*) ∈ *failures P*
⟨*proof*⟩

lemma *process-rule-2-failures* [*rule-format*]:
(*xs* @ *xs'*, *X*) ∈ *failures P* ⟶ (*xs*, {}) ∈ *failures P*
⟨*proof*⟩

lemma *process-rule-2-futures*:

$(ys \mathbin{@} ys',\ Y) \in \mathit{futures}\ P\ xs \Longrightarrow (ys,\ \{\}) \in \mathit{futures}\ P\ xs$
⟨*proof*⟩

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 $(y \mathbin{\#} ys,\ Y)$, $(zs,\ Z) \in \mathit{futures}\ P\ xs$, both of the following conditions are satisfied:

- (*ipurge-tr I D* $(D\ y)$ *ys*, *ipurge-ref I D* $(D\ y)$ *ys Y*) ∈ *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 $(y \mathbin{\#} ys,\ Y)$ 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*)
  ∈ *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, $(zs, Z) \in \mathit{futures}\ P\ xs$ just in case $(xs \mathbin{@} zs, Z) \in \mathit{failures}\ P$, which by virtue of rule *process-rule-2-failures* implies that *xs* is a trace of P. Therefore, formula $(zs, Z) \in \mathit{futures}\ P\ xs$ 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*⟩
termination ⟨*proof*⟩

function *ipurge-tr* :: $('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd \Rightarrow 'a\ list \Rightarrow 'a\ list$ **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*⟩
termination ⟨*proof*⟩

definition *ipurge-ref* ::
$('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow 'd \Rightarrow 'a\ list \Rightarrow 'a\ set \Rightarrow 'a\ set$ **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* :: $'a\ process \Rightarrow ('d \times 'd)\ set \Rightarrow ('a \Rightarrow 'd) \Rightarrow bool$ **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*⟩

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*⟩

lemma *sinks-cons-nonint*:
assumes *A*: (*u*, *D x*) ∉ *I*
shows *sinks I D u* (*x # xs*) = *sinks I D u xs*
⟨*proof*⟩

lemma *sinks-empty* [*rule-format*]:
sinks I D u xs = {} ⟶ *ipurge-tr I D u xs* = *xs*
⟨*proof*⟩

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*⟩

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

primrec *c-ipurge* :: $('d \times 'd)$ *set* $\Rightarrow$ $('a \Rightarrow 'd) \Rightarrow 'd \Rightarrow 'a$ *list* $\Rightarrow 'a$ *list* **where**
c-ipurge - - - [] = [] |
c-ipurge I D u (*x* # *xs*) = (*if D x* $\in$ *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* ::
$('s \Rightarrow 'a \Rightarrow 's) \Rightarrow ('s \Rightarrow 'a \Rightarrow 'o) \Rightarrow 's \Rightarrow ('d \times 'd)$ *set* $\Rightarrow ('a \Rightarrow 'd) \Rightarrow$ *bool*
where
c-secure step out s_0 *I D* $\equiv$
$\forall$ *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* :: $('d \times 'd)$ *set* $\Rightarrow$ $('a \Rightarrow 'd) \Rightarrow 'd$ *set* $\Rightarrow 'a$ *list* $\Rightarrow 'd$ *set*
where
c-sources-aux - - U [] = *U* |
c-sources-aux I D U (*x* # *xs*) = (*if* $\exists$ *v* $\in$ *c-sources-aux I D U xs*. (*D x*, *v*) $\in$ *I*
then insert (*D x*) (*c-sources-aux I D U xs*)
else c-sources-aux I D U xs)

primrec *c-ipurge-aux* :: $('d \times 'd)$ *set* $\Rightarrow$ $('a \Rightarrow 'd) \Rightarrow 'd$ *set* $\Rightarrow 'a$ *list* $\Rightarrow 'a$ *list*
where
c-ipurge-aux - - - [] = [] |
c-ipurge-aux I D U (*x* # *xs*) = (*if D x* $\in$ *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*
⟨*proof*⟩

lemma *c-ipurge-aux-singleton*: *c-ipurge-aux I D* {*u*} *xs* = *c-ipurge I D u xs*
⟨*proof*⟩

lemma *c-sources-aux-singleton-2*:

$D\ x \in$ *c-sources-aux I D U* $[x] = (D\ x \in U \vee (\exists\, v \in U.\ (D\ x,\ v) \in I))$
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *c-sources-append-1*:
c-sources I D (*D x*) (*xs* @ $[x]$) = *c-sources I D* (*D x*) *xs*
⟨*proof*⟩

lemma *c-ipurge-append-1*:
c-ipurge I D (*D x*) (*xs* @ $[x]$) = *c-ipurge I D* (*D x*) *xs* @ $[x]$
⟨*proof*⟩

lemma *c-sources-append-2*:
(*D x*, *u*) ∉ *I* ⟹ *c-sources I D u* (*xs* @ $[x]$) = *c-sources I D u xs*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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* ⇒ '*a* ⇒ '*s*) ⇒ ('*s* ⇒ '*a* ⇒ '*o*) ⇒ '*s* ⇒ ('*a* × '*o*) *failure set*
for *step* :: '*s* ⇒ '*a* ⇒ '*s* **and** *out* :: '*s* ⇒ '*a* ⇒ '*o* **and** s_0 :: '*s* **where**
R0: ([], {(*x*, *p*). *p* ≠ *out* s_0 *x*}) ∈ *c-failures step out* s_0 |
R1: ⟦(*xps*, -) ∈ *c-failures step out* s_0; *s* = *foldl step* s_0 (*map fst xps*)⟧ ⟹
(*xps* @ [(*x*, *out s x*)], {(*y*, *p*). *p* ≠ *out* (*step s x*) *y*}) ∈ *c-failures step out* s_0 |
R2: ⟦(*xps*, *Y*) ∈ *c-failures step out* s_0; *X* ⊆ *Y*⟧ ⟹
(*xps*, *X*) ∈ *c-failures step out* s_0

definition *c-process* ::
('*s* ⇒ '*a* ⇒ '*s*) ⇒ ('*s* ⇒ '*a* ⇒ '*o*) ⇒ '*s* ⇒ ('*a* × '*o*) *process* **where**
c-process step out s_0 ≡ *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*⟩

lemma *c-process-prop-2* [*simp*]: *process-prop-2* (*c-failures step out* s_0, {})
⟨*proof*⟩

lemma *c-process-prop-3* [*simp*]: *process-prop-3* (*c-failures step out* s_0, {})
⟨*proof*⟩

lemma *c-process-prop-4* [*simp*]: *process-prop-4* (*c-failures step out* s_0, {})
⟨*proof*⟩

lemma *c-process-prop-5* [*simp*]: *process-prop-5* (*F*, {})
⟨*proof*⟩

lemma *c-process-prop-6* [*simp*]: *process-prop-6* (*F*, {})
⟨*proof*⟩

theorem *c-process-process*: (*c-failures step out* s_0, {}) ∈ *process-set*
⟨*proof*⟩

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*) ∈ *c-failures step out* s_0 ⟹ *xps* ≠ [] ⟶
snd (*last xps*) = *out* (*foldl step* s_0 (*butlast* (*map fst xps*))) (*last* (*map fst xps*))
⟨*proof*⟩

lemma *c-failures-ref*:
(*xps*, *X*) ∈ *c-failures step out* s_0 ⟹
X ⊆ {(*x*, *p*). *p* ≠ *out* (*foldl step* s_0 (*map fst xps*)) *x*}
⟨*proof*⟩

lemma *c-failures-failures*: *failures* (*c-process step out* s_0) = *c-failures step out* s_0
⟨*proof*⟩

lemma *c-futures-failures*:
(*yps*, *Y*) ∈ *futures* (*c-process step out* s_0) *xps* =
((*xps* @ *yps*, *Y*) ∈ *c-failures step out* s_0)

⟨*proof*⟩

lemma *c-traces*:
xps ∈ *traces* (*c-process step out* s_0) = (∃ *X*. (*xps*, *X*) ∈ *c-failures step out* s_0)
⟨*proof*⟩

lemma *c-refusals*:
X ∈ *refusals* (*c-process step out* s_0) *xps* = ((*xps*, *X*) ∈ *c-failures step out* s_0)
⟨*proof*⟩

lemma *c-next-events*:
xp ∈ *next-events* (*c-process step out* s_0) *xps* =
(∃ *X*. (*xps* @ [*xp*], *X*) ∈ *c-failures step out* s_0)
⟨*proof*⟩

lemma *c-traces-failures*:
xps ∈ *traces* (*c-process step out* s_0) ⟹
(*xps*, {(*x*, *p*). *p* ≠ *out* (*foldl step* s_0 (*map fst xps*)) *x*}) ∈ *c-failures step out* s_0
⟨*proof*⟩

theorem *c-process-deterministic*: *deterministic* (*c-process step out* s_0)
⟨*proof*⟩

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*⟩
termination ⟨*proof*⟩

lemma *c-tr-length*: *length* (*c-tr step out s xs*) = *length xs*
⟨*proof*⟩

lemma *c-tr-map*: *map fst* (*c-tr step out s xs*) = *xs*
⟨*proof*⟩

lemma *c-tr-singleton*: *c-tr step out s* [*x*] = [(*x*, *out s x*)]
⟨*proof*⟩

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*⟩

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*⟩

lemma *c-failures-tr*:
(*xps*, *X*) ∈ *c-failures step out* s_0 ⟹ *xps* = *c-tr step out* s_0 (*map fst xps*)
⟨*proof*⟩

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*⟩

lemma *c-tr-failures*:
(*c-tr step out* s_0 *xs*, {(*x*, *p*). *p* ≠ *out* (*foldl step* s_0 *xs*) *x*})
∈ *c-failures step out* s_0
⟨*proof*⟩

lemma *c-tr-futures*:
(*c-tr step out* (*foldl step* s_0 *xs*) *ys*,
{(*x*, *p*). *p* ≠ *out* (*foldl step* (*foldl step* s_0 *xs*) *ys*) *x*})
∈ *futures* (*c-process step out* s_0) (*c-tr step out* s_0 *xs*)
⟨*proof*⟩

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* ⇒ '*d*) ⇒ ('*a* × '*o*) ⇒ '*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)
⟨*proof*⟩

lemma *c-dom-sinks*: *sinks I (c-dom D) u xps = sinks I D u (map fst xps)*
⟨*proof*⟩

lemma *c-tr-sources*:
c-sources I (c-dom D) u (c-tr step out s xs) = c-sources I D u xs
⟨*proof*⟩

lemma *c-tr-sinks*: *sinks I (c-dom D) u (c-tr step out s xs) = sinks I D u xs*
⟨*proof*⟩

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*⟩

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*⟩

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 s₀ xs) (take n ys)), Y)
∈ futures (c-process step out s₀) (c-tr step out s₀ xs)
shows *n ∈ {..<length ys} ⟶ D (ys ! n) ∉ sinks I D u (take (Suc n) ys) ⟶*
out (foldl step (foldl step s₀ xs) (ipurge-tr I D u (take n ys))) (ys ! n) =
out (foldl step (foldl step s₀ xs) (take n ys)) (ys ! n)
⟨*proof*⟩

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 s₀ xs) (take n ys)), Y)
∈ futures (c-process step out s₀) (c-tr step out s₀ xs)
shows *ipurge-tr I (c-dom D) u (c-tr step out (foldl step s₀ xs) ys) =*
c-tr step out (foldl step s₀ xs) (ipurge-tr I D u ys)
⟨*proof*⟩

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* s_0) I (*c-dom* D)
shows *out* (*foldl step* (*foldl step* s_0 *xs*) *ys*) x =
out (*foldl step* (*foldl step* s_0 *xs*) (*c-ipurge* I D (D x) *ys*)) x
⟨*proof*⟩

theorem *secure-implies-c-secure*:
assumes S: *secure* (*c-process step out* s_0) I (*c-dom* D)
shows *c-secure step out* s_0 I D
⟨*proof*⟩

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*⟩

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) ∈ *futures* (*c-process step out* s_0) *xps* ⟶
(*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*
⟨*proof*⟩

lemma *c-secure-futures-2*:
assumes R: *refl* I **and** S: *c-secure step out* s_0 I D
shows (*yps* @ [*yp*], A) ∈ *futures* (*c-process step out* s_0) *xps* ⟹
(*yps*, Y) ∈ *futures* (*c-process step out* s_0) *xps* ⟹
(*yps* @ [*yp*], {x ∈ Y. (*c-dom* D *yp*, *c-dom* D x) ∉ I})
∈ *futures* (*c-process step out* s_0) *xps*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *g-sinks*: *sinks g-I L High xs* ⊆ {*High*}
⟨*proof*⟩

lemma *g-ipurge-tr*: *ipurge-tr g-I L High xs* = [*x*←*xs*. *L x* = *Low*]
⟨*proof*⟩

theorem *secure-implies-g-secure*:
assumes *S*: *secure P g-I L*
shows *g-secure P L*
⟨*proof*⟩

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*)
⟨*proof*⟩

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 $g\text{-}s_0 :: g\text{-}state$ **where**
$g\text{-}s_0 \equiv Even$

lemma *g-secure-counterexample*:
$g\text{-}secure\ (c\text{-}process\ g\text{-}step\ g\text{-}out\ g\text{-}s_0)\ (c\text{-}dom\ g\text{-}D)$
⟨*proof*⟩

lemma *not-c-secure-counterexample*:
$\neg\ c\text{-}secure\ g\text{-}step\ g\text{-}out\ g\text{-}s_0\ g\text{-}I\ g\text{-}D$
⟨*proof*⟩

theorem *not-g-secure-implies-c-secure*:
$\neg\ (g\text{-}secure\ (c\text{-}process\ g\text{-}step\ g\text{-}out\ g\text{-}s_0)\ (c\text{-}dom\ g\text{-}D) \longrightarrow$
$c\text{-}secure\ g\text{-}step\ g\text{-}out\ g\text{-}s_0\ g\text{-}I\ g\text{-}D)$
⟨*proof*⟩

theorem *not-g-secure-implies-secure*:
$\neg\ (g\text{-}secure\ (c\text{-}process\ g\text{-}step\ g\text{-}out\ g\text{-}s_0)\ (c\text{-}dom\ g\text{-}D) \longrightarrow$
$secure\ (c\text{-}process\ g\text{-}step\ g\text{-}out\ g\text{-}s_0)\ g\text{-}I\ (c\text{-}dom\ g\text{-}D))$
⟨*proof*⟩

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.