HOL-CSP Version 2.0

Safouan Taha Burkhart Wolff Lina Ye

December 1, 2023

Contents

Chapter 1

Context

1.1 Preface

This is a formalization in Isabelle/HOL of the work of Hoare and Roscoe on the denotational semantics of the Failure/Divergence Model of CSP. It follows essentially the presentation of CSP in Roscoe's Book "Theory and Practice of Concurrency" [12] and the semantic details in a joint Paper of Roscoe and Brooks "An improved failures model for communicating processes" [2].

The original version of this formalization, called HOL-CSP 1.0 [13], revealed minor, but omnipresent foundational errors in key concepts like the process invariant. A correction was proposed slightly before the apparition of Roscoe's book (all three authors were in e-mail contact at that time).

In recent years, a team of authors undertook the task to port HOL-CSP 1.0 to modern Isabelle/HOL and structured proof techniques. This results in the present version are called HOL-CSP 2.0.

The effort is motivated by the following assets of CSP:

- the theory is interesting in itself, and reworking its formal structure might help to make it more widely accessible, given that it is a particularly advanced example of the shallow embedding technique using the denotational semantics of a language,
- it is interesting to *compare* the ancient, imperative, ML-heavy proof style to the more recent declarative one in Isabelle/Isar; this comparison (not presented here) gives a source of empirical evidence that such proofs are more stable wrt. the constant changes in the Isabelle itself,
- the *semantic* presentation of CSP lends itself to a semantically clean and well-understood *combination* of specification languages, which represents a major step to our longterm goal of heterogenuous, yet seman-

tically clean system specifications consisting of different formalisms describing components or system aspects separately,

- the resulting HOL-CSP environment could one day be used as a tool that certifies traces of other CSP model-checkers such as FDR4 or PAT.

In contrast to HOL-CSP 1.0, which came with an own fixpoint theory partly inspired by previous work of Alberto Camilieri under HOL4 [3], it is the goal of the redesign of HOL-CSP 2.0 to reuse the HOLCF theory that emmerged from Franz Regensburgers work and was substantially extended by Brian Huffman. Thus, the footprint of the HOL-CSP 2.0 theory should be reduced drastically. Moreover, all proofs have been heavily revised or reconstructed to reflect the drastically improved state of the art of interactive theory development with Isabelle.

1.2 Introduction

DRAWN FROM THE PAPER [13]

In his invited lecture at FME'96, C.A.R. Hoare presented his view on the status quo of formal methods in industry. With respect to formal proof methods, he ruled that they "are now sufficiently advanced that a [...] formal methodologist could occasionally detect [...] obscure latent errors before they occur in practice" and asked for their publication as a possible "milestone in the acceptance of formal methods" in industry.

In this paper, we report of a larger verification effort as part of the UniForM project [7]. It revealed an obscure latent error that was not detected within a decade. It cannot be said that the object of interest is a "large software system" whose failure may "cost millions", but it is a well-known subject in the center of academic interest considered foundational for several formal methods tools: the theory of the failure- divergence model of CSP ([5], [2]). And indeed we hope that this work may further encourage the use of formal proof methods at least in the academic community working on formal methods.

Implementations of proof support for a formal method can roughly be divided into two categories. In direct tools like FDR [1], the logical rules of a method (possibly integrated into complex proof techniques) are hard-wired into the code of their implementation. Such tools tend to be difficult to modify and to formally reason about, but can possess enviable automatic proof power in specific problem domains and comfortable user interfaces.

The other category can be labelled as logical embeddings. Formal methods such as CSP or Z can be logically embedded into an LCF-style tactical theorem prover such as HOL [4] or Isabelle[8]. Coming with an open system

design going back to Milner, these provers allow for user-programmed extensions in a logically sound way. Their strength is flexibility, generality and expressiveness that makes them to symbolic programming environments.

In this paper we present a tool of the latter category (as a step towards a future combination with the former). After a brief introduction into the failure divergence semantics in the traditional CSP-literature, we will discuss the revealed problems and present a correction. Although the error is not "mathematically deep", it stings since its correction affects many definitions. It is shown that the corrected CSP still fulfils the desired algebraic laws. The addition of fixpoint-theory and specialised tactics extends the embedding in Isabelle/HOL to a formally proven consistent proof environment for CSP. Its use is demonstrated in a final example.

1.3 An Outline of Failure-Divergence Semantics

A very first approach to give denotational semantics to CSP is to view it as a kind of a regular expression. This way, it can be understood as an automata and the denotations are just the language of the automaton; this way, synchronization and concurrency can be basically understood as the construction of a product automaton with potential interleaving. The semantics becomes compositional, and internal communication between subcomponents of a component can be modeled by the concealment operator.

Hoares work [5] was strongly inspired by this initial idea. However, it became quickly clear that the simplistic automata vision is not a satisfying paradigm for all aspects of concurrency. Particularly regarding the nature of communication, where one "sends" actively information and the other "receives" it, the bi-directional product construction seems to be misleading. Furthermore, it is an obvious difference if a group of processes remains in a passive deadlock because all possible communications contradict each other, or if a group of processes is too busy with internal chatter and never reaches the point where this component is again ready for communication.

Hoare solved these apparent problems by presenting a multi-layer approach, in which the denotational models were refined more and more allowing to distinguish the above critical situations. An ingenious concept in the overall scheme is to distinguish *non-deterministic* choice from *deterministic* one [1] in order to solve the sender/receiver problem.

Hoare proposed 3 denotational semantics for CSP:

- the *trace* model, which is basically the above naive automata model not allowing to distinguish non-deterministic choice from deterministic

[1] which in itself produces problems with recursion which had to be overcome by some restrictions on its use.

one, neither to distinguish deadlock from infinite internal chatter,

- the *failure* model is able to distinguish non-deterministic choice from deterministic one by different maximum refusal sets, which is however cannot differentiate deadlock from infinite internal chatter,
- the *failure-divergence* model overcomes additionally the unresolved problem of failure model.

In the sequel, we explain these two problems in more detail, giving some motivation for the daunting complexity of the latter model. It is this complexity which finally raises general interest in a formal verification.

1.3.1 Non-Determinism

Let a and b be any two events in some set of events Σ. The two processes

$$(a \rightarrow \mathit{Stop}) \ \Box \ b \rightarrow \mathit{Stop} \tag{1}$$

and

$$(a \rightarrow \mathit{Stop}) \sqcap b \rightarrow \mathit{Stop} \tag{2}$$

cannot be distinguished under the trace semantics, in which both processes are capable of performing the same sequences of events, i.e. both have the same set of traces $\{\{\}, \{a\}, \{b\}\}$. This is because both processes can either engage in a and then *Stop*, or engage in b and then *Stop*. We would, however, like to distinguish between *a deterministic* choice of a or b (1) and *a non-deterministic* choice of a or b (2).

This can be done by considering the events that a process can refuse to engage in when these events are offered by the environment; it cannot refuse either, so we say its maximal refusal set is the set containing all elements of Σ other than a and b, written $\Sigma - \{a, b\}$, i.e. it can refuse all elements in Σ other than a or b. In the case of the non-deterministic process (2), however, we wish to express that if the environment offers the event a say, the process non-deterministically chooses either to engage in a, to refuse it and engage in b (likewise for b). We say therefore, that process (2) has two maximal refusal sets, $\Sigma - \{a\}$ and $\Sigma - \{b\}$, because it can refuse to engage in either a or b, but not both. The notion of refusal sets is in this way used to distinguish non-determinism from determinism in choices.

1.3.2 Infinite Chatter

Consider the infinite process $\mu\ x.\ a \rightarrow x$ which performs an infinite stream of a's. If one now conceals the event a in this process by writing

$$(\mu\ x.\ a \rightarrow x) \setminus a \tag{3}$$

it no longer becomes possible to distinguish the behaviour of this process from that of the deadlock process Stop. We would like to be able to make such a distinction, since the former process has clearly not stopped but is engaging in an unbounded sequence of internal actions invisible to the environment. We say the process has diverged, and introduce the notion of a divergence set to denote all sequences events that can cause a process to diverge. Hence, the process Stop is assigned the divergence set $\{\}$, since it can not diverge, whereas the process (3) above diverges on any sequence of events since the process begins to diverge immediately, i.e. its divergence set is Σ^* , where Σ^* denotes the set of all sequences with elements in Σ. Divergence is undesirable and so it is essential to be able to express it to ensure that it is avoided.

1.3.3 The Global Architecture of HOL-CSP 2.0

The global architecture of HOL-CSP 2.0 has been substantially simplified compared to HOL-CSP 1.0: the fixpoint reasoning is now entirely based on HOLCF (which meant that the continuity proofs for CSP operators had basically been re-done).

The theory `Process` establishes the basic common notions for events, traces, ticks and tickfree-ness, the type definitions for failures and divergences as well as the global constraints on them (called the "axioms" in Hoare's Book.) captured in a predicate `is_process`. On this basis, the set of failures and divergences satisfying `is_process` is turned into the type `'a process` via a type-definition (making `is_process` as the central data invariant). In the sequel, it is shown that `'a process` belongs to the type-class *cpo* stemming from *HOLCF* which makes the concepts of complete partial order, continuity, fixpoint-induction and general recursion available to all expressions of type `'a process`.

The theory `Process` also establishes the two partial orderings $P \leq P'$ for refinements and $P \sqsubseteq P'$ for the approximation on processes used to give semantics to recursion. The latter is well-known to be logically weaker than the former. Note that, unfortunately, the use of these two symbols in HOL-CSP 2.0, where the latter is already used in the *HOLCF*-theory, is just the other way round as in the literature.

Each CSP operator is described in an own theory which comprises:

- The denotational core definition in terms of a pair of Failures and Divergences

- The establishment of `is_process` for the Failures and Divergences in the range of the given operator (thus, the invariance of `is_process` for this operator)
- The proof of the projections `T` and `F` and `D` for this operator
- The proof of continuity of the operator (which is always possible except for hide if applied to infinite hide-sets).

Finally, the theory CSP contains the "Laws" of CSP, i.e. the derived rules allowing abstractly to reason over CSP processes. The overall dependency graph is shown in Figure 1.1.

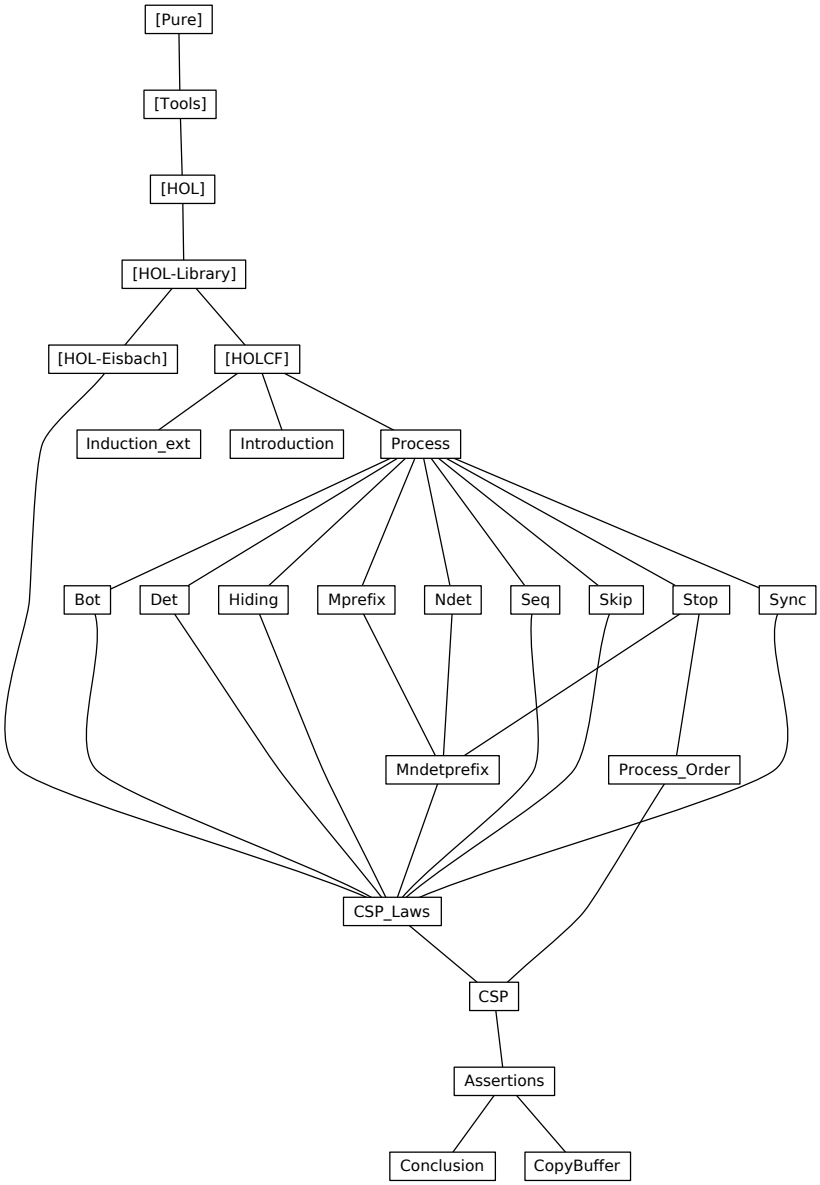

Figure 1.1: The HOL-CSP 2.0 Theory Graph

Chapter 2

The Notion of Processes

As mentioned earlier, we base the theory of CSP on HOLCF, a Isabelle/HOL library providing a theory of continuous functions, fixpoint induction and recursion.

theory *Process*
  imports *HOLCF*
begin

Since HOLCF sets the default type class to *cpo*, while our Process theory establishes links between standard types and *pcpo* types. Consequently, we reset the default type class to the default in HOL.

default-sort *type*

2.1 Pre-Requisite: Basic Traces and tick-Freeness

The denotational semantics of CSP assumes a distinguishable special event, called `tick` and written ✓, that is required to occur only in the end of traces in order to signalize successful termination of a process. (In the original text of Hoare, this treatment was more liberal and lead to foundational problems: the process invariant could not be established for the sequential composition operator of CSP; see [13] for details.)

datatype $'\alpha$ *event* = *ev* $'\alpha$ | *tick*

type-synonym $'\alpha$ *trace* = ($'\alpha$ *event*) *list*

We chose as standard ordering on traces the prefix ordering.

instantiation *list* :: (*type*) *order*
begin

definition *le-list-def* : (s::$'a$ *list*) $\leq t \longleftrightarrow (\exists\ r.\ s\ @\ r = t)$

definition *less-list-def*: (s::$'a$ *list*) $< t \longleftrightarrow s \leq t \wedge s \neq t$

lemma A : $((x::'a\ list) < y) = (x \leq y \wedge \neg\ y \leq x)$
by(*auto simp*: *le-list-def less-list-def*)

instance
proof
  fix $x\ y\ z ::'a\ list$
  show $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
    by(*auto simp*: *le-list-def less-list-def*)
  show $x \leq x$ **by**(*simp add*: *le-list-def*)
  assume A:$x \leq y$ **and** B:$y \leq z$ **thus** $x \leq z$
    apply(*insert A B*, *simp add*: *le-list-def*, *safe*)
    apply(*rename-tac r ra*, *rule-tac x=r@ra* **in** *exI*, *simp*)
    done
  assume A:$x \leq y$ **and** B:$y \leq x$ **thus** $x = y$
    by(*insert A B*, *auto simp*: *le-list-def*)
qed

end

Some facts on the prefix ordering.

lemma *nil-le*[*simp*]: $[] \leq s$
by(*induct s*, *simp-all*, *auto simp*: *le-list-def*)

lemma *nil-le2*[*simp*]: $s \leq [] = (s = [])$
by(*induct s*, *auto simp*:*le-list-def*)

lemma *nil-less*[*simp*]: $\neg\ t < []$
by(*simp add*: *less-list-def*)

lemma *nil-less2*[*simp*]: $[] < t\ @\ [a]$
by(*simp add*: *less-list-def*)

lemma *less-self*[*simp*]: $t < t@[a]$
by(*simp add*:*less-list-def le-list-def*)

lemma *le-length-mono*: $s \leq t \Longrightarrow length\ s \leq length\ t$
by(*auto simp*: *le-list-def*)

lemma *less-length-mono*: $s < t \Longrightarrow length\ s < length\ t$
by(*auto simp*: *less-list-def le-list-def*)

lemma *less-cons*: $s < t \Longrightarrow a\ \#\ s < a\ \#\ t$
by (*simp add*: *le-list-def less-list-def*)

lemma *less-append*: $s < t \Longrightarrow a\ @\ s < a\ @\ t$
by (*simp add*: *le-list-def less-list-def*)

lemma *less-tail*: $s \neq [] \Longrightarrow s < t \Longrightarrow tl\ s < tl\ t$

by (*auto simp add*: *le-list-def less-list-def*)

— should be in the List library
lemma *list-nonMt-append*: $s \neq [] \Longrightarrow \exists\ a\ t.\ s = t\ @\ [a]$
by(*erule rev-mp*,*induct s*,*simp-all*,*case-tac s* = [],*auto*)

lemma *append-eq-first-pref-spec*[*rule-format*]: $s\ @\ t = r\ @\ [x] \wedge t \neq [] \longrightarrow s \leq r$
  apply(*rule-tac x*=*s* **in** *spec*)
  apply(*induct r*,*auto*)
    apply(*erule rev-mp*)
    apply(*rename-tac xa*, *rule-tac list*=*xa* **in** *list.induct*, *simp-all*)
  apply(*simp add*: *le-list-def*)
  apply(*drule list-nonMt-append*, *auto*)
  done

lemma *prefixes-fin*: *let prefixes* = $\{t.\ \exists\, t2.\ x = t\ @\ t2\}$ *in finite prefixes* $\wedge$ *card prefixes* = *length x* + *1*
proof(*induct x*, *simp*)
  case (*Cons a x*)
  hence *A*:*finite* $\{t.\ (\exists\, t2.\ x = t\ @\ t2)\}$ **using** *not-add-less2* **by** *fastforce*
  have *B*:*inj-on Cons UNIV* **by** (*metis injI list.inject*)
  from *Cons A B inj-on-iff-eq-card* **have** *C*:*card* $((\lambda x.\ a\#x)`\{t.\ (\exists\, t2.\ x = t\ @\ t2)\})$ = *length x* + *1*
    by *fastforce*
  have *D*:$\{t.\ \exists\, t2.\ a\ \#\ x = t\ @\ t2\} = \{[]\} \cup (\lambda x.\ a\#x)`\{t.\ (\exists\, t2.\ x = t\ @\ t2)\}$
    by(*intro set-eqI iffI*, *auto simp add*:*Cons-eq-append-conv*)
  from *C D card-insert-if*[*of* $(\lambda x.\ a\#x)`\{t.\ (\exists\, t2.\ x = t\ @\ t2)\}$] **show** *?case*
   by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-eq-plus1 Un-insert-left finite.insertI*

      finite-imageI image-iff list.distinct(*1*) *list.size*(*4*) *local.A sup-bot.left-neutral*)
qed

lemma *sublists-fin*: *finite* $\{t.\ \exists\, t1\ t2.\ x = t1\ @\ t\ @\ t2\}$
  proof(*induct x*, *simp*)
    case (*Cons a x*)
    have $\{t.\ \exists\, t1\ t2.\ a\ \#\ x = t1\ @\ t\ @\ t2\} \subseteq \{t.\ \exists\, t1\ t2.\ x = t1\ @\ t\ @\ t2\} \cup \{t.\ \exists\, t2.\ a\#x = t\ @\ t2\}$
      apply(*auto*) **by** (*metis Cons-eq-append-conv*)
    with *Cons prefixes-fin*[*of a*#*x*] **show** *?case* **by** (*meson finite-UnI finite-subset*)
  qed

lemma *suffixes-fin*: *finite* $\{t.\ \exists\, t1.\ x = t1\ @\ t\}$
  apply(*subgoal-tac* $\{t.\ \exists\, t1.\ x = t1\ @\ t\} \subseteq \{t.\ \exists\, t1\ t2.\ x = t1\ @\ t\ @\ t2\}$)
  using *infinite-super sublists-fin* **apply** *blast*
  by *blast*

For the process invariant, it is a key element to reduce the notion of traces to traces that may only contain one tick event at the very end. This is captured by the definition of the predicate `front_tickFree` and its stronger version

`tickFree`. Here is the theory of this concept.

definition *tickFree* :: $'\alpha$ *trace* $\Rightarrow$ *bool*
where *tickFree s* = (*tick* $\notin$ *set s*)

definition *front-tickFree* :: $'\alpha$ *trace* $\Rightarrow$ *bool*
where *front-tickFree s* = (*s* =[] $\vee$ *tickFree*(*tl*(*rev s*)))

lemma *tickFree-Nil* [*simp*]: *tickFree* []
by(*simp add*: *tickFree-def*)

lemma *tickFree-Cons* [*simp*]: *tickFree* (*a* # *t*) = (*a* $\neq$ *tick* $\wedge$ *tickFree t*)
by(*auto simp add*: *tickFree-def*)

lemma *tickFree-tl* : [|*s* ~= [] ; *tickFree s*|] ==> *tickFree*(*tl s*)
by(*case-tac s*, *simp-all*)

lemma *tickFree-append*[*simp*]: *tickFree*(*s*@*t*) = (*tickFree s* $\wedge$ *tickFree t*)
by(*simp add*: *tickFree-def member-def*)

lemma *non-tickFree-tick* [*simp*]: $\neg$ *tickFree* [*tick*]
by(*simp add*: *tickFree-def*)

lemma *non-tickFree-implies-nonMt*: $\neg$ *tickFree s* $\Longrightarrow$ *s* $\neq$ []
by(*simp add*:*tickFree-def*,*erule rev-mp*, *induct s*, *simp-all*)

lemma *tickFree-rev* : *tickFree*(*rev t*) = (*tickFree t*)
by(*simp add*: *tickFree-def member-def*)

lemma *front-tickFree-Nil*[*simp*]: *front-tickFree* []
by(*simp add*: *front-tickFree-def*)

lemma *front-tickFree-single*[*simp*]: *front-tickFree* [*a*]
by(*simp add*: *front-tickFree-def*)

lemma *tickFree-implies-front-tickFree*: *tickFree s* $\Longrightarrow$ *front-tickFree s*
apply(*simp add*: *tickFree-def front-tickFree-def member-def*,*safe*)
apply(*erule contrapos-np*, *simp*,(*erule rev-mp*)+)
apply(*rule-tac xs*=*s* **in** *List.rev-induct*,*simp-all*)
done

lemma *front-tickFree-charn*: *front-tickFree s* = (*s* = [] $\vee$ ($\exists$ *a t*. *s* = *t* @ [*a*] $\wedge$ *tickFree t*))
apply(*simp add*: *front-tickFree-def*)
apply(*cases s*=[], *simp-all*)
apply(*drule list-nonMt-append*, *auto simp*: *tickFree-rev*)
done

lemma *front-tickFree-implies-tickFree*: *front-tickFree* (*t* @ [*a*]) $\Longrightarrow$ *tickFree t*
by(*simp add*: *tickFree-def front-tickFree-def member-def*)

lemma *tickFree-implies-front-tickFree-single*: *tickFree t* $\Longrightarrow$ *front-tickFree* (*t* @ [*a*])
by(*simp add*:*front-tickFree-charn*)

lemma *nonTickFree-n-frontTickFree*: $[\![\neg$ *tickFree s*; *front-tickFree s* $]\!] \Longrightarrow \exists\, t.\ s =$ *t* @ [*tick*]
apply(*frule non-tickFree-implies-nonMt*)
apply(*drule front-tickFree-charn*[*THEN iffD1*], *auto*)
done

lemma *front-tickFree-dw-closed* : *front-tickFree* (*s* @ *t*) $\Longrightarrow$ *front-tickFree s*
apply(*erule rev-mp*, *rule-tac x*= *s* **in** *spec*)
apply(*rule-tac xs*=*t* **in** *List.rev-induct*, *simp*, *safe*)
apply(*rename-tac x xs xa*)
apply(*simp only*: *append-assoc*[*symmetric*])
apply(*rename-tac x xs xa*, *erule-tac x*=*xa* @ *xs* **in** *all-dupE*)
apply(*drule front-tickFree-implies-tickFree*)
apply(*erule-tac x*=*xa* **in** *allE*, *auto*)
apply(*auto dest*!:*tickFree-implies-front-tickFree*)
done

lemma *front-tickFree-append*: $[\![$ *tickFree s*; *front-tickFree t*$]\!] \Longrightarrow$ *front-tickFree* (*s* @ *t*)
apply(*drule front-tickFree-charn*[*THEN iffD1*], *auto*)
apply(*erule tickFree-implies-front-tickFree*)
apply(*subst append-assoc*[*symmetric*])
apply(*rule tickFree-implies-front-tickFree-single*)
apply(*auto intro*: *tickFree-append*)
done

lemma *front-tickFree-mono*: *front-tickFree* (*t* @ *r*) $\wedge$ *r* $\neq$ [] $\longrightarrow$ *tickFree t* $\wedge$ *front-tickFree r*
by(*metis append-assoc append-butlast-last-id front-tickFree-charn*
front-tickFree-implies-tickFree tickFree-append)

2.2 Basic Types, Traces, Failures and Divergences

type-synonym $'\alpha$ *refusal* = ($'\alpha$ *event*) *set*
type-synonym $'\alpha$ *failure* = $'\alpha$ *trace* $\times$ $'\alpha$ *refusal*
type-synonym $'\alpha$ *divergence* = $'\alpha$ *trace set*
type-synonym $'\alpha$ *process*$_0$ = $'\alpha$ *failure set* $\times$ $'\alpha$ *divergence*

definition *FAILURES* :: $'\alpha$ *process*$_0$ $\Rightarrow$ ($'\alpha$ *failure set*)
where *FAILURES P* = *fst P*

definition *TRACES* :: $'\alpha$ *process*$_0$ $\Rightarrow$ ($'\alpha$ *trace set*)
where *TRACES P* = $\{$*tr*. $\exists$ *a*. *a* $\in$ *FAILURES P* $\wedge$ *tr* = *fst a*$\}$

definition *DIVERGENCES* :: $'\alpha$ *process*$_0$ $\Rightarrow$ $'\alpha$ *divergence*
where *DIVERGENCES P* = *snd P*

definition *REFUSALS* :: $'\alpha$ *process*$_0$ $\Rightarrow$ ($'\alpha$ *refusal set*)
where *REFUSALS P* = {*ref*. $\exists$ *F*. *F* $\in$ *FAILURES P* $\wedge$ *F* = ([],*ref*)}

2.3 The Process Type Invariant

definition *is-process* :: $'\alpha$ *process*$_0$ $\Rightarrow$ *bool* **where**
is-process P =
(([],{}) $\in$ *FAILURES P* $\wedge$
($\forall$ *s X*. (*s*,*X*) $\in$ *FAILURES P* $\longrightarrow$ *front-tickFree s*) $\wedge$
($\forall$ *s t* . (*s*@*t*,{}) $\in$ *FAILURES P* $\longrightarrow$ (*s*,{}) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X Y*. (*s*,*Y*) $\in$ *FAILURES P* & *X* <= *Y* $\longrightarrow$ (*s*,*X*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X Y*. (*s*,*X*) $\in$ *FAILURES P* $\wedge$
($\forall$ *c*. *c* $\in$ *Y* $\longrightarrow$ ((*s*@[*c*],{})$\notin$*FAILURES P*)) $\longrightarrow$
(*s*,*X* $\cup$ *Y*)$\in$*FAILURES P*) $\wedge$
($\forall$ *s X*. (*s*@[*tick*],{}) : *FAILURES P* $\longrightarrow$ (*s*,*X*$-${*tick*}) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s t*. *s* $\in$ *DIVERGENCES P* $\wedge$ *tickFree s* $\wedge$ *front-tickFree t*
$\longrightarrow$ *s*@*t* $\in$ *DIVERGENCES P*) $\wedge$
($\forall$ *s X*. *s* $\in$ *DIVERGENCES P* $\longrightarrow$ (*s*,*X*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s*. *s* @ [*tick*] : *DIVERGENCES P* $\longrightarrow$ *s* $\in$ *DIVERGENCES P*))

lemma *is-process-spec*:
is-process P =
(([],{}) $\in$ *FAILURES P* $\wedge$
($\forall$ *s X*. (*s*,*X*) $\in$ *FAILURES P* $\longrightarrow$ *front-tickFree s*) $\wedge$
($\forall$ *s t* . (*s* @ *t*,{}) $\notin$ *FAILURES P* $\vee$ (*s*,{}) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X Y*. (*s*,*Y*) $\notin$ *FAILURES P* $\vee$ $\neg$(*X*$\subseteq$*Y*) | (*s*,*X*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X Y*.(*s*,*X*) $\in$ *FAILURES P* $\wedge$
($\forall$ *c*. *c* $\in$ *Y* $\longrightarrow$ ((*s*@[*c*],{}) $\notin$ *FAILURES P*)) $\longrightarrow$(*s*,*X* $\cup$ *Y*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X*. (*s*@[*tick*],{}) $\in$ *FAILURES P* $\longrightarrow$ (*s*,*X* $-$ {*tick*}) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s t*. *s* $\notin$ *DIVERGENCES P* $\vee$ $\neg$*tickFree s* $\vee$ $\neg$*front-tickFree t*
$\vee$ *s* @ *t* $\in$ *DIVERGENCES P*) $\wedge$
($\forall$ *s X*. *s* $\notin$ *DIVERGENCES P* $\vee$ (*s*,*X*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s*. *s* @ [*tick*] $\notin$ *DIVERGENCES P* $\vee$ *s* $\in$ *DIVERGENCES P*))
by(*simp only*: *is-process-def HOL.nnf-simps*(*1*) *HOL.nnf-simps*(*3*) [*symmetric*]

HOL.imp-conjL[*symmetric*])

lemma *Process-eqI* :
assumes *A*: *FAILURES P* = *FAILURES Q*
assumes *B*: *DIVERGENCES P* = *DIVERGENCES Q*
shows (*P*::$'\alpha$ *process*$_0$) = *Q*
apply(*insert A B*, *unfold FAILURES-def DIVERGENCES-def*)
apply(*rule-tac t*=*P* **in** *surjective-pairing*[*symmetric*,*THEN subst*])

apply(*rule-tac t=Q* **in** *surjective-pairing*[*symmetric*,*THEN subst*])
apply(*simp*)
done

lemma *process-eq-spec*:
$((P::'a\ process_0) = Q) = (FAILURES\ P = FAILURES\ Q \wedge DIVERGENCES\ P = DIVERGENCES\ Q)$
apply(*auto simp*: *FAILURES-def DIVERGENCES-def*)
apply(*rule-tac t=P* **in** *surjective-pairing*[*symmetric*,*THEN subst*])
apply(*rule-tac t=Q* **in** *surjective-pairing*[*symmetric*,*THEN subst*])
apply(*simp*)
done

lemma *process-surj-pair*: $(FAILURES\ P, DIVERGENCES\ P) = P$
by(*auto simp*:*FAILURES-def DIVERGENCES-def*)

lemma *Fa-eq-imp-Tr-eq*: $FAILURES\ P = FAILURES\ Q \Longrightarrow TRACES\ P = TRACES\ Q$
by(*auto simp*:*FAILURES-def DIVERGENCES-def TRACES-def*)

lemma *is-process1*: $is\text{-}process\ P \Longrightarrow ([],\{\}) \in FAILURES\ P$
by(*auto simp*: *is-process-def*)

lemma *is-process2*: $is\text{-}process\ P \Longrightarrow \forall\ s\ X.\ (s,X) \in FAILURES\ P \longrightarrow front\text{-}tickFree\ s$
by(*simp only*: *is-process-spec*, *metis*)

lemma *is-process3*: $is\text{-}process\ P \Longrightarrow \forall\ s\ t.\ (s\ @\ t,\{\}) \in FAILURES\ P \longrightarrow (s, \{\}) \in FAILURES\ P$
by(*simp only*: *is-process-spec*, *metis*)

lemma *is-process3-S-pref*: $[\![is\text{-}process\ P;\ (t, \{\}) \in FAILURES\ P;\ s \leq t]\!] \Longrightarrow (s, \{\}) \in FAILURES\ P$
by(*auto simp*: *le-list-def intro*: *is-process3* [*rule-format*])

lemma *is-process4*: $is\text{-}process\ P \Longrightarrow \forall s\ X\ Y.\ (s,\ Y) \notin FAILURES\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in FAILURES\ P$
by(*simp only*: *is-process-spec*, *simp*)

lemma *is-process4-S*: $[\![is\text{-}process\ P;\ (s,\ Y) \in FAILURES\ P;\ X \subseteq Y]\!] \Longrightarrow (s,\ X) \in FAILURES\ P$
by(*drule is-process4*, *auto*)

lemma *is-process4-S1*: $[\![is\text{-}process\ P;\ x \in FAILURES\ P;\ X \subseteq snd\ x]\!] \Longrightarrow (fst\ x, X) \in FAILURES\ P$
by(*drule is-process4-S*, *auto*)

lemma *is-process5*:

is-process P $\Longrightarrow$
$\forall$ *sa X Y*. (*sa*, *X*) $\in$ *FAILURES P* $\wedge$ ($\forall$ *c*. *c* $\in$ *Y* $\longrightarrow$ (*sa* @ [*c*], {}) $\notin$ *FAILURES P*) $\longrightarrow$ (*sa*, *X* $\cup$ *Y*) $\in$ *FAILURES P*
by(*drule is-process-spec*[*THEN iffD1*],*metis*)

lemma *is-process5-S*:
$[\![$*is-process P*; (*sa*, *X*) $\in$ *FAILURES P*; $\forall$ *c*. *c* $\in$ *Y* $\longrightarrow$ (*sa* @ [*c*], {}) $\notin$ *FAILURES P*$]\!] \Longrightarrow$ (*sa*, *X* $\cup$ *Y*) $\in$ *FAILURES P*
by(*drule is-process5*, *metis*)

lemma *is-process5-S1*:
$[\![$*is-process P*; (*sa*, *X*) $\in$ *FAILURES P*; (*sa*, *X* $\cup$ *Y*) $\notin$ *FAILURES P*$]\!] \Longrightarrow \exists$ *c*. *c* $\in$ *Y* $\wedge$ (*sa* @ [*c*], {}) $\in$ *FAILURES P*
by(*erule contrapos-np*, *drule is-process5-S*, *simp-all*)

lemma *is-process6*: *is-process P* $\Longrightarrow$ $\forall$ *s X*. (*s*@[*tick*],{}) $\in$ *FAILURES P* $\longrightarrow$ (*s*,*X*$-${*tick*}) $\in$ *FAILURES P*
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process6-S*: $[\![$*is-process P* ;(*s*@[*tick*],{}) $\in$ *FAILURES P*$]\!] \Longrightarrow$ (*s*,*X*$-${*tick*}) $\in$ *FAILURES P*
by(*drule is-process6*, *metis*)

lemma *is-process7*:
is-process P $\Longrightarrow \forall$ *s t*. *s* $\notin$ *DIVERGENCES P* $\vee \neg$ *tickFree s* $\vee \neg$ *front-tickFree t* $\vee$ *s* @ *t* $\in$ *DIVERGENCES P*
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process7-S*:
$[\![$ *is-process P*;*s* : *DIVERGENCES P*;*tickFree s*;*front-tickFree t*$]\!]$
$\Longrightarrow$ *s* @ *t* $\in$ *DIVERGENCES P*
by(*drule is-process7*, *metis*)

lemma *is-process8*: *is-process P* $\Longrightarrow \forall$ *s X*. *s* $\notin$ *DIVERGENCES P* $\vee$ (*s*,*X*) $\in$ *FAILURES P*
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process8-S*: $[\![$ *is-process P*; *s* $\in$ *DIVERGENCES P* $]\!] \Longrightarrow$ (*s*,*X*) $\in$ *FAILURES P*
by(*drule is-process8*, *metis*)

lemma *is-process9*: *is-process P* $\Longrightarrow \forall$ *s*. *s*@[*tick*] $\notin$ *DIVERGENCES P* $\vee$ *s* $\in$ *DIVERGENCES P*
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process9-S*: $[\![$ *is-process P*;*s*@[*tick*] $\in$ *DIVERGENCES P* $]\!] \Longrightarrow$ *s* $\in$ *DIVERGENCES P*
by(*drule is-process9*, *metis*)

lemma *Failures-implies-Traces*: $[\![$*is-process P*; $(s,\ X) \in$ *FAILURES P*$]\!] \Longrightarrow s \in$ *TRACES P*
by(*simp add*: *TRACES-def*, *metis*)

lemma *is-process5-sing*:
$[\![$ *is-process P* ; $(s,\{x\}) \notin$ *FAILURES P*;$(s,\{\}) \in$ *FAILURES P*$]\!] \Longrightarrow (s$ @ $[x],\{\}) \in$ *FAILURES P*
by(*drule-tac X*={} **in** *is-process5-S1*, *auto*)

lemma *is-process5-singT*:
$[\![$ *is-process P* ; $(s,\{x\}) \notin$ *FAILURES P*;$(s,\{\}) \in$ *FAILURES P*$]\!] \Longrightarrow s$ @ $[x] \in$ *TRACES P*
apply(*drule is-process5-sing*, *auto*)
by(*simp add*: *TRACES-def*, *auto*)

lemma *front-trace-is-tickfree*:
assumes *A*: *is-process P* **and** *B*: $(t$ @ $[tick],X) \in$ *FAILURES P*
shows *tickFree t*
proof −
have *C*: *front-tickFree*(t @ $[tick]$) **by**(*insert A B*, *drule is-process2*, *metis*)
show *?thesis* **by**(*rule front-tickFree-implies-tickFree*[*OF C*])
qed

lemma *trace-with-Tick-implies-tickFree-front* : $[\![$*is-process P*; t @ $[tick] \in$ *TRACES P*$]\!] \Longrightarrow$ *tickFree t*
by(*auto simp*: *TRACES-def intro*: *front-trace-is-tickfree*)

2.4 The Abstraction to the process-Type

typedef
$'\alpha$ *process* = $\{p$ :: $'\alpha$ *process*$_0$. *is-process p*$\}$
proof −
have $(\{(s,\ X).\ s = []\},\{\}) \in \{p{::}'\alpha$ *process*$_0$. *is-process p*$\}$
by(*simp add*: *is-process-def front-tickFree-def*
FAILURES-def TRACES-def DIVERGENCES-def)
thus *?thesis* **by** *auto*
qed

definition *Failures* :: $'\alpha$ *process* $\Rightarrow$ ($'\alpha$ *failure set*) ($\mathcal{F}$)
where $\mathcal{F}$ *P* = *FAILURES* (*Rep-process P*)

definition *Traces* :: $'\alpha$ *process* $\Rightarrow$ ($'\alpha$ *trace set*) ($\mathcal{T}$)
where $\mathcal{T}$ *P* = *TRACES* (*Rep-process P*)

definition *Divergences* :: $'\alpha$ *process* $\Rightarrow$ $'\alpha$ *divergence* ($\mathcal{D}$)
where $\mathcal{D}$ *P* = *DIVERGENCES* (*Rep-process P*)

lemmas *D-def* = *Divergences-def*

definition R :: $'\alpha$ *process* $\Rightarrow$ ($'\alpha$ *refusal set*) ($\mathcal{R}$)
where $\mathcal{R}$ P = *REFUSALS* (*Rep-process* P)

lemma *is-process-Rep* : *is-process* (*Rep-process* P)
by(*rule-tac P=is-process* **in** *CollectD*, *rule Rep-process*)

lemma *Process-spec*: *Abs-process* ($\mathcal{F}$ P, $\mathcal{D}$ P) = P
by(*simp add*: *Failures-def FAILURES-def D-def*
DIVERGENCES-def Rep-process-inverse)

lemma *Process-eq-spec*: (P = Q) = ($\mathcal{F}$ P = $\mathcal{F}$ Q $\wedge$ $\mathcal{D}$ P = $\mathcal{D}$ Q)
apply(*rule iffI*,*simp*)
apply(*rule-tac t=P* **in** *Process-spec*[*THEN subst*])
apply(*rule-tac t=Q* **in** *Process-spec*[*THEN subst*])
by *simp*

lemma *Process-eq-spec-optimized*: (P = Q) = ($\mathcal{D}$ P = $\mathcal{D}$ Q $\wedge$ ($\mathcal{D}$ P = $\mathcal{D}$ Q $\longrightarrow$ $\mathcal{F}$ P = $\mathcal{F}$ Q))
using *Process-eq-spec* **by** *auto*

lemma *is-processT*:
([],{}) $\in$ $\mathcal{F}$ P $\wedge$
($\forall$ s X. (s,X) $\in$ $\mathcal{F}$ P $\longrightarrow$ *front-tickFree* s) $\wedge$
($\forall$ s t .(s@t,{}) $\in$ $\mathcal{F}$ P $\longrightarrow$ (s,{}) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s X Y.(s,Y) $\in$ $\mathcal{F}$ P $\wedge$ ($X$$\subseteq$$Y$) $\longrightarrow$ (s,X) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s X Y.(s,X) $\in$ $\mathcal{F}$ P $\wedge$ ($\forall$ c. c $\in$ Y $\longrightarrow$((s@[c],{})$\notin$ $\mathcal{F}$ P)) $\longrightarrow$ (s,X $\cup$ Y) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s X. (s@[*tick*],{}) $\in$ $\mathcal{F}$ P $\longrightarrow$ (s, X−{*tick*}) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s t. s $\in$ $\mathcal{D}$ P $\wedge$ *tickFree* s $\wedge$ *front-tickFree* t $\longrightarrow$ s @ t $\in$ $\mathcal{D}$ P) $\wedge$
($\forall$ s X. s $\in$ $\mathcal{D}$ P $\longrightarrow$ (s,X) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s. s@[*tick*] $\in$ $\mathcal{D}$ P $\longrightarrow$ s $\in$ $\mathcal{D}$ P)
apply(*simp only*: *Failures-def D-def Traces-def*)
apply(*rule is-process-def*[*THEN iffD1*])
apply(*rule is-process-Rep*)
done

lemma *process-charn*:
([], {}) $\in$ $\mathcal{F}$ P $\wedge$
($\forall$ s X. (s, X) $\in$ $\mathcal{F}$ P $\longrightarrow$ *front-tickFree* s) $\wedge$
($\forall$ s t. (s @ t, {}) $\notin$ $\mathcal{F}$ P $\vee$ (s, {}) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s X Y. (s, Y) $\notin$ $\mathcal{F}$ P $\vee$ $\neg$ X $\subseteq$ Y $\vee$ (s, X) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s X Y. (s, X) $\in$ $\mathcal{F}$ P $\wedge$ ($\forall$ c. c $\in$ Y $\longrightarrow$ (s @ [c], {}) $\notin$ $\mathcal{F}$ P) $\longrightarrow$ (s, X $\cup$ Y) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s X. (s @ [*tick*], {}) $\in$ $\mathcal{F}$ P $\longrightarrow$ (s, X − {*tick*}) $\in$ $\mathcal{F}$ P) $\wedge$
($\forall$ s t. s $\notin$ $\mathcal{D}$ P $\vee$ $\neg$ *tickFree* s $\vee$ $\neg$ *front-tickFree* t $\vee$ s @ t $\in$ $\mathcal{D}$ P) $\wedge$

$(\forall s\ X.\ s \notin \mathcal{D}\ P \vee (s,\ X) \in \mathcal{F}\ P) \wedge (\forall s.\ s\ @\ [tick] \notin \mathcal{D}\ P \vee s \in \mathcal{D}\ P)$
proof −
{
have A: $(\forall s\ t.\ (s\ @\ t,\ \{\}) \notin \mathcal{F}\ P \vee\ (s,\ \{\}) \in \mathcal{F}\ P) =$
$(\forall s\ t.\ (s\ @\ t,\ \{\}) \in \mathcal{F}\ P \longrightarrow (s,\ \{\}) \in \mathcal{F}\ P)$
by *metis*

have B : $(\forall s\ X\ Y.\ (s,\ Y) \notin \mathcal{F}\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in \mathcal{F}\ P)\ =$
$(\forall s\ X\ Y.\ (s,\ Y) \in \mathcal{F}\ P \wedge X \subseteq Y \longrightarrow (s,\ X) \in \mathcal{F}\ P)$
by *metis*

have C : $(\forall s\ t.\ s \notin \mathcal{D}\ P \vee \neg\ tickFree\ s \vee$
$\neg\ front\text{-}tickFree\ t \vee s\ @\ t \in \mathcal{D}\ P)\ =$
$(\forall s\ t.\ s \in \mathcal{D}\ P \wedge tickFree\ s \wedge front\text{-}tickFree\ t \longrightarrow s\ @\ t \in \mathcal{D}\ P)$
by *metis*

have D:$(\forall s\ X.\ s \notin \mathcal{D}\ P \vee (s,\ X) \in \mathcal{F}\ P) = (\forall s\ X.\ s \in \mathcal{D}\ P \longrightarrow (s,\ X) \in \mathcal{F}$
$P)$
by *metis*

have E:$(\forall s.\ s\ @\ [tick] \notin \mathcal{D}\ P \vee s \in \mathcal{D}\ P) =$
$(\forall s.\ s\ @\ [tick] \in \mathcal{D}\ P \longrightarrow s \in \mathcal{D}\ P)$
by *metis*

note $A\ B\ C\ D\ E$
}
note $a = this$
then
show *?thesis*
apply(*simp only*: a)
apply(*rule is-processT*)
done
qed

split of `is_processT`:

lemma *is-processT1*: $([],\ \{\}) \in \mathcal{F}\ P$
by(*simp add*:*process-charn*)

lemma *is-processT2*: $\forall s\ X.\ (s,\ X) \in \mathcal{F}\ P \longrightarrow front\text{-}tickFree\ s$
by(*simp add*:*process-charn*)

lemma *is-processT2-TR* : $\forall s.\ s \in \mathcal{T}\ P \longrightarrow front\text{-}tickFree\ s$
apply(*simp add*: *Failures-def* [*symmetric*] *Traces-def TRACES-def*, *safe*)
apply (*drule is-processT2*[*rule-format*], *assumption*)
done

lemma *is-proT2*:
assumes A : $(s,\ X) \in \mathcal{F}\ P$ **and** B : $s \neq []$
shows $tick \notin set\ (tl\ (rev\ s))$

proof −
  have *C*: *front-tickFree s* **by**(*insert A B*, *simp add*: *is-processT2*)
  show *?thesis*
  by(*insert C*,*simp add*: *B tickFree-def front-tickFree-def*)
qed

lemma *is-processT3* : $\forall s\ t.\ (s\ @\ t,\ \{\}) \in \mathcal{F}\ P \longrightarrow (s,\ \{\}) \in \mathcal{F}\ P$
by(*simp only*: *process-charn HOL.nnf-simps*(*3*), *simp*)

lemma *is-processT3-S-pref* :
$[\![(t,\ \{\}) \in \mathcal{F}\ P;\ s \leq t]\!] \Longrightarrow (s,\ \{\}) \in \mathcal{F}\ P$
apply(*simp only*: *le-list-def*, *safe*)
apply(*erule is-processT3*[*rule-format*])
done

lemma *is-processT4* : $\forall s\ X\ Y.\ (s,\ Y) \in \mathcal{F}\ P \wedge X \subseteq Y \longrightarrow (s,\ X) \in \mathcal{F}\ P$
by(*insert process-charn* [*of P*], *metis*)

lemma *is-processT4-S1* : $[\![x \in \mathcal{F}\ P;\ X \subseteq snd\ x]\!] \Longrightarrow (fst\ x,\ X) \in \mathcal{F}\ P$
apply(*rule-tac* $Y = snd\ x$ **in** *is-processT4*[*rule-format*])
apply *simp*
done

lemma *is-processT5*:
$\forall s\ X\ Y.\ (s,\ X) \in \mathcal{F}\ P \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin \mathcal{F}\ P) \longrightarrow (s,\ X \cup Y) \in \mathcal{F}\ P$
by(*simp add*: *process-charn*)

lemma *is-processT5-S1*:
$(s,\ X) \in \mathcal{F}\ P \Longrightarrow (s,\ X \cup Y) \notin \mathcal{F}\ P \Longrightarrow \exists c.\ c \in Y \wedge (s\ @\ [c],\ \{\}) \in \mathcal{F}\ P$
by(*erule contrapos-np*, *simp add*: *is-processT5*[*rule-format*])

lemma *is-processT5-S2*: $(s,\ X) \in \mathcal{F}\ P \Longrightarrow (s\ @\ [c],\ \{\}) \notin \mathcal{F}\ P \Longrightarrow (s,\ X \cup \{c\}) \in \mathcal{F}\ P$
by(*rule is-processT5*[*rule-format*,*OF conjI*], *metis*, *safe*)

lemma *is-processT5-S2a*: $(s,\ X) \in \mathcal{F}\ P \Longrightarrow (s,\ X \cup \{c\}) \notin \mathcal{F}\ P \Longrightarrow (s\ @\ [c],\ \{\}) \in \mathcal{F}\ P$
apply(*erule contrapos-np*)
apply(*rule is-processT5-S2*)
apply(*simp-all*)
done

lemma *is-processT5-S3*:
assumes *A*: $(s,\ \{\}) \in \mathcal{F}\ P$
and *B*: $(s\ @\ [c],\ \{\}) \notin \mathcal{F}\ P$
shows $(s,\ \{c\}) \in \mathcal{F}\ P$
proof −

have C : $\{c\} = (\{\}\ Un\ \{c\})$ **by** *simp*
show *?thesis*
by(*subst* C, *rule is-processT5-S2*, *simp-all add*: A B)
qed

lemma *is-processT5-S4*: $(s, \{\}) \in \mathcal{F}\ P \Longrightarrow (s, \{c\}) \notin \mathcal{F}\ P \Longrightarrow (s\ @\ [c], \{\}) \in \mathcal{F}\ P$
by(*erule contrapos-np*, *simp add*: *is-processT5-S3*)

lemma *is-processT5-S5*:
$(s, X) \in \mathcal{F}\ P \Longrightarrow \forall c.\ c \in Y \longrightarrow (s, X \cup \{c\}) \notin \mathcal{F}\ P \Longrightarrow \forall c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \in \mathcal{F}\ P$
by(*erule-tac* $Q = \forall c.\ c \in Y \longrightarrow (s, X \cup \{c\}) \notin \mathcal{F}\ P$ **in** *contrapos-pp*, *metis is-processT5-S2*)

lemma *is-processT5-S6*: $([], \{c\}) \notin \mathcal{F}\ P \Longrightarrow ([c], \{\}) \in \mathcal{F}\ P$
apply(*rule-tac* t=[c] **and** s=[]@[c] **in** *subst*, *simp*)
apply(*rule is-processT5-S4*, *simp-all add*: *is-processT1*)
done

lemma *is-processT6*: $\forall s\ X.\ (s\ @\ [tick], \{\}) \in \mathcal{F}\ P \longrightarrow (s, X - \{tick\}) \in \mathcal{F}\ P$
by(*simp add*: *process-charn*)

lemma *is-processT7*: $\forall s\ t.\ s \in \mathcal{D}\ P \wedge tickFree\ s \wedge front\text{-}tickFree\ t \longrightarrow s\ @\ t \in \mathcal{D}\ P$
by(*insert process-charn*[*of* P], *metis*)

lemmas *is-processT7-S* = *is-processT7*[*rule-format*,*OF conjI*[*THEN conjI*, *THEN conj-commute*[*THEN iffD1*]]]

lemma *is-processT8*: $\forall s\ X.\ s \in \mathcal{D}\ P \longrightarrow (s, X) \in \mathcal{F}\ P$
by(*insert process-charn*[*of* P], *metis*)

lemmas *is-processT8-S* = *is-processT8*[*rule-format*]

lemma *is-processT8-Pair*: $fst\ s \in \mathcal{D}\ P \Longrightarrow s \in \mathcal{F}\ P$
apply(*subst surjective-pairing*)
apply(*rule is-processT8-S*, *simp*)
done

lemma *is-processT9*: $\forall s.\ s\ @\ [tick] \in \mathcal{D}\ P \longrightarrow s \in \mathcal{D}\ P$
by(*insert process-charn*[*of* P], *metis*)

lemma *is-processT9-S-swap*: $s \notin \mathcal{D}\ P \Longrightarrow s\ @\ [tick] \notin \mathcal{D}\ P$
by(*erule contrapos-nn*,*simp add*: *is-processT9*[*rule-format*])

2.5 Some Consequences of the Process Characterization

lemma *no-Trace-implies-no-Failure*: $s \notin \mathcal{T}\ P \Longrightarrow (s, \{\}) \notin \mathcal{F}\ P$
by(*simp add*: *Traces-def TRACES-def Failures-def*)

lemmas *NT-NF* = *no-Trace-implies-no-Failure*

lemma *T-def-spec*: $\mathcal{T}\ P = \{tr.\ \exists a.\ a \in \mathcal{F}\ P \wedge tr = fst\ a\}$
by(*simp add*: *Traces-def TRACES-def Failures-def*)

lemma *F-T*: $(s, X) \in \mathcal{F}\ P \Longrightarrow s \in \mathcal{T}\ P$
by(*simp add*: *T-def-spec split-def*, *metis*)

lemma *F-T1*: $a \in \mathcal{F}\ P \Longrightarrow fst\ a \in \mathcal{T}\ P$
by(*rule-tac X=snd a* **in** *F-T*,*simp*)

lemma *T-F*: $s \in \mathcal{T}\ P \Longrightarrow (s, \{\}) \in \mathcal{F}\ P$
apply(*auto simp*: *T-def-spec*)
apply(*drule is-processT4-S1*, *simp-all*)
done

lemmas *is-processT4-empty* [*elim*!]= *F-T* [*THEN T-F*]

lemma *NF-NT*: $(s, \{\}) \notin \mathcal{F}\ P \Longrightarrow s \notin \mathcal{T}\ P$
by(*erule contrapos-nn*, *simp only*: *T-F*)

lemma *is-processT6-S1*: $tick \notin X \Longrightarrow (s\ @\ [tick], \{\}) \in \mathcal{F}\ P \Longrightarrow (s, X) \in \mathcal{F}\ P$
by(*subst Diff-triv*[*of X* {*tick*}, *symmetric*],
simp, *erule is-processT6*[*rule-format*])

lemmas *is-processT3-ST* = *T-F* [*THEN is-processT3*[*rule-format*,*THEN F-T*]]

lemmas *is-processT3-ST-pref* = *T-F* [*THEN is-processT3-S-pref* [*THEN F-T*]]

lemmas *is-processT3-SR* = *F-T* [*THEN T-F* [*THEN is-processT3*[*rule-format*]]]

lemmas *D-T* = *is-processT8-S* [*THEN F-T*]

lemma *D-T-subset* : $\mathcal{D}\ P \subseteq \mathcal{T}\ P$ **by**(*auto intro*!:*D-T*)

lemma *NF-ND* : $(s, X) \notin \mathcal{F}\ P \Longrightarrow s \notin \mathcal{D}\ P$
by(*erule contrapos-nn*, *simp add*: *is-processT8-S*)

lemmas *NT-ND* = *D-T-subset*[*THEN Set.contra-subsetD*]

lemma *T-F-spec* : $((t, \{\}) \in \mathcal{F}\ P) = (t \in \mathcal{T}\ P)$
by(*auto simp*:*T-F F-T*)

lemma *is-processT5-S7*: $t \in \mathcal{T}\ P \Longrightarrow (t,\ A) \notin \mathcal{F}\ P \Longrightarrow \exists x.\ x \in A \wedge t\ @\ [x] \in \mathcal{T}\ P$
apply(*erule contrapos-np*, *simp*)
apply(*rule is-processT5*[*rule-format*, *OF conjI*,*of* - {}, *simplified*])
apply(*auto simp*: *T-F-spec*)
done

lemma *is-processT5-S7'*:
$(t,\ X) \in \mathcal{F}\ P \Longrightarrow (t,\ X \cup A) \notin \mathcal{F}\ P \Longrightarrow \exists x.\ x \in A \wedge x \notin X \wedge t\ @\ [x] \in \mathcal{T}\ P$
apply(*erule contrapos-np*, *simp*, *subst Un-Diff-cancel*[*of X A*, *symmetric*])
apply(*rule is-processT5*[*rule-format*])
apply(*auto simp*: *T-F-spec*)
done

lemma *Nil-subset-T*: $\{[]\} \subseteq \mathcal{T}\ P$
by(*auto simp*: *T-F-spec*[*symmetric*] *is-processT1*)

lemma *Nil-elem-T*: $[] \in \mathcal{T}\ P$
by(*simp add*: *Nil-subset-T*[*THEN subsetD*])

lemmas *D-imp-front-tickFree* = *is-processT8-S*[*THEN is-processT2*[*rule-format*]]

lemma *D-front-tickFree-subset* : $\mathcal{D}\ P \subseteq$ *Collect front-tickFree*
by(*auto simp*: *D-imp-front-tickFree*)

lemma *F-D-part* : $\mathcal{F}\ P = \{(s,\ x).\ s \in \mathcal{D}\ P\} \cup \{(s,\ x).\ s \notin \mathcal{D}\ P \wedge (s,\ x) \in \mathcal{F}\ P\}$
by(*auto intro*:*is-processT8-Pair*)

lemma *D-F* : $\{(s,\ x).\ s \in \mathcal{D}\ P\} \subseteq \mathcal{F}\ P$
by(*auto intro*:*is-processT8-Pair*)

lemma *append-T-imp-tickFree*: $t\ @\ s \in \mathcal{T}\ P \Longrightarrow s \neq [] \Longrightarrow$ *tickFree t*
by(*frule is-processT2-TR*[*rule-format*],
simp add: *front-tickFree-def tickFree-rev*)

corollary *append-single-T-imp-tickFree* : $t\ @\ [a] \in \mathcal{T}\ P \Longrightarrow$ *tickFree t*
by (*simp add*: *append-T-imp-tickFree*)

lemma *F-subset-imp-T-subset*: $\mathcal{F}\ P \subseteq \mathcal{F}\ Q \Longrightarrow \mathcal{T}\ P \subseteq \mathcal{T}\ Q$
by(*auto simp*: *subsetD T-F-spec*[*symmetric*])

lemma *is-processT6-S2*: $tick \notin X \Longrightarrow [tick] \in \mathcal{T}\ P \Longrightarrow ([],\ X) \in \mathcal{F}\ P$
by(*erule is-processT6-S1*, *simp add*: *T-F-spec*)

lemma *is-processT9-tick*: $[tick] \in \mathcal{D}\ P \Longrightarrow$ *front-tickFree s* $\Longrightarrow s \in \mathcal{D}\ P$
apply(*rule append.simps*(*1*) [*THEN subst*, *of* - *s*])
apply(*rule is-processT7-S*, *simp-all*)
apply(*rule is-processT9* [*rule-format*], *simp*)

done

lemma *T-nonTickFree-imp-decomp*: $t \in \mathcal{T}\ P \Longrightarrow \neg$ *tickFree* $t \Longrightarrow \exists\, s.\ t = s$ @ [*tick*]
by(*auto elim*: *is-processT2-TR*[*rule-format*] *nonTickFree-n-frontTickFree*)

2.6 Process Approximation is a Partial Ordering, a Cpo, and a Pcpo

The Failure/Divergence Model of CSP Semantics provides two orderings: The *approximation ordering* (also called *process ordering*) will be used for giving semantics to recursion (fixpoints) over processes, the *refinement ordering* captures our intuition that a more concrete process is more deterministic and more defined than an abstract one.

We start with the key-concepts of the approximation ordering, namely the predicates *min_elems* and *Ra* (abbreviating *refusals after*). The former provides just a set of minimal elements from a given set of elements of type-class *ord* ...

definition *min-elems* :: ('*s*::*ord*) *set* ⇒ '*s set*
where *min-elems* $X = \{s \in X.\ \forall\, t.\ t \in X \longrightarrow \neg\ (t < s)\}$

lemma *Nil-min-elems* : $[] \in A \Longrightarrow [] \in$ *min-elems* A
by(*simp add*: *min-elems-def*)

lemma *min-elems-le-self*[*simp*] : (*min-elems* A) $\subseteq A$
by(*auto simp*: *min-elems-def*)

lemmas *elem-min-elems* = *Set.set-mp*[*OF min-elems-le-self*]

lemma *min-elems-Collect-ftF-is-Nil* : *min-elems* (*Collect front-tickFree*) = {[]}
apply(*auto simp*: *min-elems-def le-list-def*)
apply(*drule front-tickFree-charn*[*THEN iffD1*])
apply(*auto dest*!: *tickFree-implies-front-tickFree*)
done

lemma *min-elems5* :
assumes A: (x::'*a list*) $\in A$
shows $\exists\, s{\leq}x.\ s \in$ *min-elems* A
proof −
have ∗ : !! (x::'*a list*) (A::'*a list set*) (n::*nat*).
$x \in A \wedge$ *length* $x \leq n \longrightarrow (\exists\, s{\leq}x.\ s \in$ *min-elems* $A)$
apply(*rule-tac* x=x **in** *spec*)
apply(*rule-tac* n=n **in** *nat-induct*)
apply(*auto simp*: *Nil-min-elems*)
apply(*case-tac* $\exists\ y.\ \ y \in A \wedge y < x$,*auto*)

apply(*rename-tac A na x y*, *erule-tac x=y* **in** *allE*, *simp*)
apply(*erule impE*,*drule less-length-mono*, *arith*)
apply(*safe*, *rename-tac s*, *rule-tac x=s* **in** *exI*,*simp*)
apply(*rule-tac x=x* **in** *exI*, *simp add*:*min-elems-def*)
done
show *?thesis* **by**(*rule-tac n=length x* **in** $*$[*rule-format*],*simp add*:*A*)
qed

lemma *min-elems4*: $A \neq \{\} \Longrightarrow \exists\, s.\ (s :: 'a\ trace) \in min\text{-}elems\ A$
by(*auto dest*: *min-elems5*)

lemma *min-elems-charn*: $t \in A \Longrightarrow \exists\ t'\ r.\ t = (t'\ @\ r) \wedge t' \in min\text{-}elems\ A$
by(*drule min-elems5*[*simplified le-list-def*], *auto*)

lemmas *min-elems-ex* = *min-elems-charn*

lemma *min-elems-no*: $(x{::}'a\ list) \in min\text{-}elems\ A \Longrightarrow t \in A \Longrightarrow t \leq x \Longrightarrow x = t$
by (*metis* (*no-types*, *lifting*) *less-list-def mem-Collect-eq min-elems-def*)

...while the second returns the set of possible refusal sets after a given trace s and a given process P:

definition $Ra :: ['\alpha\ process,\ '\alpha\ trace] \Rightarrow ('\alpha\ refusal\ set)\ (\mathcal{R}_a)$
where $\mathcal{R}_a\ P\ s = \{X.\ (s,\ X) \in \mathcal{F}\ P\}$

In the following, we link the process theory to the underlying fixpoint/domain theory of HOLCF by identifying the approximation ordering with HOLCF's pcpo's.

instantiation
process :: (*type*) *below*
begin

declares approximation ordering $\_ \sqsubseteq \_$ also written `_ << _`.

definition $le\text{-}approx\text{-}def : P \sqsubseteq Q \equiv \mathcal{D}\ Q \subseteq \mathcal{D}\ P \wedge$
$(\forall\, s.\ s \notin \mathcal{D}\ P \longrightarrow \mathcal{R}_a\ P\ s = \mathcal{R}_a\ Q\ s) \wedge$
$min\text{-}elems\ (\mathcal{D}\ P) \subseteq \mathcal{T}\ Q$

The approximation ordering captures the fact that more concrete processes should be more defined by ordering the divergence sets appropriately. For defined positions in a process, the failure sets must coincide pointwise; moreover, the minimal elements (wrt. prefix ordering on traces, i.e. lists) must be contained in the trace set of the more concrete process.

instance ..

end

lemma *le-approx1*: $P \sqsubseteq Q \Longrightarrow \mathcal{D}\ Q \subseteq \mathcal{D}\ P$

by(*simp add*: *le-approx-def*)

lemma *le-approx2*: $[\![P \sqsubseteq Q;\ s \notin \mathcal{D}\ P]\!] \Longrightarrow (s,X) \in \mathcal{F}\ Q = ((s,X) \in \mathcal{F}\ P)$
by(*auto simp*: *Ra-def le-approx-def*)

lemma *le-approx3*: $P \sqsubseteq Q \Longrightarrow$ *min-elems*$(\mathcal{D}\ P) \subseteq \mathcal{T}\ Q$
by(*simp add*: *le-approx-def*)

lemma *le-approx2T*: $[\![P \sqsubseteq Q;\ s \notin \mathcal{D}\ P]\!] \Longrightarrow\ s \in \mathcal{T}\ Q = (s \in \mathcal{T}\ P)$
by(*auto simp*: *le-approx2 T-F-spec*[*symmetric*])

lemma *le-approx-lemma-F* : $P \sqsubseteq Q \Longrightarrow \mathcal{F}\ Q \subseteq \mathcal{F}\ P$
apply(*subst F-D-part*[*of Q*], *subst F-D-part*[*of P*])
apply(*auto simp*:*le-approx-def Ra-def min-elems-def*)
done

lemmas *order-lemma* = *le-approx-lemma-F*

lemma *le-approx-lemma-T*: $P \sqsubseteq Q \Longrightarrow \mathcal{T}\ Q \subseteq \mathcal{T}\ P$
by(*auto dest*!:*le-approx-lemma-F simp*: *T-F-spec*[*symmetric*])

lemma *proc-ord2a* : $P \sqsubseteq Q \Longrightarrow s \notin \mathcal{D}\ P \Longrightarrow ((s,\ X) \in \mathcal{F}\ P) = ((s,\ X) \in \mathcal{F}\ Q)$
by(*auto simp*: *le-approx-def Ra-def*)

instance
  process :: (*type*) *po*
proof
  fix P::$'\alpha$ *process*
  show $P \sqsubseteq P$ **by**(*auto simp*: *le-approx-def min-elems-def elim*: *Process.D-T*)
 next
  fix $P\ Q$ ::$'\alpha$ *process*
  assume A:$P \sqsubseteq Q$ **and** B:$Q \sqsubseteq P$ **thus** $P = Q$
  apply(*insert A*[*THEN le-approx1*] *B*[*THEN le-approx1*])
  apply(*insert A*[*THEN le-approx-lemma-F*] *B*[*THEN le-approx-lemma-F*])
  by(*auto simp*: *Process-eq-spec*)
 next
  fix $P\ Q\ R$ ::$'\alpha$ *process*
  assume A: $P \sqsubseteq Q$ **and** B: $Q \sqsubseteq R$ **thus** $P \sqsubseteq R$
  proof −
    have C : $\mathcal{D}\ R \subseteq \mathcal{D}\ P$
        by(*insert A*[*THEN le-approx1*] *B*[*THEN le-approx1*], *auto*)
    have D : $\forall s.\ s \notin \mathcal{D}\ P \longrightarrow \{X.\ (s,\ X) \in \mathcal{F}\ P\} = \{X.\ (s,\ X) \in \mathcal{F}\ R\}$
        apply(*rule allI*, *rule impI*, *rule set-eqI*, *simp*)
        apply(*frule A*[*THEN le-approx1*, *THEN Set.contra-subsetD*])
        apply(*frule B*[*THEN le-approx1*, *THEN Set.contra-subsetD*])
        apply(*drule A*[*THEN le-approx2*], *drule B*[*THEN le-approx2*])

```
        apply auto
        done
    have E : min-elems (𝒟 P) ⊆ 𝒯 R
        apply(insert B[THEN le-approx3] A[THEN le-approx3] )
        apply(insert B[THEN le-approx-lemma-T] A[THEN le-approx1] )
        apply(rule subsetI, simp add: min-elems-def, auto)
        apply(rename-tac x, case-tac x ∈ 𝒟 Q)
        apply(drule-tac B = 𝒯 R and t=x
            in subset-iff[THEN iffD1,rule-format], auto)
        apply(subst B [THEN le-approx2T],simp)
        apply(rename-tac x, drule-tac B = 𝒯 Q and t=x
            in subset-iff[THEN iffD1,rule-format],auto)
        done
    show ?thesis
    by(insert C D E, simp add: le-approx-def Ra-def)
  qed
qed
```

At this point, we inherit quite a number of facts from the underlying HOLCF theory, which comprises a library of facts such as `chain`, `directed`(sets), upper bounds and least upper bounds, etc.

Some facts from the theory of complete partial orders:

- `Porder.chainE` : $chain\ ?Y \Longrightarrow ?Y\ ?i \sqsubseteq ?Y\ (Suc\ ?i)$
- `Porder.chain_mono` : $[\![chain\ ?Y;\ ?i \leq ?j]\!] \Longrightarrow ?Y\ ?i \sqsubseteq ?Y\ ?j$
- `Porder.is_ubD` : $[\![?S <|\ ?u;\ ?x \in ?S]\!] \Longrightarrow ?x \sqsubseteq ?u$
- `Porder.ub_rangeI` :
  $(\bigwedge i.\ ?S\ i \sqsubseteq ?x) \Longrightarrow range\ ?S <|\ ?x$
- `Porder.ub_imageD` : $[\![?f\ `\ ?S <|\ ?u;\ ?x \in ?S]\!] \Longrightarrow ?f\ ?x \sqsubseteq ?u$
- `Porder.is_ub_upward` : $[\![?S <|\ ?x;\ ?x \sqsubseteq ?y]\!] \Longrightarrow ?S <|\ ?y$
- `Porder.is_lubD1` : $?S <<|\ ?x \Longrightarrow ?S <|\ ?x$
- `Porder.is_lubI` : $[\![?S <|\ ?x;\ \bigwedge u.\ ?S <|\ u \Longrightarrow ?x \sqsubseteq u]\!] \Longrightarrow ?S <<|$
  $?x$
- `Porder.is_lub_maximal` : $[\![?S <|\ ?x;\ ?x \in ?S]\!] \Longrightarrow ?S <<|\ ?x$
- `Porder.is_lub_lub` : $?M <<|\ ?x \Longrightarrow ?M <<|\ lub\ ?M$
- `Porder.is_lub_range_shift`:
  $chain\ ?S \Longrightarrow range\ (\lambda i.\ ?S\ (i + ?j)) <<|\ ?x = range\ ?S <<|\ ?x$
- `Porder.is_lub_rangeD1`: $range\ ?S <<|\ ?x \Longrightarrow ?S\ ?i \sqsubseteq ?x$

- `Porder.lub_eqI`: $?M <<| ?l \Longrightarrow lub\ ?M = ?l$
- `Porder.is_lub_unique`:$[\![?S <<| ?x;\ ?S <<| ?y]\!] \Longrightarrow ?x = ?y$

definition *lim-proc* :: $('\alpha\ process)\ set \Rightarrow '\alpha\ process$
where $lim\text{-}proc\ (X) = Abs\text{-}process\ (\bigcap\ (\mathcal{F}\ `\ X), \bigcap\ (\mathcal{D}\ `\ X))$

lemma *min-elems3*: $s\ @\ [c] \in \mathcal{D}\ P \Longrightarrow s\ @\ [c] \notin min\text{-}elems\ (\mathcal{D}\ P) \Longrightarrow s \in \mathcal{D}\ P$
apply(*auto simp*: *min-elems-def le-list-def less-list-def*)
apply(*rename-tac t r*)
apply(*subgoal-tac* $t \leq s$)
apply(*subgoal-tac* $r \neq []$)
apply(*simp add*: *le-list-def*)
apply(*auto intro*!: *is-processT7-S append-eq-first-pref-spec*)
apply(*auto dest*!: *D-T*)
apply(*simp-all only*: *append-assoc*[*symmetric*],
drule append-T-imp-tickFree,
simp-all add: *tickFree-implies-front-tickFree*)+
done

lemma *min-elems1*: $s \notin \mathcal{D}\ P \Longrightarrow s\ @\ [c] \in \mathcal{D}\ P \Longrightarrow s\ @\ [c] \in min\text{-}elems\ (\mathcal{D}\ P)$
by(*erule contrapos-np*, *auto elim*!: *min-elems3*)

lemma *min-elems2*: $s \notin \mathcal{D}\ P \Longrightarrow s\ @\ [c] \in \mathcal{D}\ P \Longrightarrow P \sqsubseteq S \Longrightarrow Q \sqsubseteq S \Longrightarrow (s\ @\ [c], \{\}) \in \mathcal{F}\ Q$
apply(*frule-tac P=Q* **and** *Q=S* **in** *le-approx-lemma-T*)
apply(*simp add*: *T-F-spec*)
apply(*rule-tac* $A{=}\mathcal{T}\ S$ **in** *subsetD*, *assumption*)
apply(*rule-tac* $A{=}min\text{-}elems(\mathcal{D}\ P)$ **in** *subsetD*)
apply(*simp-all add*: *le-approx-def min-elems1*)
done

lemma *min-elems6*: $s \notin \mathcal{D}\ P \Longrightarrow s\ @\ [c] \in \mathcal{D}\ P \Longrightarrow P \sqsubseteq S \Longrightarrow (s\ @\ [c], \{\}) \in \mathcal{F}\ S$
by(*auto intro*!: *min-elems2*)

lemma *ND-F-dir2*: $s \notin \mathcal{D}\ P \Longrightarrow (s, \{\}) \in \mathcal{F}\ P \Longrightarrow P \sqsubseteq S \Longrightarrow Q \sqsubseteq S \Longrightarrow (s, \{\}) \in \mathcal{F}\ Q$
apply(*frule-tac P=Q* **and** *Q=S* **in** *le-approx-lemma-T*)
apply(*simp add*: *le-approx-def Ra-def T-F-spec*, *safe*)
apply((*erule-tac x=s* **in** *allE*)+,*simp*)
apply(*drule-tac x={}* **in** *eqset-imp-iff*, *auto simp*: *T-F-spec*)
done — orig version

lemma *ND-F-dir2'*: $s \notin \mathcal{D}\ P \Longrightarrow s \in \mathcal{T}\ P \Longrightarrow P \sqsubseteq S \Longrightarrow Q \sqsubseteq S \Longrightarrow s \in \mathcal{T}\ Q$
apply(*frule-tac P=Q* **and** *Q=S* **in** *le-approx-lemma-T*)
apply(*simp add*: *le-approx-def Ra-def T-F-spec*, *safe*)
apply((*erule-tac x=s* **in** *allE*)+,*simp*)
apply(*drule-tac x={}* **in** *eqset-imp-iff*, *auto simp*: *T-F-spec*)
done

lemma *chain-lemma*: $[\![chain\ S]\!] \Longrightarrow S\ i \sqsubseteq S\ k \vee S\ k \sqsubseteq S\ i$
by(*case-tac* $i \leq k$, *auto intro*:*chain-mono chain-mono-less*)

lemma *is-process-REP-LUB*:
assumes *chain*: *chain S*
shows *is-process* ($\bigcap$ ($\mathcal{F}$ ‘ *range S*), $\bigcap$ ($\mathcal{D}$ ‘ *range S*))

proof (*auto simp*: *is-process-def*)
show ([], {}) $\in$ *FAILURES* ($\bigcap$ *a*::*nat*. $\mathcal{F}$ (*S a*), $\bigcap$ *a*::*nat*. $\mathcal{D}$ (*S a*))
by(*auto simp*: *FAILURES-def is-processT*)
next
fix *s*::'*a trace* **fix** *X*::'*a event set*
assume *A* : (*s*, *X*) $\in$ (*FAILURES* ($\bigcap$ *a* :: *nat*. $\mathcal{F}$ (*S a*), $\bigcap$ *a* :: *nat*. $\mathcal{D}$ (*S a*)))
thus *front-tickFree s*
by(*auto simp*: *DIVERGENCES-def FAILURES-def*
intro!: *is-processT2*[*rule-format*])
next
fix *s t*::'*a trace*
assume (*s* @ *t*, {}) $\in$ *FAILURES* ($\bigcap$ *a*::*nat*. $\mathcal{F}$ (*S a*), $\bigcap$ *a*::*nat*. $\mathcal{D}$ (*S a*))
thus (*s*, {}) $\in$ *FAILURES* ($\bigcap$ *a*::*nat*. $\mathcal{F}$ (*S a*), $\bigcap$ *a*::*nat*. $\mathcal{D}$ (*S a*))
by(*auto simp*: *DIVERGENCES-def FAILURES-def*
intro: *is-processT3*[*rule-format*])
next
fix *s*::'*a trace* **fix** *X Y* ::'*a event set*
assume (*s*, *Y*) $\in$ *FAILURES* ($\bigcap$ *a*::*nat*. $\mathcal{F}$ (*S a*), $\bigcap$ *a*::*nat*. $\mathcal{D}$(*S a*)) **and** $X \subseteq Y$
thus (*s*, *X*) $\in$ *FAILURES* ($\bigcap$ *a*::*nat*. $\mathcal{F}$ (*S a*), $\bigcap$ *a*::*nat*. $\mathcal{D}$(*S a*))
by(*auto simp*: *DIVERGENCES-def FAILURES-def*
intro: *is-processT4*[*rule-format*])
next
fix *s*::'*a trace* **fix** *X Y* :: '*a event set*
assume *A*:(*s*, *X*) $\in$ *FAILURES* ($\bigcap$ *a*::*nat*. $\mathcal{F}$ (*S a*), $\bigcap$ *a*::*nat*. $\mathcal{D}$ (*S a*))
assume *B*:$\forall$ *c*. *c*$\in$*Y* $\longrightarrow$ (*s*@[*c*],{})$\notin$*FAILURES*($\bigcap$ *a*::*nat*. $\mathcal{F}$(*S a*),$\bigcap$ *a*::*nat*. $\mathcal{D}$(*S a*))
thus (*s*, $X \cup Y$) $\in$ *FAILURES* ($\bigcap$ *a*::*nat*. $\mathcal{F}$ (*S a*), $\bigcap$ *a*::*nat*. $\mathcal{D}$ (*S a*))

What does this mean: All trace prolongations c in all Y, which are blocking in the limit, will also occur in the refusal set of the limit.

using *A B chain*
proof (*auto simp*: *DIVERGENCES-def FAILURES-def*,
case-tac $\forall$ *x*. *x* $\in$ (*range S*) $\longrightarrow$ (*s*, $X \cup Y$) $\in$ $\mathcal{F}$ *x*,
simp-all add:*DIVERGENCES-def FAILURES-def*,*rename-tac a*,
case-tac s $\notin$ $\mathcal{D}$ (*S a*),*simp-all add*: *is-processT8*)
fix *a*::*nat*
assume *X*: $\forall$ *a*. (*s*, *X*) $\in$ $\mathcal{F}$ (*S a*)
have *X-ref-at-a*: (*s*, *X*) $\in$ $\mathcal{F}$ (*S a*)

```
            using X by auto
    assume Y: ∀ c. c ∈ Y ⟶ (∃ a. (s @ [c], {}) ∉ ℱ (S a))
    assume defined: s ∉ 𝒟 (S a)
    show   (s::'a trace, X ∪ Y) ∈ ℱ (S a)
    proof(auto simp:X-ref-at-a
            intro!: is-processT5[rule-format],
        frule Y[THEN spec, THEN mp], erule exE,
        erule-tac Q=(s @ [c], {}) ∈ ℱ (S a) in contrapos-pp)
      fix c::'a event fix a' :: nat
      assume s-c-trace-not-trace-somewhere: (s @ [c], {}) ∉ ℱ (S a')
      show (s @ [c], {}) ∉ ℱ (S a)
      proof(insert chain-lemma[OF chain, of a a'],erule disjE)
        assume before: S a ⊑ S a'
        show (s @ [c], {}) ∉ ℱ (S a)
          using s-c-trace-not-trace-somewhere  before
          apply(case-tac s @ [c] ∉ 𝒟 (S a),
                simp-all add: T-F-spec before[THEN le-approx2T,symmetric])
          apply(erule contrapos-nn)
          apply(simp only: T-F-spec[symmetric])
          apply(auto dest!:min-elems6[OF defined])
          done
      next
        assume after:S a' ⊑ S a
        show (s @ [c], {}) ∉ ℱ (S a)
          using s-c-trace-not-trace-somewhere
          by(simp add:T-F-spec after[THEN le-approx2T]
                    s-c-trace-not-trace-somewhere[THEN NF-ND])
      qed
    qed
  qed
next
  fix s::'a trace   fix X::'a event set
  assume (s @ [tick], {}) ∈ FAILURES (⋂ a::nat. ℱ (S a), ⋂ a::nat. 𝒟 (S a))
  thus   (s, X − {tick}) ∈ FAILURES (⋂ a::nat. ℱ (S a), ⋂ a::nat. 𝒟 (S a))
        by(auto simp: DIVERGENCES-def FAILURES-def
                intro! : is-processT6[rule-format])
next
  fix s t ::'a trace
  assume s : DIVERGENCES (⋂ a::nat. ℱ (S a), ⋂ a::nat. 𝒟 (S a))
  and    tickFree s and  front-tickFree t
  thus   s @ t ∈ DIVERGENCES (⋂ a::nat. ℱ (S a), ⋂ a::nat. 𝒟 (S a))
        by(auto simp: DIVERGENCES-def FAILURES-def
                intro: is-processT7[rule-format])
next
  fix s::'a trace  fix X::'a event set
  assume s ∈ DIVERGENCES (⋂ a::nat. ℱ (S a), ⋂ a::nat. 𝒟 (S a))
  thus   (s, X) ∈ FAILURES (⋂ a::nat. ℱ (S a), ⋂ a::nat. 𝒟 (S a))
        by(auto simp: DIVERGENCES-def FAILURES-def
                    intro: is-processT8[rule-format])
```

next
  fix *s::'a trace*
  assume s @ [*tick*] $\in$ *DIVERGENCES* ($\bigcap$ *a::nat.* $\mathcal{F}$ (*S a*), $\bigcap$ *a::nat.* $\mathcal{D}$ (*S a*))
  thus $s \in$ *DIVERGENCES* ($\bigcap$ *a::nat.* $\mathcal{F}$ (*S a*), $\bigcap$ *a::nat.* $\mathcal{D}$ (*S a*))
    by(*auto simp*: *DIVERGENCES-def FAILURES-def*
      intro: *is-processT9*[*rule-format*])
qed

lemmas *Rep-Abs-LUB* = *Abs-process-inverse*[*simplified Rep-process*,
  simplified, *OF is-process-REP-LUB*,
  simplified]

lemma *F-LUB*: *chain S* $\Longrightarrow$ $\mathcal{F}$(*lim-proc*(*range S*)) = $\bigcap$ ($\mathcal{F}$ ‘ *range S*)
by(*simp add*: *lim-proc-def* , *subst Failures-def*, *auto simp*: *FAILURES-def Rep-Abs-LUB*)

lemma *D-LUB*: *chain S* $\Longrightarrow$ $\mathcal{D}$(*lim-proc*(*range S*)) = $\bigcap$ ($\mathcal{D}$ ‘ *range S*)
by(*simp add*: *lim-proc-def* , *subst D-def*, *auto simp*: *DIVERGENCES-def Rep-Abs-LUB*)

lemma *T-LUB*: *chain S* $\Longrightarrow$ $\mathcal{T}$ (*lim-proc*(*range S*)) = $\bigcap$ ($\mathcal{T}$ ‘ *range S*)
apply(*simp add*: *lim-proc-def* , *subst Traces-def*)
apply(*simp add*: *TRACES-def FAILURES-def Rep-Abs-LUB*)
apply(*auto intro*: *F-T*, *rule-tac x*={} **in** *exI*, *auto intro*: *T-F*)
done

schematic-goal *D-LUB-2*: *chain S* $\Longrightarrow$ $t \in$ $\mathcal{D}$(*lim-proc*(*range S*)) = *?X*
apply(*subst D-LUB*, *simp*)
apply(*rule trans*, *simp*)
apply(*simp*)
done

schematic-goal *T-LUB-2*: *chain S* $\Longrightarrow$ ($t \in$ $\mathcal{T}$ (*lim-proc* (*range S*))) = *?X*
apply(*subst T-LUB*, *simp*)
apply(*rule trans*, *simp*)
apply(*simp*)
done

2.7 Process Refinement is a Partial Ordering

The following type instantiation declares the refinement order _ $\leq$ _ written `_ <= _`. It captures the intuition that more concrete processes should be more deterministic and more defined.

instantiation
  process :: (*type*) *ord*
begin

definition *le-ref-def* : $P \leq Q \equiv \mathcal{D}\ Q \subseteq \mathcal{D}\ P \wedge \mathcal{F}\ Q \subseteq \mathcal{F}\ P$

definition *less-ref-def* : $(P::'a\ process) < Q \equiv P \leq Q \wedge P \neq Q$

instance ..

end

Note that this just another syntax to our standard process refinement order defined in the theory Process.

lemma *le-approx-implies-le-ref*: $(P::'\alpha\ process) \sqsubseteq Q \Longrightarrow P \leq Q$
by(*simp add*: *le-ref-def le-approx1 le-approx-lemma-F*)

lemma *le-ref1*: $P \leq Q \Longrightarrow \mathcal{D}\ Q \subseteq \mathcal{D}\ P$
by(*simp add*: *le-ref-def*)

lemma *le-ref2*: $P \leq Q \Longrightarrow \mathcal{F}\ Q \subseteq \mathcal{F}\ P$
by(*simp add*: *le-ref-def*)

lemma *le-ref2T* : $P \leq Q \Longrightarrow \mathcal{T}\ Q \subseteq \mathcal{T}\ P$
by (*rule subsetI*) (*simp add*: *T-F-spec*[*symmetric*] *le-ref2*[*THEN subsetD*])

instance *process* :: (*type*) *order*
proof
fix $P\ Q\ R$ :: $'\alpha$ *process*
{
show $(P < Q) = (P \leq Q \wedge \neg\ Q \leq\ P)$
by(*auto simp*: *le-ref-def less-ref-def Process-eq-spec*)
next
show $P \leq P$ **by**(*simp add*: *le-ref-def*)
next
assume $P \leq Q$ **and** $Q \leq R$ **then show** $P \leq R$
by (*simp add*: *le-ref-def*, *auto*)
next
assume $P \leq Q$ **and** $Q \leq P$ **then show** $P = Q$
by(*auto simp*: *le-ref-def Process-eq-spec*)
}
qed

lemma *lim-proc-is-ub*:*chain S* $\Longrightarrow$ *range S* <| *lim-proc* (*range S*)
apply(*auto simp*: *is-ub-def le-approx-def F-LUB D-LUB T-LUB Ra-def*)
using *chain-lemma is-processT8 le-approx2* **apply** *blast*
using *D-T chain-lemma le-approx2T le-approx-def* **by** *blast*

lemma *lim-proc-is-lub1*: *chain S* $\Longrightarrow \forall\ u\ .\ (range\ S <|\ u \longrightarrow\ \mathcal{D}\ u \subseteq \mathcal{D}\ (lim\text{-}proc\ (range\ S)))$
by(*auto simp*: *D-LUB*, *frule-tac i=a* **in** *Porder.ub-rangeD*, *auto dest*: *le-approx1*)

lemma *lim-proc-is-lub2*:

chain S $\Longrightarrow \forall$ *u . range S* $<|$ *u* $\longrightarrow$ ($\forall$ *s.* $s \notin \mathcal{D}$ (*lim-proc* (*range S*))
$\longrightarrow$ *Ra* (*lim-proc* (*range S*)) *s* = *Ra u s*)
apply(*auto simp*: *is-ub-def D-LUB F-LUB Ra-def*)
using *proc-ord2a* **apply** *blast*
using *is-processT8-S proc-ord2a* **by** *blast*

lemma *lim-proc-is-lub3a*: *front-tickFree s* $\Longrightarrow s \notin \mathcal{D}\ P \longrightarrow (\forall t.\ t \in \mathcal{D}\ P \longrightarrow \neg\ t < s\ @\ [c])$
apply(*rule impI*, *erule contrapos-np*, *auto*)
apply(*auto simp*: *le-list-def less-list-def*)
by (*metis D-def butlast-append butlast-snoc*
front-tickFree-mono is-process7 is-process-Rep self-append-conv)

lemma *lim-proc-is-lub3b*:
assumes *1* : $\forall x.\ x \in X \longrightarrow (\forall xa.\ xa \in \mathcal{D}\ x \wedge (\forall t.\ t \in \mathcal{D}\ x \longrightarrow \neg\ t < xa) \longrightarrow xa \in \mathcal{T}\ \ u)$
and *2* : $xa \in X$
and *3* : $\forall xa.\ xa \in X \longrightarrow x \in \mathcal{D}\ xa$
and *4* : $\forall t.\ (\forall x.\ x \in X \longrightarrow t \in \mathcal{D}\ x) \longrightarrow \neg\ t < x$
shows $x \in \mathcal{T}\ \ u$
proof (*cases* $\forall t.\ t \in \mathcal{D}\ xa \longrightarrow \neg\ t < x$)
case *True* **from** ‹$\forall t.\ t \in \mathcal{D}\ xa \longrightarrow \neg\ t < x$› **show** *?thesis* **using** *1 2 3* **by** *simp*
next
case *False* **from** ‹$\neg(\forall t.\ t \in \mathcal{D}\ xa \longrightarrow \neg\ t < x)$› **and** *3 4*
have *A*: $\exists y\ r\ c.\ y \in X \wedge r \notin \mathcal{D}\ y \wedge r \in \mathcal{D}\ xa \wedge r\ @\ [c] = x$
by (*metis D-imp-front-tickFree front-tickFree-charn*
less-self lim-proc-is-lub3a nil-less
tickFree-implies-front-tickFree)
show *?thesis*
apply(*insert A*) **apply**(*erule exE*)+
using *1 3 D-imp-front-tickFree lim-proc-is-lub3a* **by** *blast*
qed

lemma *lim-proc-is-lub3c*:
assumes $*$:*chain S*
and $**$:*X* = *range S* — protection for range - otherwise auto unfolds and gets lost
shows $\forall$ *u.* $X <|\ u \longrightarrow$ *min-elems*($\mathcal{D}$ (*lim-proc X*)) $\subseteq \mathcal{T}\ \ u$
proof −
have *B* : $\mathcal{D}$ (*lim-proc X*) = $\bigcap$ ($\mathcal{D}$ ‘ *X*) **by**(*simp add*: $*$ $**$ *D-LUB*)
show *?thesis*
apply(*auto simp*: *is-ub-def* $*$ $**$)
apply(*auto simp*: *B min-elems-def le-approx-def HOL.imp-conjR HOL.all-conj-distrib Ball-def*)
apply(*simp add*: *subset-iff imp-conjL*[*symmetric*])
apply(*rule lim-proc-is-lub3b*[*of range S*, *simplified*])

```
  using ** B by auto
qed

lemma limproc-is-lub: chain S ⟹ range S <<| lim-proc (range S)
apply (auto simp: is-lub-def lim-proc-is-ub)
apply (simp add: le-approx-def is-lub-def lim-proc-is-ub)
by (simp add: lim-proc-is-lub1 lim-proc-is-lub2 lim-proc-is-lub3c)

lemma limproc-is-thelub: chain S ⟹ Lub S = lim-proc (range S)
by (frule limproc-is-lub,frule Porder.po-class.lub-eqI, simp)


instance
  process :: (type) cpo
proof
  fix S ::nat ⇒ 'α process
  assume C:chain S
  thus ∃ x. range S <<| x using limproc-is-lub by blast
qed

instance
  process :: (type) pcpo
proof
  show ∃ x::'a process. ∀ y::'a process. x ⊑ y
  proof −
    have is-process-witness :
        is-process({(s,X). front-tickFree s},{d. front-tickFree d})
        apply(auto simp:is-process-def FAILURES-def DIVERGENCES-def)
        apply(auto elim!: tickFree-implies-front-tickFree front-tickFree-dw-closed
                          front-tickFree-append)
        done
    have bot-inverse :
       Rep-process(Abs-process({(s, X). front-tickFree s},Collect front-tickFree))=
                    ({(s, X). front-tickFree s}, Collect front-tickFree)
       by(subst Abs-process-inverse, simp-all add: Rep-process is-process-witness)
    have divergences-frontTickFree:
        ⋀y x. x ∈ snd (Rep-process y) ⟹ front-tickFree x
        by(rule D-imp-front-tickFree, simp add: D-def DIVERGENCES-def)
    have failures-frontTickFree:
        ⋀y s x. (s, x) ∈ fst (Rep-process y) ⟹ front-tickFree s
        by(rule is-processT2[rule-format],
          simp add: Failures-def FAILURES-def)
    have minelems-contains-mt:
        ⋀y x. x ∈ min-elems (Collect front-tickFree) ⟹ x = []
        by(simp add: min-elems-def front-tickFree-charn,safe,
          auto simp: Nil-elem-T)
    show ?thesis
    apply(rule-tac x=Abs-process ({(s,X). front-tickFree s},{d. front-tickFree d})
```

```
        in exI)
    apply(auto simp: le-approx-def bot-inverse Ra-def
                Failures-def D-def FAILURES-def DIVERGENCES-def
                divergences-frontTickFree failures-frontTickFree)
    apply (metis minelems-contains-mt Nil-elem-T )
    done
  qed
qed
```

2.8 Process Refinement is Admissible

```
lemma le-adm[simp]: cont (u::('a::cpo) ⇒ 'b process) ⟹ monofun v ⟹ adm(λx.
u x ≤ v x)
proof(auto simp add:le-ref-def cont2contlubE adm-def, goal-cases)
  case (1 Y x)
  hence v (Y i) ⊑ v (⨆i. Y i) for i by (simp add: is-ub-thelub monofunE)
  hence 𝒟 (v (⨆i. Y i)) ⊆ 𝒟 (u (Y i)) for i using 1(4) le-approx1 by blast
  then show ?case
   using D-LUB[OF ch2ch-cont[OF 1(1) 1(3)]] limproc-is-thelub[OF ch2ch-cont[OF
1(1) 1(3)]] 1(5) by force
next
  case (2 Y a b)
  hence v (Y i) ⊑ v (⨆i. Y i) for i by (simp add: is-ub-thelub monofunE)
  hence ℱ (v (⨆i. Y i)) ⊆ ℱ (u (Y i)) for i using 2(4) le-approx-lemma-F by
blast
  then show ?case
   using F-LUB[OF ch2ch-cont[OF 2(1) 2(3)]] limproc-is-thelub[OF ch2ch-cont[OF
2(1) 2(3)]] 2(5) by force
qed

lemmas le-adm-cont[simp] = le-adm[OF - cont2mono]
```

2.9 The Conditional Statement is Continuous

The conditional operator of CSP is obtained by a direct shallow embedding. Here we prove it continuous

```
lemma if-then-else-cont[simp]:
  assumes *:(⋀x. P x ⟹ cont ((f::'c ⇒ ('a::cpo) ⇒ 'b process) x))
  and     **:(⋀x. ¬ P x ⟹ cont (g x))
  shows ⋀x. cont(λy. if P x then f x y else g x y)
  using * ** by (auto simp:cont-def)


end
```

Chapter 3

The CSP Operators

3.1 The Undefined Process

theory *Bot*
imports *Process*
begin

definition *BOT* :: $'\alpha$ *process*
where *BOT* $\equiv$ *Abs-process* ({(*s*,*X*). *front-tickFree s*}, {*d*. *front-tickFree d*})

lemma *is-process-REP-Bot* :
is-process ({(*s*,*X*). *front-tickFree s*}, {*d*. *front-tickFree d*})
by(*auto simp*: *tickFree-implies-front-tickFree is-process-def*
FAILURES-def DIVERGENCES-def
elim: *Process.front-tickFree-dw-closed*
elim: *Process.front-tickFree-append*)

lemma *Rep-Abs-Bot* :*Rep-process* (*Abs-process* ({(*s*,*X*). *front-tickFree s*},{*d*. *front-tickFree d*})) =
({(*s*,*X*). *front-tickFree s*},{*d*. *front-tickFree d*})
by(*subst Abs-process-inverse*, *simp-all only*: *CollectI Rep-process is-process-REP-Bot*)

lemma *F-Bot*: $\mathcal{F}$ *BOT* = {(*s*,*X*). *front-tickFree s*}
by(*simp add*: *BOT-def FAILURES-def Failures-def Rep-Abs-Bot*)

lemma *D-Bot*: $\mathcal{D}$ *BOT* = {*d*. *front-tickFree d*}
by(*simp add*: *BOT-def DIVERGENCES-def D-def Rep-Abs-Bot*)

lemma *T-Bot*: $\mathcal{T}$ *BOT* = {*s*. *front-tickFree s*}
by(*simp add*: *BOT-def TRACES-def Traces-def FAILURES-def Rep-Abs-Bot*)

This is the key result: $\perp$ — which we know to exist from the process instantiation — is equal Bot .

lemma *BOT-is-UU*[*simp*]: *BOT* = $\perp$

apply(*auto simp*: *Pcpo.eq-bottom-iff Process.le-approx-def Ra-def*
min-elems-Collect-ftF-is-Nil Process.Nil-elem-T
F-Bot D-Bot T-Bot
elim: *D-imp-front-tickFree*)
apply(*metis Process.is-processT2*)
done

lemma *F-UU*: $\mathcal{F}\ \bot = \{(s,X).\ \textit{front-tickFree}\ s\}$
using *F-Bot* **by** *auto*

lemma *D-UU*: $\mathcal{D}\ \bot = \{d.\ \textit{front-tickFree}\ d\}$
using *D-Bot* **by** *auto*

lemma *T-UU*: $\mathcal{T}\ \bot = \{s.\ \textit{front-tickFree}\ s\}$
using *T-Bot* **by** *auto*

end

3.2 The SKIP Process

theory *Skip*
imports *Process*
begin

definition *SKIP* :: *'a process*
where $\textit{SKIP} \equiv \textit{Abs-process}\ (\{(s,\ X).\ s = [] \wedge \textit{tick} \notin X\} \cup \{(s,\ X).\ s = [\textit{tick}]\},$
$\{\})$

lemma *is-process-REP-SKIP*:
$\textit{is-process}\ (\{(s,\ X).\ s = [] \wedge \textit{tick} \notin X\} \cup \{(s,\ X).\ s = [\textit{tick}]\},\ \{\})$
apply(*auto simp*: *FAILURES-def DIVERGENCES-def front-tickFree-def is-process-def*)
apply(*erule contrapos-np*,*drule neq-Nil-conv*[*THEN iffD1*], *auto*)
done

lemma *is-process-REP-SKIP2*:
$\textit{is-process}\ (\{[]\} \times \{X.\ \textit{tick} \notin X\} \cup \{(s,\ X).\ s = [\textit{tick}]\},\ \{\})$
using *is-process-REP-SKIP* **by** *auto*

lemmas *process-prover* = *Rep-process Abs-process-inverse*
FAILURES-def TRACES-def
DIVERGENCES-def is-process-REP-SKIP

lemma *F-SKIP*:
$\mathcal{F}\ \textit{SKIP} = \{(s,\ X).\ s = [] \wedge \textit{tick} \notin X\} \cup \{(s,\ X).\ s = [\textit{tick}]\}$
by(*simp add*: *process-prover SKIP-def Failures-def is-process-REP-SKIP2*)

lemma *D-SKIP*: $\mathcal{D}\ \textit{SKIP} = \{\}$

by(*simp add*: *process-prover SKIP-def D-def is-process-REP-SKIP2*)

lemma *T-SKIP*: $\mathcal{T}$ *SKIP* ={[],[*tick*]}
by(*auto simp*: *process-prover SKIP-def Traces-def is-process-REP-SKIP2*)

end

3.3 The STOP Process

theory *Stop*
imports *Process*
begin

definition *STOP* :: $'\alpha$ *process*
where *STOP* $\equiv$ *Abs-process* ({(s, X). s = []}, {})

lemma *is-process-REP-STOP*: *is-process* ({(s, X). s = []},{})
by(*simp add*: *is-process-def FAILURES-def DIVERGENCES-def*)

lemma *Rep-Abs-STOP* : *Rep-process* (*Abs-process* ({(s, X). s = []},{})) = ({(s, X). s = []},{})
by(*subst Abs-process-inverse*, *simp add*: *Rep-process is-process-REP-STOP*, *auto*)

lemma *F-STOP* : $\mathcal{F}$ *STOP* = {(s,X). s = []}
by(*simp add*: *STOP-def FAILURES-def Failures-def Rep-Abs-STOP*)

lemma *D-STOP*: $\mathcal{D}$ *STOP* = {}
by(*simp add*: *STOP-def DIVERGENCES-def D-def Rep-Abs-STOP*)

lemma *T-STOP*: $\mathcal{T}$ *STOP* = {[]}
by(*simp add*: *STOP-def TRACES-def FAILURES-def Traces-def Rep-Abs-STOP*)

end

3.4 Deterministic Choice Operator Definition

theory *Det*
imports *Process*
begin

3.4.1 Definition

definition
Det :: [$'\alpha$ *process*,$'\alpha$ *process*] $\Rightarrow$ $'\alpha$ *process* (**infixl** [+] *79*)
where P [+] Q $\equiv$ *Abs-process*({(s,X). s = [] $\wedge$ (s,X) $\in$ $\mathcal{F}$ P $\cap$ $\mathcal{F}$ Q}
$\cup$ {(s,X). $s \neq$ [] $\wedge$ (s,X) $\in$ $\mathcal{F}$ P $\cup$ $\mathcal{F}$ Q}

$\cup \{(s,X).\ s = [] \wedge s \in \mathcal{D}\ P \cup \mathcal{D}\ Q\}$
$\cup \{(s,X).\ s = [] \wedge tick \notin X \wedge [tick] \in \mathcal{T}\ P \cup \mathcal{T}\ Q\}$,
$\mathcal{D}\ P \cup \mathcal{D}\ Q)$

notation
Det (**infixl** $\Box$ *79*)

term ‹$(A \Box B) \Box D' = C$›

lemma *is-process-REP-Det*:
is-process $(\{(s,\ X).\ s = [] \wedge (s,\ X) \in \mathcal{F}\ P \cap \mathcal{F}\ Q\} \cup$
$\{(s,\ X).\ s \neq [] \wedge (s,\ X) \in \mathcal{F}\ P \cup \mathcal{F}\ Q\} \cup$
$\{(s,\ X).\ s = [] \wedge s \in \mathcal{D}\ P \cup \mathcal{D}\ Q\} \cup$
$\{(s,\ X).\ s = [] \wedge tick \notin X \wedge [tick] \in \mathcal{T}\ P \cup \mathcal{T}\ Q\}$,
$\mathcal{D}\ P \cup \mathcal{D}\ Q)$
(**is** *is-process*(*?T*,*?D*))
proof (*simp only*: *fst-conv snd-conv Rep-process is-process-def*
DIVERGENCES-def FAILURES-def, *intro conjI*)
show $([],\ \{\}) \in ?T$
by(*simp add*: *is-processT1*)
next
show $\forall s\ X.\ (s,\ X) \in ?T \longrightarrow$ *front-tickFree s*
by(*auto simp*: *is-processT2*)
next
show $\forall s\ t.\quad (s\ @\ t,\ \{\}) \in ?T \longrightarrow (s,\ \{\}) \in ?T$
by(*auto simp*: *is-processT1 dest*!: *is-processT3*[*rule-format*])
next
show $\forall s\ X\ Y.\ (s,\ Y) \in ?T \wedge X \subseteq Y \longrightarrow (s,\ X) \in ?T$
by(*auto dest*: *is-processT4*[*rule-format*,*OF conj-commute*[*THEN iffD1*],*OF conjI*])
next
show $\forall sa\ X\ Y.\ (sa,\ X) \in ?T \wedge (\forall c.\ c \in Y \longrightarrow (sa\ @\ [c],\ \{\}) \notin ?T) \longrightarrow (sa,\ X \cup Y) \in ?T$
by(*auto simp*: *is-processT5 T-F*)
next
show $\forall s\ X.\ (s\ @\ [tick],\ \{\}) \in ?T \longrightarrow (s,\ X - \{tick\}) \in ?T$
apply((*rule allI*)+, *rule impI*, *rename-tac s X*)
apply(*case-tac s*=[], *simp-all add*: *is-processT6*[*rule-format*] *T-F-spec*)
by(*auto simp*: *is-processT6*[*rule-format*] *T-F-spec*)
next
show $\forall s\ t.\ s \in ?D \wedge$ *tickFree s* $\wedge$ *front-tickFree t* $\longrightarrow s\ @\ t \in ?D$
by(*auto simp*: *is-processT7*)
next
show $\forall s\ X.\ s \in ?D \longrightarrow (s,\ X) \in ?T$
by(*auto simp*: *is-processT8*[*rule-format*])
next
show $\forall s.\ s\ @\ [tick] \in ?D \longrightarrow s \in ?D$
by(*auto intro*!:*is-processT9*[*rule-format*])
qed

lemma *Rep-Abs-Det*:
Rep-process
(*Abs-process*
$(\{(s, X).\ s = [] \wedge (s, X) \in \mathcal{F}\ P \cap \mathcal{F}\ Q\} \cup$
$\{(s, X).\ s \neq [] \wedge (s, X) \in \mathcal{F}\ P \cup \mathcal{F}\ Q\} \cup$
$\{(s, X).\ s = [] \wedge s \in \mathcal{D}\ P \cup \mathcal{D}\ Q\} \cup$
$\{(s, X).\ s = [] \wedge tick \notin X \wedge [tick] \in \mathcal{T}\ P \cup \mathcal{T}\ Q\},$
$\mathcal{D}\ P \cup \mathcal{D}\ Q)) =$
$(\{(s, X).\ s = [] \wedge (s, X) \in \mathcal{F}\ P \cap \mathcal{F}\ Q\} \cup$
$\{(s, X).\ s \neq [] \wedge (s, X) \in \mathcal{F}\ P \cup \mathcal{F}\ Q\} \cup$
$\{(s, X).\ s = [] \wedge s \in \mathcal{D}\ P \cup \mathcal{D}\ Q\} \cup$
$\{(s, X).\ s = [] \wedge tick \notin X \wedge [tick] \in \mathcal{T}\ P \cup \mathcal{T}\ Q\},$
$\mathcal{D}\ P \cup \mathcal{D}\ Q)$
by(*simp only*: *CollectI Rep-process Abs-process-inverse is-process-REP-Det*)

3.4.2 The Projections

lemma *F-Det* :
$\mathcal{F}\ (P \ \Box\ Q) = \{(s,X).\ s = [] \wedge (s,X) \in \mathcal{F}\ P \cap \mathcal{F}\ Q\}$
$\cup \{(s,X).\ s \neq [] \wedge (s,X) \in \mathcal{F}\ P \cup \mathcal{F}\ Q\}$
$\cup \{(s,X).\ s = [] \wedge s \in \mathcal{D}\ P \cup \mathcal{D}\ Q\}$
$\cup \{(s,X).\ s = [] \wedge tick \notin X \wedge [tick] \in \mathcal{T}\ P \cup \mathcal{T}\ Q\}$
by(*subst Failures-def*, *simp only*: *Det-def Rep-Abs-Det FAILURES-def fst-conv*)

3.4.3 Basic Laws

The following theorem of Commutativity helps to simplify the subsequent continuity proof by symmetry breaking. It is therefore already developed here:

lemma *D-Det*: $\mathcal{D}\ (P \ \Box\ Q) = \mathcal{D}\ P \cup \mathcal{D}\ Q$
by(*subst D-def*, *simp only*: *Det-def Rep-Abs-Det DIVERGENCES-def snd-conv*)

lemma *T-Det*: $\mathcal{T}\ (P \ \Box\ Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$
apply(*subst Traces-def*, *simp only*: *Det-def Rep-Abs-Det TRACES-def FAILURES-def*
fst-def snd-conv)
apply(*auto simp*: *T-F F-T is-processT1 Nil-elem-T*)
apply(*rule-tac x*={} **in** *exI*, *simp add*: *T-F F-T is-processT1 Nil-elem-T*)+
done

lemma *Det-commute*:$(P \ \Box\ Q) = (Q \ \Box\ P)$
by(*auto simp*: *Process-eq-spec F-Det D-Det*)

3.4.4 The Continuity-Rule

lemma *mono-Det1*: $P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (Q \ \Box\ S) \subseteq \mathcal{D}\ (P \ \Box\ S)$

apply (*drule le-approx1*)
by (*auto simp*: *Process-eq-spec F-Det D-Det*)

lemma *mono-Det2*:
assumes *ordered*: $P \sqsubseteq Q$
shows $(\forall\ s.\ s \notin \mathcal{D}\ (P \sqcap S) \longrightarrow Ra\ (P \sqcap S)\ s = Ra\ (Q \sqcap S)\ s)$
proof −
have *A*:$\bigwedge s\ t.\ [] \notin \mathcal{D}\ P \Longrightarrow [] \notin \mathcal{D}\ S \Longrightarrow s = [] \Longrightarrow$
$([],\ t) \in \mathcal{F}\ P \Longrightarrow ([],\ t) \in \mathcal{F}\ S \Longrightarrow ([],\ t) \in \mathcal{F}\ Q$
by (*insert ordered*,*frule-tac X*=*t* **and** *s*=[] **in** *proc-ord2a*, *simp-all*)
have *B*:$\bigwedge s\ t.\ s \notin \mathcal{D}\ P \Longrightarrow s \notin \mathcal{D}\ S \Longrightarrow$
$(s \neq [] \wedge ((s,\ t) \in \mathcal{F}\ P \vee (s,\ t) \in \mathcal{F}\ S) \longrightarrow (s,\ t) \in \mathcal{F}\ Q \vee (s,\ t) \in \mathcal{F}\ S) \wedge$
$(s = [] \wedge tick \notin t \wedge ([tick] \in \mathcal{T}\ P \vee [tick] \in \mathcal{T}\ S) \longrightarrow$
$([],\ t) \in \mathcal{F}\ Q \wedge ([],\ t) \in \mathcal{F}\ S \vee [] \in \mathcal{D}\ Q \vee\ [tick] \in \mathcal{T}\ Q \vee\ [tick] \in \mathcal{T}\ S)$
apply(*intro conjI impI*, *elim conjE disjE*, *rule disjI1*)
apply(*simp-all add*: *proc-ord2a*[*OF ordered*,*symmetric*])
apply(*elim conjE disjE*,*subst le-approx2T*[*OF ordered*])
apply(*frule is-processT9-S-swap*, *simp-all*)
done
have *C*: $\bigwedge s.\ s \notin \mathcal{D}\ P \Longrightarrow s \notin \mathcal{D}\ S \Longrightarrow$
$\{X.\ s = [] \wedge (s,\ X) \in \mathcal{F}\ Q \wedge (s,\ X) \in \mathcal{F}\ S \vee$
$s \neq [] \wedge ((s,\ X) \in \mathcal{F}\ Q \vee (s,\ X) \in \mathcal{F}\ S) \vee$
$s = [] \wedge s \in \mathcal{D}\ Q \vee s = [] \wedge tick \notin X \wedge$
$([tick] \in \mathcal{T}\ Q \vee [tick] \in \mathcal{T}\ S)\}$
$\subseteq \{X.\ s = [] \wedge (s,\ X) \in \mathcal{F}\ P \wedge (s,\ X) \in \mathcal{F}\ S \vee$
$s \neq [] \wedge ((s,\ X) \in \mathcal{F}\ P \vee (s,\ X) \in \mathcal{F}\ S) \vee$
$s = [] \wedge tick \notin X \wedge ([tick] \in \mathcal{T}\ P \vee [tick] \in \mathcal{T}\ S)\}$
apply(*intro subsetI*, *frule is-processT9-S-swap*, *simp*)
apply(*elim conjE disjE*, *simp-all add*: *proc-ord2a*[*OF ordered*,*symmetric*] *is-processT8-S*)
apply(*insert le-approx1*[*OF ordered*] *le-approx-lemma-T*[*OF ordered*])
by (*auto simp*: *proc-ord2a*[*OF ordered*,*symmetric*])
show *?thesis*
apply(*simp add*: *Process-eq-spec F-Det D-Det Ra-def min-elems-def*)
apply(*intro allI impI equalityI C*, *simp-all*)
apply(*intro allI impI subsetI*, *simp*)
apply(*metis A B*)
done
qed

lemma *mono-Det3*: $P \sqsubseteq Q \Longrightarrow min\text{-}elems\ (\mathcal{D}\ (P \sqcap S)) \subseteq \mathcal{T}\ (Q \sqcap S)$
apply (*frule le-approx3*)
apply (*simp add*: *Process-eq-spec F-Det T-Det D-Det Ra-def min-elems-def subset-iff*)
apply (*auto intro*:*D-T*)
done

lemma *mono-Det*[*simp*] : $P \sqsubseteq Q \Longrightarrow (P \sqcap S) \sqsubseteq (Q \sqcap S)$

by (*auto simp*: *le-approx-def mono-Det1 mono-Det2 mono-Det3*)

lemma *mono-Det-sym*[*simp*] : $P \sqsubseteq Q \Longrightarrow (S \,\Box\, P) \sqsubseteq (S \,\Box\, Q)$
by (*simp add*: *Det-commute*)

lemma *cont-Det0*:
assumes C : *chain* Y
shows (*lim-proc* (*range* Y) $\Box$ S) = *lim-proc* (*range* ($\lambda i.\ Y\ i \,\Box\, S$))
proof −
  have C':*chain* ($\lambda i.\ Y\ i \,\Box\, S$)
    by(*auto intro*!:*chainI simp*:*chainE*[*OF C*])
  show *?thesis*
  apply (*subst Process-eq-spec*)
  apply (*simp add*: *D-Det F-Det*)
  apply(*simp add*: *F-LUB*[*OF C*] *D-LUB*[*OF C*] *T-LUB*[*OF C*] *F-LUB*[*OF C'*] *D-LUB*[*OF C'*] *T-LUB*[*OF C'*])
  apply(*safe*)
  apply(*auto simp*: *D-Det F-Det T-Det*)
  using *NF-ND* **apply** *blast*
  using *is-processT6-S2 is-processT8* **apply** *blast*
  apply (*metis D-T append-Nil front-tickFree-single process-charn tickFree-Nil*)
  using *NF-ND is-processT6-S2* **apply** *blast*
  using *NF-ND is-processT6-S2* **by** *blast*
qed

lemma *cont-Det*:
assumes C: *chain* Y
shows $((\bigsqcup\ i.\ Y\ i) \,\Box\, S) = (\bigsqcup\ i.\ (Y\ i \,\Box\, S))$
apply(*insert C*)
apply(*subst limproc-is-thelub*, *simp*)
apply(*subst limproc-is-thelub*)
apply(*rule chainI*)
apply(*frule-tac i=i* **in** *chainE*)
apply(*simp*)
apply(*erule cont-Det0*)
done

lemma *cont-Det'*:
assumes *chain*:*chain* Y
shows $((\bigsqcup\ i.\ Y\ i) \,\Box\, S) = (\bigsqcup\ i.\ (Y\ i \,\Box\, S))$
proof −
  have *chain'*:*chain* ($\lambda i.\ Y\ i \,\Box\, S$)
    by(*auto intro*!:*chainI simp*: *chainE*[*OF chain*])
  have B:$\mathcal{F}$ (*lim-proc* (*range* Y) $\Box$ S) $\subseteq$ $\mathcal{F}$ (*lim-proc* (*range* ($\lambda i.\ Y\ i \,\Box\, S$)))
    apply(*simp add*: *D-Det F-Det F-LUB T-LUB D-LUB chain chain'*)

```
        apply(intro conjI subsetI, simp-all)
        by(auto split:prod.split prod.split-asm)
  have C:𝓕 (lim-proc (range (λi. Y i □ S))) ⊆ 𝓕(lim-proc (range Y) □ S)
    proof −
    have C1 : ⋀ba ab ac. ∀ a. ([], ba) ∈ 𝓕 (Y a) ∧ ([], ba) ∈ 𝓕 S ∨ [] ∈ 𝓓 (Y a)
⟹
              [] ∉ 𝓓 (Y ab) ⟹ [] ∉ 𝓓 S ⟹  tick ∈ ba ⟹ ([], ba) ∈ 𝓕 (Y ac)
            using is-processT8 by auto
    have C2 : ⋀ba ab ac ad D. ∀ a. ([], ba) ∈ 𝓕 (Y a) ∧ ([], ba) ∈ 𝓕 S ∨ [] ∈ 𝓓
(Y a)
                        ∨ tick ∉ ba ∧ [tick] ∈ 𝓣 (Y a) ⟹
              []∉D(Y ab) ⟹ []∉D S ⟹ ([],ba)∉ 𝓕(Y ac) ⟹ [tick] ∉ 𝓣 S ⟹
[tick]∈𝓣(Y ad)
            using NF-ND is-processT6-S2 by blast
    have C3: ⋀ba ab ac. ∀ a. [] ∈ 𝓓 (Y a) ∨ tick ∉ ba ∧ [tick] ∈ 𝓣 (Y a) ⟹
              [] ∉ 𝓓 (Y ab) ⟹ [] ∉ 𝓓 S ⟹ ([], ba) ∉ 𝓕 S ⟹
              [tick] ∉ 𝓣 S ⟹ [tick] ∈ 𝓣 (Y ac)
              by (metis D-T append-Nil front-tickFree-single process-charn tick-
Free-Nil)
    show ?thesis
        apply(simp add: D-Det F-Det F-LUB T-LUB D-LUB chain chain')
        apply(rule subsetI, simp)
        apply(simp split:prod.split prod.split-asm)
        apply(intro allI impI,simp)
        by (metis C3 is-processT6-S2 is-processT8-S)
    qed
  have D:𝓓 (lim-proc (range Y) □ S) = 𝓓 (lim-proc (range (λi. Y i □ S)))
        by(simp add: D-Det F-Det F-LUB T-LUB D-LUB chain chain')
  show ?thesis
  by(simp add: chain chain' limproc-is-thelub Process-eq-spec equalityI B C D)
qed

lemma Det-cont[simp]:
assumes f:cont f
and     g:cont g
shows     cont (λx. f x □ g x)
proof −
  have A : ⋀x. cont f ⟹ cont (λy. y □ f x)
      apply (rule contI2,rule monofunI)
      apply (auto)
      apply (subst Det-commute,subst cont-Det)
      apply (auto simp: Det-commute)
      done
  have B : ⋀y. cont f ⟹ cont (λx. y □ f x)
      apply (rule-tac c=(λ x. y □ x) in cont-compose)
      apply (rule contI2,rule monofunI)
      apply (auto)
      apply (subst Det-commute,subst cont-Det)
      by (simp-all add: Det-commute)
```

show *?thesis* **using** *f g*
apply(*rule-tac f*=($\lambda x.$ (λ *g*. *f x* $\Box$ *g*)) **in** *cont-apply*)
apply(*auto intro*:*contI2 monofunI simp*:*Det-commute A B*)
done
qed

end

3.5 Nondeterministic Choice Operator Definition

theory *Ndet*
imports *Process*
begin

3.5.1 Definition and Consequences

definition
Ndet :: [$'\alpha$ *process*,$'\alpha$ *process*] $\Rightarrow$ $'\alpha$ *process* (**infixl** |−| *80*)
where P |−| $Q \equiv$ *Abs-process*($\mathcal{F}\ P \cup \mathcal{F}\ Q$, $\mathcal{D}\ P \cup \mathcal{D}\ Q$)

notation
Ndet (**infixl** $\sqcap$ *80*)

lemma *is-process-REP-Ndet*:
is-process ($\mathcal{F}\ P \cup \mathcal{F}\ Q$, $\mathcal{D}\ P \cup \mathcal{D}\ Q$)
proof(*simp only*: *fst-conv snd-conv Rep-process is-process-def DIVERGENCES-def FAILURES-def*,
intro conjI)
show ([], {}) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$)
by(*simp add*: *is-processT1*)
next
show $\forall s\ X.$ (s, X) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$) $\longrightarrow$ *front-tickFree s*
by(*auto simp*: *is-processT2*)
next
show $\forall s\ t.$ (s @ t, {}) $\in$($\mathcal{F}\ P \cup \mathcal{F}\ Q$) $\longrightarrow$ (s, {}) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$)
by (*auto simp*: *is-processT1 dest*!: *is-processT3*[*rule-format*])
next
show $\forall s\ X\ Y.$ (s, Y) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$) $\wedge$ $X \subseteq Y$ $\longrightarrow$ (s, X) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$)
by(*auto dest*: *is-processT4*[*rule-format*,*OF conj-commute*[*THEN iffD1*], *OF conjI*])
next
show $\forall sa\ X\ Y.$ (sa, X) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$) $\wedge$ ($\forall c.\ c \in Y \longrightarrow$ (sa @ [c], {}) $\notin$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$))
$\longrightarrow$ (sa, $X \cup Y$) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$)
by(*auto simp*: *is-processT5 T-F*)
next
show $\forall s\ X.$ (s @ [*tick*], {}) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$) $\longrightarrow$ (s, $X - \{tick\}$) $\in$ ($\mathcal{F}\ P \cup \mathcal{F}\ Q$)

apply((*rule allI*)+, *rule impI*)
apply(*rename-tac s X*, *case-tac s*=[], *simp-all add*: *is-processT6*[*rule-format*] *T-F-spec*)
apply(*erule disjE*,*simp-all add*: *is-processT6*[*rule-format*] *T-F-spec*)
apply(*erule disjE*,*simp-all add*: *is-processT6*[*rule-format*] *T-F-spec*)
done
next
show $\forall s\ t.\ s \in \mathcal{D}\ P \cup \mathcal{D}\ Q \wedge$ *tickFree s* $\wedge$ *front-tickFree t* $\longrightarrow s\ @\ t \in \mathcal{D}\ P \cup \mathcal{D}\ Q$
by(*auto simp*: *is-processT7*)
next
show $\forall s\ X.\ s \in \mathcal{D}\ P \cup \mathcal{D}\ Q \longrightarrow (s, X) \in (\mathcal{F}\ P \cup \mathcal{F}\ Q)$
by(*auto simp*: *is-processT8*[*rule-format*])
next
show $\forall s.\ s\ @\ [tick] \in \mathcal{D}\ P \cup \mathcal{D}\ Q \longrightarrow s \in \mathcal{D}\ P \cup \mathcal{D}\ Q$
by(*auto intro*!:*is-processT9*[*rule-format*])
qed

lemma *Rep-Abs-Ndet*:
Rep-process(*Abs-process*($\mathcal{F}\ P \cup \mathcal{F}\ Q$, $\mathcal{D}\ P \cup \mathcal{D}\ Q$)) = ($\mathcal{F}\ P \cup \mathcal{F}\ Q$, $\mathcal{D}\ P \cup \mathcal{D}\ Q$)
by(*simp only*:*CollectI Rep-process Abs-process-inverse is-process-REP-Ndet*)

3.5.2 Some Laws

The commutativity of the operator helps to simplify the subsequent continuity proof and is therefore developed here:

lemma *F-Ndet* : $\mathcal{F}\ (P \sqcap Q) = \mathcal{F}\ P \cup \mathcal{F}\ Q$
apply (*subst Failures-def*)
apply (*simp only*: *Ndet-def*)
by (*simp add*: *FAILURES-def Rep-Abs-Ndet*)

lemma *D-Ndet* : $\mathcal{D}(P \sqcap Q) = \mathcal{D}\ P \cup \mathcal{D}\ Q$
by(*subst D-def*, *simp only*: *Ndet-def Rep-Abs-Ndet DIVERGENCES-def snd-conv*)

lemma *T-Ndet* : $\mathcal{T}\ (P \sqcap Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$
apply(*subst Traces-def*, *simp only*: *Ndet-def Rep-Abs-Ndet TRACES-def FAILURES-def*
fst-def snd-conv)
apply(*auto simp*: *T-F F-T is-processT1 Nil-elem-T*)
apply(*rule-tac x*={} **in** *exI*, *simp add*: *T-F F-T is-processT1 Nil-elem-T*)+
done

lemma *Ndet-commute*: $(P \sqcap Q) = (Q \sqcap P)$
by(*auto simp*: *Process-eq-spec F-Ndet D-Ndet*)

3.5.3 The Continuity Rule

lemma *mono-Ndet1*: $P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (Q \sqcap S) \subseteq \mathcal{D}\ (P \sqcap S)$
apply(*drule le-approx1*)
by (*auto simp*: *Process-eq-spec F-Ndet D-Ndet*)

lemma *mono-Ndet2*: $P \sqsubseteq Q \Longrightarrow (\forall\ s.\ s \notin \mathcal{D}\ (P \sqcap S) \longrightarrow Ra\ (P \sqcap S)\ s = Ra\ (Q \sqcap S)\ s)$
apply(*subst* (*asm*) *le-approx-def*)
by (*auto simp*: *Process-eq-spec F-Ndet D-Ndet Ra-def*)

lemma *mono-Ndet3*: $P \sqsubseteq Q \Longrightarrow min\text{-}elems\ (\mathcal{D}\ (P \sqcap S)) \subseteq \mathcal{T}\ (Q \sqcap S)$
apply(*auto dest*!:*le-approx3 simp*: *min-elems-def subset-iff F-Ndet D-Ndet T-Ndet*)
apply(*erule-tac x=t* **in** *allE*, *auto*)
by (*erule-tac x=[]* **in** *allE*, *auto simp*: *less-list-def Nil-elem-T D-T*)

lemma *mono-Ndet*[*simp*] : $P \sqsubseteq Q \Longrightarrow (P \sqcap S) \sqsubseteq (Q \sqcap S)$
by(*auto simp*:*le-approx-def mono-Ndet1 mono-Ndet2 mono-Ndet3*)

lemma *mono-Ndet-sym*[*simp*] : $P \sqsubseteq Q \Longrightarrow (S \sqcap P) \sqsubseteq (S \sqcap Q)$
by (*auto simp*: *Ndet-commute*)

lemma *cont-Ndet1*:
assumes *chain*:*chain Y*
shows $((\bigsqcup\ i.\ Y\ i) \sqcap S) = (\bigsqcup\ i.\ (Y\ i \sqcap S))$
proof –
  have A : *chain* $(\lambda i.\ Y\ i \sqcap S)$
    apply(*insert chain*,*rule chainI*)
    apply(*frule-tac i=i* **in** *chainE*)
    by(*simp*)
  show *?thesis* **using** *chain*
    by(*auto simp add*: *limproc-is-thelub Process-eq-spec D-Ndet F-Ndet F-LUB D-LUB A*)
qed

lemma *Ndet-cont*[*simp*]:
assumes f: *cont f*
and g: *cont g*
shows *cont* $(\lambda x.\ f\ x \sqcap g\ x)$
proof –
  have A:$\bigwedge x.$ *cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *cont* $(\lambda X.\ (f\ x) \sqcap X)$
    apply(*rule contI2*, *rule monofunI*)
    apply(*auto*)
    apply(*subst Ndet-commute*, *subst cont-Ndet1*)
    by (*auto simp*:*Ndet-commute*)
  have B:$\bigwedge y.$ *cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *cont* $(\lambda x.\ f\ x \sqcap y)$
    apply(*rule-tac c=*$(\lambda\ g.\ g \sqcap y)$ **in** *cont-compose*)
    apply(*rule contI2*,*rule monofunI*)

```
        by   (simp-all add: cont-Ndet1)
  show ?thesis using f g
  by (rule-tac f=(λ x. (λ g. f x ⊓ g)) in cont-apply, auto simp: A B)
qed


end
```

3.6 The Sequence Operator

```
theory Seq
 imports Process
begin
```

3.6.1 Definition

```
abbreviation div-seq P Q    ≡ {t1 @ t2 |t1 t2. t1 ∈ 𝒟 P ∧ tickFree t1 ∧
front-tickFree t2}
                            ∪ {t1 @ t2 |t1 t2. t1 @ [tick] ∈ 𝒯 P ∧ t2 ∈ 𝒟 Q}

definition Seq :: ['a process,'a process] ⇒ 'a process (infixl ; 74)
where      P ; Q ≡ Abs-process
                   ({(t, X). (t, X ∪ {tick}) ∈ ℱ P ∧ tickFree t} ∪
                    {(t, X). ∃t1 t2. t = t1 @ t2 ∧ t1 @ [tick] ∈ 𝒯 P ∧ (t2,
X) ∈ ℱ Q} ∪
                    {(t, X). t ∈ div-seq P Q},
                    div-seq P Q)

lemma Seq-maintains-is-process:
    is-process    ({(t, X). (t, X ∪ {tick}) ∈ ℱ P ∧ tickFree t} ∪
                  {(t, X). ∃t1 t2. t = t1 @ t2 ∧ t1 @ [tick] ∈ 𝒯 P ∧ (t2, X) ∈
ℱ Q} ∪
                  {(t, X). t ∈ div-seq P Q},
                  div-seq P Q)
     (is is-process(?f, ?d))
proof(simp only: fst-conv snd-conv Failures-def is-process-def FAILURES-def DI-
VERGENCES-def,
    fold FAILURES-def DIVERGENCES-def,fold Failures-def,intro conjI)
  show ([], {}) ∈ ?f
   apply(cases [tick] ∈ 𝒯 P, simp-all add: is-processT1)
   using F-T is-processT5-S6 by blast
next
  show ∀s X. (s, X) ∈ ?f ⟶ front-tickFree s
   by (auto simp:is-processT2 append-T-imp-tickFree front-tickFree-append D-imp-front-tickFree)

next
  show ∀s t. (s @ t, {}) ∈ ?f ⟶ (s, {}) ∈ ?f
   apply auto
      apply (metis F-T append-Nil2 is-processT is-processT3-SR is-processT5-S3)
```

apply(*simp add*:*append-eq-append-conv2*)
apply (*metis T-F-spec append-Nil2 is-processT is-processT5-S4*)
apply (*metis append-self-conv front-tickFree-append front-tickFree-mono is-processT2-TR*
no-Trace-implies-no-Failure non-tickFree-implies-nonMt non-tickFree-tick)
apply (*metis* (*mono-tags*, *lifting*) *F-T append-Nil2 is-processT5-S3 process-charn*)
apply (*metis front-tickFree-append front-tickFree-mono self-append-conv*)
apply(*simp add*:*append-eq-append-conv2*)
apply (*metis T-F-spec append-Nil2 is-processT is-processT5-S4*)
by (*metis D-imp-front-tickFree append-T-imp-tickFree front-tickFree-append*
front-tickFree-mono not-Cons-self2 self-append-conv)
next
show $\forall s\ X\ Y.\ (s,\ Y) \in ?f \wedge X \subseteq Y \longrightarrow (s,\ X) \in ?f$
apply *auto*
apply (*metis insert-mono is-processT4-S1 prod.sel*(*1*) *prod.sel*(*2*))
apply (*metis is-processT4*)
apply (*simp add*: *append-T-imp-tickFree*)
by (*metis process-charn*)
next
{ fix *sa X Y*
have $(sa,\ X \cup \{tick\}) \in \mathcal{F}\ P \Longrightarrow$
$tickFree\ sa \Longrightarrow$
$\forall c.\ c \in Y \wedge c \neq tick \longrightarrow (sa\ @\ [c],\ \{\}) \notin \mathcal{F}\ P \Longrightarrow$
$(sa,\ X \cup Y \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ sa$
apply(*rule-tac* $t{=}X \cup Y \cup \{tick\}$ **and** $s{=}X \cup \{tick\} \cup (Y{-}\{tick\})$ **in** *subst*,*auto*)
by (*metis DiffE Un-insert-left is-processT5 singletonI*)
} **note** *is-processT5-SEQH3* = *this*
have *is-processT5-SEQH4* :
$\bigwedge sa\ X\ Y.\ (sa,\ X \cup \{tick\}) \in \mathcal{F}\ P$
$\Longrightarrow tickFree\ sa$
$\Longrightarrow \forall c.\ c \in Y \longrightarrow (sa@[c],\{tick\}) \notin \mathcal{F}\ P \vee \neg\ tickFree(sa@[c])$
$\Longrightarrow \forall c.\ c \in Y \longrightarrow (\forall t1\ t2.\ sa@[c]{\neq}t1@t2 \vee t1@[tick]{\notin}\mathcal{T}\ P \vee (t2,\{\}){\notin}\mathcal{F}\ Q)$
$\Longrightarrow (sa,\ X \cup Y \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ sa$
by (*metis append-Nil2 is-processT1 is-processT5-S3 is-processT5-SEQH3*
no-Trace-implies-no-Failure tickFree-Cons tickFree-append)
let $?f1 = \{(t,\ X).\ (t,\ X \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ t\} \cup$
$\{(t,\ X).\ \exists t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in \mathcal{T}\ P \wedge (t2,\ X) \in \mathcal{F}\ Q\}$
have *is-processT5-SEQ2*: $\bigwedge sa\ X\ Y.\ (sa,\ X) \in ?f1$
$\Longrightarrow (\forall c.\ c \in Y \longrightarrow (sa@[c],\ \{\}){\notin}?f)$
$\Longrightarrow (sa,\ X{\cup}Y) \in ?f1$
apply (*clarsimp*,*rule is-processT5-SEQH4*[*simplified*])
by (*auto simp*: *is-processT5*)
show $\forall s\ X\ Y.\ (s,\ X) \in ?f \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin ?f) \longrightarrow (s,\ X \cup Y) \in ?f$
apply(*intro allI impI*, *elim conjE UnE*)
apply(*rule rev-subsetD*)

apply(*rule is-processT5-SEQ2*)
apply *auto*
using *is-processT5-S1* **apply** *force*
apply (*simp add*: *append-T-imp-tickFree*)
using *is-processT5*[*rule-format*, *OF conjI*] **by** *force*
next
show $\forall s\ X.\ (s\ @\ [tick],\ \{\}) \in ?f \longrightarrow (s,\ X - \{tick\}) \in ?f$
apply *auto*
apply (*metis* (*no-types*, *lifting*) *append-T-imp-tickFree butlast-append butlast-snoc is-processT2 is-processT6 nonTickFree-n-frontTickFree non-tickFree-tick tickFree-append*)
apply (*metis append-T-imp-tickFree front-tickFree-append front-tickFree-mono is-processT2 non-tickFree-implies-nonMt non-tickFree-tick*)
apply (*metis process-charn*)
apply (*metis front-tickFree-append front-tickFree-implies-tickFree*)
apply (*metis D-T T-nonTickFree-imp-decomp append-T-imp-tickFree append-assoc append-same-eq non-tickFree-implies-nonMt non-tickFree-tick process-charn tickFree-append*)
by (*metis D-imp-front-tickFree append-T-imp-tickFree front-tickFree-append front-tickFree-mono non-tickFree-implies-nonMt non-tickFree-tick*)
next
show $\forall s\ t.\ s \in ?d \wedge tickFree\ s \wedge front\text{-}tickFree\ t \longrightarrow s\ @\ t \in ?d$
apply *auto*
using *front-tickFree-append* **apply** *blast*
by (*metis process-charn*)
next
show $\forall s\ X.\ s \in ?d \longrightarrow (s,\ X) \in ?f$
by *blast*
next
show $\forall s.\ s\ @\ [tick] \in ?d \longrightarrow s \in ?d$
apply *auto*
apply (*metis append-Nil2 front-tickFree-implies-tickFree process-charn*)
by (*metis append1-eq-conv append-assoc front-tickFree-implies-tickFree is-processT2-TR nonTickFree-n-frontTickFree non-tickFree-tick process-charn tickFree-append*)
qed

3.6.2 The Projections

lemmas *Rep-Abs-Seq*[*simp*] = *Abs-process-inverse*[*simplified*, *OF Seq-maintains-is-process*]

lemma
F-Seq : $\mathcal{F}\ (P\ ;\ Q) = \{(t,\ X).\ (t,\ X \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ t\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in \mathcal{T}\ P \wedge (t2,$

$X) \in \mathcal{F}\ Q\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1 \in \mathcal{D}\ P \wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*$\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in \mathcal{T}\ P \wedge t2 \in \mathcal{D}\ Q\}$
unfolding *Seq-def* **by**(*subst Failures-def*, *simp only*:*Rep-Abs-Seq*, *auto simp*: *FAILURES-def*)

lemma
D-Seq : $\mathcal{D}\ (P\ ;\ Q) = \{t1\ @\ t2\ |t1\ t2.\ t1 \in \mathcal{D}\ P \wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*$\} \cup$
$\{t1\ @\ t2\ |t1\ t2.\ t1\ @\ [tick] \in \mathcal{T}\ P \wedge t2 \in \mathcal{D}\ Q\}$
unfolding *Seq-def* **by**(*subst D-def*,*simp only*:*Rep-Abs-Seq*, *simp add*:*DIVERGENCES-def*)

lemma
T-Seq : $\mathcal{T}\ (P\ ;\ Q) = \{t.\ \exists\ X.\ (t,\ X \cup \{tick\}) \in \mathcal{F}\ P \wedge$ *tickFree t*$\} \cup$
$\{t.\ \exists\ t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in \mathcal{T}\ P \wedge t2 \in \mathcal{T}\ Q\} \cup$
$\{t1\ @\ t2\ |t1\ t2.\ t1 \in \mathcal{D}\ P \wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*$\} \cup$
$\{t1\ @\ t2\ |t1\ t2.\ t1\ @\ [tick] \in \mathcal{T}\ P \wedge t2 \in \mathcal{D}\ Q\} \cup$
$\{t1\ @\ t2\ |t1\ t2.\ t1 \in \mathcal{D}\ P \wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*$\} \cup$
$\{t1\ @\ t2\ |t1\ t2.\ t1\ @\ [tick] \in \mathcal{T}\ P \wedge t2 \in \mathcal{D}\ Q\}$
unfolding *Seq-def*
apply(*subst Traces-def*, *simp only*: *Rep-Abs-Seq*, *auto simp*: *TRACES-def FAILURES-def*)
using *F-T* **apply** *fastforce*
apply (*simp add*: *append-T-imp-tickFree*)
using *F-T* **apply** *fastforce*
using *T-F* **by** *blast*

3.6.3 Continuity Rule

lemma *mono-Seq-D11*:
$P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (Q\ ;\ S) \subseteq \mathcal{D}\ (P\ ;\ S)$
apply(*auto simp*: *D-Seq*)
using *le-approx1* **apply** *blast*
using *le-approx-lemma-T* **by** *blast*

lemma *mono-Seq-D12*:
assumes *ordered*: $P \sqsubseteq Q$
shows $(\forall\ s.\ s \notin \mathcal{D}\ (P\ ;\ S) \longrightarrow Ra\ (P\ ;\ S)\ s = Ra\ (Q\ ;\ S)\ s)$
proof −
have *mono-SEQI2a*:$P \sqsubseteq Q \Longrightarrow \forall s.\ s \notin \mathcal{D}\ (P\ ;\ S) \longrightarrow Ra\ (Q\ ;\ S)\ s \subseteq Ra\ (P\ ;\ S)\ s$
apply(*simp add*: *Ra-def D-Seq F-Seq*)
apply(*insert le-approx-lemma-F*[*of P Q*] *le-approx-lemma-T*[*of P Q*], *auto*)
using *le-approx1* **by** *blast+*
have *mono-SEQI2b*:$P \sqsubseteq Q \Longrightarrow \forall s.\ s \notin \mathcal{D}\ (P\ ;\ S) \longrightarrow Ra\ (P\ ;\ S)\ s \subseteq Ra\ (Q\ ;$

S) *s*
apply(*simp add*: *Ra-def D-Seq F-Seq*)
apply(*insert le-approx-lemma-F*[*of P Q*] *le-approx-lemma-T*[*of P Q*]
le-approx1[*of P Q*] *le-approx2T*[*of P Q*], *auto*)
using *le-approx2* **apply** *fastforce*
apply (*metis front-tickFree-implies-tickFree is-processT2-TR process-charn*)
apply (*simp add*: *append-T-imp-tickFree*)
by (*metis front-tickFree-implies-tickFree is-processT2-TR process-charn*)
show *?thesis*
using *ordered mono-SEQI2a mono-SEQI2b* **by**(*blast*)
qed

lemma *minSeqInclu*:
min-elems($\mathcal{D}$ (*P* **;** *S*))
$\subseteq$ *min-elems*($\mathcal{D}$ *P*) $\cup$ {*t1*@*t2*|*t1 t2*. *t1*@[*tick*]$\in\mathcal{T}$ *P*$\wedge$*t1*$\notin\mathcal{D}$ *P*$\wedge$*t2*$\in$*min-elems*($\mathcal{D}$ *S*)}
apply(*auto simp*: *D-Seq min-elems-def*)
apply (*meson process-charn*)
apply (*metis append-Nil2 front-tickFree-Nil front-tickFree-append front-tickFree-mono*

le-list-def less-list-def)
apply (*metis* (*no-types*, *lifting*) *D-imp-front-tickFree Nil-is-append-conv append-T-imp-tickFree*
append-butlast-last-id butlast-append process-charn less-self lim-proc-is-lub3a

list.distinct(*1*) *nil-less*)
by (*metis D-imp-front-tickFree append-Nil2 front-tickFree-Nil front-tickFree-mono process-charn*
list.distinct(*1*) *nonTickFree-n-frontTickFree*)

lemma *mono-Seq-D13* :
assumes *ordered*: *P* $\sqsubseteq$ *Q*
shows *min-elems* ($\mathcal{D}$ (*P* **;** *S*)) $\subseteq$ $\mathcal{T}$ (*Q* **;** *S*)
apply (*insert ordered*, *frule le-approx3*, *rule subset-trans* [*OF minSeqInclu*])
apply (*auto simp*: *F-Seq T-Seq min-elems-def append-T-imp-tickFree*)
apply(*rule-tac x*={} **in** *exI*, *rule is-processT5-S3*)
apply (*metis* (*no-types*, *lifting*) *T-F elem-min-elems le-approx3 less-list-def min-elems5 subset-eq*)
using *Nil-elem-T no-Trace-implies-no-Failure* **apply** *fastforce*
apply (*metis* (*no-types*, *lifting*) *less-self nonTickFree-n-frontTickFree process-charn*)

apply(*rule-tac x*={} **in** *exI*, *metis* (*no-types*, *lifting*) *le-approx2T process-charn*)
by (*metis* (*no-types*, *lifting*) *less-self nonTickFree-n-frontTickFree process-charn*)

lemma *mono-Seq*[*simp*] : *P* $\sqsubseteq$ *Q* $\Longrightarrow$ (*P* **;** *S*) $\sqsubseteq$ (*Q* **;** *S*)
by (*auto simp*: *le-approx-def mono-Seq-D11 mono-Seq-D12 mono-Seq-D13*)

lemma *mono-Seq-D21*:

$P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (S\ ;\ Q) \subseteq \mathcal{D}\ (S\ ;\ P)$
apply(*auto simp*: *D-Seq*)
using *le-approx1* **by** *blast*

lemma *mono-Seq-D22*:
assumes *ordered*: $P \sqsubseteq Q$
shows $(\forall\ s.\ s \notin \mathcal{D}\ (S\ ;\ P) \longrightarrow Ra\ (S\ ;\ P)\ s = Ra\ (S\ ;\ Q)\ s)$
proof −
have *mono-SEQI2a*:$P \sqsubseteq Q \Longrightarrow \forall s.\ s \notin \mathcal{D}\ (S\ ;\ P) \longrightarrow Ra\ (S\ ;\ Q)\ s \subseteq Ra\ (S\ ;\ P)\ s$
apply(*simp add*: *Ra-def D-Seq F-Seq*)
apply(*insert le-approx-lemma-F*[*of P Q*] *le-approx-lemma-T*[*of P Q*], *auto*)
using *le-approx1* **by** *fastforce*+
have *mono-SEQI2b*:$P \sqsubseteq Q \Longrightarrow \forall s.\ s \notin \mathcal{D}\ (S\ ;\ P) \longrightarrow Ra\ (S\ ;\ P)\ s \subseteq Ra\ (S\ ;\ Q)\ s$
apply(*simp add*: *Ra-def D-Seq F-Seq*)
apply(*insert le-approx-lemma-F*[*of P Q*] *le-approx-lemma-T*[*of P Q*]
le-approx1[*of P Q*] *le-approx2T*[*of P Q*], *auto*)
using *le-approx2* **by** *fastforce*+
show *?thesis*
using *ordered mono-SEQI2a mono-SEQI2b* **by**(*blast*)
qed

lemma *mono-Seq-D23* :
assumes *ordered*: $P \sqsubseteq Q$
shows $\textit{min-elems}\ (\mathcal{D}\ (S\ ;\ P)) \subseteq \mathcal{T}\ (S\ ;\ Q)$
apply (*insert ordered*, *frule le-approx3*, *auto simp*: *D-Seq T-Seq min-elems-def*)
apply (*metis* (*no-types*, *lifting*) *D-imp-front-tickFree Nil-elem-T append.assoc below-refl*
front-tickFree-charn less-self min-elems2 no-Trace-implies-no-Failure)
apply (*simp add*: *append-T-imp-tickFree*)
by (*metis* (*no-types*, *lifting*) *D-def D-imp-front-tickFree append-butlast-last-id append-is-Nil-conv*
butlast-append butlast-snoc is-process9 is-process-Rep less-self nonTickFree-n-frontTickFree)

lemma *mono-Seq-sym*[*simp*] : $P \sqsubseteq Q \Longrightarrow (S\ ;\ P) \sqsubseteq (S\ ;\ Q)$
by (*auto simp*: *le-approx-def mono-Seq-D21 mono-Seq-D22 mono-Seq-D23*)

lemma *chain-Seq1*: $\textit{chain}\ Y \Longrightarrow \textit{chain}\ (\lambda i.\ Y\ i\ ;\ S)$
by(*simp add*: *chain-def*)

lemma *chain-Seq2*: $\textit{chain}\ Y \Longrightarrow \textit{chain}\ (\lambda i.\ S\ ;\ Y\ i)$
by(*simp add*: *chain-def*)

lemma *limproc-Seq-D1*: $\textit{chain}\ Y \Longrightarrow \mathcal{D}\ (\textit{lim-proc}\ (\textit{range}\ Y)\ ;\ S) = \mathcal{D}\ (\textit{lim-proc}\ (\textit{range}\ (\lambda i.\ Y\ i\ ;\ S)))$
apply(*auto simp*:*Process-eq-spec D-Seq F-Seq F-LUB D-LUB T-LUB chain-Seq1*)
apply(*blast*)

```
  proof −
    fix x
     assume A:∀ xa. (∃ t1 t2. x = t1 @ t2 ∧ t1 ∈ 𝒟 (Y xa) ∧ tickFree t1 ∧
front-tickFree t2) ∨
              (∃ t1 t2. x = t1 @ t2 ∧ t1 @ [tick] ∈ 𝒯 (Y xa) ∧ t2 ∈ 𝒟 S)
    and B: ∀ t1 t2. x = t1 @ t2 ⟶ (∃ x. t1 @ [tick] ∉ 𝒯 (Y x)) ∨ t2 ∉ 𝒟 S
    and C: chain Y
    thus ∃ t1 t2. x = t1 @ t2 ∧ (∀ x. t1 ∈ 𝒟 (Y x)) ∧ tickFree t1 ∧ front-tickFree
t2
      proof (cases ∃ n. ∀ t1 ≤ x. t1 ∉ 𝒟 (Y n))
        case True
        then obtain n where ∀ t1 ≤ x. t1 ∉ 𝒟 (Y n) by blast
        with A B C show ?thesis
          apply(erule-tac x=n in allE, elim exE disjE, auto simp add:le-list-def)
          by (metis D-T chain-lemma is-processT le-approx2T)
      next
        case False
        from False obtain t1 where D:t1 ≤ x ∧ (∀ n. ∀ t ≤ x. t ∈ 𝒟 (Y n) ⟶ t
≤ t1) by blast
        with False have E:∀ n. t1 ∈ 𝒟 (Y n)
       by (metis append-Nil2 append-T-imp-tickFree front-tickFree-append front-tickFree-mono

                  is-processT le-list-def local.A not-Cons-self2)
        from A B C D E show ?thesis
        by (metis D-imp-front-tickFree Traces-def append-Nil2 front-tickFree-append

                  front-tickFree-implies-tickFree front-tickFree-mono is-processT
                  is-process-Rep le-list-def nonTickFree-n-frontTickFree
                  trace-with-Tick-implies-tickFree-front)
      qed
  qed

lemma limproc-Seq-F1: chain Y ⟹ ℱ (lim-proc (range Y) ; S) = ℱ (lim-proc
(range (λi. Y i ; S)))
 apply(auto simp add:Process-eq-spec D-Seq F-Seq F-LUB D-LUB T-LUB chain-Seq1)
  proof (auto, goal-cases)
    case (1 a b x)
    then show ?case
       apply(erule-tac x=x in allE, elim disjE exE, auto simp add: is-processT7
is-processT8-S)
        apply(rename-tac t1 t2, erule-tac x=t1 in allE, erule-tac x=t1 in allE,
erule-tac x=t1 in allE)
     apply (metis D-T append-T-imp-tickFree chain-lemma is-processT le-approx2T
not-Cons-self2)
       by (metis D-T append-T-imp-tickFree chain-lemma is-processT le-approx2T
not-Cons-self2)
  next
    case (2 a b)
    assume A1:∀ t1 t2. a = t1 @ t2 ⟶ (∃ x. t1 @ [tick] ∉ 𝒯 (Y x)) ∨ t2 ∉ 𝒟 S
```

and *A2*:$\forall$ *t1*. *tickFree t1* $\longrightarrow$ ($\forall$ *t2*. *a* = *t1* @ *t2* $\longrightarrow$ ($\exists$ *x*. *t1* $\notin \mathcal{D}$ (*Y x*)) $\vee$ $\neg$ *front-tickFree t2*)
and *A3*:$\forall$ *t1 t2*. *a* = *t1* @ *t2* $\longrightarrow$ ($\exists$ *x*. *t1* @ [*tick*] $\notin \mathcal{T}$ (*Y x*)) $\vee$ (*t2*, *b*) $\notin \mathcal{F}$ *S*
and *B*: $\forall$ *x*. (*a*, *insert tick b*) $\in \mathcal{F}$ (*Y x*) $\wedge$ *tickFree a* $\vee$
($\exists$ *t1 t2*. *a* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in \mathcal{T}$ (*Y x*) $\wedge$ (*t2*, *b*) $\in \mathcal{F}$ *S*) $\vee$
($\exists$ *t1 t2*. *a* = *t1* @ *t2* $\wedge$ *t1* $\in \mathcal{D}$ (*Y x*) $\wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*) $\vee$
($\exists$ *t1 t2*. *a* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in \mathcal{T}$ (*Y x*) $\wedge$ *t2* $\in \mathcal{D}$ *S*)
and *C*:*chain Y*
have *E*:$\neg$ *tickFree a* $\Longrightarrow$ *False*
proof −
assume *F*:$\neg$ *tickFree a*
from *A* **obtain** *f*
where *D*:*f* = (λ*t2*. {*n*. $\exists$ *t1*. *a* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in \mathcal{T}$ (*Y n*) $\wedge$ (*t2*, *b*) $\in \mathcal{F}$ *S*}
$\cup$ {*n*. $\exists$ *t1*. *a* = *t1* @ *t2* $\wedge$ *t1* $\in \mathcal{D}$ (*Y n*) $\wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*})
by *simp*
with *B F* **have** $\forall$ *n*. *n* $\in$ ($\bigcup$ *x*$\in${*t*. $\exists$ *t1*. *a* = *t1* @ *t*}. *f x*)
(**is** $\forall$ *n*. *n* $\in$ *?S f*) **using** *NF-ND* **by** *fastforce*
hence *infinite* (*?S f*) **by** (*simp add*: *Sup-set-def*)
then obtain *t2* **where** *E*:*t2*$\in${*t*. $\exists$ *t1*. *a* = *t1* @ *t*} $\wedge$ *infinite* (*f t2*) **using** *suffixes-fin* **by** *blast*
{ assume *E1*:*infinite*{*n*. $\exists$ *t1*. *a* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in \mathcal{T}$ (*Y n*) $\wedge$ (*t2*, *b*) $\in \mathcal{F}$ *S*}
(**is** *infinite ?E1*)
with *E* **obtain** *t1* **where** *F*:*a* = *t1* @ *t2* $\wedge$ (*t2*, *b*) $\in \mathcal{F}$ *S* **using** *D not-finite-existsD* **by** *blast*
with *A3* **obtain** *m* **where** *G*:*t1*@ [*tick*] $\notin \mathcal{T}$ (*Y m*) **by** *blast*
with *E1* **obtain** *n* **where** *n* $\geq$ *m* $\wedge$ *n* $\in$ *?E1* **by** (*meson finite-nat-set-iff-bounded-le nat-le-linear*)
with *D* **have** *n* $\geq$ *m* $\wedge$ *t1*@ [*tick*] $\in \mathcal{T}$ (*Y n*) **by** (*simp add*: *F*)
with *G C* **have** *False* **using** *le-approx-lemma-T chain-mono* **by** *blast*
} **note** *E1* = *this*
{ assume *E2*:*infinite*{*n*. $\exists$ *t1*. *a* = *t1* @ *t2* $\wedge$ *t1* $\in \mathcal{D}$ (*Y n*) $\wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*}
(**is** *infinite ?E2*)
with *E* **obtain** *t1* **where** *F*:*a* = *t1* @ *t2* $\wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*
using *D not-finite-existsD* **by** *blast*
with *A2* **obtain** *m* **where** *G*:*t1* $\notin \mathcal{D}$ (*Y m*) **by** *blast*
with *E2* **obtain** *n* **where** *n* $\geq$ *m* $\wedge$ *n* $\in$ *?E2* **by** (*meson finite-nat-set-iff-bounded-le nat-le-linear*)
with *D* **have** *n* $\geq$ *m* $\wedge$ *t1* $\in \mathcal{D}$ (*Y n*) **by** (*simp add*: *F*)
with *G C* **have** *False* **using** *le-approx1 chain-mono* **by** *blast*
} **note** *E2* = *this*
from *D E E1 E2* **show** *False* **by** *blast*
qed
from *E* **show** *tickFree a* **by** *blast*

qed

lemma *cont-left-D-Seq* : *chain Y* $\Longrightarrow$ (($\bigsqcup$ *i. Y i*) **;** *S*) = ($\bigsqcup$ *i.* (*Y i* **;** *S*))
by (*simp add*: *Process-eq-spec chain-Seq1 limproc-Seq-D1 limproc-Seq-F1 limproc-is-thelub*)

lemma *limproc-Seq-D2*: *chain Y* $\Longrightarrow$ $\mathcal{D}$ (*S* **;** *lim-proc* (*range Y*)) = $\mathcal{D}$ (*lim-proc* (*range* (λ*i. S* **;** *Y i*)))
apply(*auto simp add*:*Process-eq-spec D-Seq F-Seq F-LUB D-LUB T-LUB chain-Seq2*)
apply(*blast*)
proof −
fix *x*
assume *A*:$\forall$ *t1. t1* @ [*tick*] $\in$ $\mathcal{T}$ *S* $\longrightarrow$ ($\forall$ *t2. x* = *t1* @ *t2* $\longrightarrow$ ($\exists$ *x. t2* $\notin$ $\mathcal{D}$ (*Y x*)))
and *B*: $\forall$ *n.* $\exists$ *t1 t2. x* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in$ $\mathcal{T}$ *S* $\wedge$ *t2* $\in$ $\mathcal{D}$ (*Y n*)
and *C*: *chain Y*
from *A* **obtain** *f* **where** *D*:*f* = (λ*t2.* {*n.* $\exists$ *t1. x* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in$ $\mathcal{T}$ *S* $\wedge$ *t2* $\in$ $\mathcal{D}$ (*Y n*)})
by *simp*
with *B* **have** $\forall$ *n. n* $\in$ ($\bigcup$*x*$\in${*t.* $\exists$ *t1. x* = *t1* @ *t*}. *f x*) (**is** $\forall$ *n. n* $\in$ *?S f*) **by** *fastforce*
hence *infinite* (*?S f*) **by** (*simp add*: *Sup-set-def*)
then obtain *t2* **where** *E*:*t2*$\in${*t.* $\exists$ *t1. x* = *t1* @ *t*} $\wedge$ *infinite* (*f t2*) **using** *suffixes-fin* **by** *blast*
then obtain *t1* **where** *F*:*x* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in$ $\mathcal{T}$ *S* **using** *D not-finite-existsD* **by** *blast*
from *A F* **obtain** *m* **where** *G*:*t2* $\notin$ $\mathcal{D}$ (*Y m*) **by** *blast*
with *E* **obtain** *n* **where** *n* $\geq$ *m* $\wedge$ *n* $\in$ (*f t2*) **by** (*meson finite-nat-set-iff-bounded-le nat-le-linear*)
with *D* **have** *n* $\geq$ *m* $\wedge$ *t2* $\in$ $\mathcal{D}$ (*Y n*) **by** *blast*
with *G C* **have** *False* **using** *le-approx1 po-class.chain-mono* **by** *blast*
thus $\exists$ *t1 t2. x* = *t1* @ *t2* $\wedge$ *t1* $\in$ $\mathcal{D}$ *S* $\wedge$ *tickFree t1* $\wedge$ *front-tickFree t2* **..**
qed

lemma *limproc-Seq-F2*:
chain Y $\Longrightarrow$ $\mathcal{F}$ (*S* **;** *lim-proc* (*range Y*)) = $\mathcal{F}$ (*lim-proc* (*range* (λ*i. S* **;** *Y i*)))
apply(*auto simp*:*Process-eq-spec D-Seq F-Seq T-Seq F-LUB D-LUB D-LUB-2 T-LUB T-LUB-2 chain-Seq2 del*:*conjI*)
apply(*auto*)[*1*]
apply(*auto*)[*1*]
proof−
fix *x X*
assume *A*:$\forall$ *t1. t1* @ [*tick*] $\in$ $\mathcal{T}$ *S* $\longrightarrow$ ($\forall$ *t2. x* = *t1* @ *t2* $\longrightarrow$ ($\exists$ *m.* (*t2*, *X*) $\notin$ $\mathcal{F}$ (*Y m*)))
and *B*: $\forall$ *n.* ($\exists$ *t1 t2. x* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in$ $\mathcal{T}$ *S* $\wedge$ (*t2*, *X*) $\in$ $\mathcal{F}$ (*Y n*)) $\vee$
($\exists$ *t1 t2. x* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in$ $\mathcal{T}$ *S* $\wedge$ *t2* $\in$ $\mathcal{D}$ (*Y n*))
and *C*: *chain Y*
hence *D*:$\forall$ *n.* ($\exists$ *t1 t2. x* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in$ $\mathcal{T}$ *S* $\wedge$ (*t2*, *X*) $\in$ $\mathcal{F}$ (*Y n*)) **by** (*meson NF-ND*)
from *A* **obtain** *f* **where** *D*:*f* = (λ*t2.* {*n.* $\exists$ *t1. x* = *t1* @ *t2* $\wedge$ *t1* @ [*tick*] $\in$ $\mathcal{T}$

$S \wedge (t2,\ X) \in \mathcal{F}\ (Y\ n)\})$
by *simp*
with D **have** $\forall n.\ n \in (\bigcup x{\in}\{t.\ \exists t1.\ x = t1\ @\ t\}.\ f\ x)$ **using** B *NF-ND* **by** *fastforce*
hence *infinite* $(\bigcup x{\in}\{t.\ \exists t1.\ x = t1\ @\ t\}.\ f\ x)$ **by** (*simp add*: *Sup-set-def*)
then obtain $t2$ **where** $E{:}t2{\in}\{t.\ \exists t1.\ x = t1\ @\ t\} \wedge$ *infinite* $(f\ t2)$ **using** *suffixes-fin* **by** *blast*
then obtain $t1$ **where** $F{:}x = t1\ @\ t2 \wedge t1\ @\ [tick] \in \mathcal{T}\ S$ **using** D *not-finite-existsD* **by** *blast*
from $A\ F$ **obtain** m **where** $G{:}(t2,\ X) \notin \mathcal{F}\ (Y\ m)$ **by** *blast*
with E **obtain** n **where** $n \geq m \wedge n \in (f\ t2)$ **by** (*meson finite-nat-set-iff-bounded-le nat-le-linear*)
with D **have** $n \geq m \wedge (t2,\ X) \in \mathcal{F}\ (Y\ n)$ **by** *blast*
with $G\ C$ **have** *False* **using** *is-processT8 po-class.chain-mono proc-ord2a* **by** *blast*
thus $(x,\ insert\ tick\ X) \in \mathcal{F}\ S \wedge tickFree\ x$ **..**
qed

lemma *cont-right-D-Seq* : $chain\ Y \Longrightarrow (S\ ;\ (\bigsqcup\ i.\ Y\ i)) = (\bigsqcup\ i.\ (S\ ;\ Y\ i))$
by (*simp add*: *Process-eq-spec chain-Seq2 limproc-Seq-D2 limproc-Seq-F2 limproc-is-thelub*)

lemma *Seq-cont*[*simp*]:
assumes $f{:}cont\ f$
and $g{:}cont\ g$
shows $cont\ (\lambda x.\ f\ x\ ;\ g\ x)$
proof −
have $A : \bigwedge x.\ cont\ g \Longrightarrow cont\ (\lambda y.\ y\ ;\ g\ x)$
apply (*rule contI2*, *rule monofunI*)
apply (*auto*)
by (*simp add*: *cont-left-D-Seq*)
have $B : \bigwedge y.\ cont\ g \Longrightarrow cont\ (\lambda x.\ y\ ;\ g\ x)$
apply (*rule-tac* $c{=}(\lambda\ x.\ y\ ;\ x)$ **in** *cont-compose*)
apply (*rule contI2*,*rule monofunI*)
by (*auto simp add*: *chain-Seq2 cont-right-D-Seq*)
show *?thesis* **using** $f\ g$
apply(*rule-tac* $f{=}(\lambda x.\ (\lambda\ f.\ f\ ;\ g\ x))$ **in** *cont-apply*)
by(*auto intro*:*contI2 monofunI simp*:$A\ B$)
qed

end

Chapter 4

Concurrent CSP Operators

4.1 The Hiding Operator

theory *Hiding*
imports *Process*
begin

4.1.1 Preliminaries : primitives and lemmas

abbreviation *trace-hide t A* $\equiv$ *filter* $(\lambda x.\ x \notin A)$ *t*

lemma *Hiding-tickFree* : *tickFree s* $\longleftrightarrow$ *tickFree* (*trace-hide s* (*ev*'*A*))
  by (*auto simp add*: *tickFree-def*)

lemma *Hiding-fronttickFree* : *front-tickFree s* $\Longrightarrow$ *front-tickFree* (*trace-hide s* (*ev*'*A*))
  apply(*auto simp add*: *front-tickFree-charn tickFree-def split*:*if-splits*)
  by (*metis Hiding-tickFree list-nonMt-append tickFree-append tickFree-def*)

lemma *trace-hide-union*[*simp*] : *trace-hide t* (*ev* ' $(A \cup B)$) = *trace-hide* (*trace-hide t* (*ev* ' *A*)) (*ev* ' *B*)
  by (*subgoal-tac ev* ' $(A \cup B)$ = (*ev* ' *A*) $\cup$ (*ev* ' *B*)) *auto*

abbreviation *isInfHiddenRun f P A* $\equiv$ *strict-mono f* $\wedge$ $(\forall\ i.\ f\ i \in \mathcal{T}\ P) \wedge$
$(\forall\ i.$ *trace-hide* $(f\ i)$ (*ev* ' *A*) = *trace-hide* $(f\ (0{::}nat))$ (*ev* ' *A*))

lemma *isInfHiddenRun-1*: *isInfHiddenRun f P A* $\longleftrightarrow$ *strict-mono f* $\wedge$ $(\forall\ i.\ f\ i \in \mathcal{T}\ P) \wedge$
$(\forall i.\ \exists t.\ f\ i = f\ 0\ @\ t \wedge set\ t \subseteq$ (*ev* ' *A*))
(**is** *?A* $\longleftrightarrow$ *?B*)
proof
  assume *A*: *?A*
  {
    fix *i*
    from *A* **have** $f\ 0 \leq f\ i$ **using** *strict-mono-less-eq* **by** *blast*

```
    then obtain t where B:f i = f 0 @ t by (metis le-list-def)
    hence trace-hide (f i) (ev ‘ A) = (trace-hide (f 0) (ev ‘ A)) @ (trace-hide t (ev
‘ A)) by simp
    with A have trace-hide t (ev ‘ A) = [] by (metis append-self-conv)
    with B have ∃ t. f i = f 0 @ t ∧ set t ⊆ (ev ‘ A)
      using filter-empty-conv[of λx. x ∉ (ev ‘ A)] by auto
  }
  with A show ?B by simp
next
  assume B: ?B
  {
    fix i
    from B obtain t where B:f i = f 0 @ t and set t ⊆ (ev ‘ A) by blast
     hence trace-hide (f i) (ev ‘ A) = (trace-hide (f 0) (ev ‘ A)) by (simp add:
subset-iff)
  }
  with B show ?A by simp
qed
```

4.1.2 The Hiding Operator Definition

```
abbreviation div-hide P A ≡ {s. ∃ t u. front-tickFree u ∧
                                tickFree t ∧ s = trace-hide t (ev ‘ A) @ u ∧
                                  (t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P A ∧ t ∈
range f))}

definition Hiding :: ['α process ,'α set] => 'α process      (‹- \ -› [57,56] 57)
where
  ‹P \ A ≡ Abs-process ({(s,X). ∃ t. s = trace-hide t (ev‘A) ∧ (t,X ∪ (ev‘A)) ∈
ℱ P} ∪
                    {(s,X). s ∈ div-hide P A}, div-hide P A)›

lemma inf-hidden:
  assumes as1:∀ t. trace-hide t (ev ‘ A) = trace-hide s (ev ‘ A) ⟶ (t, ev ‘ A) ∉
ℱ P
      and as2:s ∈ 𝒯 P
    shows ∃ f. isInfHiddenRun f P A ∧ s ∈ range f
proof
  define f where A:f = rec-nat s (λi t. (let e = SOME e. e ∈ ev‘A ∧ t @ [e] ∈
𝒯 P in t @ [e]))
  hence B:strict-mono f by (simp add:strict-mono-def lift-Suc-mono-less-iff)
  from A have C:s ∈ range f
    by (metis (mono-tags, lifting) old.nat.simps(6) range-eqI)
  { fix i
    have f i ∈ 𝒯 P ∧ trace-hide (f i) (ev ‘ A) = (trace-hide s (ev ‘ A))
    proof(induct i, simp add: A Nil-elem-T as2)
      case (Suc i)
      with as1[THEN spec, of f i] have a:∃ e. e ∈ ev‘A ∧ f i @ [e] ∈ 𝒯 P
        using is-processT5-S7 by force
```

```
    from A have b:f (Suc i) = (let e = SOME e. e ∈ ev`A ∧ f i @ [e] ∈ 𝒯 P in
f i @ [e])
      by simp
    with Suc a[THEN someI-ex] show ?case by simp
  qed
 }
 with B C show isInfHiddenRun f P A ∧ s ∈ range f by simp
qed
```

4.1.3 Consequences

```
lemma trace-hide-append: s @ t = trace-hide ta (ev ` A) ⟹ ∃ ss tt. ta = ss@tt
∧
                                                    s = trace-hide ss (ev ` A) ∧
                                                      t = trace-hide tt (ev ` A)
proof(induct ta arbitrary:s t)
 case Nil thus ?case by simp
next
 case (Cons a ta) thus ?case
 proof(cases a ∈ (ev ` A))
   case True
   hence s @ t = trace-hide ta (ev ` A) by (simp add: Cons)
   with Cons obtain ss tt where ta = ss @ tt ∧ s = trace-hide ss (ev ` A)
                              ∧ t = trace-hide tt (ev ` A) by blast
   with True Cons show ?thesis by (rule-tac x=a#ss in exI, rule-tac x=tt in
exI, auto)
 next
   case False thus ?thesis
   proof(cases s)
     case Nil thus ?thesis using Cons by fastforce
   next
     case Cons2:(Cons aa tls)
     with False have A:a = aa ∧ tls @ t = trace-hide ta (ev ` A) using Cons by
auto
     with Cons obtain ss tt where ta = ss @ tt ∧ tls = trace-hide ss (ev ` A)
                              ∧ t = trace-hide tt (ev ` A) by blast
     with Cons2 A False show ?thesis by (rule-tac x=a#ss in exI, rule-tac x=tt
in exI, auto)
   qed
 qed
qed

lemma Hiding-maintains-is-process:
    is-process    ({(s,X). ∃ t. s = trace-hide t (ev`A) ∧ (t,X ∪ (ev`A)) ∈ ℱ P} ∪
                    {(s,X). s ∈ div-hide P A}, div-hide P A) (is is-process(?f, ?d))
proof (simp only: fst-conv snd-conv Failures-def is-process-def FAILURES-def DI-
VERGENCES-def,
   fold FAILURES-def DIVERGENCES-def,fold Failures-def,intro conjI, goal-cases)
 case 1 thus ?case
```

```
  proof (auto, rule not-not[THEN iffD1], rule notI, simp, goal-cases)
    case 1
     from inf-hidden[of A [] P] obtain f where A:isInfHiddenRun f P A ∧ [] ∈
range f
      using 1(2) Nil-elem-T by auto
   from A 1(1)[THEN spec, of []] filter.simps(1) tickFree-Nil show ?case by auto
  qed
next
  case 2 thus ?case
     using is-processT2 Hiding-fronttickFree front-tickFree-append Hiding-tickFree
by blast+
next
  case 3 thus ?case
  proof(auto del:disjE, goal-cases)
    case (1 s t ta) show ?case
    proof(cases tickFree s)
      case True
      from 1(2) obtain ss tt where A:ta = ss@tt ∧ s = trace-hide ss (ev ‘ A) ∧
ss ∈ 𝒯 P
      using trace-hide-append[of s t A ta, OF 1(1)] by (metis F-T is-processT3-SR)
      with True have B:tickFree ss
        by (metis event.distinct(1) filter-set imageE member-filter tickFree-def)
      show ?thesis
      using 1(3) A B inf-hidden[of A ss P] by (metis append-Nil2 front-tickFree-Nil)
    next
      case False
      with 1(1,2) obtain ss tt where A:s = ss@[tick] ∧ ta = tt@[tick]
        by (metis append-Nil2 contra-subsetD filter-is-subset front-tickFree-mono
                    Hiding-fronttickFree is-processT nonTickFree-n-frontTickFree
tickFree-def)
      with 1(1,2) have ss = trace-hide tt (ev ‘ A)
        by (metis (no-types, lifting) butlast-append butlast-snoc contra-subsetD
            filter.simps(2) filter-append filter-is-subset front-tickFree-implies-tickFree

                    front-tickFree-single is-processT nonTickFree-n-frontTickFree
non-tickFree-tick
                self-append-conv2 tickFree-append tickFree-def)
      with False 1(1,2) A show ?thesis
       by (metis append-Nil2 front-tickFree-mono Hiding-fronttickFree is-processT)
    qed
  next
    case (2 s t ta u) show ?case
    proof(cases length s ≤ length (trace-hide ta (ev ‘ A)))
      case True
      with append-eq-append-conv2[THEN iffD1, OF 2(3)]
        obtain tt where s@tt = trace-hide ta (ev ‘ A) by auto
      with 2(4) obtain ss ttt where A:ta = ss@ttt ∧ s = trace-hide ss (ev ‘ A) ∧
ss ∈ 𝒯 P
       using trace-hide-append[of s tt A ta] by (metis D-T imageE is-processT3-ST)
```

```
      with 2(2) have B:tickFree ss using tickFree-append by blast
      show ?thesis
        using 2(4,5) A B inf-hidden[of A ss P]
        by (metis (no-types, lifting) append-Nil2 is-processT)
    next
      case False
      with 2(3) obtain uu uuu where A:s = trace-hide ta (ev ‘ A) @ uu ∧ u =
uu @ uuu
      by (auto simp add: append-eq-append-conv2, metis le-length-mono le-list-def)

      with 2(1,2,4) 2(5)[rule-format, of ta uu] show ?thesis
        using front-tickFree-dw-closed by blast
    qed
  qed
next
  case 4 thus ?case
   by (auto, metis (mono-tags) Un-subset-iff is-processT4 sup.cobounded2 sup.coboundedI1)
next
  case 5 thus ?case
  proof(intro impI allI, auto, rule not-not[THEN iffD1], rule notI, simp, goal-cases)
    case (1 X Y t)
    from 1(3) 1(4)[THEN spec, of t, simplified] obtain c where A1:c ∈ Y and
A2:c ∉ (ev‘A)
                                                  and A3:t@[c] ∈ 𝒯 P
      using is-processT5-S7'[of t X ∪ (ev‘A) P Y] by (metis UnCI sup-commute
sup-left-commute)
    hence trace-hide t (ev ‘ A) @ [c] = trace-hide (t@[c]) (ev ‘ A) by simp
    thus ?case using 1(1)[rule-format, OF A1] inf-hidden[of A t@[c], rotated, OF
A3]
      by (metis (no-types, lifting) append.right-neutral append-T-imp-tickFree
                                    front-tickFree-Nil is-processT5-S7 not-Cons-self2
rangeE)
  qed
next
  case 6 thus ?case
  proof (auto, goal-cases)
    case (1 s X t)
    hence front-tickFree t by (simp add:is-processT2)
    with 1(1) obtain t' where A:t = t'@[tick]
      by (metis filter-is-subset nonTickFree-n-frontTickFree non-tickFree-tick
          subset-iff tickFree-append tickFree-def)
    with 1(1,2) have B:s = trace-hide t' (ev ‘ A)
      by (auto simp add:tickFree-def split:if-splits)
    with A 1(1,2) is-processT6[of P, THEN spec, THEN spec, of t' X ∪ ev ‘ A]
        is-processT4-empty[of t ev ‘ A P] show ?case
    by (auto simp add: Un-Diff split:if-splits)
  next
    case (2 s X t u)
    then obtain u' where A:u = u'@[tick]
```

```
    by (metis filter-is-subset nonTickFree-n-frontTickFree non-tickFree-tick
        subset-iff tickFree-append tickFree-def)
    with 2(3) have B:s = trace-hide t (ev ‘ A) @ u′
      by (auto simp add:tickFree-def split:if-splits)
    with 2(1,2,5) 2(4)[THEN spec, THEN spec, of t u′] show ?case
      using front-tickFree-dw-closed A by blast
  next
    case (3 s X u f x)
    then obtain u′ where A:u = u′@[tick]
     by (metis filter-is-subset nonTickFree-n-frontTickFree non-tickFree-tick
        subset-iff tickFree-append tickFree-def)
     with 3(3) have B:s = trace-hide (f x) (ev ‘ A) @ u′
      by (auto simp add:tickFree-def split:if-splits)
     with 3(1,2,3,5,6,7) 3(4)[THEN spec, THEN spec, of f x u′] show ?case
      using front-tickFree-dw-closed[of u′ [tick]] by auto
  qed
next
  case 7 thus ?case using front-tickFree-append by (auto, blast +)
next
  case 8 thus ?case by simp
next
  case 9 thus ?case
  proof (intro allI impI, simp, elim exE, goal-cases)
    case (1 s t u)
    then obtain u′ where u = u′@[tick]
     by (metis Hiding-tickFree nonTickFree-n-frontTickFree non-tickFree-tick tick-
Free-append)
    with 1 show ?case
      apply(rule-tac x=t in exI, rule-tac x=u′ in exI)
      using front-tickFree-dw-closed by auto
  qed
qed

lemma Rep-Abs-Hiding: Rep-process (Abs-process
                        ({(s,X). ∃ t. s = trace-hide t (ev‘A) ∧ (t,X ∪ (ev‘A))
∈ ℱ P} ∪
                        {(s,X). s ∈ div-hide P A}, div-hide P A))
                      = ({(s,X). ∃ t. s = trace-hide t (ev‘A) ∧ (t,X ∪ (ev‘A))
∈ ℱ P} ∪
                        {(s,X). s ∈ div-hide P A}, div-hide P A)
  by (simp only:CollectI Rep-process Abs-process-inverse Hiding-maintains-is-process)
```

4.1.4 Projections

```
lemma F-Hiding: ‹ℱ (P \ A) = {(s,X). ∃ t. s = trace-hide t (ev‘A) ∧ (t,X ∪
(ev‘A)) ∈ ℱ P} ∪
                        {(s,X). s ∈ div-hide P A}›
  by (subst Failures-def, simp only: Hiding-def Rep-Abs-Hiding FAILURES-def
fst-conv)
```

lemma *D-Hiding*: ‹$\mathcal{D}$ (P \ A) = *div-hide P A*›
by (*subst D-def*, *simp only*: *Hiding-def Rep-Abs-Hiding DIVERGENCES-def snd-conv*)

lemma *T-Hiding*: ‹$\mathcal{T}$ (P \ A) = {s. ∃ t. s = *trace-hide t* (*ev*‘A) ∧ (t, *ev*‘A) ∈ $\mathcal{F}$ P} ∪ *div-hide P A*›
apply (*unfold Traces-def*, *simp only*:*Rep-Abs-Hiding Hiding-def TRACES-def FAILURES-def fst-conv*, *auto*)
apply (*metis is-processT sup.cobounded2*)
apply (*metis FAILURES-def Failures-def NF-NT range-eqI*)
apply (*metis sup.idem*)
by (*metis FAILURES-def F-T Failures-def range-eqI*)

4.1.5 Continuity Rule

lemma *mono-Hiding*[*simp*]: ‹(P::'*a process*) ⊑ Q ⟹ (P \ A) ⊑ (Q \ A)›
proof (*auto simp only*:*le-approx-def D-Hiding Ra-def F-Hiding T-Hiding*, *goal-cases*)
case (*1 t u*) **thus** *?case* **by** *blast*
next
case (*2 u f xa*) **thus** *?case*
apply(*rule-tac x=f xa* **in** *exI*, *rule-tac x=u* **in** *exI*)
by (*metis D-T Ra-def le-approx2T le-approx-def rangeI*)
next
case (*3 x t*)
hence *A*:*front-tickFree t* **by** (*meson is-processT2*)
show *?case*
proof(*cases tickFree t*)
case *True* **thus** *?thesis*
by (*metis 3*(*2*) *3*(*4*) *3*(*5*) *3*(*6*) *front-tickFree-charn mem-Collect-eq self-append-conv*)
next
case *False*
with *A* **obtain** t' **where** $t = t'$@[*tick*] **using** *nonTickFree-n-frontTickFree* **by** *blast*
with *3* **show** *?thesis*
by (*metis* (*no-types*, *lifting*) *filter.simps*(*1*) *filter.simps*(*2*) *front-tickFree-mono front-tickFree-Nil filter-append is-processT9 list.distinct*(*1*) *local.A mem-Collect-eq*)
qed
next
case (*4 x t*) **thus** *?case* **using** *NF-ND* **by** *blast*
next
case (*5 x t u*) **thus** *?case* **by** *blast*
next
case (*6 x u f xa*) **thus** *?case* **by** (*metis D-T Ra-def le-approx2T le-approx-def rangeI*)
next
case (*7 x*)
from *7*(*4*) **have** *A*:x ∈ *min-elems* (*div-hide P A*) **by** *simp*
from *elem-min-elems*[*OF 7*(*4*), *simplified*] **obtain** *t u*

where *B1*:x = *trace-hide* t (*ev* ‘ A) @ u **and** *B2*:*tickFree* t **and** *B3*:*front-tickFree* u **and**
B4:($t \in \mathcal{D}\ P \vee (\exists (f{::}\ nat \Rightarrow 'a\ event\ list).\ strict\text{-}mono\ f \wedge (\forall i.\ f\ i \in \mathcal{T}\ P) \wedge$
$(\forall i.\ trace\text{-}hide\ (f\ i)\ (ev\ ‘\ A) = trace\text{-}hide\ (f\ 0)\ (ev\ ‘\ A)) \wedge t \in range\ f))$
by *blast*
show *?case* **proof**(*cases* $t \in \mathcal{D}\ P$)
case *True*
then obtain $t'\ t''$ **where** *C1*:$t'@t''=t$ **and** *C2*:$t' \in$ *min-elems* ($\mathcal{D}\ P$) **by** (*metis min-elems-charn*)
hence *C3*:*trace-hide* t' (*ev* ‘ A) $\in$ *div-hide* P A
apply (*simp*, *rule-tac* $x=t'$ **in** *exI*, *rule-tac* $x=[]$ **in** *exI*, *simp*)
using *B2 elem-min-elems tickFree-append* **by** *blast*
from *C1 B1* **have** *D1*:*trace-hide* t' (*ev* ‘ A) @ *trace-hide* t'' (*ev* ‘ A) = *trace-hide* t (*ev* ‘ A)
by *fastforce*
from *B1 C1 D1 min-elems-no*[*OF A C3*] **have** *E1*:x=*trace-hide* t' (*ev* ‘ A)
by (*metis* (*no-types*, *lifting*) *append.assoc le-list-def*)
with *B1 B2 B3 C1 D1 7*(*5*)[*simplified*, *rule-format*, *of* t' []]
have *E2*:($\forall (f{::}\ nat \Rightarrow 'a\ event\ list).\ strict\text{-}mono\ f \longrightarrow (\exists i.\ f\ i \notin \mathcal{T}\ Q) \vee$
$(\exists i.\ trace\text{-}hide\ (f\ i)\ (ev\ ‘\ A) \neq trace\text{-}hide\ (f\ 0)\ (ev\ ‘\ A)) \vee t' \notin range\ f$)
apply *simp*
using *front-tickFree-append Hiding-tickFree tickFree-append* **by** *blast*
with *E1 7*(*3*) *C2 inf-hidden*[*of A* t' *Q*] **show** *?thesis*
by (*metis* (*no-types*, *lifting*) *contra-subsetD*)
next
case *False*
from *B1 B2 B3 B4* **have** *C*:*trace-hide* t (*ev* ‘ A) $\in$ *div-hide* P A
by (*simp*, *rule-tac* $x=t$ **in** *exI*, *rule-tac* $x=[]$ **in** *exI*, *simp*)
from *B1 min-elems-no*[*OF A C*] **have** *E1*:x=*trace-hide* t (*ev* ‘ A)
using *le-list-def* **by** *auto*
from *B4 False 7*(*2*)[*rule-format*, *of t*, *OF False*] **have** $t \in \mathcal{T}\ Q$ **using** *F-T T-F* **by** *blast*
with *E1 7*(*5*)[*simplified*, *rule-format*, *of t* [], *simplified*, *OF E1 B2*] *inf-hidden*[*of A t Q*] **show** *?thesis*
by *metis*
qed
qed

lemma *cont-Hiding1* :‹*chain* $Y \Longrightarrow$ *chain* (λ i. ($Y\ i \setminus A$))›
by (*simp add*: *po-class.chain-def*)

lemma *KoenigLemma*:
assumes $*$:*infinite* (*Tr*::$'a$ *list set*) **and** $**$:$\forall i.$ *finite*$\{t.\ \exists t' \in Tr.\ t = take\ i\ t'\}$
shows $\exists (f{::}nat \Rightarrow 'a\ list).\ strict\text{-}mono\ f \wedge range\ f \subseteq \{t.\ \exists t' \in Tr.\ t \leq t'\}$
proof −
define Tr' **where** $Tr' = \{t.\ \exists t' \in Tr.\ t \leq t'\}$
have *a*:*infinite* Tr'
by (*metis* (*mono-tags*, *lifting*) $*$ Tr'*-def infinite-super mem-Collect-eq order-refl*

subsetI)
{ fix *i*
have $\{t \in Tr'.\ length\ t = i\} \subseteq \{t.\ \exists\, t' \in Tr.\ t = take\ i\ t'\}$
by (*auto simp add*:*Tr'-def*, *metis append-eq-conv-conj le-list-def*)
with ∗∗ **have** *finite*($\{t \in Tr'.\ length\ t = i\}$) **using** *infinite-super* **by** *blast*
} **note** *b*=*this*
{ fix *t*
define *E* **where** $E = \{e\ |e.\ t@[e] \in Tr'\}$
have *aa*:*finite E*
proof −
have *inj-on* ($\lambda e.\ t\ @\ [e]$) *E* **by** (*simp add*: *inj-on-def*)

with *b*[*of Suc* (*length t*)] *inj-on-finite*[*of* $\lambda e.\ t@[e]$ *E* $\{t' \in Tr'.\ length\ t' = Suc\ (length\ t)\}$]
show *?thesis* **by** (*simp add*: *E-def image-Collect-subsetI*)
qed
hence *bb*:*finite* $\{t@[e]\ |e.\ e \in E\}$ **by** *simp*
have $\{t' \in Tr'.\ t < t'\} = \{t@[e]\ |e.\ e \in E\} \cup (\bigcup e \in E.\ \{t' \in Tr'.\ t@[e] < t'\})$
proof (*auto simp add*:*Let-def E-def Tr'-def*, *goal-cases*)
case (*1 x xa*)
then obtain *u e* **where** $x = t\ @\ [e]\ @\ u$
by (*metis A append-Cons append-Nil append-Nil2 le-list-def list.exhaust*)
with *1 1*(*4*)[*rule-format*, *of e*] **show** *?case* **by** (*metis append-assoc le-list-def less-list-def*)
next
case (*2 x xa xb xc*)
thus *?case* **by** (*meson less-self less-trans*)
qed
with *aa bb* **have** *infinite* $\{t' \in Tr'.\ t < t'\} \Longrightarrow \exists\, e.\ infinite\ \{t' \in Tr'.\ t@[e] < t'\}$ **by** *auto*
} **note** *c*=*this*
define *ff* **where** *d*:*ff* =*rec-nat* [] ($\lambda i\ t.$ (*let* $e = SOME\ e.\ infinite\ \{t' \in Tr'.\ t@[e] < t'\}$ *in* $t\ @\ [e]$))
hence *dd*:$\forall\, n.\ ff\ (Suc\ n) > ff\ n$ **by** *simp*
hence *e*:*strict-mono ff* **by** (*simp add*: *lift-Suc-mono-less strict-monoI*)
{ fix *n*
have $ff\ n \in Tr' \wedge infinite\ \{t' \in Tr'.\ ff\ n < t'\}$
proof(*induct n*)
case *0*
from *a Tr'-def* **have** $Tr' = \{t' \in Tr'.\ [] < t'\} \cup \{[]\}$ **by** (*auto simp add*: *le-neq-trans*)
with *a* **have** *infinite* $\{t' \in Tr'.\ [] < t'\}$
by (*metis* (*no-types*, *lifting*) *finite.emptyI finite.insertI finite-UnI*)
with *d Tr'-def* **show** *?case* **by** *auto*
next
case (*Suc n*)
from *d* **have** $ff\ (Suc\ n) = (let\ e = SOME\ e.\ infinite\ \{t' \in Tr'.\ ff\ n@[e] < t'\}\ in\ ff\ n\ @\ [e])$ **by** *simp*
with *c*[*rule-format*, *of ff n*] **obtain** *e* **where**

a1:*ff* (*Suc n*) = (*ff n*) @ [*e*] ∧ *infinite* {*t*′ ∈ *Tr*′. *ff n* @ [*e*] < *t*′}
by (*metis* (*no-types*, *lifting*) *Suc.hyps someI-ex*)
then obtain *i* **where** *i* ∈ *Tr*′ ∧ *ff* (*Suc n*) < *i* **using** *not-finite-existsD* **by** *auto*
with *Tr*′*-def* **have** *ff* (*Suc n*) ∈ *Tr*′ **using** *dual-order.trans less-imp-le* **by** *fastforce*
with *a1* **show** *?case* **by** *simp*
qed
} **note** *g*=*this*
hence *h*:*range ff* ⊆ *Tr*′ **by** *auto*
show *?thesis* **using** *Tr*′*-def e h* **by** *blast*
qed

lemma *div-Hiding-lub* :
‹*finite* (*A*::′*a set*) ⟹ *chain Y* ⟹ 𝒟 (⨆ *i*. (*Y i* \ *A*)) ⊆ 𝒟 ((⨆ *i*. *Y i*) \ *A*)›
proof (*auto simp add*:*limproc-is-thelub cont-Hiding1 D-Hiding T-Hiding D-LUB T-LUB*, *goal-cases*)
case (*1 x*)
{ **fix** *xa t u f*
assume *a*:*front-tickFree u* ∧ *tickFree t* ∧ *x* = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧
isInfHiddenRun f (*Y xa*) *A* ∧ (∀ *i*. *f i* ∉ 𝒟 (*Y xa*)) ∧ *t* ∈ *range f*
hence (∀ *i n*. *f i* ∈ 𝒯 (*Y n*)) **using** *1*(*2*) *NT-ND chain-lemma le-approx2T* **by** *blast*
with *a* **have** *?case* **by** *blast*
} **note** *aa*=*this*
{ **fix** *xa t u f j*
assume *a*:*front-tickFree u* ∧ *tickFree t* ∧ *x* = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧
isInfHiddenRun f (*Y xa*) *A* ∧ (*f j* ∈ 𝒟 (*Y xa*)) ∧ *t* ∈ *range f*
hence ∃ *t u*. *front-tickFree u* ∧ *tickFree t* ∧ *x* = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧ *t* ∈ 𝒟 (*Y xa*)
apply(*rule-tac x*=*f j* **in** *exI*, *rule-tac x*=*u* **in** *exI*)
using *Hiding-tickFree*[*of f j A*] *Hiding-tickFree*[*of t A*] **by** (*metis imageE*)
} **note** *bb*=*this*
have *cc*: ∀ *xa*. ∃ *t u*. *front-tickFree u* ∧ *tickFree t* ∧ *x* = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧ *t* ∈ 𝒟(*Y xa*)
⟹ *?case* (**is** ∀ *xa*. ∃ *t*. *?S t xa* ⟹ *?case*)
proof −
assume *dd*:∀ *xa*. ∃ *t u*. *front-tickFree u* ∧ *tickFree t* ∧ *x* = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧ *t* ∈ 𝒟(*Y xa*)
(**is** ∀ *xa*. ∃ *t*. *?S t xa*)
define *f* **where** *f* = (λ*n*. *SOME t*. *?S t n*)
thus *?case*
proof (*cases finite*(*range f*))
case *True*
obtain *t* **where** *gg*:*infinite* (*f* −‘ {*t*}) **using** *f-def True inf-img-fin-dom* **by** *blast*
then obtain *k* **where** *f k* = *t* **using** *finite-nat-set-iff-bounded-le* **by** *blast*
then obtain *u* **where** *uu*:*front-tickFree u* ∧ *x* = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧ *tickFree t*

```
        using f-def dd[rule-format, of k] some-eq-ex[of λt. ?S t k] by blast
      { fix m
        from gg obtain n where gg:n ≥ m ∧ n ∈ (f −‘ {t})
          by (meson finite-nat-set-iff-bounded-le nat-le-linear)
       hence t ∈ 𝒟 (Y n) using f-def dd[rule-format, of n] some-eq-ex[of λt. ?S
t n] by auto
        with gg 1(2) have t ∈ 𝒟 (Y m) by (meson contra-subsetD le-approx-def
po-class.chain-mono)
      }
      with gg uu show ?thesis by blast
    next
      case False
      { fix t
        assume t ∈ range f
        then obtain k where f k = t using finite-nat-set-iff-bounded-le by blast
         then obtain u where uu:front-tickFree u ∧ x = trace-hide t (ev ‘ A) @
u ∧ tickFree t
          using f-def dd[rule-format, of k] some-eq-ex[of λt. ?S t k] by blast
        hence set t ⊆ set x ∪ (ev ‘ A) by auto
      } note ee=this
      { fix i
        have finite {(take i t)|t. t ∈ range f}
        proof(induct i, simp)
          case (Suc i)
          have ff:{take (Suc i) t|t. t ∈ range f} ⊆ {(take i t)|t. t ∈ range f} ∪
                  (⋃ e∈(set x ∪ (ev ‘ A)). {(take i t)@[e]|t. t ∈ range f}) (is ?AA
⊆ ?BB)
            proof
              fix t
              assume t ∈ ?AA
              then obtain t′ where hh:t′ ∈ range f ∧ t = take (Suc i) t′
                using finite-nat-set-iff-bounded-le by blast
              with ee[of t′] show t ∈ ?BB
                proof(cases length t′ > i)
                  case True
                       hence ii:take (Suc i) t′ = take i t′ @ [t′!i] by (simp add:
take-Suc-conv-app-nth)
                  with ee[of t′] have t′!i ∈ set x ∪ (ev ‘ A)
                    by (meson True contra-subsetD hh nth-mem)
                  with ii hh show ?thesis by blast
                next
                  case False
                  hence take (Suc i) t′ = take i t′ by fastforce
                  with hh show ?thesis by auto
                qed
            qed
          { fix e
           have {x @ [e] |x. ∃ t. x = take i t ∧ t ∈ range f} = {take i t′ @ [e] |t′.
t′ ∈ range f}
```

```
            by auto
          hence finite({(take i t') @ [e] |t'. t'∈range f})
            using finite-image-set[of - λt. t@[e], OF Suc] by auto
        } note gg=this
        have finite(set x ∪ (ev ‘ A)) using 1(1) by simp
      with ff gg Suc show ?case by (metis (no-types, lifting) finite-UN finite-Un
finite-subset)
      qed
    } note ff=this
    hence ∀ i. {take i t |t. t ∈ range f} = {t. ∃ t'. t = take i (f t')} by auto
    with KoenigLemma[of range f, OF False] ff
    obtain f' where gg:strict-mono (f'::nat ⇒ 'a event list) ∧
                                  range f' ⊆ {t. ∃ t'∈range f. t ≤ t'} by auto
    { fix n
      define M where M = {m. f' n ≤ f m }
      assume finite M
      hence l1:finite {length (f m)|m. m ∈ M} by simp
      obtain lm where l2:lm = Max {length (f m)|m. m ∈ M} by blast
      { fix k
        have length (f' k) ≥ k
              by(induct k, simp, metis (full-types) gg lessI less-length-mono
linorder-not-le
                                  not-less-eq-eq order-trans strict-mono-def)
      }
          with gg obtain m where r1:length (f' m) > lm by (meson lessI
less-le-trans)
      from gg obtain r where r2:f' (max m n) ≤ f r by blast
      with gg have r3: r ∈ M
            by (metis (no-types, lifting) M-def max.bounded-iff mem-Collect-eq
order-refl
                                  order-trans strict-mono-less-eq)
      with l1 l2 have f1:length (f r) ≤ lm using Max-ge by blast
      from r1 r2 have f2:length (f r) > lm
          by (meson dual-order.strict-trans1 gg le-length-mono max.bounded-iff
order-refl
                strict-mono-less-eq)
      from f1 f2 have False by simp
    } note ii=this
    { fix i n
      from ii obtain m where jj:m ≥ n ∧ f m ≥ f' i
        by (metis finite-nat-set-iff-bounded-le mem-Collect-eq nat-le-linear)
      have kk: (f m) ∈ 𝒟 (Y m) using f-def dd[rule-format, of m] some-eq-ex[of
λt. ?S t m] by auto
      with jj gg have (f' i) ∈ 𝒯 (Y m) by (meson D-T is-processT3-ST-pref)
     with jj 1(2) have (f' i) ∈ 𝒯 (Y n) using D-T le-approx2T po-class.chain-mono
by blast
    } note jj=this
    from gg have kk:mono (λn. trace-hide (f' n) (ev ‘ A))
      unfolding strict-mono-def mono-def
```

by (*metis* (*no-types*, *lifting*) *filter-append gg le-list-def mono-def strict-mono-mono*)
{ fix *n*
from *gg* **obtain** *k r* **where** *f k* = *f′ n* @ *r* **by** (*metis ii le-list-def not-finite-existsD*)
hence *trace-hide* (*f′ n*) (*ev* ‘ *A*) ≤ *x*
using *f-def dd*[*rule-format*, *of k*] *some-eq-ex*[*of* λ*t*. *?S t k*] *le-list-def* **by** *auto blast*
} note *ll*=*this*
{ assume *llll*:∀ *m*. ∃ *n*. *trace-hide* (*f′ n*) (*ev* ‘ *A*) > *trace-hide* (*f′ m*) (*ev* ‘ *A*)
hence *lll*:∀ *m*. ∃ *n*. *length* (*trace-hide* (*f′ n*) (*ev* ‘ *A*)) > *length* (*trace-hide* (*f′ m*) (*ev* ‘ *A*))
using *less-length-mono* **by** *blast*
define *ff* **where** *lll′*:*ff* = *rec-nat* (*length* (*trace-hide* (*f′ 0*) (*ev* ‘ *A*)))
(λ*i t*. (*let n* = *SOME n*. (*length* (*trace-hide* (*f′ n*) (*ev* ‘ *A*))) > *t*
in length (*trace-hide* (*f′ n*) (*ev* ‘ *A*))))
{ fix *n*
from *lll′ lll*[*rule-format*, *of n*] **have** *ff* (*Suc n*) > *ff n*
apply *simp* **apply** (*cases n*)
apply (*metis* (*no-types*, *lifting*) *old.nat.simps*(*6*) *someI-ex*)
by (*metis* (*no-types*, *lifting*) *llll less-length-mono old.nat.simps*(*7*) *someI-ex*)
} note *lll″*=*this*
with *lll″* **have** *strict-mono ff* **by** (*simp add*: *lll″ lift-Suc-mono-less strict-monoI*)
hence *lll‴*:*infinite*(*range ff*) **using** *finite-imageD strict-mono-imp-inj-on* **by** *auto*
from *lll lll′* **have** *range ff* ⊆ *range* (λ*n*. *length* (*trace-hide* (*f′ n*) (*ev* ‘ *A*)))

by (*auto*, *metis* (*mono-tags*, *lifting*) *old.nat.exhaust old.nat.simps*(*6*) *old.nat.simps*(*7*) *range-eqI*)
with *lll‴* **have** *infinite* (*range* (λ*n*. *length* (*trace-hide* (*f′ n*) (*ev* ‘ *A*))))
using *finite-subset* **by** *auto*
hence ∃ *m*. *length* (*trace-hide* (*f′ m*) (*ev* ‘ *A*)) > *length x*
using *finite-nat-set-iff-bounded-le* **by** (*metis* (*no-types*, *lifting*) *not-less rangeE*)
with *ll* **have** *False* **using** *le-length-mono not-less* **by** *blast*
}
then obtain *m* **where** *mm*:∀ *n*. *trace-hide* (*f′ n*) (*ev* ‘ *A*) ≤ *trace-hide* (*f′ m*) (*ev* ‘ *A*)
by (*metis* (*mono-tags*, *lifting*) *A kk le-cases mono-def*)
with *gg* **obtain** *k* **where** *nn*:*f k* ≥ *f′ m* **by** *blast*
then obtain *u* **where** *oo*:*front-tickFree u* ∧ *x* = *trace-hide* (*f′ m*) (*ev* ‘ *A*) @ *u* ∧ *tickFree* (*f′ m*)
using *f-def dd*[*rule-format*, *of k*] *some-eq-ex*[*of* λ*t*. *?S t k*]
by (*auto*, *metis* (*no-types*, *lifting*) *contra-subsetD filter-is-subset front-tickFree-append*

front-tickFree-mono le-list-def ll tickFree-Nil tickFree-append
tickFree-def tickFree-implies-front-tickFree)

```
        show ?thesis
          apply(rule-tac x=f' m in exI, rule-tac x=u in exI)
          apply(simp add:oo, rule disjI2, rule-tac x=λn. f' (n + m) in exI)
        using gg jj kk mm apply (auto simp add: strict-mono-def dual-order.antisym
mono-def)
          by (metis plus-nat.add-0 rangeI)
      qed
    qed
  show ?case
  proof (cases ∃ xa t u f. front-tickFree u ∧ tickFree t ∧ (∀ i. f i ∉ 𝒟 (Y xa)) ∧
t ∈ range f ∧
                          x = trace-hide t (ev ‘ A) @ u ∧ isInfHiddenRun f (Y xa) A)
    case True
    then show ?thesis using aa by blast
  next
    case False
    have dd:∀ xa. ∃ t u. front-tickFree u ∧ tickFree t ∧ x = trace-hide t (ev ‘ A) @
u ∧
            (t ∈ 𝒟 (Y xa) ∨ (∃ f. isInfHiddenRun f (Y xa) A ∧ (∃ i. f i ∈ 𝒟 (Y xa))
∧ t ∈ range f))
            (is ∀ xa. ?dd xa)
      proof (rule-tac allI)
        fix xa
        from 1(3) obtain t u where
              front-tickFree u ∧ tickFree t ∧ x = trace-hide t (ev ‘ A) @ u ∧
              (t ∈ 𝒟 (Y xa) ∨ (∃ f. isInfHiddenRun f (Y xa) A ∧ t ∈ range f))
          by blast
        thus ?dd xa
         apply(rule-tac x=t in exI, rule-tac x=u in exI, intro conjI, simp-all, elim
conjE disjE, simp-all)
          using 1(1) False NT-ND chain-lemma le-approx2T by blast
      qed
    hence ee:∀ xa. ∃ t u. front-tickFree u ∧ tickFree t ∧ x = trace-hide t (ev ‘ A) @
u ∧ t ∈ 𝒟(Y xa)
      using bb by blast
    with cc show ?thesis by simp
  qed
qed

lemma cont-Hiding2 : ‹finite A ⟹ chain Y ⟹ ((⨆ i. Y i) \ A) = (⨆ i. (Y i
\ A))›
proof(auto simp add:limproc-is-thelub cont-Hiding1 Process-eq-spec
                    D-Hiding Ra-def F-Hiding T-Hiding F-LUB  D-LUB T-LUB,
goal-cases)
  case (1 b x t u) thus ?case by blast
next
  case (2 b x u f xa) thus ?case by blast
next
  case (3 s X)
```

```
  hence ‹s ∉ 𝒟 ((⨆ i. Y i) \ A)›
    by (simp add:limproc-is-thelub cont-Hiding1 F-LUB  D-LUB T-LUB D-Hiding)
  with 3(1,2) obtain n where a: ‹s ∉ 𝒟 (Y n \ A)›
    by (metis (no-types, lifting) D-LUB-2 div-Hiding-lub subsetCE limproc-is-thelub
cont-Hiding1)
  with 3(3) obtain t where b:s = trace-hide t (ev ‘ A) ∧ (t, X ∪ ev ‘ A) ∈ ℱ
(Y n)
    unfolding D-Hiding by blast
  hence c:front-tickFree t using is-processT2 by blast
  have d:t ∉ 𝒟(Y n)
    proof(cases tickFree t)
      case True
      with a b show ?thesis using front-tickFree-Nil
        by (simp add: D-Hiding)
    next
      case False
     with c obtain t′ where t = t′@[tick] using nonTickFree-n-frontTickFree by
blast
      with a b show ?thesis
         apply(simp add: D-Hiding, erule-tac x=t′ in allE, erule-tac x=[tick] in
allE, simp)
         by (metis event.distinct(1) filter.simps(1) front-tickFree-implies-tickFree
imageE is-processT)
    qed
    with b show ?case using 3(2) NF-ND chain-lemma proc-ord2a by blast
next
  case (4 xa t u) thus ?case by blast
next
  case (5 xa u f xb) thus ?case by blast
next
  case (6 s)
  hence ‹s ∈ 𝒟 (⨆ i. (Y i \ A))›
    by (simp add:limproc-is-thelub cont-Hiding1 6(1) D-Hiding D-LUB)
  with div-Hiding-lub[OF 6(1,2)] have ‹s ∈ 𝒟 ((⨆ i. Y i) \ A)› by blast
  thus ?case by (simp add:limproc-is-thelub 6(2) D-Hiding D-LUB T-LUB)
qed

lemma cont-Hiding-base[simp] : ‹finite A ⟹ cont (λx. x \ A)›
  by (simp add: cont-def cont-Hiding1 cont-Hiding2 cpo-lubI)

lemma Hiding-cont[simp]: ‹finite A ⟹ cont f ⟹ cont (λx. f x \ A)›
  by (rule-tac f=f in cont-compose, simp-all)

end
```

4.2 The Synchronizing Operator

```
theory Sync
  imports Process HOL−Library.Infinite-Set
```

begin

4.2.1 Basic Concepts

fun *setinterleaving*:: $'a$ *trace* $\times$ ($'a$ *event*) *set* $\times$ $'a$ *trace* $\Rightarrow$ ($'a$ *trace*)*set*
where

si-empty1: *setinterleaving*([], X, []) = {[]}
| *si-empty2*: *setinterleaving*([], X, (y # t)) =
(*if* ($y \in X$)
then {}
else {z.$\exists$ u. z = (y # u) $\wedge$ $u \in$ *setinterleaving* ([], X, t)})
| *si-empty3*: *setinterleaving*((x # s), X, []) =
(*if* ($x \in X$)
then {}
else {z.$\exists$ u. z = (x # u) $\wedge$ $u \in$ *setinterleaving* (s, X, [])})

| *si-neq* : *setinterleaving*((x # s), X, (y # t)) =
(*if* ($x \in X$)
then if ($y \in X$)
then if ($x = y$)
then {z.$\exists$ u. z = (x#u) $\wedge$ $u \in$ *setinterleaving*(s, X, t)}
else {}
else {z.$\exists$ u. z = (y#u) $\wedge$ $u \in$ *setinterleaving* ((x#s), X, t)}
else if ($y \notin X$)
then {z.$\exists$ u. z = (x # u) $\wedge$ $u \in$ *setinterleaving* (s, X, (y # t))}
$\cup$ {z.$\exists$ u. z = (y # u) $\wedge$ $u \in$ *setinterleaving*((x # s), X, t)}
else {z.$\exists$ u. z = (x # u) $\wedge$ $u \in$ *setinterleaving* (s, X, (y # t))})

fun *setinterleavingList*::$'a$ *trace* $\times$ ($'a$ *event*) *set* $\times$ $'a$ *trace* $\Rightarrow$ ($'a$ *trace*)*list*
where

si-empty1l: *setinterleavingList*([], X, []) = [[]]
| *si-empty2l*: *setinterleavingList*([], X, (y # t)) =
(*if* ($y \in X$)
then []
else map (λz. y#z) (*setinterleavingList* ([], X, t)))
| *si-empty3l*: *setinterleavingList*((x # s), X, []) =
(*if* ($x \in X$)
then []
else map (λz. x#z) (*setinterleavingList* (s, X, [])))

| *si-neql* : *setinterleavingList*((x # s), X, (y # t)) =
(*if* ($x \in X$)
then if ($y \in X$)
then if ($x = y$)
then map (λz. x#z) (*setinterleavingList*(s, X, t))

else []
else map ($\lambda z.\ y\#z$) (*setinterleavingList* ((*x*#*s*), *X*, *t*))
else if ($y \notin X$)
then map ($\lambda z.\ x\#z$) (*setinterleavingList* (*s*, *X*, (*y* # *t*)))
@ *map* ($\lambda z.\ y\#z$) (*setinterleavingList* ((*x*#*s*), *X*, *t*))
else map ($\lambda z.\ x\#z$) (*setinterleavingList* (*s*, *X*, (*y* # *t*))))

lemma *finiteSetinterleavingList*: *finite* (*set* (*setinterleavingList*(*s*, *X*, *t*)))
by *auto*

lemma *sym* : *setinterleaving*(*s*, *X*, *t*)= *setinterleaving*(*t*, *X*, *s*)
by (*rule setinterleaving.induct*[*of* λ(*s*,*X*,*t*). *setinterleaving* (*s*, *X*, *t*)
= *setinterleaving* (*t*, *X*, *s*) (*s*, *X*, *t*), *simplified*], *auto*)

abbreviation *setinterleaves-syntax* (- *setinterleaves* '(()'(-, -')(), -') [*60*,*0*,*0*,*0*]*70*)
where *u setinterleaves* ((*s*, *t*), *X*) == (*u* $\in$ *setinterleaving*(*s*, *X*, *t*))

4.2.2 Definition

definition *Sync* :: ['*a process*,'*a set*,'*a process*] => '*a process* ((*3*(- ⟦-⟧/ -)) [*64*,*0*,*65*] *64*)
where *P* ⟦ *A* ⟧ *Q* ≡ *Abs-process*({(*s*,*R*).∃ *t u X Y*. (*t*,*X*) $\in \mathcal{F}$ *P* ∧ (*u*,*Y*) $\in \mathcal{F}$ *Q* ∧
(*s setinterleaves* ((*t*,*u*),(*ev*'*A*) ∪ {*tick*})) ∧
R = (*X* ∪ *Y*) ∩ ((*ev*'*A*) ∪ {*tick*}) ∪ *X* ∩ *Y*} ∪
{(*s*,*R*).∃ *t u r v*. *front-tickFree v* ∧ (*tickFree r* ∨ *v*=[]) ∧
s = *r*@*v* ∧
(*r setinterleaves* ((*t*,*u*),(*ev*'*A*) ∪ {*tick*})) ∧
(*t* $\in \mathcal{D}$ *P* ∧ *u* $\in \mathcal{T}$ *Q* ∨ *t* $\in \mathcal{D}$ *Q* ∧ *u* $\in \mathcal{T}$ *P*)},

{*s*. ∃ *t u r v*. *front-tickFree v* ∧ (*tickFree r* ∨ *v*=[]) ∧
s = *r*@*v* ∧
(*r setinterleaves* ((*t*,*u*),(*ev*'*A*) ∪ {*tick*})) ∧
(*t* $\in \mathcal{D}$ *P* ∧ *u* $\in \mathcal{T}$ *Q* ∨ *t* $\in \mathcal{D}$ *Q* ∧ *u* $\in \mathcal{T}$ *P*)})

4.2.3 Consequences

lemma *emptyLeftProperty*:∀ *s*. *s setinterleaves* (([], *t*), *A*)⟶*s*=*t*
apply(*induct-tac t*)
apply *simp*
by *auto*

lemma *emptyLeftSelf*: (∀ *t1*. *t1*∈ *set t*⟶*t1*∉*A*)⟶*t setinterleaves* (([], *t*), *A*)
apply(*induct-tac t*)
apply *simp*

```
  by auto

lemma emptyLeftNonSync: ∀ s. s setinterleaves ((([], t), A)⟶(∀ t1. t1∈ set t⟶t1∉A)
  apply(induct-tac t)
   apply simp
proof−
  fix a list
  assume a: ∀ s. s setinterleaves ((([], list), A) ⟶ (∀ t1. t1 ∈ set list ⟶ t1 ∉ A)
  thus ∀ s. s setinterleaves ((([], a # list), A) ⟶ (∀ t1. t1 ∈ set (a # list) ⟶ t1
∉ A)
  proof−
    have th: s setinterleaves ((([], a # list), A)⟶a∉A by auto
     obtain s1 where th1: s setinterleaves ((([], a # list), A)⟶s1 setinterleaves
((([], list), A)
      using mem-Collect-eq th by fastforce
     from a th1 have th2: ∀ s. s setinterleaves ((([], a # list), A)⟶(∀ t. t∈ set
list⟶t ∉ A)
      by auto
    with a show ?thesis
      by (metis (no-types, lifting) empty-iff set-ConsD setinterleaving.simps(2))
  qed
qed


lemma ftf-Sync1: a ∉ set(u) ∧ a ∉ set(t) ∧ s setinterleaves ((t, u), A) ⟶ a ∉
set(s)
proof (induction length t + length u arbitrary:s t u rule:nat-less-induct)
  case 1
  show ?case
    apply(cases t) using sym emptyLeftProperty apply blast
    apply(cases u) using sym emptyLeftProperty apply blast
    apply(simp split:if-splits, intro conjI impI, elim conjE disjE exE)
       apply (metis 1.hyps add-less-le-mono length-Cons lessI order-refl set-ConsD)
        apply (metis 1.hyps add-Suc-right length-Cons lessI set-ConsD)
       apply (metis 1.hyps add-less-mono length-Cons lessI set-ConsD)
      apply (metis 1.hyps add-Suc-right length-Cons lessI set-ConsD)
       apply (metis (no-types, opaque-lifting) 1.hyps add.commute add-Suc-right
insert-iff
                                    length-Cons lessI list.simps(15))
    by (metis 1.hyps add-less-le-mono length-Cons lessI order-refl set-ConsD)
qed

lemma addNonSyncEmpty: sa setinterleaves ((([], u), A) ∧ y1 ∉ A ⟶
    (sa @ [y1]) setinterleaves ((([y1], u), A) ∧ (sa @ [y1]) setinterleaves ((([], u @
[y1]), A)
  proof (induction length u arbitrary:sa u rule:nat-less-induct)
    case 1
    then show ?case
      apply(cases u)
```

```
      apply simp
    proof−
      fix a list
      assume a: ∀ m<length u. ∀ x. m = length x ⟶(∀ xa. xa setinterleaves (([], x), A) ∧ y1 ∉ A ⟶
      (xa @ [y1]) setinterleaves (([y1], x), A) ∧ (xa @ [y1]) setinterleaves (([], x @ [y1]), A))
      and b: u = a # list
      from b have th: sa setinterleaves (([], u), A)⟹(tl sa) setinterleaves (([], list), A)
      by (metis emptyLeftNonSync emptyLeftProperty emptyLeftSelf list.distinct(1) list.sel(3)
        list.set-sel(2))
      from a b th have th1: sa setinterleaves (([], u), A) ∧ y1 ∉ A⟶ ((tl sa) @ [y1])
      setinterleaves (([y1], list), A)∧((tl sa)@[y1]) setinterleaves (([], list@[y1]), A)
by auto
      thus sa setinterleaves (([], u), A) ∧ y1 ∉ A ⟶
        (sa @ [y1]) setinterleaves (([y1], u), A) ∧ (sa @ [y1]) setinterleaves (([], u @ [y1]), A)
        using b th by auto
    qed
  qed

lemma addNonSync: sa setinterleaves ((t,u),A)∧ y1 ∉ A⟶
(sa@[y1]) setinterleaves ((t@[y1],u),A)∧(sa@[y1]) setinterleaves ((t,u@[y1]),A)
proof (induction length t + length u arbitrary:sa t u rule:nat-less-induct)
  case 1
  then show ?case
    apply(cases t)
     apply (simp add: addNonSyncEmpty)
    apply (cases u)
     apply (metis Sync.sym addNonSyncEmpty append-self-conv2)
  proof−
    fix a list aa lista
    assume a: ∀ m<length t + length u. ∀ x xa. m = length x + length xa ⟶
    (∀ xb. xb setinterleaves ((x, xa), A) ∧ y1 ∉ A ⟶(xb @ [y1]) setinterleaves ((x @ [y1], xa), A)
    ∧ (xb @ [y1]) setinterleaves ((x, xa @ [y1]), A))
    and b:  t = a # list
    and c: u = aa # lista
    thus  sa setinterleaves ((t, u), A) ∧ y1 ∉ A ⟶
     (sa @ [y1]) setinterleaves ((t @ [y1], u), A) ∧ (sa @ [y1]) setinterleaves ((t, u @ [y1]), A)
     proof−
       from b c have th0pre: a=aa∧aa∈ A⟹sa setinterleaves ((t, u), A) ⟹
                        (tl sa) setinterleaves ((list,lista), A) by auto
       from th0pre a b c have th0pre1: a=aa∧aa∈ A⟹sa setinterleaves ((t, u), A)∧ y1 ∉ A⟹
```

((*tl sa*) @ [*y1*]) *setinterleaves* ((*list*@ [*y1*], *lista*), *A*)∧
((*tl sa*) @ [*y1*]) *setinterleaves* ((*list*, *lista*@ [*y1*]), *A*)
by (*metis add-less-mono length-Cons lessI*)
from *th0pre th0pre1 b c* **have** *th0*: *a*=*aa*∧*aa*∈ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*) ∧ *y1* ∉ *A* ⟶
(*sa* @ [*y1*]) *setinterleaves* ((*t* @ [*y1*], *u*), *A*)∧(*sa* @ [*y1*]) *setinterleaves* ((*t*, *u* @ [*y1*]), *A*)
by *auto*
from *b c* **have** *th1pre*: *a*∉*A*∧*aa*∈ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*) ⟹
(*tl sa*) *setinterleaves* ((*list*,*aa*#*lista*), *A*) **by** *auto*
from *th1pre a b c* **have** *th1pre1*: *a*∉*A*∧*aa*∈ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*)∧ *y1* ∉ *A*⟹
((*tl sa*) @ [*y1*]) *setinterleaves* ((*list*@ [*y1*], *aa*#*lista*), *A*)∧
((*tl sa*) @ [*y1*]) *setinterleaves* ((*list*, *aa*#*lista*@ [*y1*]), *A*)
by (*metis add-Suc append-Cons length-Cons lessI*)
from *th1pre th1pre1 b c* **have** *th1*: *a*∉*A*∧*aa*∈ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*) ∧ *y1* ∉ *A* ⟶
(*sa* @ [*y1*]) *setinterleaves* ((*t* @ [*y1*], *u*), *A*)∧(*sa* @ [*y1*]) *setinterleaves* ((*t*, *u* @ [*y1*]), *A*)
by *auto*
from *b c* **have** *th2pre*: *aa*∉*A*∧*a*∈ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*) ⟹
(*tl sa*) *setinterleaves* ((*a*#*list*,*lista*), *A*) **by** *auto*
from *th2pre a b c* **have** *th2pre1*: *aa*∉*A*∧*a*∈ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*)∧ *y1* ∉ *A*⟹
((*tl sa*) @ [*y1*]) *setinterleaves* ((*a*#*list*@ [*y1*], *lista*), *A*)∧
((*tl sa*) @ [*y1*]) *setinterleaves* ((*a*#*list*, *lista*@ [*y1*]), *A*)
by (*metis add-Suc-right append-Cons length-Cons lessI*)
from *th2pre th2pre1 b c* **have** *th2*: *aa*∉*A*∧*a*∈ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*) ∧ *y1* ∉ *A* ⟶
(*sa* @ [*y1*]) *setinterleaves* ((*t* @ [*y1*], *u*), *A*)∧(*sa* @ [*y1*]) *setinterleaves* ((*t*, *u* @ [*y1*]), *A*)
by *auto*
from *b c* **have** *th3pre*: *aa*∉*A*∧*a*∉ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*)⟹
(*tl sa*) *setinterleaves* ((*a*#*list*,*lista*), *A*)∨(*tl sa*)*setinterleaves* ((*list*,*aa*#*lista*), *A*) **by** *auto*
from *th3pre a b c* **have** *th3pre1*: *aa*∉*A*∧*a*∉ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*)∧
(*tl sa*) *setinterleaves* ((*a*#*list*,*lista*), *A*)∧ *y1* ∉ *A* ⟹
((*tl sa*) @ [*y1*]) *setinterleaves* ((*a*#*list*@ [*y1*], *lista*), *A*)∧((*tl sa*) @ [*y1*]) *setinterleaves*
((*a*#*list*, *lista*@ [*y1*]), *A*)
by (*metis add-Suc-right append-Cons length-Cons lessI*)
from *th3pre a b c* **have** *th3pre2*: *aa*∉*A*∧*a*∉ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*)∧
(*tl sa*) *setinterleaves* ((*list*,*aa*#*lista*), *A*)∧ *y1* ∉ *A* ⟹((*tl sa*) @ [*y1*]) *setinterleaves*
((*list*@ [*y1*], *aa*#*lista*), *A*)∧((*tl sa*) @ [*y1*]) *setinterleaves* ((*list*, *aa*#*lista* @ [*y1*]), *A*)
by (*metis add-Suc append-Cons length-Cons lessI*)
from *th3pre th3pre1 th3pre2 a b c* **have** *th3*: *aa*∉*A*∧*a*∉ *A*⟹*sa setinterleaves* ((*t*, *u*), *A*) ∧

```
      y1 ∉ A ⟶(sa @ [y1]) setinterleaves ((t @ [y1], u), A) ∧
      (sa @ [y1]) setinterleaves ((t, u @ [y1]), A) by auto
      show ?thesis
        using b c th0 th1 th2 th3 by auto
    qed
  qed
qed

lemma addSyncEmpty: sa setinterleaves (([], u), A) ∧ y1 ∈ A ⟶
                  (sa @ [y1]) setinterleaves ((y1], u @ [y1]), A)
  proof (induction length u arbitrary:sa u rule:nat-less-induct)
    case 1
    then show ?case
      apply(cases u)
       apply simp
    proof−
      fix a list
       assume a: ∀ m<length u. ∀ x. m = length x ⟶(∀ xa. xa setinterleaves ((([],
x), A) ∧ y1 ∈ A
                ⟶ (xa @ [y1]) setinterleaves ((y1], x @ [y1]), A))
      and b: u = a # list
     from b have th: sa setinterleaves ((([], u), A)⟹(tl sa) setinterleaves ((([], list),
A)
      by (metis emptyLeftNonSync emptyLeftProperty emptyLeftSelf list.distinct(1)
list.sel(3)
          list.set-sel(2))
      from a th have th1: sa setinterleaves ((([], u), A) ∧ y1 ∈ A⟹
      ((tl sa) @ [y1]) setinterleaves ((y1], list @ [y1]), A) using b by auto
      thus sa setinterleaves ((([], u), A) ∧ y1 ∈ A ⟶
          (sa @ [y1]) setinterleaves ((y1], u @ [y1]), A) using b th by auto
    qed
  qed

lemma addSync: sa setinterleaves ((t,u),A)∧ y1 ∈ A⟶(sa@[y1]) setinterleaves
((t@[y1],u@[y1]),A)
proof (induction length t + length u arbitrary:sa t u rule:nat-less-induct)
  case 1
  then show ?case
    apply(cases t)
     apply (simp add: addSyncEmpty)
    apply(cases u)
     apply (metis Sync.sym addSyncEmpty append.left-neutral)
  proof−
    fix a list aa lista
    assume a: ∀ m<length t + length u. ∀ x xa. m = length x + length xa ⟶
                  (∀ xb. xb setinterleaves ((x, xa), A) ∧ y1 ∈ A ⟶ (xb @ [y1])
setinterleaves
              ((x @ [y1], xa @ [y1]), A))
    and b: t = a # list
```

and *c*: $u = aa \# lista$
thus *sa setinterleaves*$((t, u),A)\wedge y1 \in A \longrightarrow (sa @ [y1])$ *setinterleaves* $((t @ [y1], u @ [y1]), A)$
proof−
from *b c* **have** *th0pre*: $a{=}aa\wedge aa\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A) \Longrightarrow$
$(tl\ sa)$ *setinterleaves* $((list,lista), A)$ **by** *auto*
from *th0pre a b c* **have** *th0pre1*: $a{=}aa\wedge aa\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A)\wedge y1 \in A \Longrightarrow$
$((tl\ sa) @ [y1])$ *setinterleaves* $((list@ [y1], lista @ [y1]), A)$
by (*metis add-less-mono length-Cons lessI*)
from *th0pre th0pre1 b c* **have** *th0*: $a{=}aa\wedge a\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A) \wedge y1 \in A \longrightarrow$
$(sa @ [y1])$ *setinterleaves* $((t @ [y1], u @ [y1]), A)$ **by** *auto*
from *b c* **have** *th1pre*: $a\notin A\wedge aa\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A) \Longrightarrow$
$(tl\ sa)$ *setinterleaves* $((list,aa\#lista), A)$ **by** *auto*
from *th1pre a b c* **have** *th1pre1*: $a\notin A\wedge aa\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A)\wedge y1 \in A \Longrightarrow$
$((tl\ sa) @ [y1])$ *setinterleaves* $((list@ [y1], aa\#lista @ [y1]), A)$
by (*metis add-Suc append-Cons length-Cons lessI*)
from *th1pre th1pre1 b c* **have** *th1*: $a\notin A\wedge aa\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A) \wedge y1 \in A \longrightarrow$
$(sa @ [y1])$ *setinterleaves* $((t @ [y1], u @ [y1]), A)$ **by** *auto*
from *b c* **have** *th2pre*: $aa\notin A\wedge a\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A) \Longrightarrow$
$(tl\ sa)$ *setinterleaves* $((a\#list,lista), A)$ **by** *auto*
from *th2pre a b c* **have** *th2pre1*: $aa\notin A\wedge a\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A)\wedge y1 \in A \Longrightarrow$
$((tl\ sa) @ [y1])$ *setinterleaves* $((a\#list@ [y1], lista @ [y1]), A)$
by (*metis add-Suc-right append-Cons length-Cons lessI*)
from *th2pre th2pre1 b c* **have** *th2*: $aa\notin A\wedge a\in A \Longrightarrow sa$ *setinterleaves* $((t, u), A) \wedge y1 \in A \longrightarrow$
$(sa @ [y1])$ *setinterleaves* $((t @ [y1], u @ [y1]), A)$ **by** *auto*
from *b c* **have** *th3pre*: $aa\notin A\wedge a\notin A \Longrightarrow sa$ *setinterleaves* $((t, u), A) \Longrightarrow$
$(tl\ sa)$ *setinterleaves*$((a\#list,lista), A)\vee(tl\ sa)$*setinterleaves* $((list,aa\#lista), A)$ **by** *auto*
from *th3pre a b c* **have** *th3pre1*: $aa\notin A\wedge a\notin A \Longrightarrow sa$ *setinterleaves* $((t, u), A)\wedge$
$(tl\ sa)$ *setinterleaves* $((a\#list,lista), A)\wedge y1 \in A \Longrightarrow$
$((tl\ sa) @ [y1])$ *setinterleaves* $((a\#list@ [y1], lista @ [y1]), A)$
by (*metis add-Suc-right append-Cons length-Cons lessI*)
from *th3pre a b c* **have** *th3pre2*: $aa\notin A\wedge a\notin A \Longrightarrow sa$ *setinterleaves* $((t, u), A)\wedge$
$(tl\ sa)$ *setinterleaves* $((list,aa\#lista), A)\wedge y1 \in A \Longrightarrow$
$((tl\ sa) @ [y1])$ *setinterleaves* $((list@ [y1], aa\#lista @ [y1]), A)$
by (*metis add-Suc append-Cons length-Cons lessI*)
from *th3pre th3pre1 th3pre2 a b c* **have** *th3*: $aa\notin A\wedge a\notin A \Longrightarrow sa$ *setinterleaves* $((t, u), A) \wedge$
$y1 \in A \longrightarrow (sa @ [y1])$ *setinterleaves* $((t @ [y1], u @ [y1]), A)$ **by** *auto*
show *?thesis*
using *b c th0 th1 th2 th3* **by** *auto*
qed
qed

qed

lemma *doubleReverse*: *s1 setinterleaves* ((*t*, *u*), *A*)⟶(*rev s1*) *setinterleaves* ((*rev t*, *rev u*), *A*)
proof (*induction length t* + *length u arbitrary*:*s1 t u rule*:*nat-less-induct*)
  case *1*
  then show *?case*
    apply(*cases t*)
    using *emptyLeftNonSync*
     apply (*metis emptyLeftProperty emptyLeftSelf rev.simps*(*1*) *set-rev*)
    apply(*cases u*)
    using *sym*
    apply (*metis* (*no-types*, *lifting*) *emptyLeftNonSync emptyLeftProperty emptyLeft-Self*
     rev.simps(*1*) *set-rev*)
  proof−
    fix *a list aa lista*
    assume *a*: ∀ *m*<*length t* + *length u*. ∀ *x xa*. *m* = *length x* + *length xa* ⟶
          (∀ *xb*. *xb setinterleaves* ((*x*, *xa*), *A*) ⟶*rev xb setinterleaves* ((*rev x*, *rev xa*), *A*))
    and *b*: *t* = *a* # *list*
    and *c*: *u* = *aa* # *lista*
    thus *s1 setinterleaves* ((*t*, *u*), *A*) ⟶*rev s1 setinterleaves* ((*rev t*, *rev u*), *A*)
    proof−
      from *b c* **have** *th0pre*: *a*=*aa*∧*aa*∈ *A*⟹*s1 setinterleaves* ((*t*, *u*), *A*)⟹
      (*tl s1*) *setinterleaves* ((*list*,*lista*), *A*) **by** *auto*
       from *th0pre a b c* **have** *th0pre1*: *a*=*aa*∧*aa*∈ *A*⟹*s1 setinterleaves* ((*t*, *u*), *A*)⟹
      ((*rev* (*tl s1*)) *setinterleaves* ((*rev list*, *rev lista*), *A*))
        by (*metis add-less-mono length-Cons lessI*)
      from *th0pre th0pre1 b c* **have** *th0*: *a*=*aa*∧*aa*∈ *A*⟹*s1 setinterleaves* ((*t*, *u*), *A*) ⟶
      rev s1 setinterleaves ((*rev t*, *rev u*), *A*) **using** *addSync* **by** *fastforce*
      from *b c* **have** *th1pre*: *a*∉*A*∧*aa*∈ *A*⟹*s1 setinterleaves* ((*t*, *u*), *A*)⟹
      (*tl s1*) *setinterleaves* ((*list*,*aa*#*lista*), *A*) **by** *auto*
       from *th1pre a b c* **have** *th1pre1*: *a*∉*A*∧*aa*∈ *A*⟹*s1 setinterleaves* ((*t*, *u*), *A*)⟹
      ((*rev* (*tl s1*)) *setinterleaves* ((*rev list*, *rev* (*aa*#*lista*)), *A*))
        by (*metis add-less-mono1 length-Cons lessI*)
      from *th1pre th1pre1 b c* **have** *th1*: *a*∉*A*∧*aa*∈ *A*⟹*s1 setinterleaves* ((*t*, *u*), *A*) ⟶
      rev s1 setinterleaves ((*rev t*, *rev u*), *A*) **using** *addNonSync* **by** *fastforce*
      from *b c* **have** *th2pre*: *aa*∉*A*∧*a*∈ *A*⟹*s1 setinterleaves* ((*t*, *u*), *A*)⟹
      (*tl s1*) *setinterleaves* ((*a*#*list*,*lista*), *A*) **by** *auto*
       from *th2pre a b c* **have** *th2pre1*: *aa*∉*A*∧*a*∈ *A*⟹*s1 setinterleaves* ((*t*, *u*), *A*)⟹
      ((*rev* (*tl s1*)) *setinterleaves* ((*rev* (*a*#*list*), *rev lista*), *A*))
        by (*metis add-Suc-right length-Cons lessI*)

```
    from th2pre th2pre1 b c have th2: aa∉A∧a∈ A⟹s1 setinterleaves ((t, u),
A) ⟶
    rev s1 setinterleaves ((rev t, rev u), A)
      using addNonSync by fastforce
    from b c have th3pre: aa∉A∧a∉ A⟹s1 setinterleaves ((t, u), A)⟹
     (tl s1) setinterleaves ((a#list,lista),A)∨(tl s1) setinterleaves((list,aa#lista),
A) by auto
    from th3pre a b c have th3pre1: aa∉A∧a∉ A⟹s1 setinterleaves ((t, u), A)∧
    (tl s1) setinterleaves ((a#list,lista), A) ⟹
    ((rev (tl s1)) setinterleaves ((rev (a#list), rev (lista)), A))
      by (metis add-Suc-right length-Cons lessI)
    from th3pre a b c have th3pre2: aa∉A∧a∉ A⟹s1 setinterleaves ((t, u), A)∧
    (tl s1) setinterleaves ((list,aa#lista), A) ⟹
    ((rev (tl s1)) setinterleaves ((rev (list), rev (aa#lista)), A))
      by (metis add-less-mono1 length-Cons lessI)
    from th3pre1 have th3pre31: aa∉A∧a∉ A⟹s1 setinterleaves ((t, u), A)∧
    ((rev (tl s1)) setinterleaves ((rev (a#list), rev (lista)), A))⟶
    ((rev (tl s1))@[aa]) setinterleaves ((rev (a#list), (rev (lista))@[aa]), A)
      by (simp add: addNonSync)
    from th3pre2 have th3pre32: aa∉A∧a∉ A⟹s1 setinterleaves ((t, u), A)∧
    ((rev (tl s1)) setinterleaves ((rev (list), rev (aa#lista)), A))⟶
    ((rev (tl s1))@[a]) setinterleaves (((rev (list)@[a]), (rev (aa#lista))), A)
      by (simp add: addNonSync)
     from th3pre th3pre1 th3pre2 th3pre31 th3pre32 a b c have th3: aa∉A∧a∉
A⟹
     s1 setinterleaves ((t, u), A) ⟶rev s1 setinterleaves ((rev t, rev u), A) by
force
    show ?thesis
      using b c th0 th1 th2 th3 by auto
  qed
 qed
qed


lemma ftf-Sync21: (a∈set(u)∧a∉set(t)∨a∈set(t)∧a∉set(u))∧a∈A⟶setinterleaving(u,
A ,t)= {}
proof (induction length t + length u arbitrary: t u rule:nat-less-induct)
 case 1
 then show ?case
  apply(cases t)
  using Sync.sym emptyLeftNonSync apply fastforce
  apply(cases u)
   apply auto[1]
  apply(simp split:if-splits, intro conjI impI, elim conjE disjE exE)
          apply blast
         apply auto[1]
        apply blast
       apply auto[1]
      apply auto[1]
```

```
    using less-SucI apply blast
      apply auto[1]
     apply auto[1]
    using list.simps(15) apply auto[1]
    by auto
qed

lemma ftf-Sync32: u=u1@[tick]∧t=t1@[tick]∧s setinterleaves ((t, u), insert tick
(ev ‘ A))⟹
                ∃ s1. s1 setinterleaves ((t1, u1), insert tick (ev ‘ A))∧ (s=s1@[tick])
proof−
  assume h: u=u1@[tick]∧t=t1@[tick]∧s setinterleaves ((t, u), insert tick (ev ‘
A))
  thus ∃ s1. s1 setinterleaves ((t1, u1), insert tick (ev ‘ A))∧ (s=s1@[tick])
  proof−
    from h have a:rev u=tick#rev u1 by auto
    from h have b:rev t=tick#rev t1 by auto
    from h have ab: (rev s)  setinterleaves ((tick#rev t1, tick#rev u1), insert tick
(ev ‘ A))
      using doubleReverse by fastforce
    from h obtain ss where c: ss  setinterleaves  ((rev t1, rev u1), insert tick (ev
‘ A))∧
    ss=tl(rev s) using ab by auto
    from c have d: (rev ss)  setinterleaves  (( t1,  u1), insert tick (ev ‘ A))
      using doubleReverse by fastforce
    from d have e: rev s=tick#ss
      using ab append-Cons-eq-iff c by auto
    show ?thesis
      using d e by blast
  qed
qed


lemma SyncWithTick-imp-NTF:
  (s @ [tick]) setinterleaves ((t, u),insert tick (ev ‘ A))
   ⟹ t∈ 𝒯 P ⟹ u ∈ 𝒯 Q
   ⟹ ∃ t1 u1. t=t1@[tick] ∧ u=u1@[tick]∧ s setinterleaves ((t1, u1), insert tick
(ev ‘ A))
proof−
  assume h: (s @ [tick]) setinterleaves ((t, u), insert tick (ev ‘ A))
  and h1: t∈ 𝒯 P
  and h2: u∈ 𝒯 Q
  thus ∃ t1 u1. t=t1@[tick] ∧ u=u1@[tick]∧ s setinterleaves ((t1, u1), insert tick
(ev ‘ A))
  proof−
    from h have a: (tick#rev s) setinterleaves ((rev t, rev u), insert tick (ev ‘ A))
      using doubleReverse by fastforce
    from h obtain tt uu where b: t=tt@[tick]∧u=uu@[tick]
       by (metis T-nonTickFree-imp-decomp empty-iff ftf-Sync1 ftf-Sync21 h1 h2
```

insertI1
non-tickFree-tick tickFree-append tickFree-def)
from *h b* **have** *d*: *s setinterleaves* ((*tt*, *uu*), *insert tick* (*ev* ‘ *A*))
using *ftf-Sync32* **by** *blast*
show *?thesis*
using *b d* **by** *blast*
qed
qed

lemma *synPrefix1*:
ta = []
⟹ ∃ *t1 u1*. (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟶
t1 ≤ *ta* ∧ *u1* ≤ *u* ∧ *s setinterleaves* ((*t1*, *u1*), *A*)
proof−
assume *a*: *ta* = []
obtain *u1* **where** *th*: *u1*=*s* **by** *blast*
from *a* **have** *th2*: (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟶ (*s* @ *t*)= *u*
by (*simp add*: *a emptyLeftProperty*)
from *a* **have** *th3*: (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟶ (∀ *t1*. *t1*∈*set u*⟶*t1*∉*A*)

by (*simp add*: *emptyLeftNonSync*)
from *a th* **have** *th1*: (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟶ [] ≤ *ta* ∧ *u1* ≤ *u*
using *le-list-def th2* **by** *blast*
from *a th* **have** *thh1*: (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*)∧ (∀ *t1*. *t1*∈*set u1*⟶*t1*∉*A*)
⟶
s setinterleaves ((([], *u1*), *A*) **by** (*simp add*: *emptyLeftSelf*)
thus ∃ *t1 u1*. (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟶ *t1* ≤ *ta* ∧ *u1* ≤ *u* ∧
s setinterleaves ((*t1*, *u1*), *A*) **using** *th th1 th2 th3 thh1* **by** *fastforce*
qed

lemma *synPrefix*: ∃ *t1 u1*. (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*)⟶
t1≤*ta*∧*u1*≤*u*∧*s setinterleaves* ((*t1*, *u1*), *A*)
proof (*induction length ta* + *length u arbitrary*: *s t ta u rule*:*nat-less-induct*)
case *1*
then show *?case*
apply(*cases ta*)
using *synPrefix1* **apply** *fastforce*
apply(*cases u*)
using *sym synPrefix1* **apply** *metis*
proof−
fix *a list aa lista s t*
assume *a*: ∀ *m*<*length ta* + *length u*. ∀ *x xa*. *m* = *length x* + *length xa* ⟶
(∀ *xb xc*. ∃ *t1 u1*. (*xb* @ *xc*) *setinterleaves* ((*x*, *xa*), *A*) ⟶
t1 ≤ *x* ∧ *u1* ≤ *xa* ∧ *xb setinterleaves* ((*t1*, *u1*), *A*))
and *b*: *ta* = *a* # *list*
and *c*: *u* = *aa* # *lista*
thus ∃ *t1 u1*. (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟶ *t1* ≤ *ta* ∧ *u1* ≤ *u* ∧
s setinterleaves ((*t1*, *u1*), *A*)

proof−
from *a* **have** *th0*: ∀ *xb xc*. ∃ *t1 u1*. (*xb* @ *xc*) *setinterleaves* ((*list*, *lista*), *A*) ⟶
t1 ≤ *list* ∧ *u1* ≤ *lista* ∧ *xb setinterleaves* ((*t1*, *u1*), *A*)
by (*metis add-less-le-mono b c impossible-Cons le-cases not-le-imp-less*)
from *b c* **obtain** *yb* **where** *thp*: *a*=*aa*∧*a*∈ *A*∧(*s*@*t*) *setinterleaves* ((*ta*, *u*), *A*)∧*length s* >*1* ⟹
yb=*tl s* **by** *blast*
with *b c* **have** *thp4*: *a*=*aa*∧*a*∈ *A*∧(*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*)∧*length s* >*1* ⟹*s*=*a*#*yb*
by (*auto*, *metis Cons-eq-append-conv list.sel*(*3*) *list.size*(*3*) *not-less0*)
have *thp5*: *a*=*aa*∧*a*∈ *A*∧(*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*)∧*length s* >*1* ⟹
(*yb* @ *t*) *setinterleaves* ((*list*, *lista*), *A*) **using** *b c thp4* **by** *auto*
from *th0* **obtain** *yt yu* **where** *thp1*: *a* = *aa* ∧ *a* ∈ *A* ⟹ (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ∧
1 < *length s*⟹*yb setinterleaves* ((*yt*, *yu*), *A*)∧*yt*≤*list*∧*yu*≤*lista* **using** *thp5* **by** *blast*
from *thp thp1* **have** *thp2*: *a*=*aa*∧*a*∈ *A*⟹ (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*)∧ *length s* >*1*⟶
s setinterleaves ((*a*#*yt*, *aa*#*yu*), *A*) **using** *thp4* **by** *auto*
from *b c* **have** *thp3*: *a*=*aa*∧*a*∈ *A*⟹ (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ∧ *length s*=*1*⟶
s setinterleaves (([*a*], [*aa*]), *A*)
using *append-eq-append-conv2*[*of s t* [*aa*]] **by** (*auto*, *metis append-Nil2 append-eq-append-conv length-Cons list.size*(*3*))
have *thp6*: *a*=*aa*∧*a*∈ *A*⟹ (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ∧ *length s*=*0*⟶
s setinterleaves (([], []), *A*) **by** *auto*
from *thp1* **have** *thp7*: *a*=*aa*∧*a*∈ *A*⟹ (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*)∧ *length s* >*1*⟹
(*a* # *yt*)≤*ta* ∧(*aa* # *yu*)≤*u* **by** (*metis b c le-less less-cons*)
have *th*: *a*=*aa*∧*a*∈ *A*⟹∃ *t1 u1*. (*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟶
t1 ≤ *ta* ∧ *u1* ≤ *u* ∧ *s setinterleaves* ((*t1*, *u1*), *A*)
proof −
assume *dd*:*a*=*aa* ∧ *a*∈ *A*
consider (*aa*) *length s* = *0* | (*bb*) *length s* = *1* | (*cc*) *length s* > *1*
by *linarith*
then show *?thesis*
proof *cases*
case *aa*
with *thp6* **show** *?thesis*
by (*rule-tac x*=[] **in** *exI*, *rule-tac x*=[] **in** *exI*, *simp*)
next
case *bb*
with *dd thp3 b c* **show** *?thesis*
by (*rule-tac x*=[*a*] **in** *exI*, *rule-tac x*=[*a*] **in** *exI*, *auto simp add*: *le-list-def*)

next
case *cc*

with *dd thp2 thp7 b c* **show** *?thesis*
by (*rule-tac x=a#yt* **in** *exI*, *rule-tac x=a#yu* **in** *exI*, *auto simp add*: *le-list-def*)
qed
qed
from *b c* **have** *th1pre*: *a∉A∧aa∈ A∧(s @ t) setinterleaves ((ta, u), A) ⟶ tl (s @ t) setinterleaves ((list, u), A)∧hd (s@t)=a* **by** *auto*
from *a b c* **obtain** *yt1 yu1* **where** *th1pre1*: *a∉A∧aa∈ A∧(s @ t) setinterleaves ((ta, u), A)∧*
length s>0⟶yt1≤list∧ yu1 ≤ u∧ tl s setinterleaves ((yt1, yu1), A)
by (*metis* (*no-types*, *lifting*) *length-Cons length-greater-0-conv lessI plus-nat.simps*(*2*)

th1pre tl-append2)
from *b* **have** *th1pre2*: *yt1≤list⟶a#yt1≤ta*
by (*simp add*: *le-less less-cons*)
from *b c* **have** *th1pre3*: *a∉A∧aa∈ A∧tl s setinterleaves ((yt1, yu1), A)⟶ (a#(tl s)) setinterleaves ((a#yt1, yu1), A)*
by (*metis* (*mono-tags*, *lifting*) *addNonSync doubleReverse rev.simps*(*2*) *rev-rev-ident*)
from *b c th1pre1 th1pre2* **have** *th1pre4*: *a∉A∧aa∈ A∧(s @ t) setinterleaves ((ta, u), A)∧*
length s>0 ⟶a#yt1≤ta∧ yu1 ≤u∧ s setinterleaves ((a#yt1, yu1), A)
using *th1pre th1pre3* **by** *fastforce*
have *th1pre5*:*a∉A∧aa∈ A∧(s @ t) setinterleaves ((ta, u), A)∧length s=0 ⟶[]≤ta∧ [] ≤u∧*
s setinterleaves ((([], []), A) **by** *auto*
have *th1*: *a∉A∧aa∈ A⟹∃ t1 u1. (s @ t) setinterleaves ((ta, u), A) ⟶ t1 ≤ ta ∧ u1 ≤ u ∧ s setinterleaves ((t1, u1), A)* **using** *th1pre4 th1pre5* **by** *blast*
from *b c* **have** *th2pre*: *aa∉A∧a∈ A∧(s @ t) setinterleaves ((ta, u), A) ⟶ tl (s @ t) setinterleaves ((ta, lista), A)∧hd (s@t)=aa* **by** *auto*
from *a b c* **obtain** *yt2 yu2* **where** *th2pre1*: *aa∉A∧a∈ A∧(s @ t) setinterleaves ((ta, u), A)∧*
length s>0⟶yt2≤ta∧ yu2 ≤ lista∧ (tl s) setinterleaves ((yt2, yu2), A)
by (*metis* (*no-types*, *lifting*) *add-Suc-right length-Cons length-greater-0-conv lessI*

th2pre tl-append2)
from *c* **have** *th2pre2*: *yu2≤lista⟶aa#yu2≤u*
by (*simp add*: *le-less less-cons*)
from *b c* **have** *th2pre3*: *aa∉A∧a∈ A∧tl s setinterleaves ((yt2, yu2), A)⟶ (aa#(tl s)) setinterleaves ((yt2, aa#yu2), A)*
by (*metis* (*mono-tags*, *lifting*) *addNonSync doubleReverse rev.simps*(*2*) *rev-rev-ident*)
from *b c th2pre1 th2pre2* **have** *th2pre4*: *aa∉A∧a∈ A∧(s @ t) setinterleaves ((ta, u), A)∧*
length s>0 ⟶yt2≤ta∧ aa#yu2 ≤u∧ s setinterleaves ((yt2, aa#yu2), A)
using *th2pre th2pre3* **by** *fastforce*
have *th2pre5*:*aa∉A∧a∈ A∧(s @ t) setinterleaves ((ta, u), A)∧length s=0 ⟶ []≤ta∧ [] ≤u∧ s setinterleaves ((([], []), A)* **by** *auto*

have *th2*: $aa \notin A \wedge a \in A \Longrightarrow \exists\, t1\ u1.\ (s\ @\ t)$ *setinterleaves* $((ta,\ u),\ A) \longrightarrow$
$t1 \leq ta \wedge u1 \leq u \wedge s$ *setinterleaves* $((t1,\ u1),\ A)$ **using** *th2pre4 th2pre5* **by** *blast*

from *b c* **have** *th3pre*: $aa \notin A \wedge a \notin A \wedge (s\ @\ t)$ *setinterleaves* $((ta,\ u),\ A) \longrightarrow tl\ (s\ @\ t)$
setinterleaves $((ta,\ lista),\ A) \wedge hd\ (s@t) = aa \vee tl\ (s\ @\ t)$*setinterleaves*$((list,\ u),\ A) \wedge hd(s@t) = a$
by *auto*

from *a b c th3pre* **obtain** *yt31 yu31* **where** *th3pre1*: $aa \notin A \wedge a \notin A \wedge$
$(s\ @\ t)$ *setinterleaves* $((ta,\ u),\ A) \wedge tl\ (s\ @\ t)$ *setinterleaves* $((ta,\ lista),\ A) \wedge$
$hd\ (s@t) = aa \wedge length\ s > 0 \longrightarrow yt31 \leq ta \wedge\ yu31\ \leq\ lista \wedge\ (tl\ s)$ *setinterleaves* $((yt31,\ yu31),\ A)$
by(*metis*(*no-types*,*lifting*)*add-Suc-right length-Cons length-greater-0-conv lessI tl-append2*)

from *a b c th3pre* **obtain** *yt32 yu32* **where** *th3pre2*: $aa \notin A \wedge a \notin A \wedge (s@t)$*setinterleaves* $((ta,\ u),A) \wedge$
$tl\ (s\ @\ t)$ *setinterleaves* $((list,\ u),\ A) \wedge hd\ (s@t) = a \wedge length\ s > 0 \longrightarrow yt32 \leq list \wedge$
$yu32\ \leq u \wedge$
$(tl\ s)$ *setinterleaves* $((yt32,\ yu32),\ A)$
by (*metis add-less-mono1 impossible-Cons le-neq-implies-less length-greater-0-conv*

nat-le-linear tl-append2)

from *b c* **have** *th3pre3*: $aa \notin A \wedge a \notin A \wedge tl\ s$ *setinterleaves* $((yt31,\ yu31),\ A) \longrightarrow$
$(aa\#(tl\ s))$ *setinterleaves* $((yt31,\ aa\#yu31),\ A)$
by (*metis* (*mono-tags*, *lifting*) *addNonSync doubleReverse rev.simps*(*2*) *rev-rev-ident*)

from *b c th3pre1 th3pre2* **have** *th3pre4*: $aa \notin A \wedge a \notin A \wedge (s\ @\ t)$ *setinterleaves* $((ta,\ u),\ A) \wedge$
$length\ s > 0 \wedge tl\ (s\ @\ t)$ *setinterleaves* $((ta,\ lista),\ A) \wedge hd\ (s@t) = aa \longrightarrow yt31 \leq ta \wedge$
$aa\#yu31\ \leq u \wedge$
s *setinterleaves* $((yt31,\ aa\#yu31),\ A)$
by (*metis hd-append2 le-less length-greater-0-conv less-cons list.exhaust-sel th3pre3*)

from *b c* **have** *th3pre5*: $aa \notin A \wedge a \notin A \wedge tl\ s$ *setinterleaves* $((yt32,\ yu32),\ A) \longrightarrow$
$(a\#(tl\ s))$ *setinterleaves* $((a\#yt32,\ yu32),\ A)$
by (*metis* (*mono-tags*, *lifting*) *addNonSync doubleReverse rev.simps*(*2*) *rev-rev-ident*)

from *b c th3pre1 th3pre2* **have** *th3pre6*: $aa \notin A \wedge a \notin A \wedge (s\ @\ t)$ *setinterleaves* $((ta,\ u),\ A) \wedge$
$length\ s > 0 \wedge tl\ (s\ @\ t)$ *setinterleaves* $((list,\ u),\ A) \wedge hd\ (s@t) = a \longrightarrow a\#yt32 \leq ta \wedge$
$yu32\ \leq u \wedge$
s *setinterleaves* $((a\#yt32,\ yu32),\ A)$
using *th3pre th3pre5* **by** (*metis hd-append2 le-less length-greater-0-conv less-cons*

list.exhaust-sel)

have *th3pre7*:$aa \notin A \wedge a \notin A \wedge (s\ @\ t)$ *setinterleaves* $((ta,\ u),\ A) \wedge length\ s = 0 \longrightarrow$
$[] \leq ta \wedge\ [] \leq u \wedge\ s$ *setinterleaves* $(([],\ []),\ A)$ **by** *auto*

have *th3*: $aa \notin A \wedge a \notin A \Longrightarrow \exists\, t1\ u1.\ (s\ @\ t)$ *setinterleaves* $((ta,\ u),\ A) \longrightarrow$
$t1 \leq ta \wedge u1 \leq u \wedge s$ *setinterleaves* $((t1,\ u1),\ A)$

```
      using th3pre th3pre4 th3pre6 th3pre7 by blast
    with a b c th th1 th2 show ?thesis
      by fastforce
  qed
 qed
qed

lemma SyncWithTick-imp-NTF1:
(s @ [tick]) setinterleaves ((t, u), insert tick (ev ‘ A))
 ⟹ t∈ 𝒯 P ⟹ u∈ 𝒯 Q
 ⟹ ∃ t u Xa Y. (t, Xa) ∈ ℱ P ∧ ( (u, Y) ∈ ℱ Q ∧
              s setinterleaves ((t, u), insert tick (ev ‘ A)) ∧
              X − {tick} = (Xa ∪ Y) ∩ insert tick (ev ‘ A) ∪ Xa ∩ Y)
 apply (drule SyncWithTick-imp-NTF)
  apply simp
  apply simp
proof −
 assume A: t ∈ 𝒯 P
 and B: u ∈ 𝒯 Q
 and C: ∃ t1 u1. t = t1 @ [tick]∧u = u1@[tick]∧s setinterleaves ((t1, u1), insert
tick (ev ‘ A))
 from C obtain t1 u1
   where D: t = t1 @ [tick] ∧ u = u1 @ [tick] ∧
              s setinterleaves ((t1, u1), insert tick (ev ‘ A)) by blast
 from A B D have E: (t1, X−{tick}) ∈ ℱ P ∧ (u1, X−{tick}) ∈ ℱ Q
   by (simp add: T-F process-charn)
 thus  ∃ t u Xa Y. (t, Xa) ∈ ℱ P ∧ (u, Y) ∈ ℱ Q ∧
                s setinterleaves ((t, u), insert tick (ev ‘ A)) ∧
                X − {tick} = (Xa ∪ Y) ∩ insert tick (ev ‘ A) ∪ Xa ∩ Y
   using A B C D E by blast
qed

lemma ftf-Sync:
 front-tickFree u
  ⟹ front-tickFree t
  ⟹ s setinterleaves ((t, u),insert tick (ev ‘ A))
  ⟹ front-tickFree s
proof−
 assume A: front-tickFree u
 assume B: front-tickFree t
 assume C: s setinterleaves ((t, u), insert tick (ev ‘ A))
 thus front-tickFree s
 proof−
   from A obtain u1 where A1: ∀ u2. u1≤u∧tickFree u1∧  (u2≤u∧tickFree
u2⟶u2≤u1)
     by (metis append.right-neutral append-eq-first-pref-spec front-tickFree-charn
le-list-def
       tickFree-Nil)
   from A A1 have AA1: u1=u∨u=u1@[tick]
```

by (*metis*(*no-types*, *lifting*) *antisym append-Nil2 append-eq-first-pref-spec ap-pend-is-Nil-conv*
front-tickFree-charn le-list-def less-list-def less-self nonTickFree-n-frontTickFree)
from *B* **obtain** *t1* **where** *B1*: ∀ *t2*. *t1*≤*t*∧*tickFree t1*∧ (*t2*≤*t*∧*tickFree t2*⟶*t2*≤*t1*)
by (*metis append.right-neutral append-eq-first-pref-spec front-tickFree-charn le-list-def*
tickFree-Nil)
from *B B1* **have** *BB1*: *t1*=*t*∨*t*=*t1*@[*tick*]
by (*metis*(*no-types*, *lifting*) *antisym append-Nil2 append-eq-first-pref-spec ap-pend-is-Nil-conv*
front-tickFree-charn le-list-def less-list-def less-self nonTickFree-n-frontTickFree)
from *A1 B1* **have** *C1*: ∀ *s1*. *s1 setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *A*))⟶*tickFree s1*
by (*meson ftf-Sync1 tickFree-def*)
from *AA1 BB1* **have** *CC1*: *u1*=*u*∧*t1*=*t*⟶*tickFree s*
by (*simp add*: *C C1*)
from *AA1 BB1* **have** *CC2*: *u1*=*u*∧*t*=*t1*@[*tick*]⟶*tickFree s* **using** *ftf-Sync21*

by (*metis A1 C equals0D insertI1 non-tickFree-tick tickFree-append tick-Free-def*)
from *AA1 BB1* **have** *CC3*: *u*=*u1*@[*tick*]∧*t1*=*t*⟶*tickFree s* **using** *ftf-Sync21*

by (*metis B1 C insertI1 insert-not-empty mk-disjoint-insert non-tickFree-tick tickFree-append*
tickFree-def)
from *AA1 BB1* **have** *CC4*: *u*=*u1*@[*tick*]∧*t*=*t1*@[*tick*]⟹
∃ *s1*. *s1 setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *A*))∧(*s*=*s1*@[*tick*])
using *C ftf-Sync32* **by** *blast*
from *C1 CC4* **have** *CC5*: *front-tickFree s*
using *AA1 BB1 CC1 CC2 CC3 tickFree-implies-front-tickFree*
tickFree-implies-front-tickFree-single **by** *auto*
with *A1 B1 C1 AA1 BB1* **show** *?thesis*
using *CC1 CC2 CC3 tickFree-implies-front-tickFree* **by** *auto*
qed
qed

lemma *is-processT5-SYNC2*:
⋀ *sa Y t u Xa Ya*. (*t*,*Xa*) ∈ ℱ *P* ∧ (*u*,*Ya*) ∈ ℱ *Q* ∧ (*sa setinterleaves* ((*t*,*u*),*insert tick* (*ev* ‘ *A*)))
⟹∀ *c*. *c* ∈ *Y* ⟶
(∀ *t1 u1*. (*sa*@[*c*]) *setinterleaves* ((*t1*,*u1*),*insert tick* (*ev* ‘ *A*))⟶
(((*t1*,{}) ∈ ℱ *P*⟶(*u1*, {}) ∉ ℱ *Q*)∧((*t1*,{}) ∈ ℱ *Q*⟶(*u1*, {}) ∉ ℱ *P*)))
⟹ ∃ *t2 u2 Xb*. (*t2*, *Xb*) ∈ ℱ *P* ∧
(∃ *Yb*. (*u2*, *Yb*) ∈ ℱ *Q* ∧
sa setinterleaves ((*t2*, *u2*), *insert tick* (*ev* ‘ *A*)) ∧
(*Xa* ∪ *Ya*) ∩ *insert tick* (*ev* ‘ *A*) ∪ *Xa* ∩ *Ya* ∪ *Y* =
(*Xb* ∪ *Yb*) ∩ *insert tick* (*ev* ‘ *A*) ∪ *Xb* ∩ *Yb*)
proof −

fix *sa Y t u Xa Ya*
assume *A*: $(t,Xa) \in \mathcal{F}\ P \wedge (u,Ya) \in \mathcal{F}\ Q \wedge$(*sa setinterleaves* ((*t*,*u*),*insert tick* (*ev* ‘ *A*)))
and *B*: $\forall$ *c*. *c* $\in$ *Y* $\longrightarrow$
($\forall$ *t1 u1*. (*sa*@[*c*]) *setinterleaves* ((*t1*,*u1*),*insert tick* (*ev* ‘ *A*)) $\longrightarrow$
(((*t1*,{}) $\in \mathcal{F}\ P \longrightarrow$
(*u1*, {}) $\notin \mathcal{F}\ Q$) $\wedge$ ((*t1*,{}) $\in \mathcal{F}\ Q \longrightarrow$(*u1*, {}) $\notin \mathcal{F}\ P$)))
thus $\exists$ *t2 u2 Xb*. (*t2*, *Xb*) $\in \mathcal{F}\ P \wedge$
($\exists$ *Yb*. (*u2*, *Yb*) $\in \mathcal{F}\ Q \wedge$ *sa setinterleaves* ((*t2*, *u2*), *insert tick* (*ev* ‘ *A*)) $\wedge$
(*Xa* $\cup$ *Ya*) $\cap$ *insert tick* (*ev* ‘ *A*) $\cup$ *Xa* $\cap$ *Ya* $\cup$ *Y* =
(*Xb* $\cup$ *Yb*) $\cap$ *insert tick* (*ev* ‘ *A*) $\cup$ *Xb* $\cap$ *Yb*)
proof−
from *A B* **obtain** *Y1 Y2*
where *A1*: (*Y1*=(*Y*$\cap$*insert tick* (*ev* ‘ *A*)))$\wedge$(*Y2*=(*Y* − *insert tick* (*ev* ‘ *A*)))
by *blast*
from *A1* **have** *AA1*: *Y1* $\cup$ *Y2* = *Y*
by *blast*
from *A1* **have** *newAA*: $\forall$ *y*$\in$*Y1*. *y*$\in$ *insert tick* (*ev* ‘ *A*)
by *blast*
from *A1* **have** *newAAA*: $\forall$ *y*$\in$*Y2*. *y*$\notin$ *insert tick* (*ev* ‘ *A*)
by *blast*
from *B A1* **have** *AA2*: $\forall$ *z*$\in$*Y1*.(*t*@[*z*], {}) $\notin \mathcal{F}\ P \vee$ (*u*@[*z*], {}) $\notin \mathcal{F}\ Q$
proof(*cases* $\exists$ *z* $\in$ *Y1*. (*t*@[*z*], {})$\in\mathcal{F}\ P \wedge$ (*u*@[*z*], {}) $\in \mathcal{F}\ Q$)
case *True*
from *True* **obtain** *z1* **where** *TrA*: *z1*$\in$*Y1*$\wedge$(*t*@[*z1*], {})$\in\mathcal{F}\ P \wedge$ (*u*@[*z1*], {}) $\in \mathcal{F}\ Q$
by *blast*
from *TrA A* **have** *TrB*:
⟦*sa setinterleaves* ((*t*,*u*),*insert tick*(*ev* ‘ *A*));
z1$\in$*insert tick* (*ev* ‘ *A*)⟧
$\Longrightarrow$(*sa*@[*z1*]) *setinterleaves* ((*t*@[*z1*],(*u*@[*z1*])),*insert tick* (*ev* ‘ *A*))
proof−
have *a*: (*rev sa*) *setinterleaves* ((*rev t*,*rev u*),*insert tick* (*ev* ‘ *A*))
using *local.A doubleReverse* **by** *blast*
have *b*: (*z1*#*rev sa*) *setinterleaves* ((*z1*#*rev t*,*z1*# *rev u*),*insert tick* (*ev* ‘ *A*))
using *TrA a newAA* **by** *auto*
show *?thesis*
using *b doubleReverse* **by** *fastforce*
qed
then show *?thesis*
using *A1 B TrA local.A* **by** *blast*
next
case *False*
then show *?thesis*
by *blast*
qed

from *A B A1* **obtain** *Z1 Z2* **where** *A2*: $(Z1=Y1\cap\{z.(t@[z], \{\})\notin\mathcal{F}\ P\})\wedge(Z2=Y1-\{z.(t@[z], \{\})\notin\mathcal{F}\ P\})$
by *blast*
from *A2* **have** *BA*: $Y1=Z1\cup Z2$
by *blast*
from *A2* **have** *BAA*: $\forall z\in Z1.\ (t@[z], \{\}) \notin \mathcal{F}\ P$
by *blast*
from *A2* **have** *BAAA*: $\forall z\in Z2.\ (u@[z], \{\})\notin \mathcal{F}\ Q$
using *AA2* **by** *blast*
from *A2 BA BAA* **have** *A3*: $(t, Z1)\in \mathcal{F}\ P$
by (*meson F-T NF-NT is-processT5-S7 local.A*)
from *A A1* **have** *A43*: $\forall y\in Y2.\ (t@[y], \{\})\notin \mathcal{F}\ P \wedge (u@[y], \{\})\notin \mathcal{F}\ Q$
proof(*cases* $\exists y\in Y2.\ ((t@[y], \{\})\in \mathcal{F}\ P)\vee((u@[y], \{\})\in \mathcal{F}\ Q)$)
case *True*
from *True* **obtain** *y1* **where** *innAA*: $y1\in Y2\wedge(((t@[y1], \{\})\in \mathcal{F}\ P)\vee((u@[y1], \{\})\in \mathcal{F}\ Q))$ **by** *blast*
from *innAA* **have** *innAAA*: $y1\notin$ *insert tick* (*ev* ‘ *A*)
using *newAAA* **by** *auto*
from *innAA* **have** *innAA1*:
$[\![$*sa setinterleaves* ((*t*,*u*),*insert tick* (*ev* ‘ *A*));
y1 $\notin$ *insert tick* (*ev* ‘ *A*)$]\!]$
$\Longrightarrow$ $((t@[y1], \{\})\in \mathcal{F}\ P$
$\longrightarrow$(*sa*@[*y1*]) *setinterleaves* ((*t*@[*y1*],*u*),*insert tick* (*ev* ‘ *A*))) $\wedge$
$((u@[y1], \{\})\in \mathcal{F}\ Q\longrightarrow$(*sa*@[*y1*]) *setinterleaves*((*t*,*u*@[*y1*]),*insert tick* (*ev*‘*A*)))
by (*simp add*: *addNonSync*)
with *A B innAA1 innAA* **show** *?thesis*
by (*metis A1 DiffD1 innAAA is-processT4-empty*)
next
case *False*
then show *?thesis* **by** *blast*
qed
from *A1 A2* **obtain** *Xbb* **where** *B1*: $Xbb=(Xa\cup Z1\cup Y2)$
by *blast*
from *A B1 A3 A43* **have** *A5*: $(t, Xbb)\in \mathcal{F}\ P$
by (*meson BAA is-processT5-S1*)
from *A1 A2* **obtain** *Ybb* **where** *B2*: $Ybb=(Ya\cup Z2\cup Y2)$
by *blast*
from *A B* **have** *A52*: $(u, Ybb)\in \mathcal{F}\ Q$
by (*metis A43 B2 BAAA is-processT5-S1*)
from *A1 A2* **have** *A61*: $(Xbb \cup Ybb)\cap$*insert tick* (*ev* ‘ *A*)$=((Xa\cup Ya\cup Y1\cup Y2)\cap$ *insert tick* (*ev* ‘ *A*))
using *B1 B2* **by** *blast*
from *A1* **have** *A62*: $(Y1) \cap$ *insert tick* (*ev* ‘ *A*)$=Y1$
using *inf.orderE* **by** *auto*
from *A1* **have** *A63*: $(Y2) \cap$ *insert tick* (*ev* ‘ *A*)$=\{\}$
by (*simp add*: *AA1 Int-commute*)
from *A61 A62 A63* **have** *A64*:$((Xa\cup Ya\cup Y1\cup Y2)\cap$*insert tick*(*ev* ‘*A*))$=((Xa\cup Ya)\cap$*insert tick*(*ev* ‘*A*)$\cup Y1)$

by (*simp add*: *Int-Un-distrib2*)
from *AA1 BA B1 B2* **have** *A65*: $(Xbb \cap Ybb) \subseteq ((Xa \cap Ya) \cup Y)$
by *auto*
from *AA1 BA B1 B2* **have** *A66*: $(Xbb \cap Ybb) \supseteq ((Xa \cap Ya) \cup Y2)$
by *auto*
from *A1 A2* **have** *A66*: $((Xa \cup Ya) \cap$ *insert tick* $(ev$ ‘ $A) \cup Xa \cap Ya \cup Y)$
$= ((Xbb \cup Ybb) \cap$
insert tick $(ev$ ‘ $A) \cup Xbb \cap Ybb)$
proof −
have *f1*: $(Xa \cup Ya) \cap$ *insert tick* $(ev$ ‘ $A) \cup Xa \cap Ya \cup Y = Y \cup Xa \cap Ya$
$\cup ((Xa \cup Ya) \cap$
insert tick $(ev$ ‘ $A) \cup Xbb \cap Ybb)$**using** *A65* **by** *auto*
have $Xa \cap Ya \cup Y2 \cup Xbb \cap Ybb = Xbb \cap Ybb$
by (*meson A66 le-iff-sup*)
then have $(Xa \cup Ya) \cap$ *insert tick* $(ev$ ‘ $A) \cup Xa \cap Ya \cup Y = Xbb \cap Ybb$
$\cup ((Xa \cup Ya) \cap$
insert tick $(ev$ ‘ $A) \cup Y1)$ **using** *f1 A1* **by** *auto*
then show *?thesis*
using *A61 A64* **by** *force*
qed
with *A5 A52 A66* **show** *?thesis*
using *local.A* **by** *blast*
qed
qed

lemma *pt3*:
$ta \in \mathcal{T}\ P$
$\Longrightarrow u \in \mathcal{T}\ Q \Longrightarrow (s\ @\ t)$ *setinterleaves* $((ta,\ u),$ *insert tick* $(ev$ ‘ $A))$
$\Longrightarrow \exists\, t\ u\ X.\ (t,\ X) \in \mathcal{F}\ P \wedge$
$(\exists\, Y.\ (u,\ Y) \in \mathcal{F}\ Q \wedge$
s *setinterleaves* $((t,\ u),$ *insert tick* $(ev$ ‘ $A)) \wedge$
$tick \notin X \wedge tick \notin Y \wedge (X \cup Y) \cap ev$ ‘ $A = \{\} \wedge X \cap Y = \{\})$
proof−
assume *a1*: $ta \in \mathcal{T}\ P$
assume *a2*: $u \in \mathcal{T}\ Q$
assume *a3*: $(s\ @\ t)$ *setinterleaves* $((ta,\ u),$ *insert tick* $(ev$ ‘ $A))$
have *aa*: $ta \in \mathcal{T}\ P \Longrightarrow u \in \mathcal{T}\ Q \Longrightarrow (s\ @\ t)$ *setinterleaves* $((ta,\ u),$ *insert tick*
$(ev$ ‘ $A)) \Longrightarrow$
$\exists\, t\ u.\ t \in \mathcal{T}\ P \wedge (u \in \mathcal{T}\ Q \wedge s$ *setinterleaves* $((t,\ u),$ *insert tick* $(ev$ ‘
$A)))$
using *is-processT3-ST-pref synPrefix* **by** *blast*
thus $\exists\, t\ u\ X.\ (t,\ X) \in \mathcal{F}\ P \wedge (\exists\, Y.\ (u,\ Y) \in \mathcal{F}\ Q \wedge$
s *setinterleaves* $((t,\ u),$ *insert tick* $(ev$ ‘ $A)) \wedge$
$tick \notin X \wedge tick \notin Y \wedge (X \cup Y) \cap ev$ ‘ $A = \{\} \wedge X \cap Y = \{\})$
using *NF-NT a1 a2 a3* **by** *blast*
qed

lemma *Sync-maintains-is-process*:

```
is-process ({(s,R).∃ t u X Y. (t,X) ∈ 𝓕 P ∧ (u,Y) ∈ 𝓕 Q ∧
                        (s setinterleaves ((t,u),(ev‘A) ∪ {tick})) ∧
                        R = (X ∪ Y) ∩ ((ev‘A) ∪ {tick}) ∪ X ∩ Y} ∪
           {(s,R).∃ t u r v. front-tickFree v ∧ (tickFree r ∨ v=[]) ∧
                        s = r@v ∧
                        (r setinterleaves ((t,u),(ev‘A) ∪ {tick})) ∧
                        (t ∈ 𝓓 P ∧ u ∈ 𝓣 Q ∨ t ∈ 𝓓 Q ∧ u ∈ 𝓣 P)},

           {s.    ∃ t u r v. front-tickFree v ∧ (tickFree r ∨ v=[]) ∧
                        s = r@v ∧
                        (r setinterleaves ((t,u),(ev‘A) ∪ {tick})) ∧
                        (t ∈ 𝓓 P ∧ u ∈ 𝓣 Q ∨ t ∈ 𝓓 Q ∧ u ∈ 𝓣 P)})
           (is is-process(?f, ?d))
proof(simp only: fst-conv snd-conv Failures-def is-process-def FAILURES-def DI-
VERGENCES-def,
     fold FAILURES-def DIVERGENCES-def,fold Failures-def,intro conjI)
  show ([], {}) ∈ ?f
    apply auto
    by (metis Int-commute Int-empty-right Sync.si-empty1 Un-empty empty-iff in-
sert-iff is-processT1)
next
  show  ∀ s X. (s, X) ∈ ?f ⟶ front-tickFree s
      apply (auto simp:is-processT2 append-T-imp-tickFree front-tickFree-append
D-imp-front-tickFree)
      using ftf-Sync is-processT2 apply blast
     using D-imp-front-tickFree ftf-Sync is-processT2-TR apply blast
    using D-imp-front-tickFree ftf-Sync is-processT2-TR by blast
next
  show ∀ s t. (s @ t, {}) ∈ ?f ⟶ (s, {}) ∈ ?f
    apply auto
       apply(drule F-T)
       apply(drule F-T)
  proof(goal-cases)
    case (1 s t ta u X Y)
    then show ?case
      using pt3 by fastforce
next
  case (2 s t ta u r v)
  have aa: front-tickFree v
           ⟹ s@t = r @ v ⟹ r setinterleaves ((ta, u), insert tick (ev ‘ A))
           ⟹ ∀ t u r. r setinterleaves ((t,u),insert tick(ev ‘A)) ⟶
                    (∀ v. s=r@v⟶front-tickFree v ⟶
                    ¬tickFree r ∧ v≠[]∨(t ∈ 𝓓 P ⟶
                    u ∉ 𝓣 Q) ∧ (t ∈ 𝓓 Q ⟶ u ∉ 𝓣 P))
           ⟹ tickFree r ⟹ ta ∈ 𝓓 P ⟹ u ∈ 𝓣 Q ⟹ s<r
    apply(simp add: append-eq-append-conv2)
   by (metis append-Nil2 front-tickFree-Nil front-tickFree-dw-closed le-list-def less-list-def)
  from 2(1,2,3,4,5,6,7) Sync.sym aa have a1: s<r by blast
  have aaa: r setinterleaves ((ta, u), insert tick (ev ‘ A))
```

```
          ⟹ tickFree r ⟹ ta ∈ 𝒟 P
          ⟹ u ∈ 𝒯 Q ⟹ s<r
          ⟹ ∃ t u X. (t, X) ∈ ℱ P ∧ (∃ Y. (u, Y) ∈ ℱ Q ∧
                s setinterleaves ((t, u), insert tick (ev ‘ A)) ∧
               tick ∉ X ∧ tick ∉ Y ∧ (X ∪ Y) ∩ ev ‘ A = {} ∧ X ∩ Y = {})
    apply(drule D-T)
    apply(simp add: le-list-def less-list-def)
    using Sync.sym pt3 by blast
  have aab: ∃ t u X. (t, X) ∈ ℱ P ∧(∃ Y. (u, Y) ∈ ℱ Q ∧
                s setinterleaves ((t, u), insert tick (ev ‘ A)) ∧
               tick ∉ X ∧ tick ∉ Y ∧ (X ∪ Y) ∩ ev ‘ A = {} ∧ X ∩ Y = {})
    using 2(1,2,3,4,5,6,7) aa aaa Sync.sym by auto
  then show ?case by auto
next
  case (3 s t ta u r v)
  have aa: front-tickFree v ⟹s @ t = r @ v
          ⟹ r setinterleaves ((ta, u), insert tick (ev ‘ A))
          ⟹ ∀ t u r. r setinterleaves ((t, u), insert tick (ev ‘ A)) ⟶
               (∀ v. s = r @ v ⟶front-tickFree v ⟶¬ tickFree r ∧ v ≠ [] ∨
                    (t ∈ 𝒟 P ⟶ u ∉ 𝒯 Q) ∧ (t ∈ 𝒟 Q ⟶ u ∉ 𝒯 P))
          ⟹ tickFree r
          ⟹ ta ∈ 𝒟 Q ⟹ u ∈ 𝒯 P ⟹ s<r
    apply(simp add: append-eq-append-conv2)
   by (metis append-Nil2 front-tickFree-Nil front-tickFree-dw-closed le-list-def less-list-def)

  have aaa: r setinterleaves ((ta, u), insert tick (ev ‘ A))
          ⟹ tickFree r ⟹ ta ∈ 𝒟 Q ⟹ u ∈ 𝒯 P ⟹s<r
          ⟹ ∃ t u X. (t, X) ∈ ℱ P ∧
                (∃ Y. (u, Y) ∈ ℱ Q ∧s setinterleaves ((t, u), insert tick (ev ‘
A)) ∧
                 tick ∉ X ∧ tick ∉ Y ∧ (X ∪ Y) ∩ ev ‘ A = {} ∧ X ∩ Y =
{})
    apply(drule D-T)
    apply(simp add: le-list-def less-list-def)
    using Sync.sym pt3 by blast
  from 3(1,2,3,4,5,6,7) Sync.sym aa have a1: s<r
    by blast
  have aab: ∃ t u X. (t, X) ∈ ℱ P ∧ (∃ Y. (u, Y) ∈ ℱ Q ∧
                s setinterleaves ((t, u), insert tick (ev ‘ A)) ∧
               tick ∉ X ∧ tick ∉ Y ∧ (X ∪ Y) ∩ ev ‘ A = {} ∧ X ∩ Y = {})
    using 3(1,2,3,4,5,6,7) aa aaa Sync.sym by auto
  then show ?case
    by auto
next
  case (4 s t ta u)
  from 4(3) have a1: ta ∈ 𝒯 P
    by (simp add: D-T)
  then show ?case
    using 4(1) 4(4) pt3 by blast
```

```
next
  case (5 s t ta u)
  from 5(3) have a1: ta ∈ 𝒯 Q
    by (simp add: D-T)
  then show ?case
    using 5(1) 5(4) Sync.sym pt3 by blast
qed
next
  show ∀s X Y. (s, Y) ∈ ?f ∧ X ⊆ Y ⟶ (s, X) ∈ ?f
    apply auto
  proof(goal-cases)
    case (1 s X t u Xa Ya)
     have aa: X ⊆ ((Xa ∪ Ya) ∩ insert tick (ev ‘ A) ∪ Xa ∩ Ya )⟹∃x1 y1.
x1⊆Xa ∧ y1⊆Ya ∧
    X=((x1 ∪ y1) ∩ insert tick (ev ‘ A) ∪ x1 ∩ y1)
      apply(simp add: Set.subset-iff-psubset-eq)
      apply(erule disjE)
       defer
       apply blast
    proof−
      assume A: X ⊂ (Xa ∪ Ya) ∩ insert tick (ev ‘ A) ∪ Xa ∩ Ya
      from A obtain X1 where B: X1=((Xa ∪ Ya) ∩ insert tick (ev ‘ A) ∪ Xa
∩ Ya)−X
        by blast
      from A obtain x where C: x=Xa−X1
        by blast
      from A obtain y where D: y=Ya−X1
        by blast
      from A B C D have E: X = (x ∪ y) ∩ insert tick (ev ‘ A) ∪ x ∩ y
        by blast
     thus ∃x1. (x1 ⊂ Xa ∨ x1 = Xa) ∧ (∃y1. (y1 ⊂ Ya ∨ y1 = Ya) ∧ X = (x1
∪ y1) ∩
      insert tick (ev ‘ A) ∪ x1 ∩ y1)
       using A B C D E by (metis Un-Diff-Int inf.strict-order-iff inf-sup-absorb)
    qed
    then show ?case using 1(1,2,3,4,5)
      by (meson process-charn)
  qed

next
  let ?f1 = {(s,R). ∃ t u X Y. (t,X) ∈ ℱ P ∧ (u,Y) ∈ ℱ Q ∧
                  (s setinterleaves ((t,u),(ev‘A) ∪ {tick})) ∧
                  R = (X ∪ Y) ∩ ((ev‘A) ∪ {tick}) ∪ X ∩ Y}
  have is-processT5-SYNC3:
          ⋀sa X Y.(sa, X) ∈?f1 ⟹(∀c. c∈Y⟶(sa@[c],{})∉?f)⟹(sa, X∪Y)
∈ ?f1
    apply(clarsimp)
    apply(rule is-processT5-SYNC2[simplified])
     apply(auto simp:is-processT5) apply blast
```

by (*metis Sync.sym Un-empty-right inf-sup-absorb inf-sup-aci*(*5*) *insert-absorb insert-not-empty*)
show $\forall s\ X\ Y.\ (s,\ X) \in ?f \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin ?f) \longrightarrow (s,\ X \cup Y) \in ?f$
apply(*intro allI impI*, *elim conjE UnE*)
apply(*rule rev-subsetD*)
apply(*rule is-processT5-SYNC3*)
by(*auto*)
next
show $\forall s\ X.\ (s\ @\ [tick],\ \{\}) \in ?f \longrightarrow (s,\ X - \{tick\}) \in ?f$
apply *auto*
apply(*drule F-T*)
apply(*drule F-T*)
using *SyncWithTick-imp-NTF1* **apply** *fastforce*
apply(*metis append-assoc append-same-eq front-tickFree-dw-closed nonTick-Free-n-frontTickFree*
non-tickFree-tick tickFree-append)
apply(*metis butlast-append butlast-snoc front-tickFree-charn non-tickFree-tick tickFree-append*
tickFree-implies-front-tickFree)
apply (*metis NT-ND append-Nil2 front-tickFree-Nil is-processT3-ST is-processT9-S-swap*

SyncWithTick-imp-NTF)
by (*metis NT-ND append-Nil2 front-tickFree-Nil is-processT3-ST is-processT9-S-swap*

SyncWithTick-imp-NTF)

next
show $\forall s\ t.\ s \in ?d \wedge tickFree\ s \wedge front\text{-}tickFree\ t \longrightarrow s\ @\ t \in ?d$
apply *auto*
using *front-tickFree-append* **apply** *blast*
using *front-tickFree-append* **apply** *blast*
apply *blast*
by *blast*
next
show $\forall s\ X.\ s \in ?d \longrightarrow (s,\ X) \in ?f$
by *blast*
next
show $\forall s.\ s\ @\ [tick] \in ?d \longrightarrow s \in ?d$
apply *auto*
apply (*metis butlast-append butlast-snoc front-tickFree-charn non-tickFree-tick*

tickFree-append tickFree-implies-front-tickFree)
apply (*metis butlast-append butlast-snoc front-tickFree-charn non-tickFree-tick*

tickFree-append tickFree-implies-front-tickFree)
apply (*metis D-T append.right-neutral front-tickFree-Nil is-processT3-ST is-processT9*
SyncWithTick-imp-NTF)

by (*metis D-T append.right-neutral front-tickFree-Nil is-processT3-ST is-processT9*

SyncWithTick-imp-NTF)
qed

lemmas *Rep-Abs-Sync*[*simp*] = *Abs-process-inverse*[*simplified*, *OF Sync-maintains-is-process*]

4.2.4 Projections

lemma
F-Sync : $\mathcal{F}\ (P\ [\![\ A\ ]\!]\ Q) =$
$\{(s,R).\exists\ t\ u\ X\ Y.\ (t,X) \in \mathcal{F}\ P\ \wedge$
$(u,Y) \in \mathcal{F}\ Q\ \wedge$
s *setinterleaves* $((t,u),(ev`A) \cup \{tick\})\ \wedge$
$R = (X \cup Y) \cap ((ev`A) \cup \{tick\}) \cup X \cap Y\} \cup$
$\{(s,R).\exists\ t\ u\ r\ v.$ *front-tickFree* $v\ \wedge$
(*tickFree* $r \vee v=[]$) $\wedge$
$s = r@v\ \wedge$
r *setinterleaves* $((t,u),(ev`A)\cup\{tick\})\ \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\}$

unfolding *Sync-def*
apply(*subst Failures-def*)
apply(*simp only*: *Rep-Abs-Sync*)
by(*auto simp*: *FAILURES-def*)

lemma
D-Sync : $\mathcal{D}\ (P\ [\![\ A\ ]\!]\ Q) =$
$\{s.\ \ \exists\ t\ u\ r\ v.$ *front-tickFree* $v\ \wedge$ (*tickFree* $r \vee v=[]$) $\wedge$
$s = r@v\ \wedge\ r$ *setinterleaves* $((t,u),(ev`A) \cup \{tick\})\ \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\}$

unfolding *Sync-def*
apply(*subst D-def*)
apply(*simp only*: *Rep-Abs-Sync*)
by(*simp add*: *DIVERGENCES-def*)

lemma
T-Sync : $\mathcal{T}\ (P\ [\![\ A\ ]\!]\ Q) =$
$\{s.\ \ \exists\ t\ u.\ t \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q\ \wedge$
s *setinterleaves* $((t,u),(ev`A) \cup \{tick\})\}\cup$
$\{s.\ \ \exists\ t\ u\ r\ v.$ *front-tickFree* $v\ \wedge$ (*tickFree* $r \vee v=[]$) $\wedge$
$s = r@v\ \wedge\ r$ *setinterleaves* $((t,u),(ev`A) \cup \{tick\})\ \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\}$

unfolding *Sync-def*

apply(*subst Traces-def*, *simp only*: *Rep-Abs-Sync*, *simp add*:*TRACES-def FAILURES-def*)
apply *auto*
apply (*meson F-T*)
using *T-F* **by** *blast*

4.2.5 Syntax for Interleave and Parallel Operator

abbreviation *Inter-syntax* ((-|||-) [*72*,*73*] *72*)
where P ||| $Q \equiv (P$ ⟦ {} ⟧ $Q)$

abbreviation *Par-syntax* ((-||-) [*74*,*75*] *74*)
where P || $Q \equiv (P$ ⟦ *UNIV* ⟧ $Q)$

4.2.6 Continuity Rule

lemma *mono-Sync-D1*:
$P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (Q$ ⟦ A ⟧ $S) \subseteq \mathcal{D}\ (P$ ⟦ A ⟧ $S)$
apply(*auto simp*: *D-Sync*)
using *le-approx1* **apply** *blast*
using *le-approx-lemma-T* **apply** *blast*
apply (*metis append-Nil2 le-approx1 subsetCE tickFree-Nil tickFree-implies-front-tickFree*)

by (*metis append-Nil2 le-approx-lemma-T subsetCE tickFree-Nil tickFree-implies-front-tickFree*)

lemma *mono-Sync-D2*:
$P \sqsubseteq Q \Longrightarrow (\forall\ s.\ s \notin \mathcal{D}\ (P$ ⟦ A ⟧ $S) \longrightarrow Ra\ (P$ ⟦ A ⟧ $S)\ s = Ra\ (Q$ ⟦ A ⟧ $S)\ s)$
apply *auto*
apply(*simp add*: *Ra-def D-Sync F-Sync*)
apply (*metis* (*no-types*, *lifting*) *F-T append-Nil2 front-tickFree-Nil le-approx2*)
apply(*simp add*: *Ra-def D-Sync F-Sync*)
apply(*insert le-approx-lemma-F*[*of P Q*] *le-approx-lemma-T*[*of P Q*]
le-approx1[*of P Q*] *le-approx2T*[*of P Q*], *auto*)
apply *blast*
apply *blast*
apply *blast*
using *front-tickFree-Nil* **apply** *blast*
using *front-tickFree-Nil* **by** *blast*

lemma *interPrefixEmpty*: *s setinterleaves* $((t,[]),A) \wedge t1<t \Longrightarrow \exists\ s1<s.$ *s1 setinterleaves* $((t1,\ []),\ A)$
by (*metis* (*no-types*, *lifting*) *Sync.sym emptyLeftProperty le-list-def nil-le2 order.strict-implies-order synPrefix1*)

lemma *interPrefix*:$\exists\ u1\ s1.$ *s setinterleaves*$((t,u),A) \wedge t1<t \longrightarrow u1 \leq u \wedge s1<s \wedge$ *s1 setinterleaves*$((t1,u1),A)$
proof (*induction length t* + *length u arbitrary*:*s t u t1 rule*:*nat-less-induct*)
case *1*
then show *?case*
apply(*cases t*)

```
    apply auto[1]
   apply(cases u)
    apply (meson interPrefixEmpty nil-le2)
  proof−
   fix a list aa lista
    assume a: ∀m<length t + length u. ∀x xa. m = length x + length xa ⟶
(∀xb xc. ∃u1 s1.
   xb setinterleaves((x, xa),A)∧xc < x ⟶ u1 ≤ xa ∧ s1 < xb ∧ s1 setinterleaves
((xc, u1), A))
   and b: t = a # list
   and c: u = aa # lista
   thus ∃u1 s1. s setinterleaves((t, u),A)∧t1<t⟶u1 ≤u∧s1<s∧s1 setinterleaves
((t1, u1), A)
   proof−
     from b c have th0pre: a=aa∧aa∈ A⟹s setinterleaves ((t, u), A) ⟹
     (tl s) setinterleaves ((list,lista), A) by auto
     from a obtain yu ys
       where th0pre1: a=aa∧aa∈ A ∧ s setinterleaves ((t, u), A)∧ t1 < a # list
∧ length t1>0
                    ⟹ yu ≤ lista ∧ ys<tl s ∧ ys setinterleaves ((tl t1, yu), A)
       apply (simp add: le-list-def less-list-def append-eq-Cons-conv)
       by (metis add-less-mono b c length-Cons lessI list.sel(3) th0pre)
     from th0pre th0pre1  b c
     have th0pre2: a=aa ∧ aa∈A
                 ⟹ s setinterleaves ((a # list, u), A) ∧ t1<a#list∧length t1>0
             ⟹ (a#ys)setinterleaves((a#tl t1, a#yu), A)∧a#ys<s∧a#yu≤u∧t1=a#tl
t1
       apply (simp add: le-list-def less-list-def  Cons-eq-append-conv)
       by (metis append-Cons list.exhaust-sel list.inject list.sel(3))
      have th0pre3: a=aa∧aa∈A⟹s setinterleaves ((a # list, u), A)∧t1 < a #
list∧length t1=0⟹
     []<s∧[]≤u∧ [] setinterleaves ((t1, []), A)
       apply (simp add: le-list-def less-list-def) using c by auto
      from  th0pre2 th0pre3 c  have th0 : a=aa∧aa∈A⟹s setinterleaves ((a #
list, u), A)∧
     t1 < a # list⟹∃u1≤u. ∃s1<s. s1 setinterleaves ((t1, u1), A)  by fastforce
     from b c have th1pre: a∉A∧aa∈ A⟹s setinterleaves ((a # list, u), A) ⟹
     (tl s) setinterleaves ((list,aa#lista), A) by auto
     from a obtain yu1 ys1  where th1pre1: a∉A∧aa∈ A∧s setinterleaves ((a #
list, u), A)∧
      t1 < a # list∧length t1>0⟹yu1≤u∧ys1<tl s∧ys1 setinterleaves ((tl t1,
yu1), A)
       apply (simp add: le-list-def less-list-def append-eq-Cons-conv)
       by (metis add-Suc b c length-Cons lessI list.sel(3) th1pre)
      from th1pre1 have th1pre2: a∉A∧aa∈ A ∧ ys1 setinterleaves ((tl t1, yu1),
A)⟹
     (a#ys1) setinterleaves ((a#tl t1, yu1), A) apply simp
         by (metis (mono-tags, lifting) addNonSync doubleReverse rev.simps(2)
rev-rev-ident)
```

from *th1pre th1pre1 b c* **have** *th1pre3*: $a\notin A\wedge aa\in A\Longrightarrow s$ *setinterleaves* $((a$ # *list*, *u*), *A*)∧
t1<*a*#*list*∧*length t1*>*0*⟹(*a*#*ys1*) *setinterleaves* ((*a*#*tl t1*, *yu1*), *A*)∧*a*#*ys1*<*s*∧*yu1*≤*u*∧*t1*=*a*#*tl t1*
apply (*simp add*: *le-list-def less-list-def*)
by (*metis append-Cons list.exhaust-sel list.inject list.sel*(*3*) *th1pre2*)
have *th1preEmpty*: $a\notin A\wedge aa\in A\Longrightarrow s$ *setinterleaves* ((*a* # *list*, *u*), *A*)∧*t1* < *a* # *list*∧*length t1*=*0*
⟹[]<*s*∧[]≤*u*∧ [] *setinterleaves* ((*t1*, []), *A*) **apply** (*simp add*: *le-list-def less-list-def*)
using *c* **by** *auto*
from *c* **have** *th1*: $a\notin A\wedge aa\in A\Longrightarrow s$ *setinterleaves* ((*a* # *list*, *u*), *A*)∧*t1* < *a* # *list*⟹
∃ *u1*≤*u*. ∃ *s1*<*s*. *s1 setinterleaves* ((*t1*, *u1*), *A*) **using** *th1pre3 th1preEmpty* **by** *fastforce*
from *b c* **have** *th2pre*: $aa\notin A\wedge a\in A\Longrightarrow s$ *setinterleaves* ((*a* # *list*, *u*), *A*) ⟹
(*tl s*) *setinterleaves* ((*a*#*list*,*lista*), *A*) **by** *auto*
from *a th2pre* **obtain** *yu2 ys2* **where** *th2pre1*: $aa\notin A\wedge a\in A\wedge s$ *setinterleaves* ((*a* # *list*, *u*), *A*)∧
t1 < *a* # *list*∧*length t1*>*0*⟹*yu2*≤*lista*∧*ys2*<*tl s*∧*ys2 setinterleaves* ((*t1*, *yu2*), *A*)
apply (*simp add*: *le-list-def less-list-def*)
by (*metis add-Suc-right b c length-Cons lessI*)
from *th2pre1* **have** *th2pre2*: $aa\notin A\wedge a\in A$ ∧ *ys2 setinterleaves* ((*t1*, *yu2*), *A*)⟹
(*aa*#*ys2*) *setinterleaves* ((*t1*, *aa*#*yu2*), *A*) **apply** *simp*
by (*metis* (*mono-tags*, *lifting*) *addNonSync doubleReverse rev.simps*(*2*) *rev-rev-ident*)
from *th2pre th2pre1 b c* **have** *th2pre3*: $aa\notin A\wedge a\in A\Longrightarrow s$ *setinterleaves* ((*a* # *list*, *u*), *A*)∧
t1 < *a* # *list*∧*length t1*>*0*⟹(*aa*#*ys2*) *setinterleaves* ((*t1*, *aa*#*yu2*), *A*)∧*aa*#*ys2*<*s*∧*aa*#*yu2*≤*u*
apply (*simp add*: *le-list-def less-list-def*) **using** *th2pre2* **by** *auto*
have *th2preEmpty*: $aa\notin A\wedge a\in A\Longrightarrow s$ *setinterleaves* ((*a* # *list*, *u*), *A*)∧*t1* < *a* # *list*∧*length t1*=*0*
⟹[]<*s*∧[]≤*u*∧ [] *setinterleaves* ((*t1*, []), *A*) **apply** (*simp add*: *le-list-def less-list-def*)
using *c* **by** *auto*
from *th2pre3 b c* **have** *th2*: $aa\notin A\wedge a\in A\Longrightarrow s$ *setinterleaves* ((*a* # *list*, *u*), *A*)∧*t1* < *a* # *list*⟹
∃ *u1*≤*u*. ∃ *s1*<*s*. *s1 setinterleaves* ((*t1*, *u1*), *A*)
using *th2preEmpty* **by** *blast*
from *b c* **have** *th3pre*: $aa\notin A\wedge a\notin A\Longrightarrow s$ *setinterleaves* ((*a* # *list*, *u*), *A*) ⟹
(*tl s*) *setinterleaves* ((*a*#*list*,*lista*), *A*)∧*hd s*=*aa*∨(*tl s*) *setinterleaves* ((*list*,*u*), *A*)∧*hd s*=*a*
by *auto*
from *a c b th3pre* **obtain** *yu3 ys3* **where** *th3pre1*: ($aa\notin A\wedge a\notin A\wedge s$ *setinterleaves*
((*a* # *list*, *u*), *A*)∧ *t1* < *a* # *list*∧*length t1*>*0*∧(*tl s*) *setinterleaves* ((*a*#*list*,*lista*),

A)⟶
yu3≤*lista*∧*ys3*<*tl s*∧*ys3 setinterleaves*((*t1*,*yu3*),*A*)) **apply**(*simp add*:*le-list-def less-list-def*)
by (*metis* (*no-types*, *lifting*) *add-Suc length-Cons lessI*)
have *adsmallPrefix*: *t1*<*a* # *list*∧*length t1*>*0*⟹*tl t1*<*list*
using *less-tail* **by** *fastforce*
from *a b c th3pre adsmallPrefix* **obtain** *yu30 ys30* **where** *th3pre12*: (*aa*∉*A*∧*a*∉ *A*⟹
(*tl s*) *setinterleaves* ((*list*,*u*), *A*)∧*hd s*=*a*∧*tl t1*< *list*⟶*yu30*≤*u*∧*ys30*<*tl s*∧
ys30 setinterleaves ((*tl t1*, *yu30*), *A*)) **apply** (*simp add*: *le-list-def less-list-def*)

by (*metis* (*no-types*, *lifting*) *add-Suc-right length-Cons lessI*)
have *th3pre1more1*: *yu3*≤*lista* ⟹*aa*#*yu3*≤*u* **using** *c*
by (*metis le-less less-cons*)
have *th3pre1more2*: *aa*∉*A*∧*a*∉ *A*∧*s setinterleaves* ((*a* # *list*, *aa*#*lista*), *A*) ∧
(*tl s*) *setinterleaves* ((*a*#*list*,*lista*), *A*)∧*hd s*=*aa*∧*ys3*<*tl s* ⟹*aa*#*ys3*<*s* **using** *c*
by (*metis less-cons list.exhaust-sel list.sel*(*2*) *nil-less*)
have *th3pre1more21*: *aa* ∉ *A* ∧*a* ∉ *A* ∧ *ys3*<*tl s*∧*ys3 setinterleaves* ((*t1*, *yu3*), *A*) ∧
hd s = *aa* ⟶(*aa* # *ys3*) *setinterleaves* ((*t1*, *aa* # *yu3*), *A*)
by (*metis* (*mono-tags*, *lifting*) *addNonSync doubleReverse rev.simps*(*2*) *rev-rev-ident*)
from *th3pre1more1 th3pre1more2 th3pre1more21 th3pre1* **have** *th3pre1more3*: (*aa*∉*A*∧*a*∉ *A*∧
s setinterleaves ((*a* # *list*, *u*), *A*)∧ *t1* < *a* # *list*∧*length t1*>*0*∧(*tl s*) *setinterleaves*
((*a*#*list*,*lista*), *A*)∧*hd s*=*aa*⟶*aa*#*yu3*≤*u*∧*aa*#*ys3*<*s*∧(*aa*#*ys3*) *setinterleaves* ((*t1*, *aa*#*yu3*), *A*))
using *c* **by** *blast*
have *th3pre1more32*: *aa*∉*A*∧*a*∉ *A*∧*s setinterleaves* ((*a* # *list*, *aa*#*lista*), *A*) ∧
(*tl s*) *setinterleaves* ((*list*,*aa*#*lista*), *A*)∧*hd s*=*a*∧*ys30*<*tl s* ⟹*a*#*ys30*<*s* **using** *c*
by (*metis less-cons list.exhaust-sel list.sel*(*2*) *nil-less*)
have *th3pre1more213*: *aa* ∉ *A* ∧*a* ∉ *A* ∧ *ys30*<*tl s*∧*ys30 setinterleaves* ((*tl t1*, *yu30*), *A*) ∧
hd s = *a* ∧*t1*<*a*#*list*⟶(*a* # *ys30*) *setinterleaves* ((*a*#*tl t1*, *yu30*), *A*)
by (*metis* (*mono-tags*, *lifting*) *addNonSync doubleReverse rev.simps*(*2*) *rev-rev-ident*)
from *th3pre1more32 th3pre12 th3pre1more213* **have** *th3pre1more33*: (*aa*∉*A*∧*a*∉ *A*∧*s setinterleaves*
((*a* # *list*, *u*), *A*)∧ *t1* < *a* # *list*∧*length t1*>*0*∧(*tl s*) *setinterleaves* ((*list*,*aa*#*lista*), *A*)∧
hd s=*a*⟶*yu30*≤*u*∧*a*#*ys30*<*s*∧(*a*#*ys30*) *setinterleaves* ((*t1*, *yu30*), *A*))
by (*metis adsmallPrefix c hd-append2 le-list-def length-greater-0-conv list.exhaust-sel*

list.sel(*1*) *order.strict-implies-order*)
from *th3pre1more3 th3pre1more33 b c th3pre* **have** *th3NoEmpty*: *aa*∉*A*∧*a*∉

A⟹*s setinterleaves*
((*a* # *list*, *u*), *A*)∧*t1* < *a* # *list*∧*length t1*>*0*⟹∃ *u1*≤*u*. ∃ *s1*<*s*. *s1 setinterleaves* ((*t1*, *u1*), *A*)
apply (*simp add*: *le-list-def less-list-def*)
by (*metis append.simps*(*2*))
have *th3preEmpty*: *aa*∉*A*∧*a*∉ *A*⟹*s setinterleaves* ((*a* # *list*, *u*), *A*)∧*t1* < *a* # *list*∧*length t1*=*0*
⟹[]<*s*∧[]≤*u*∧ [] *setinterleaves* ((*t1*, []), *A*)
apply (*simp add*: *le-list-def less-list-def*) **using** *c* **by** *auto*
from *th3NoEmpty th3preEmpty* **have** *th3*: *aa*∉*A*∧*a*∉ *A*⟹*s setinterleaves* ((*a* # *list*, *u*), *A*)∧*t1* <
a # *list*⟹∃ *u1*≤*u*. ∃ *s1*<*s*. *s1 setinterleaves* ((*t1*, *u1*), *A*)
by *blast*
show *?thesis*
using *b c th0 th1 th2 th3* **by** *auto*
qed
qed
qed

lemma *mono-Sync2a*:
r∈*min-elems*(𝒟 (*P* ⟦*A*⟧ *S*))
⟹ ∃ *t u*. *r setinterleaves* ((*t*, *u*),*insert tick* (*ev* ‘ *A*)) ∧
(*t* ∈ *min-elems*(𝒟 *P*) ∧ *u* ∈ 𝒯 *S* ∨ *t* ∈ *min-elems*(𝒟 *S*) ∧
u ∈ 𝒯 *P* ∧ *u* ∈ (𝒯 *P* − (𝒟 *P*−*min-elems*(𝒟 *P*))))
proof−
fix *r*
assume *A*: *r* ∈ *min-elems*(𝒟 (*P* ⟦*A*⟧ *S*))
thus ∃ *t u*. *r setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *A*))∧(*t* ∈ *min-elems*(𝒟 *P*) ∧
u ∈ 𝒯 *S* ∨ *t* ∈ *min-elems*(𝒟 *S*) ∧ *u* ∈ 𝒯 *P* ∧ *u* ∈ (𝒯 *P* − (𝒟 *P*−*min-elems*(𝒟 *P*))))
proof(*cases* ∀ *t u r1*. *r1*≤*r*∧ *r1 setinterleaves*((*t*, *u*),*insert tick* (*ev* ‘ *A*))
⟶ (*t* ∉ *min-elems*(𝒟 *P*)) ∧ (*t* ∉ *min-elems*(𝒟 *S*)))
case *True*
from *A* **have** *AA*: *r* ∈ 𝒟 (*P* ⟦*A*⟧ *S*)
using *elem-min-elems* **by** *blast*
from *AA* **obtain** *t1 u1* **where** *A1*:
∃ *r1*≤*r*. *r1 setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *A*)) ∧
(*t1* ∈ 𝒟 *P* ∨ *t1* ∈ 𝒟 *S*) **apply**(*simp add*: *D-Sync*)
using *le-list-def* **by** *blast*
from *True A1* **have** *A2*: (*t1* ∉ *min-elems*(𝒟 *P*)) ∧ (*t1* ∉ *min-elems*(𝒟 *S*))
by *blast*
from *A1 A2 min-elems5* **obtain** *tm1 tm2*
where *tmA*: (*t1*∈𝒟 *P* ⟶
(*tm1*≤*t1*∧*tm1*∈ *min-elems*(𝒟 *P*))) ∧ (*t1*∈𝒟 *S*⟶(*tm2*≤*t1*∧*tm2*∈ *min-elems*(𝒟 *S*)))
by *metis*
from *A2 tmA* **have** *AB1*: (*t1*∈𝒟 *P*⟶*tm1*<*t1*)∧(*t1*∈𝒟 *S*⟶*tm2*<*t1*)
by (*metis A1 order.not-eq-order-implies-strict*)
from *A1 AB1* **obtain** *um1 rm1 um2 rm2* **where** *AB2*: (*t1*∈𝒟 *P*⟶*um1*≤*u1*

```
∧ rm1 setinterleaves
    ((tm1, um1),insert tick (ev ‘ A)))∧(t1∈𝒟 S⟶um2≤u1 ∧ rm2 setinterleaves
    ((tm2, um2), insert tick (ev ‘ A))) by (meson interPrefix)
  from A1 A2 True tmA AB1 have A3: (t1∈𝒟 P⟶rm1<r∧rm1∈𝒟(P⟦A⟧S))∧(t1∈𝒟
S⟶rm2<r∧rm2∈ 𝒟(P⟦A⟧S))
      by (meson dual-order.strict-implies-order interPrefix order-trans)
    from A3 AA have ATrue: r ∉ min-elems(𝒟 (P ⟦A⟧ S)) using min-elems-def

      using A1 by blast
    with A ATrue show ?thesis
      by blast
  next
    case False
    from A have dsync: r∈𝒟 (P ⟦A⟧ S)
      by (simp add: local.A elem-min-elems)
    from dsync obtain r1 t2 u2
      where C: r1≤r∧ r1 setinterleaves ((t2, u2), insert tick (ev ‘ A)) ∧
              ((t2 ∈ 𝒟 P ∧ u2 ∈ 𝒯 S) ∨ (t2 ∈ 𝒟 S ∧ u2 ∈ 𝒯 P))
              apply(simp add: D-Sync)
              using le-list-def by blast
    from C have r1D: r1 ∈ 𝒟 (P ⟦A⟧ S) apply(simp add: D-Sync)
      using front-tickFree-Nil by blast
    from A C r1D have eq: r1=r apply(simp add: min-elems-def)
      using le-neq-trans by blast
    from A False have minAa: (t2 ∈ 𝒟 P∧u2∈𝒯 S)⟶(t2 ∈ min-elems(𝒟 P))
    proof(cases t2 ∈ 𝒟 P ∧u2 ∈ 𝒯 S∧ t2 ∉ min-elems(𝒟 P))
      case True
      from True obtain t3 where inA: t3<t2∧t3∈min-elems(𝒟 P)
        by (metis le-imp-less-or-eq min-elems5)
     from inA obtain r3 u3 where inA1: u3≤u2∧r3 setinterleaves((t3,u3),insert
tick(ev ‘ A))∧r3<r
        using C eq interPrefix by blast
      from inA1 have inA3: u3 ∈ 𝒯 S
        using True is-processT3-ST-pref by blast
      from inA1 have inA2: r3 ∈ 𝒟(P ⟦A⟧ S) apply (simp add: D-Sync)
        using elem-min-elems front-tickFree-Nil inA inA3 by blast
      with A eq inA1 show ?thesis
        by(simp add: min-elems-def)
    next
      case False
      then show ?thesis
        by blast
    qed
    from A False have minAb1: (t2 ∈ 𝒟 S∧u2∈ 𝒯 P)⟶(t2 ∈ min-elems(𝒟 S))

    proof(cases t2 ∈ 𝒟 S ∧ u2 ∈ 𝒯 P∧ t2 ∉ min-elems(𝒟 S))
      case True
      from True obtain t3 r3 u3 where inA: t3<t2∧t3∈min-elems(𝒟 S)
      and inA1: u3≤u2∧r3 setinterleaves ((t3, u3), insert tick (ev ‘ A))∧r3<r
```

```
      by (metis C eq interPrefix le-less min-elems5)
    from inA1 have inA3: u3 ∈ 𝒯 P
      using True is-processT3-ST-pref by blast
    from inA1 have inA2: r3 ∈ 𝒟 (P ⟦A⟧ S) apply (simp add: D-Sync)
      using elem-min-elems front-tickFree-Nil inA inA3 by blast
    with A eq inA1 show ?thesis
      by(simp add: min-elems-def)
  next
    case False
    then show ?thesis
      by blast
  qed
    from A False have minAb2: (t2 ∈ 𝒟 S∧u2∈ 𝒯 P)⟶(u2 ∈ (𝒯 P−(𝒟
P−min-elems(𝒟 P))))
  proof(cases t2 ∈ 𝒟 S ∧u2 ∈ 𝒯 P∧ u2 ∈(𝒟 P−min-elems(𝒟 P)))
    case True
    from True have inAAA: u2∈𝒟 P∧u2∉min-elems(𝒟 P)
      by blast
    from inAAA obtain u3 where inA: u3<u2∧u3∈min-elems(𝒟 P)
      by (metis le-neq-trans min-elems5)
    from inA obtain t3 r3 where inA1: t3≤t2∧r3 setinterleaves((t3,u3),insert
tick(ev ‘ A))∧r3<r
      using C Sync.sym eq interPrefix by blast
    from inA1 have inA3: u3 ∈ 𝒟 P∧t3 ∈ 𝒯 S
      using D-T True elem-min-elems inA is-processT3-ST-pref by blast
    from inA1 have inA2: r3 ∈ 𝒟 (P ⟦A⟧ S) apply (simp add: D-Sync)
      using Sync.sym front-tickFree-Nil inA3 by blast
    with A inA1 inA2 show ?thesis
      by(simp add: min-elems-def)
  next
    case False
    then show ?thesis
      by blast
  qed
  with A C show ?thesis
    using eq minAa minAb1 by blast
 qed
qed

lemma mono-Sync-D3:
 assumes ordered: P ⊑ Q
 shows min-elems (𝒟 (P ⟦A⟧ S)) ⊆ 𝒯(Q ⟦A⟧ S)
proof−
 have mono-sync2b:P ⊑ Q ⟹ ∀ r. r ∈ min-elems(𝒟 (P⟦A⟧S))⟶r ∈ 𝒯(Q⟦A⟧S)

  apply(frule impI)
  apply(auto simp: mono-Sync2a)
 proof −
  fix r
```

assume A: $P \sqsubseteq Q$
and B: $r \in$ *min-elems*($\mathcal{D}(P[\![A]\!]S)$)
from B **obtain** t u
where E: r *setinterleaves* ((t, u), *insert tick* (*ev* ‘ A)) $\wedge$ ($t \in$ *min-elems*($\mathcal{D}$ P) $\wedge$
$u \in \mathcal{T}\ S \vee t \in$ *min-elems*($\mathcal{D}\ S$) $\wedge\ u \in (\mathcal{T}\ P - (\mathcal{D}\ P -$ *min-elems*($\mathcal{D}$ P))))
using *mono-Sync2a* **by** *blast*
from E **have** F:($t \in \mathcal{T}\ Q \wedge u \in \mathcal{T}\ S) \vee (t \in \mathcal{T}\ S \wedge u \in \mathcal{T}\ Q)$
using *le-approx2T le-approx3 ordered* **by** *blast*
thus $r \in \mathcal{T}\ (Q\ [\![A]\!]\ S)$
by (*metis* (*no-types*, *lifting*) *E Sync.sym T-Sync UnCI Un-commute insert-is-Un mem-Collect-eq*)
qed
show *?thesis*
apply(*insert ordered*)
apply(*frule mono-sync2b*)
by *blast*
qed

lemma *mono-D-Sync*: $\mathcal{D}\ (S\ [\![\ A\ ]\!]\ Q) = \mathcal{D}\ (Q\ [\![\ A\ ]\!]\ S)$
by(*auto simp*: *D-Sync*)

lemma *mono-T-Sync*: $\mathcal{T}\ (S\ [\![\ A\ ]\!]\ Q) = \mathcal{T}\ (Q\ [\![\ A\ ]\!]\ S)$
apply(*auto simp*: *T-Sync*)
using *Sync.sym* **by** *fastforce+*

lemma *mono-F-Sync*: $\mathcal{F}\ (S\ [\![\ A\ ]\!]\ Q) = \mathcal{F}\ (Q\ [\![\ A\ ]\!]\ S)$
apply *auto*
apply (*simp add*: *F-Sync*)
using *Sync.sym* **apply** *blast*
apply (*simp add*: *F-Sync*)
using *Sync.sym* **by** *blast*

lemma *mono-Ra-Sync*: $Ra\ (S\ [\![\ A\ ]\!]\ Q)\ s = Ra\ (Q\ [\![\ A\ ]\!]\ S)\ s$
apply *auto*
apply (*auto simp*: *Ra-def*)
by (*auto simp*: *mono-F-Sync*)

lemma *mono-Sync*[*simp*] : $P \sqsubseteq Q \Longrightarrow (P\ [\![\ A\ ]\!]\ S) \sqsubseteq (Q\ [\![\ A\ ]\!]\ S)$
by(*auto simp*: *le-approx-def mono-Sync-D1 mono-Sync-D2 mono-Sync-D3*)

lemma *mono-Sync-sym* [*simp*]: $P \sqsubseteq Q \Longrightarrow (S\ [\![\ A\ ]\!]\ P) \sqsubseteq (S\ [\![\ A\ ]\!]\ Q)$
by (*metis Process-eq-spec mono-D-Sync mono-F-Sync mono-Sync*)

lemma *chain-Sync1*: *chain* $Y \Longrightarrow$ *chain* ($\lambda i.\ Y\ i\ [\![\ A\ ]\!]\ S$)
by(*simp add*: *chain-def*)

```
lemma chain-Sync2: chain Y ⟹ chain (λi. S ⟦ A ⟧ Y i)
  by(simp add: chain-def)

lemma empty-setinterleaving : [] setinterleaves ((t, u), A) ⟹ t = []
  by (cases t, cases u, auto, cases u, simp-all split:if-splits)

lemma inters-fin-fund: finite{(t, u). s setinterleaves ((t, u), A)}
proof (induction length s arbitrary:s rule:nat-less-induct)
  case 1
  have A:{(t, u). s setinterleaves((t, u), A)} ⊆ {([],[])} ∪
                {(t, u). s setinterleaves ((t, u), A) ∧
                        (∃ a list. t = a#list ∧ a ∉ A) ∧ u = []} ∪
                {(t, u). s setinterleaves ((t, u), A) ∧
                        (∃ a list. u = a#list ∧ a ∉ A) ∧ t = []} ∪
                {(t, u). s setinterleaves ((t, u), A) ∧
                        (∃ a list aa lista. u = a#list ∧ t = aa#lista)}
          (is ?A ⊆ {([],[])} ∪ ?B ∪ ?C ∪ ?D)
    apply (rule subsetI, safe)
        apply(simp-all add: neq-Nil-conv)
       apply (metis Sync.si-empty2 Sync.sym empty-iff list.exhaust-sel)
      using list.exhaust-sel apply blast
     apply (metis Sync.sym emptyLeftNonSync list.exhaust-sel list.set-intros(1))

      using list.exhaust-sel apply blast
     apply (metis emptyLeftNonSync list.exhaust-sel list.set-intros(1))
    by (metis Sync.si-empty2 Sync.sym empty-iff list.exhaust-sel)
  have a1:?B ⊆ { ((hd s#list), [])| list. (tl s) setinterleaves ((list, []), A) ∧ (hd s)
∉ A}
  (is ?B ⊆ ?B1) by auto
  define f where a2:f = (λa (t, (u::'a event list)). ((a::'a event)#t, ([]::'a event
list)))
  have a3:?B1 ⊆ {(hd s # list, []) |list. tl s setinterleaves ((list, []), A)} (is ?B1
⊆ ?B2)
    by blast
  from a1 a3 have a13:?B ⊆ ?B2
    by simp
  have AA: finite ?B
  proof (cases s)
    case Nil
    then show ?thesis
      using not-finite-existsD by fastforce
  next
    case (Cons a list)
     hence aa:finite{(t,u).(tl s) setinterleaves((t, u), A)}using 1[THEN spec, of
length (tl s)]
      by (simp)
    have {(hd s#list, [])|list. tl s setinterleaves((list,[]),A)}⊆(λ(t, u).f(hd s)(t, u))
'{(t, u).
    tl s setinterleaves ((t, u), A)} using a2 by auto
```

hence *finite ?B2* **using** *finite-imageI* [*of* {(*t*, *u*). (*tl s*) *setinterleaves* ((*t*, *u*), *A*)}
λ(*t*, *u*). *f* (*hd s*) (*t*, *u*), *OF aa*] **using** *rev-finite-subset* **by** *auto*
then show *?thesis* **using** *a13*
by (*meson rev-finite-subset*)
qed
have *a1*:*?C* ⊆ { ([],(*hd s*#*list*))| *list*. (*tl s*) *setinterleaves* (([],*list*), *A*) ∧ (*hd s*) ∉ *A*}
(**is** *?C* ⊆ *?C1*) **by** *auto*
define *f* **where** *a2*:*f* = (λ*a* ((*t*::'*a event list*), *u*). (([]::'*a event list*), (*a*::'*a event*)#*u*))
have *a3*:*?C1* ⊆ {([],*hd s* # *list*) |*list*. *tl s setinterleaves* (([],*list*), *A*)} (**is** *?C1* ⊆ *?C2*)
by *blast*
from *a1 a3* **have** *a13*:*?C* ⊆ *?C2*
by *simp*
have *AAA*:*finite ?C*
proof (*cases s*)
case *Nil*
then show *?thesis*
using *not-finite-existsD* **by** *fastforce*
next
case (*Cons a list*)
hence *aa*:*finite* {(*t*,*u*).(*tl s*)*setinterleaves*((*t*, *u*), *A*)}**using** *1*[*THEN spec*, *of length* (*tl s*)]
by (*simp*)
have {([], *hd s* # *list*) |*list*. *tl s setinterleaves* (([], *list*), *A*)} ⊆ (λ(*t*, *u*).
f (*hd s*) (*t*, *u*)) ‘ {(*t*, *u*). *tl s setinterleaves* ((*t*, *u*), *A*)} **using** *a2* **by** *auto*
hence *finite ?C2* **using** *finite-imageI* [*of* {(*t*, *u*). (*tl s*) *setinterleaves* ((*t*, *u*), *A*)}
λ(*t*, *u*). *f* (*hd s*) (*t*, *u*), *OF aa*] **using** *rev-finite-subset* **by** *auto*
then show *?thesis* **using** *a13*
by (*meson rev-finite-subset*)
qed
have *dd0*:*?D* ⊆ {(*a*#*l*, *aa*#*la*)|*a aa l la*. *s setinterleaves* ((*a*#*l*, *aa*#*la*), *A*)} (**is** *?D* ⊆ *?D1*)
apply (*rule subsetI*, *auto*, *simp split*:*if-splits*)
by (*blast*, *blast*, *metis*, *blast*)
have *AAAA*:*finite ?D*
proof (*cases s*)
case *Nil*
hence *?D* ⊆ {([],[])}
using *empty-setinterleaving* **by** *auto*
then show *?thesis*
using *infinite-super* **by** *auto*
next
case (*Cons a list*)
hence *dd1*:*?D1* ⊆ {(*a*#*l*, *u*)| *l u*. *list setinterleaves* ((*l*, *u*), *A*)}
∪ {(*t*, *a*#*la*)| *t la*. *list setinterleaves* ((*t*, *la*), *A*)}

$\cup$ {(a#l, a#la)|l la. list setinterleaves ((l, la), A)}(**is** ?D1 $\subseteq$?D2 $\cup$?D3 $\cup$?D4)
by (rule-tac subsetI, auto split:if-splits)
with Cons **have** aa:finite {(t,u). list setinterleaves ((t, u), A)}
using 1[THEN spec, of length list] **by** (simp)
define f **where** a2:f = (λ (t, (u::'a event list)). ((a::'a event)# t, u))
with Cons **have** ?D2 $\subseteq$ (f ' {(t, u). list setinterleaves ((t, u), A)})
using a2 **by** auto
hence dd2:finite ?D2
using finite-imageI [of {(t, u). list setinterleaves ((t, u), A)} f, OF aa]
by (meson rev-finite-subset)
define f **where** a2:f = (λ ((t::'a event list),u). (t,(a::'a event)#u))
with Cons **have** ?D3 $\subseteq$ (f ' {(t, u). list setinterleaves ((t, u), A)})
using a2 **by** auto
hence dd3:finite ?D3
using finite-imageI [of {(t, u). list setinterleaves ((t, u), A)} f, OF aa]
by (meson rev-finite-subset)
define f **where** a2:f = (λ (t,u). ((a::'a event)#t,(a::'a event)#u))
with Cons **have** ?D4 $\subseteq$ (f ' {(t, u). list setinterleaves ((t, u), A)})
using a2 **by** auto
hence dd4:finite ?D4
using finite-imageI [of {(t, u). list setinterleaves ((t, u), A)} f, OF aa]
by (meson rev-finite-subset)
with dd0 dd1 dd2 dd3 dd4 **show** ?thesis
by (simp add: finite-subset)
qed
from A AA AAA AAAA **show** ?case
by (simp add: finite-subset)
qed

lemma inters-fin: finite{(t, u, r). r setinterleaves ((t, u), insert tick (ev ' A)) $\wedge$ ($\exists$ v. x = r @ v $\wedge$ front-tickFree v $\wedge$ (tickFree r $\vee$ v = []))}
(**is** finite ?A)
proof −
have a:?A$\subseteq$($\bigcup$ r $\in$ {r. r $\leq$ x}. {a. $\exists$ t u. a=(t,u,r)$\wedge$r setinterleaves((t,u),insert tick(ev ' A))})
(**is** ?A $\subseteq$ ($\bigcup$ r $\in$ {r. r $\leq$ x}. ?P r))
using le-list-def **by** fastforce
have b:$\forall$ (r::'a event list). finite (?P r)
proof(rule allI)
fix r::'a event list
define f **where** f= (λ((t::'a event list), (u::'a event list)). (t,u,r))
hence ?P r $\subseteq$ (f ' {(t, u). r setinterleaves ((t, u), insert tick (ev ' A))})
by auto
then show finite (?P r)
using inters-fin-fund[of r insert tick (ev ' A)] finite-imageI[of {(t, u). r setinterleaves
((t, u), insert tick (ev ' A))} f] **by** (meson rev-finite-subset)

qed
have $\{t.\ \exists\, t2.\ x = t\ @\ t2\} = \{r.\ \exists\, ra.\ r\ @\ ra = x\}$
by *blast*
hence *finite* $\{r.\ r \leq x\}$ **using** *prefixes-fin*[*of x*, *simplified Let-def*, *THEN conjunct1*]
by (*auto simp add*:*le-list-def*)
with *a b* **show** *?thesis*
by (*meson finite-UN-I infinite-super*)
qed

lemma *limproc-Sync-D*:
chain Y $\Longrightarrow \mathcal{D}$ (*lim-proc*(*range Y*)$[\![A]\!]$ *S*) = $\mathcal{D}$(*lim-proc*(*range*($\lambda i.$ *Y i* $[\![A]\!]$ *S*)))
apply(*auto simp*:*Process-eq-spec F-Sync D-Sync T-Sync F-LUB D-LUB T-LUB chain-Sync1*)
apply *blast*
apply *blast*
using *front-tickFree-Nil* **apply** *blast*
using *front-tickFree-Nil* **apply** *blast*
proof −
fix *x*
assume *A*: *chain Y*
and *B*: $\forall\, xa.\ \exists\, t\ u\ r\ v.$ *front-tickFree v* $\wedge$ (*tickFree r* $\vee$ *v* = []) $\wedge$
$x = r\ @\ v \wedge r$ *setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *A*)) $\wedge$
($t \in \mathcal{D}\ (Y\ xa) \wedge u \in \mathcal{T}\ S \vee t \in \mathcal{D}\ S \wedge u \in \mathcal{T}\ (Y\ xa)$)
thus $\exists\, t\ u\ r\ v.$ *front-tickFree v* $\wedge$ (*tickFree r* $\vee$ *v* = []) $\wedge$ $x = r\ @\ v \wedge$
r setinterleaves ((*t*, *u*), *insert tick* (*ev* ‘ *A*)) $\wedge$
($(\forall\, x.\ t \in \mathcal{D}\ (Y\ x)) \wedge u \in \mathcal{T}\ S \vee t \in \mathcal{D}\ S \wedge (\forall\, x.\ u \in \mathcal{T}\ (Y\ x))$)
proof(*cases* $\forall\, t\ u\ r.$ *r setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *A*)) $\longrightarrow$
($r \leq x \longrightarrow ((\exists\, x.\ t \notin \mathcal{D}\ (Y\ x)) \vee u \notin \mathcal{T}\ S) \wedge$
($t \in \mathcal{D}\ S \longrightarrow (\exists\, x.\ u \notin \mathcal{T}\ (Y\ x))$)))
case *True*
from *A* **obtain** *tryunion* **where** *Ctryy*:
tryunion=$\{(t,\ u,\ r).$ *r setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *A*)) $\wedge$
($\exists\, v.(x = r\ @\ v \wedge$ *front-tickFree v* $\wedge$ (*tickFree r* $\vee$ *v* = []))$\}$
by *simp*
from *Ctryy inters-fin* **have** *tolfin*: *finite tryunion*
by *auto*
obtain *tryunion1*
where *Ctryy1*: *tryunion1* = $\{n.\ \exists\, (t,u,\ r) \in$ *tryunion*.
($t \in \mathcal{D}\ (Y\ n) \wedge u \in \mathcal{T}\ S \vee t \in \mathcal{D}\ S \wedge u \in \mathcal{T}\ (Y\ n)$)$\}$ **by** *simp*
from *B Ctryy1 Ctryy* **have** *Ctryy3*: $\forall\, n.\ n \in$ *tryunion1*
by *blast*
from *Ctryy3* **have** *Ctryy2*: *infinite tryunion1*
using *finite-nat-set-iff-bounded* **by** *auto*
from *Ctryy2 Ctryy1 tolfin* **obtain** *r2 t2 u2*
where *Ee*: (*t2*,*u2*, *r2*) $\in$ *tryunion* $\wedge$

infinite $\{n.\ (t2 \in \mathcal{D}\ (Y\ n) \wedge u2 \in \mathcal{T}\ S \vee t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ n))\}$
by *auto*
from *True Ee Ctryy* **obtain** *x1 x2*
where *Ee1*: $((t2 \notin \mathcal{D}\ (Y\ x1)) \vee u2 \notin \mathcal{T}\ S) \wedge (t2 \in \mathcal{D}\ S \longrightarrow (u2 \notin \mathcal{T}\ (Y\ x2)))$
apply (*simp add*: *le-list-def*) **by** *blast*
from *Ee Ee1 Ctryy* **obtain** *x3*
where *Ee2*: $(x1 \leq x3) \wedge (x2 \leq x3) \wedge (t2 \in \mathcal{D}\ (Y\ x3) \wedge u2 \in \mathcal{T}\ S \vee t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ x3))$
apply(*simp add*: *infinite-nat-iff-unbounded-le*)
proof −
assume *a1*: *r2 setinterleaves* $((t2,\ u2)$, *insert tick* $(ev\ `\ A)) \wedge$
$(\exists\, v.\ x = r2\ @\ v \wedge$ *front-tickFree* $v \wedge$ (*tickFree* $r2 \vee v = []$)) $\wedge$
$((\forall\, m.\ \exists\, n \geq m.\ t2 \in \mathcal{D}\ (Y\ n) \wedge u2 \in \mathcal{T}\ S) \vee$
$(\forall\, m.\ \exists\, n \geq m.\ t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ n)))$
assume *a2*: $\bigwedge x3.\ x1 \leq x3 \wedge x2 \leq x3 \wedge (t2 \in \mathcal{D}\ (Y\ x3) \wedge u2 \in \mathcal{T}\ S \vee t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ x3))$
$\Longrightarrow$ *thesis*
obtain *nn* :: $nat \Rightarrow nat$
where *f3*: $\forall\, x0.\ (\exists\, v1 \geq x0.\ t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ v1)) =$
$(x0 \leq nn\ x0 \wedge t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ (nn\ x0)))$ **by** *moura*
obtain *nna* :: $nat \Rightarrow nat$
where $\forall\, x0.\ (\exists\, v1 \geq x0.\ t2 \in \mathcal{D}\ (Y\ v1) \wedge u2 \in \mathcal{T}\ S) =$
$(x0 \leq nna\ x0 \wedge t2 \in \mathcal{D}\ (Y\ (nna\ x0)) \wedge u2 \in \mathcal{T}\ S)$ **by** *moura*
then have *f4*: $(\forall\, n.\ n \leq nna\ n \wedge t2 \in \mathcal{D}\ (Y\ (nna\ n)) \wedge u2 \in \mathcal{T}\ S) \vee$
$(\forall\, n.\ n \leq nn\ n \wedge t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ (nn\ n)))$ **using** *f3 a1* **by** *presburger*
have $(\exists\, n.\ \neg\ n \leq nn\ n \vee t2 \notin \mathcal{D}\ S \vee u2 \notin \mathcal{T}\ (Y\ (nn\ n))) \vee$
$(\exists\, n \geq x1.\ x2 \leq n \wedge (t2 \in \mathcal{D}\ (Y\ n) \wedge$
$u2 \in \mathcal{T}\ S \vee t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ n)))$ **by** (*metis le-cases order.trans*)
moreover
{ assume $\exists\, n.\ \neg\ n \leq nn\ n \vee t2 \notin \mathcal{D}\ S \vee u2 \notin \mathcal{T}\ (Y\ (nn\ n))$
then have $\forall\, n.\ n \leq nna\ n \wedge t2 \in \mathcal{D}\ (Y\ (nna\ n)) \wedge u2 \in \mathcal{T}\ S$
using *f4* **by** *blast*
then have $\exists\, n \geq x1.\ x2 \leq n \wedge (t2 \in \mathcal{D}\ (Y\ n) \wedge u2 \in \mathcal{T}\ S \vee t2 \in \mathcal{D}\ S \wedge u2 \in \mathcal{T}\ (Y\ n))$
by (*metis* (*no-types*) *le-cases order.trans*)
}
ultimately show *?thesis*
using *a2* **by** *blast*
qed
from *A* **have** *Abb1*: $\forall\, n1\ n2.\ n1 \leq n2 \longrightarrow Y\ n1 \sqsubseteq Y\ n2$
by (*simp add*: *po-class.chain-mono*)
from *A Abb1* **have** *Abb2*: $\forall\, n1\ n2.\ n1 > n2 \wedge t \notin \mathcal{D}\ (Y\ n2) \longrightarrow t \notin \mathcal{D}\ (Y\ n1)$
using *le-approx1 less-imp-le-nat* **by** *blast*
from *A Abb1* **have** *Abb3*: $\forall\, n1\ n2.\ n1 > n2 \wedge t \notin \mathcal{T}\ (Y\ n2) \longrightarrow t \notin \mathcal{T}\ (Y$

n1)
by (*meson NT-ND le-approx2T less-imp-le-nat*)
from *Abb2 Ee2* **have** *Ab1*: $t2 \notin \mathcal{D}\ (Y\ x1) \longrightarrow t2 \notin \mathcal{D}\ (Y\ x3)$
by (*meson Abb1 le-approx1 less-imp-le-nat subsetCE*)
from *Abb3* **have** *Ab2*: $u2 \notin \mathcal{T}\ (Y\ x2) \longrightarrow u2 \notin \mathcal{T}\ (Y\ x3)$
by (*meson Abb1 Ee2 NT-ND le-approx2T less-imp-le-nat*)
from *True Ee Ee1 Ee2 Ab1 Ab2* **show** *?thesis*
by *blast*
next
case *False*
from *A B False* **obtain** *t u r*
where *Bb1*: r *setinterleaves* $((t,\ u),$ *insert tick* $(ev\ `\ A)) \wedge r \leq x \wedge$
$((\forall x.\ t \in \mathcal{D}\ (Y\ x)) \wedge u \in \mathcal{T}\ S \vee t \in \mathcal{D}\ S \wedge (\forall x.\ u \in \mathcal{T}\ (Y\ x)))$
by *auto*
from *B* **have** *Bb21*: *front-tickFree x*
by (*metis D-imp-front-tickFree append-Nil2 front-tickFree-append ftf-Sync is-processT2-TR*)
from *B Bb1* **have** *Bb2*: $\exists v.$ *front-tickFree* $v \wedge$ (*tickFree* $r \vee v = []) \wedge x = r$ @ v
by (*metis Bb21 front-tickFree-Nil front-tickFree-mono le-list-def*)
then show *?thesis*
using *Bb1* **by** *blast*
qed
qed

lemma *limproc-Sync-F-annex1*:
chain Y
$\Longrightarrow \forall n.\ (a,\ b) \in \mathcal{F}\ (Y\ n\ [\![A]\!]\ S)$
$\Longrightarrow \exists x.\ a \notin \mathcal{D}\ (Y\ x\ [\![A]\!]\ S)$
$\Longrightarrow \exists t\ u\ X.\ (\forall x.\ (t,\ X) \in \mathcal{F}\ (Y\ x)) \wedge$
$(\exists Y.\ (u,\ Y) \in \mathcal{F}\ S \wedge a$ *setinterleaves*$((t,\ u),$*insert tick* $(ev\ `\ A)) \wedge$
$b = (X \cup Y) \cap$ *insert tick* $(ev\ `\ A) \cup X \cap Y)$
proof −
fix *a b*
assume *A*: *chain Y*
assume *B*: $\forall n.\ (a,\ b) \in \mathcal{F}\ (Y\ n\ [\![A]\!]\ S)$
assume *C*: $\exists x.\ a \notin \mathcal{D}\ (Y\ x\ [\![A]\!]\ S)$
thus $\exists t\ u\ X.\ (\forall x.\ (t,\ X) \in \mathcal{F}\ (Y\ x)) \wedge$
$(\exists Y.\ (u,\ Y) \in \mathcal{F}\ S \wedge$
a *setinterleaves* $((t,\ u),$ *insert tick* $(ev\ `\ A)) \wedge$
$b = (X \cup Y) \cap$ *insert tick* $(ev\ `\ A) \cup X \cap Y)$
proof−
from *B C* **obtain** *x1* **where** *D*: $a \notin \mathcal{D}\ (Y\ x1\ [\![A]\!]\ S)$
by *blast*
from *B D* **obtain** *t1 u1 X1 Y1* **where** *E*:
a *setinterleaves* $((t1,\ u1),$ *insert tick* $(ev\ `\ A)) \wedge$
$(t1,\ X1) \in \mathcal{F}\ (Y\ x1) \wedge (u1,\ Y1) \in \mathcal{F}\ S \wedge$
$b = (X1 \cup Y1) \cap$ *insert tick* $(ev\ `\ A) \cup X1 \cap Y1$

apply(*auto simp*: *D-Sync F-Sync*)
by *fastforce*
from *B D E* **have** *F*: $t1 \notin \mathcal{D}\ (Y\ x1) \wedge t1 \in \mathcal{T}(Y\ x1)$ **apply**(*auto simp*: *D-Sync*)

using *F-T front-tickFree-Nil* **apply** *blast*
by (*simp add*: *F-T*)
from *F* **have** *ee*: $\forall i.\ (t1,\ X1) \in \mathcal{F}\ (Y\ i) \wedge (u1,\ Y1) \in \mathcal{F}\ S \wedge\ a$ *setinterleaves* $((t1,\ u1)$, *insert tick* $(ev\ `\ A)) \wedge\ b = (X1 \cup Y1) \cap$ *insert tick* $(ev\ `\ A) \cup X1 \cap Y1$
using *E chain-lemma is-processT8-S le-approx2 local.A* **by** *metis*
with *A B C D E F ee* **show** *?thesis*
by *blast*
qed
qed

lemma *limproc-Sync-F-annex2*:
chain Y
$\Longrightarrow \forall t\ u\ r.\ r$ *setinterleaves* $((t,\ u)$, *insert tick* $(ev\ `\ A)) \longrightarrow$
$(\forall v.\ a = r\ @\ v \longrightarrow$ *front-tickFree* $v \longrightarrow$
$\neg$ *tickFree* $r \wedge v \neq [] \vee ((\exists x.\ t \notin \mathcal{D}\ (Y\ x)) \vee u \notin \mathcal{T}\ S) \wedge$
$(t \in \mathcal{D}\ S \longrightarrow (\exists x.\ u \notin \mathcal{T}\ (Y\ x))))$
$\Longrightarrow \exists x.\ a \notin \mathcal{D}(Y\ x\ [\![\ A\ ]\!]\ S)$
apply(*auto simp*: *D-Sync*)
proof−
fix *a*
assume *A*: $\forall t\ u\ r.\ r$ *setinterleaves* $((t,\ u)$, *insert tick* $(ev\ `\ A)) \longrightarrow$
$(\forall v.\ a = r\ @\ v \longrightarrow$ *front-tickFree* $v \longrightarrow$
$\neg$ *tickFree* $r \wedge v \neq [] \vee$
$((\exists x.\ t \notin \mathcal{D}\ (Y\ x)) \vee u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow (\exists x.\ u \notin \mathcal{T}\ (Y\ x))))$
assume *B*: *chain Y*
thus $\exists x.\ \forall t\ u\ r.\ r$ *setinterleaves* $((t,\ u)$, *insert tick* $(ev\ `\ A)) \longrightarrow$
$(\forall v.\ a = r\ @\ v \longrightarrow$ *front-tickFree* $v \longrightarrow$
$\neg$ *tickFree* $r \wedge v \neq [] \vee (t \in \mathcal{D}\ (Y\ x) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ x)))$
proof−
from *B* **have** *D*: $((\exists x.\ t \notin \mathcal{D}\ (Y\ x)) \vee u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow (\exists x.\ u \notin \mathcal{T}\ (Y\ x)))$
$\Longrightarrow (\exists x.\ ((t \in \mathcal{D}\ (Y\ x) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ x))))$
by (*meson D-T chain-lemma le-approx1 le-approx2T subsetCE*)
from *A* **obtain** *tryunion*
where *Ctryy*: *tryunion*$=\{(t,\ u,\ r).\ r$ *setinterleaves* $((t,\ u)$, *insert tick* $(ev\ `\ A)) \wedge$
$(\exists v.\ a = r\ @\ v \wedge$ *front-tickFree* $v \wedge$ (*tickFree* $r \vee v = []))\}$
by *simp*
from *Ctryy* **have** *tolfin*: *finite tryunion*
using *inters-fin* **by** *auto*
from *B* **have** *Abb1*: $\forall n1\ n2.\ n1 \leq n2 \longrightarrow Y\ n1 \sqsubseteq Y\ n2$

by (*simp add*: *po-class.chain-mono*)
from *A Abb1* **have** *Abb2*: $\forall$ *n1 n2 t. n1>n2*$\wedge t \notin \mathcal{D}$ (*Y n2*)$\longrightarrow t \notin \mathcal{D}$ (*Y n1*)
using *le-approx1 less-imp-le-nat* **by** *blast*
from *A Abb1* **have** *Abb3*: $\forall$ *n1 n2 t. n1>n2*$\wedge t \notin \mathcal{T}$ (*Y n2*)$\longrightarrow t \notin \mathcal{T}$ (*Y n1*)
by (*meson NT-ND le-approx2T less-imp-le-nat*)
from *Abb2 Abb3*
have *oneIndexPre*: $\forall$ *t u.* (($\exists$ *x. t* $\notin \mathcal{D}$ (*Y x*)) $\vee$ *u* $\notin \mathcal{T}$ *S*) $\wedge$ (*t* $\in \mathcal{D}$ *S* $\longrightarrow$
($\exists$ *x. u* $\notin \mathcal{T}$ (*Y x*))) $\longrightarrow$
($\exists$ *x.* (*t* $\notin \mathcal{D}$ (*Y x*) $\vee$ *u* $\notin \mathcal{T}$ *S*) $\wedge$ (*t* $\in \mathcal{D}$ *S* $\longrightarrow$ *u* $\notin \mathcal{T}$ (*Y x*)))
by (*meson B D-T ND-F-dir2′ chain-lemma le-approx1 subsetCE*)
from *A B oneIndexPre Abb2 Abb3*
have *oneIndex*: $\forall$ *t u r. r setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *A*)) $\longrightarrow$
($\forall$ *v. a* = *r* @ *v* $\longrightarrow$*front-tickFree v* $\longrightarrow$ $\neg$ *tickFree r*$\wedge$*v*$\neq$[] $\vee$
($\exists$ *x.*(((*t* $\notin \mathcal{D}$ (*Y x*)) $\vee$ *u* $\notin \mathcal{T}$ *S*) $\wedge$ (*t* $\in \mathcal{D}$ *S* $\longrightarrow$ (*u* $\notin \mathcal{T}$ (*Y x*)))))) **by** *blast*
define *proUnion1*
where *finiUnion*: *proUnion1* = {*n.* $\exists$ (*t*,*u*, *r*)$\in$*tryunion.* (((*t* $\in \mathcal{D}$ (*Y n*) $\longrightarrow$ *u* $\notin \mathcal{T}$ *S*) $\wedge$
(*t* $\in \mathcal{D}$ *S* $\longrightarrow$ *u* $\notin \mathcal{T}$ (*Y n*)))$\wedge$
($\forall$ *m.* ((*t* $\in \mathcal{D}$ (*Y m*)$\longrightarrow$*u* $\notin \mathcal{T}$ *S*) $\wedge$
(*t* $\in \mathcal{D}$ *S*$\longrightarrow$*u* $\notin \mathcal{T}$ (*Y m*)))$\longrightarrow$*n*$\leq$*m*))}
from *B*
have *finiSet1*: $\bigwedge$*r t u.* (*t*,*u*, *r*) $\in$ *tryunion*
$\Longrightarrow$ *finite* {*n1.*(((*t* $\in \mathcal{D}$ (*Y n1*)$\longrightarrow$*u* $\notin \mathcal{T}$ *S*) $\wedge$
(*t*$\in\mathcal{D}$ *S*$\longrightarrow$*u*$\notin\mathcal{T}$(*Y n1*)))$\wedge$($\forall$ *m1.* ((*t* $\in \mathcal{D}$ (*Y m1*) $\longrightarrow$
u $\notin \mathcal{T}$ *S*) $\wedge$ (*t* $\in \mathcal{D}$ *S* $\longrightarrow$ *u* $\notin \mathcal{T}$ (*Y m1*)))$\longrightarrow$*n1*$\leq$*m1*))}

by (*metis* (*no-types*, *lifting*) *infinite-nat-iff-unbounded mem-Collect-eq not-less*)
from *B tolfin finiUnion oneIndex finiSet1* **have** *finiSet*: *finite proUnion1*
by *auto*
from *finiSet* **obtain** *proMax* **where** *ann*: *proMax*=*Max*(*proUnion1*)
by *blast*
from *ann Abb2* **have** *ann1*: $\forall$ *num*$\in$*proUnion1.* $\forall$ *t. t* $\notin \mathcal{D}$ (*Y num*)$\longrightarrow t \notin \mathcal{D}$ (*Y proMax*)
by (*meson Abb1 Max-ge finiSet le-approx1 subsetCE*)
from *ann Abb3* **have** *ann2*: $\forall$ *num*$\in$*proUnion1.* $\forall$ *t. t* $\notin \mathcal{T}$ (*Y num*)$\longrightarrow t \notin \mathcal{T}$ (*Y proMax*)
by (*meson Abb1 D-T Max-ge finiSet le-approx2T*)
from *finiUnion*
have *ann3*: $\forall$ *num*$\in$*proUnion1.* $\exists$ *r t u. r setinterleaves*((*t*, *u*),*insert tick* (*ev* ‘ *A*)) $\longrightarrow$
($\forall$ *v. a* = *r* @ *v* $\longrightarrow$*front-tickFree v* $\longrightarrow$
$\neg$ *tickFree r* $\wedge$ *v* $\neq$ [] $\vee$((((*t*$\notin\mathcal{D}$(*Y num*))$\vee$*u* $\notin \mathcal{T}$ *S*) $\wedge$
(*t* $\in \mathcal{D}$ *S* $\longrightarrow$ (*u* $\notin \mathcal{T}$ (*Y num*)))))) **by** *auto*
obtain *proUnion2*
where *ann0*: *proUnion2* ={(*t*, *u*, *r*).$\exists$ *n.* (*t*, *u*, *r*) $\in$ *tryunion* $\wedge$
((*t* $\in \mathcal{D}$ (*Y n*)$\longrightarrow$*u* $\notin \mathcal{T}$ *S*) $\wedge$

$(t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ n)))\}$ **by** *auto*
from *ann0 Ctryy finiUnion*
have *ann1*: $\bigwedge t\ u\ r.\ (t,\ u,\ r) \in$ *proUnion2*
$\Longrightarrow \exists$ *num. num*$\in$*proUnion1* $\wedge$ $((t \in \mathcal{D}\ (Y\ num) \longrightarrow u \notin \mathcal{T}\ S)$
$\wedge$
$(t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ num)))$
proof−
fix *t u r*
assume *a*: $(t,\ u,\ r) \in$ *proUnion2*
define *P* **where** *PP*:$P = (\lambda num.\ ((t \in \mathcal{D}\ (Y\ num) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}$
$S \longrightarrow u \notin \mathcal{T}\ (Y\ num))))$
thus $\exists$ *num. num*$\in$*proUnion1*$\wedge$ *P num*
proof−
from *finiUnion* **obtain** *minIndexRB* **where** *ann1preB*: *minIndexRB*=$\{n.\ (P$
$n)\}$
by *auto*
from *B ann1preB* **obtain** *minmin* **where** *ab*: (*minmin*::*nat*) = *Inf* (*minIndexRB*)

by *auto*
from *ann0 ann1preB PP* **have** *ann1preNoEmpty1*: *minIndexRB* $\neq \{\}$
using *a* **by** *blast*
from *ab ann1preNoEmpty1* **have** *ann1pre0*:*minmin* $\in$ *minIndexRB*
using *Inf-nat-def wellorder-Least-lemma*(*1*) **by** *force*
have *minmin*$\in$ $\{n.\ \exists\, t\ u\ r.\ (t,u,r) \in$ *tryunion* $\wedge$ $((t \in \mathcal{D}\ (Y\ n) \longrightarrow u \notin \mathcal{T}\ S)$
$\wedge$
$(t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ n))) \wedge (\forall m.\ (t \in \mathcal{D}\ (Y\ m) \longrightarrow$
$u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ m)) \longrightarrow n \leq m)\}$
apply(*rule CollectI*,*rule-tac x*=*t* **in** *exI*,*rule-tac x*=*u* **in** *exI*,*rule-tac x*=*r*
in *exI*,*intro conjI*)
using *a ann0* **apply** *blast*
using *PP ann1pre0 ann1preB* **apply** *blast*
using *PP ann1pre0 ann1preB* **apply** *blast*
by (*simp add*: *Inf-nat-def PP ab ann1preB wellorder-Least-lemma*(*2*))
then show *?thesis*
using *ann1pre0 ann1preB finiUnion* **by** *blast*
qed
qed
from *ann1*
have *ann12*: $\forall\, t\ u\ r.\ \exists\, num.(t,\ u,\ r) \in$ *proUnion2* $\longrightarrow$ *num*$\in$*proUnion1* $\wedge$
$((t \in \mathcal{D}\ (Y\ num) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y$
$num)))$ **by** *auto*
from *ann0 Ctryy*
have *ann2*: $\forall\, t\ u\ r.\ (t,\ u,\ r) \in$ *proUnion2* $\longrightarrow$
(*r setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *A*)) $\wedge$
($\exists\, v.\ a = r\ @\ v \wedge$ *front-tickFree v* $\wedge$ (*tickFree r* $\vee$ $v = []$)) $\wedge$
($\exists\, nu1.\ (t \in \mathcal{D}\ (Y\ nu1) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}$
$(Y\ nu1))))$ **by** *simp*
have *ann15*: $\forall\, t\ u\ r.$ (*r setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *A*)) $\wedge$
($\exists\, v.\ a = r\ @\ v \wedge$ *front-tickFree v* $\wedge$ (*tickFree r* $\vee$ $v = []$)) $\wedge$

$(\exists\, nu1.\ (t \in \mathcal{D}\ (Y\ nu1) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ nu1)))) \longrightarrow$
$(t,\ u,\ r) \in proUnion2$
using *Ctryy ann0* **by** *blast*
from *ann12 ann15*
have *ann01*: $\forall\, t\ u\ r.\ ((r\ setinterleaves\ ((t,\ u),\ insert\ tick\ (ev\ `\ A)) \wedge$
$(\exists\, v.\ a = r\ @\ v \wedge front\text{-}tickFree\ v \wedge (tickFree\ r \vee v = [])) \wedge$
$(\exists\, nu1.\ (t \in \mathcal{D}\ (Y\ nu1) \longrightarrow u \notin \mathcal{T}\ S) \wedge$
$(t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ nu1)))) \longrightarrow$
$(\exists\, num \in proUnion1.\ ((t \in \mathcal{D}\ (Y\ num) \longrightarrow u \notin \mathcal{T}\ S) \wedge$
$(t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ num)))))$ **by** *blast*
from *ann01*
have *annhelp*: $\forall\, t\ u\ r.\ r\ setinterleaves\ ((t,\ u),\ insert\ tick\ (ev\ `\ A)) \longrightarrow$
$(\forall\, v.\ a = r\ @\ v \longrightarrow front\text{-}tickFree\ v \longrightarrow (tickFree\ r \vee v = []) \longrightarrow$
$(\exists\, num \in proUnion1.((t \in \mathcal{D}\ (Y\ num) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ num)))))$
by (*metis oneIndex*)
from *Abb1 Abb2*
have *annHelp2*: $\forall\, nn\ t\ u.$
$nn \in proUnion1 \wedge (t \in \mathcal{D}\ (Y\ nn) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ nn)) \longrightarrow$
$(t \in \mathcal{D}\ (Y\ proMax) \longrightarrow u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ proMax))$
by (*metis Abb3 Max-ge ann finiSet le-neq-trans*)
from *annHelp2 annhelp*
have *annHelp1*: $\forall\, t\ u\ r.$
$r\ setinterleaves\ ((t,\ u),\ insert\ tick\ (ev\ `\ A)) \longrightarrow$
$(\forall\, v.\ a = r\ @\ v \longrightarrow$
$front\text{-}tickFree\ v \longrightarrow$
$tickFree\ r \vee v = [] \longrightarrow (t \in \mathcal{D}\ (Y\ proMax) \longrightarrow u \notin \mathcal{T}\ S) \wedge$
$(t \in \mathcal{D}\ S \longrightarrow u \notin \mathcal{T}\ (Y\ proMax)))$ **by** *blast*
with *A B* **show** *?thesis*
by *blast*
qed
qed

lemma *limproc-Sync-F*:
$chain\ Y \Longrightarrow \mathcal{F}(lim\text{-}proc\ (range\ Y)[\![\ A\ ]\!]S) = \mathcal{F}(lim\text{-}proc\ (range\ (\lambda i.\ Y\ i\ [\![\ A\ ]\!]\ S)))$
apply(*auto simp add*: *Process-eq-spec D-Sync F-Sync F-LUB D-LUB T-LUB chain-Sync1*)
apply *blast*
apply *blast*
apply *blast*
using *front-tickFree-Nil* **apply** *blast*
using *front-tickFree-Nil* **apply** *blast*
proof−
fix *a b*

assume *A1*: $\forall x.\ (\exists t\ u\ X.\ (t,\ X) \in \mathcal{F}\ (Y\ x) \wedge$
$(\exists Y.\ (u,\ Y) \in \mathcal{F}\ S \wedge$
a *setinterleaves* $((t,\ u),$ *insert tick* $(ev\ `\ A)) \wedge$
$b = (X \cup Y) \cap$ *insert tick* $(ev\ `\ A) \cup X \cap Y)) \vee$
$(\exists t\ u\ r\ v.$ *front-tickFree* $v \wedge$ (*tickFree* $r \vee v = []) \wedge\ a = r\ @\ v \wedge$
r *setinterleaves* $((t,\ u),$ *insert tick* $(ev\ `\ A)) \wedge$
$(t \in \mathcal{D}\ (Y\ x) \wedge u \in \mathcal{T}\ S \vee t \in \mathcal{D}\ S \wedge u \in \mathcal{T}\ (Y\ x)))$
and *B*: $\forall t\ u\ r.\ r$ *setinterleaves* $((t,\ u),$ *insert tick* $(ev\ `\ A)) \longrightarrow$
$(\forall v.\ a = r\ @\ v \longrightarrow$ *front-tickFree* $v \longrightarrow$
$\neg$ *tickFree* $r \wedge v \neq [] \vee$
$((\exists x.\ t \notin \mathcal{D}\ (Y\ x)) \vee u \notin \mathcal{T}\ S) \wedge (t \in \mathcal{D}\ S \longrightarrow\ (\exists x.\ u \notin \mathcal{T}$
$(Y\ x))))$
and *C*: *chain Y*
thus $\exists t\ u\ X.\ (\forall x.\ (t,\ X) \in \mathcal{F}\ (Y\ x)) \wedge (\exists Y.\ (u,\ Y) \in \mathcal{F}\ S \wedge a$ *setinterleaves*
$((t,\ u),$ *insert tick* $(ev\ `\ A)) \wedge b = (X \cup Y) \cap$ *insert tick* $(ev\ `\ A) \cup X \cap Y)$
proof (*cases* $\exists m.\ a \notin \mathcal{D}(Y\ m\ [\![\ A\ ]\!]\ S)$)
case *True*
then obtain m **where** $a \notin \mathcal{D}(Y\ m\ [\![\ A\ ]\!]\ S)$
by *blast*
with *A1* **have** *D*: $\forall n.\ (a,\ b) \in \mathcal{F}\ (Y\ n\ [\![\ A\ ]\!]\ S)$
by (*auto simp*: *F-Sync*)
with *A1 B C* **show** *?thesis*
apply(*erule-tac x=m* **in** *allE*)
apply(*frule limproc-Sync-F-annex2*)
apply *simp*
by(*simp add*: *limproc-Sync-F-annex1*)
next
case *False*
with *False* **have** *E*: $\forall n.\ \ a \in \mathcal{D}\ (Y\ n\ [\![\ A\ ]\!]\ S)$
by *blast*
with *C E B* **show** *?thesis*
apply *auto*
apply(*drule limproc-Sync-F-annex2*)
apply *blast*
by *fast*
qed
qed

lemma *cont-left-Sync* : *chain Y* $\Longrightarrow ((\bigsqcup\ i.\ Y\ i)\ [\![\ A\ ]\!]\ S) = (\bigsqcup\ i.\ (Y\ i\ [\![\ A\ ]\!]\ S))$
by(*simp add*: *Process-eq-spec chain-Sync1 limproc-Sync-D limproc-Sync-F limproc-is-thelub*)

lemma *cont-right-Sync* : *chain Y* $\Longrightarrow (S\ [\![\ A\ ]\!]\ (\bigsqcup\ i.\ Y\ i)) = (\bigsqcup\ i.\ (S\ [\![\ A\ ]\!]\ Y$
$i))$
by (*metis* (*no-types*, *lifting*) *Process-eq-spec cont-left-Sync lub-eq mono-D-Sync mono-F-Sync*)

lemma *Sync-cont*[*simp*]:
assumes *f*:*cont f*

and *g*:*cont g*
shows *cont* ($\lambda x.$ ($f\ x$) $[\![A]\!]$ ($g\ x$))
proof −
  have A : $\bigwedge x.$ *cont g* $\Longrightarrow$ *cont* ($\lambda y.$ y $[\![A]\!]$ ($g\ x$))
    apply (*rule contI2*, *rule monofunI*)
     apply(*auto*)
    by (*simp add*: *cont-left-Sync*)
  have B : $\bigwedge y.$ *cont g* $\Longrightarrow$ *cont* ($\lambda x.$ y $[\![A]\!]$ $g\ x$)
    apply(*rule-tac c*=(λ $x.$ y $[\![A]\!]$ x) **in** *cont-compose*)
     apply(*rule contI2*, *rule monofunI*)
      apply (*simp*)
     apply (*simp add*: *cont-right-Sync*)
    by *simp*
  show *?thesis* **using** *f g*
    apply(*rule-tac f*=($\lambda x.$ (λ $f.$ f $[\![A]\!]$ $g\ x$)) **in** *cont-apply*)
    by(*auto intro*:*contI2 simp*:*A B*)
qed

end

4.3 The Multi-Prefix Operator Definition

theory *Mprefix*
imports *Process*
begin

4.3.1 The Definition and some Consequences

definition *Mprefix* :: [*'a set*,*'a* => *'a process*] => *'a process* **where**
  Mprefix A P ≡ *Abs-process*(
    $\{(tr,ref).\ tr = [] \wedge ref\ Int\ (ev\ `\ A) = \{\}\} \cup$
    $\{(tr,ref).\ tr \neq [] \wedge hd\ tr \in (ev\ `\ A) \wedge$
      $(\exists\ a.\ ev\ a = (hd\ tr) \wedge (tl\ tr,ref) \in \mathcal{F}(P\ a))\}$,
    $\{d.\ d \neq [] \wedge\ hd\ d\ \in (ev\ `\ A) \wedge$
      $(\exists\ a.\ ev\ a = hd\ d \wedge tl\ d \in \mathcal{D}\ (P\ a))\})$

syntax
  -Mprefix :: [*pttrn*,*'a set*,*'a process*] ⇒ *'a process* ((*3*□(-/∈-)/ → (-))[*0*,*0*,*64*]*64*)

term *Ball A* ($\lambda x.$ *P*)

translations
  □x∈A → P ⇌ *CONST Mprefix A* ($\lambda x.$ *P*)

Syntax Check:

term ‹□x∈A → □y∈A → □z∈A → $P\ z\ x\ y = Q$›

4.3.2 Well-foundedness of Mprefix

```
lemma is-process-REP-Mprefix :
is-process ({(tr,ref). tr=[] ∧ ref ∩ (ev ‘ A) = {}} ∪
            {(tr,ref). tr ≠ [] ∧ hd tr ∈ (ev ‘ A) ∧
                    (∃ a. ev a = (hd tr) ∧ (tl tr,ref) ∈ ℱ(P a))},
            {d. d ≠ [] ∧ hd d ∈ (ev ‘ A) ∧
                    (∃ a. ev a = hd d ∧ tl d ∈ 𝒟(P a))})
 (is is-process(?f, ?d))
proof (simp only:is-process-def FAILURES-def DIVERGENCES-def
                Product-Type.fst-conv Product-Type.snd-conv,
       intro conjI allI impI)
   show ([],{}) ∈ ?f by(simp)
next
   fix s:: 'a event list fix X::'a event set
   assume H : (s, X) ∈ ?f
   show front-tickFree s
            apply(insert H, auto simp: front-tickFree-def tickFree-def
                                dest!:list-nonMt-append)
            apply(rename-tac aa ta x, case-tac ta, auto simp: front-tickFree-charn
                                dest! : is-processT2[rule-format])
            apply(simp add: tickFree-def)
            done
next
   fix s t :: 'a event list
   assume (s @ t, {}) ∈ ?f
   then show (s, {}) ∈ ?f
            by(auto elim: is-processT3[rule-format])
next
   fix s:: 'a event list fix X Y::'a event set
   assume (s, Y) ∈ ?f ∧ X ⊆ Y
   then show (s, X) ∈ ?f
            by(auto intro: is-processT4[rule-format])
next
   fix s:: 'a event list fix X Y::'a event set
   assume (s, X) ∈ ?f ∧ (∀ c. c∈Y ⟶ (s @ [c], {}) ∉ ?f)
   then show (s, X ∪ Y) ∈ ?f
            by(auto intro!: is-processT1 is-processT5[rule-format])
next
   fix s:: 'a event list fix X::'a event set
   assume (s @ [tick], {}) ∈ ?f
   then show (s, X − {tick}) ∈ ?f
            by(cases s, auto dest!: is-processT6[rule-format])
next
   fix s t:: 'a event list fix X::'a event set
   assume s ∈ ?d ∧ tickFree s ∧ front-tickFree t
   then show s @ t ∈ ?d
            by(auto intro!: is-processT7-S, cases s, simp-all)
next
   fix s:: 'a event list fix X::'a event set
```

```
    assume     s ∈ ?d
    then show  (s, X) ∈ ?f
               by(auto simp: is-processT8-S)
next
    fix s::    'a event list
    assume     s @ [tick] ∈ ?d
    then show  s ∈ ?d
               apply(auto)
               apply(cases s, simp-all)
               apply(cases s, auto intro: is-processT9[rule-format])
               done

qed



lemma is-process-REP-Mprefix' :
     is-process ({(tr,ref). tr=[] ∧  ref ∩ (ev ‘ A) = {}} ∪
                 {(tr,ref). tr ≠ [] ∧ hd tr ∈ (ev ‘ A) ∧
                      (∃ a. ev a = (hd tr) ∧ (tl tr,ref) ∈ 𝓕(P a))},
                 {d. d ≠  [] ∧  hd d ∈ (ev ‘ A) ∧
                      (∃ a. ev a = hd d  ∧ tl d ∈ 𝓓(P a))})
 (is is-process(?f, ?d))
proof  (simp only:is-process-def FAILURES-def DIVERGENCES-def
                 Product-Type.fst-conv Product-Type.snd-conv,
        intro conjI allI impI,goal-cases)
  case 1
    show ([],{}) ∈ ?f by(simp)
next
  case (2 s X)
    assume H : (s, X) ∈ ?f
    have       front-tickFree s
      apply(insert H, auto simp:  front-tickFree-def tickFree-def
                          dest!:list-nonMt-append)
      apply(rename-tac aa ta x, case-tac ta, auto simp: front-tickFree-charn
                          dest! : is-processT2[rule-format])
      apply(simp add: tickFree-def)
      done
    then show front-tickFree s  by(auto)
next
  case (3 s t)
    assume H : (s @ t, {}) ∈ ?f
    show (s, {}) ∈ ?f using H  by(auto elim: is-processT3[rule-format])
next
  case (4 s X Y)
    assume H1: (s, Y) ∈ ?f ∧ X ⊆ Y
    then show (s, X) ∈ ?f  by(auto intro: is-processT4[rule-format])
next
  case (5 s X Y)
```

```
    assume  (s, X) ∈ ?f ∧ (∀ c. c∈Y ⟶ (s @ [c], {}) ∉ ?f)
   then show (s, X ∪ Y) ∈ ?f by(auto intro!: is-processT1 is-processT5[rule-format])

next
  case (6 s X)
    assume  (s @ [tick], {}) ∈ ?f
   then show (s, X − {tick}) ∈ ?f by(cases s, auto dest!: is-processT6[rule-format])
next
  case (7 s t)
    assume H1 : s ∈  ?d ∧ tickFree s ∧ front-tickFree t
    have 7:     s @ t ∈ ?d
      using H1  by(auto intro!: is-processT7-S, cases s, simp-all)
    then show s @ t ∈ ?d using H1  by(auto)
next
  case (8 s X)
    assume H : s ∈ ?d
    then show (s, X) ∈ ?f using H by(auto simp: is-processT8-S)
next
  case (9 s)
    assume H: s @ [tick] ∈ ?d
    then have 9:   s ∈ ?d
       apply(auto)
       apply(cases s, simp-all)
       apply(cases s, auto intro: is-processT9[rule-format])
       done
    then show s ∈ ?d by(auto)
qed

lemmas Rep-Abs-Mprefix[simp] = Abs-process-inverse[simplified, OF is-process-REP-Mprefix]
thm Rep-Abs-Mprefix

lemma Rep-Abs-Mprefix'' :
assumes H1 : f = {(tr, ref). tr = [] ∧ ref ∩ ev ‘ A = {}} ∪
                {(tr, ref). tr ≠ [] ∧ hd tr ∈ ev ‘ A
                          ∧ (∃ a. ev a = hd tr ∧ (tl tr, ref) ∈ ℱ (P a))}
    and H2 : d = {d. d ≠  [] ∧  hd d ∈ (ev ‘ A) ∧
                  (∃  a. ev a = hd d  ∧ tl d ∈ 𝒟(P a))}
shows Rep-process (Abs-process (f,d)) = (f,d)
by(subst Abs-process-inverse,
   simp-all only: H1 H2 CollectI Rep-process is-process-REP-Mprefix)
```

4.3.3 Projections in Prefix

```
lemma F-Mprefix :
  ℱ (□ x ∈ A → P x) = {(tr,ref). tr=[] ∧  ref ∩ (ev ‘ A) = {}} ∪
                      {(tr,ref). tr ≠ [] ∧ hd tr ∈ (ev ‘ A) ∧
                                 (∃  a. ev a = (hd tr) ∧ (tl tr,ref) ∈ ℱ(P a))}
  by(simp add:Mprefix-def Failures-def Rep-Abs-Mprefix'' FAILURES-def)
```

lemma *D-Mprefix*:
$\mathcal{D}\ (\Box\ x \in A \rightarrow P\ x) = \{d.\ d \neq\ [] \wedge\ hd\ d \in (ev\ `\ A) \wedge$
$(\exists\ a.\ ev\ a = hd\ d\ \wedge\ tl\ d \in \mathcal{D}(P\ a))\}$
by(*simp add*:*Mprefix-def D-def Rep-Abs-Mprefix'' DIVERGENCES-def*)

lemma *T-Mprefix*:
$\mathcal{T}\ (\Box\ x \in A \rightarrow P\ x)=\{s.\ s=[] \vee (\exists\ a.\ a{\in}A \wedge s{\neq}[] \wedge hd\ s = ev\ a \wedge tl\ s \in \mathcal{T}(P\ a))\}$
by(*auto simp*: *T-F-spec*[*symmetric*] *F-Mprefix*)

4.3.4 Basic Properties

lemma *tick-T-Mprefix* [*simp*]: $[tick] \notin \mathcal{T}\ (\Box\ x \in A \rightarrow P\ x)$
by(*simp add*:*T-Mprefix*)

lemma *Nil-Nin-D-Mprefix* [*simp*]: $[] \notin \mathcal{D}\ (\Box\ x \in A \rightarrow P\ x)$
by(*simp add*: *D-Mprefix*)

4.3.5 Proof of Continuity Rule

Backpatch Isabelle 2009-1

definition
contlub :: $('a{::}cpo \Rightarrow 'b{::}cpo) \Rightarrow bool$
where
$contlub\ f = (\forall\ Y.\ chain\ Y \longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = (\bigsqcup\ i.\ f\ (Y\ i)))$

lemma *contlubE*:
$[\![contlub\ f;\ chain\ Y]\!] \Longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = (\bigsqcup\ i.\ f\ (Y\ i))$
by (*simp add*: *contlub-def*)

lemma *monocontlub2cont*: $[\![monofun\ f;\ contlub\ f]\!] \Longrightarrow cont\ f$
apply (*rule contI*)
apply (*rule thelubE*)
apply (*erule* (*1*) *ch2ch-monofun*)
apply (*erule* (*1*) *contlubE* [*symmetric*])
done

lemma *contlubI*:
$(\bigwedge Y.\ chain\ Y \Longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = (\bigsqcup\ i.\ f\ (Y\ i))) \Longrightarrow contlub\ f$
by (*simp add*: *contlub-def*)

lemma *cont2contlub*: $cont\ f \Longrightarrow contlub\ f$
apply (*rule contlubI*)
apply (*rule Porder.po-class.lub-eqI* [*symmetric*])

apply (*erule* (*1*) *contE*)
done

Core of Proof

lemma *mono-Mprefix1*:
$\forall a{\in}A.\ P\ a \sqsubseteq Q\ a \Longrightarrow \mathcal{D}\ (Mprefix\ A\ Q) \subseteq \mathcal{D}\ (Mprefix\ A\ P)$
apply(*auto simp*: *D-Mprefix*) **using** *le-approx1* **by** *blast*

lemma *mono-Mprefix2*:
$\forall x{\in}A.\ P\ x \sqsubseteq Q\ x \Longrightarrow$
$\forall s.\ s \notin \mathcal{D}\ (Mprefix\ A\ P) \longrightarrow Ra\ (Mprefix\ A\ P)\ s = Ra\ (Mprefix\ A\ Q)\ s$
apply (*auto simp*: *Ra-def D-Mprefix F-Mprefix*) **using** *proc-ord2a* **by** *blast+*

lemma *mono-Mprefix3* :
assumes $H{:}\forall x{\in}A.\ P\ x \sqsubseteq Q\ x$
shows $min\text{-}elems\ (\mathcal{D}\ (Mprefix\ A\ P)) \subseteq \mathcal{T}\ (Mprefix\ A\ Q)$
proof(*auto simp*: *min-elems-def D-Mprefix T-Mprefix image-def*, *goal-cases*)
case (*1 x a*)
with *H*[*rule-format*, *of a*, *OF 1*(*2*)] **show** *?case*
apply(*auto dest*!: *le-approx3 simp*: *min-elems-def*)
apply(*subgoal-tac* $\forall t.\ t \in \mathcal{D}\ (P\ a) \longrightarrow \neg\ t < tl\ x$, *auto*)
apply(*rename-tac t*, *erule-tac x*=(*ev a*)#*t* **in** *allE*, *auto*)
using *less-cons hd-Cons-tl* **by** *metis*
qed

lemma *mono-Mprefix0*:
$\forall x{\in}A.\ P\ x \sqsubseteq Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq Mprefix\ A\ Q$
apply(*simp add*: *le-approx-def mono-Mprefix1 mono-Mprefix3*)
apply(*rule mono-Mprefix2*)
apply(*auto simp*: *le-approx-def*)
done

lemma *mono-Mprefix*[*simp*]: *monofun*(*Mprefix A*)
by(*auto simp*: *Fun-Cpo.below-fun-def monofun-def mono-Mprefix0*)

lemma *proc-ord2-set*:
$P \sqsubseteq Q \Longrightarrow \{(s,\ X).\ s \notin \mathcal{D}\ P \wedge (s,\ X) \in \mathcal{F}\ P\} = \{(s,\ X).\ s \notin \mathcal{D}\ P \wedge (s,\ X) \in \mathcal{F}\ Q\}$
by(*auto simp*: *le-approx2*)

lemma *proc-ord-proc-eq-spec*: $P \sqsubseteq Q \Longrightarrow \mathcal{D}\ P \subseteq \mathcal{D}\ Q \Longrightarrow P = Q$
by (*metis* (*mono-tags*, *lifting*) *below-antisym below-refl le-approx-def subset-antisym*)

lemma *Mprefix-chainpreserving*: $chain\ Y \Longrightarrow chain\ (\lambda i.\ Mprefix\ A\ (Y\ i))$

apply(*rule chainI*, *rename-tac i*)
apply(*frule-tac i=i* **in** *chainE*)
by(*simp add*: *mono-Mprefix0 fun-belowD*)

lemma *limproc-is-thelub-fun*:
assumes *chain S*
shows (*Lub S c*) = *lim-proc* (*range* (λx. (*S x c*)))
proof −
have $\bigwedge$*xa*. *chain* (λx. *S x xa*)
using ‹*chain S*› **by**(*auto intro*!: *chainI simp*: *chain-def fun-belowD*)
then **show** *?thesis* **by** (*metis contlub-lambda limproc-is-thelub*)
qed

lemma *contlub-Mprefix* : *contlub*(%*P*. *Mprefix A P*)
proof(*rule contlubI*, *rule proc-ord-proc-eq-spec*)
fix *Y* :: *nat* ⇒ *'a* ⇒ *'a process*
assume *C* : *chain Y*
have *C'*: $\bigwedge$*xa*. *chain* (λx. *Y x xa*)
apply(*insert C*,*rule chainI*)
by(*auto simp*: *chain-def fun-belowD*)
show *Mprefix A* ($\bigsqcup$ *i*. *Y i*) $\sqsubseteq$ ($\bigsqcup$ *i*. *Mprefix A* (*Y i*))
by(*auto simp*: *Process.le-approx-def F-Mprefix D-Mprefix T-Mprefix C C'*
Mprefix-chainpreserving limproc-is-thelub limproc-is-thelub-fun D-T D-LUB D-LUB-2 F-LUB T-LUB-2 Ra-def min-elems-def)
next
fix *Y* :: *nat* ⇒ *'a* ⇒ *'a process*
assume *C* : *chain Y*
show $\mathcal{D}$ (*Mprefix A* ($\bigsqcup$ *i*. *Y i*)) $\subseteq$ $\mathcal{D}$ ($\bigsqcup$ *i*. *Mprefix A* (*Y i*))
apply(*auto simp*: *Process.le-approx-def F-Mprefix D-Mprefix T-Mprefix C Mprefix-chainpreserving limproc-is-thelub D-LUB-2*)
by (*meson C fun-below-iff in-mono is-ub-thelub le-approx1*)
qed

lemma *Mprefix-cont* [*simp*]:
($\bigwedge$*x*. *cont* (*f x*)) $\Longrightarrow$ *cont* (λy. □ *z* ∈ *A* → *f z y*)
apply(*rule-tac f* = λz *y*. (*f y z*) **in** *Cont.cont-compose*)
apply(*rule monocontlub2cont*)
apply(*auto intro*: *mono-Mprefix contlub-Mprefix Fun-Cpo.cont2cont-lambda*)
done

4.3.6 High-level Syntax for Read and Write

The following syntax introduces the usual channel notation for CSP. Slightly deviating from conventional wisdom, we view a channel not as a tag in a pair, rather than as a function of type $'a \Rightarrow 'b$. This paves the way for *typed* channels over a common universe of events.

definition *read* :: $['a \Rightarrow 'b, 'a\ set,\ 'a \Rightarrow 'b\ process] \Rightarrow 'b\ process$
where $read\ c\ A\ P \equiv Mprefix(c\ `\ A)\ (P\ o\ (inv\text{-}into\ A\ c))$
definition *write* :: $['a \Rightarrow 'b,\ 'a,\ 'b\ process] \Rightarrow 'b\ process$
where $write\ c\ a\ P \equiv Mprefix\ \{c\ a\}\ (\lambda\ x.\ P)$
definition *write0* :: $['a,\ 'a\ process] \Rightarrow 'a\ process$
where $write0\ a\ P \equiv Mprefix\ \{a\}\ (\lambda\ x.\ P)$

syntax
-read :: [*id*, *pttrn*, *'a process*] => *'a process*
((*3*(-**?**-) /→ -) [*0*,*0*,*78*] *78*)
-readX :: [*id*, *pttrn*, *bool*,*'a process*] => *'a process*
((*3*(-**?**-)|- /→ -) [*0*,*0*,*78*] *78*)
-readS :: [*id*, *pttrn*, *'b set*,*'a process*] => *'a process*
((*3*(-**?**-)∈- /→ -) [*0*,*0*,*78*] *78*)
-write :: [*id*, *'b*, *'a process*] => *'a process*
((*3*(-**!**-) /→ -) [*0*,*0*,*78*] *78*)
-writeS :: [*'a*, *'a process*] => *'a process*
((*3*- /→ -) [*0*,*78*]*78*)

4.3.7 CSP$_M$-Style Syntax for Communication Primitives

translations
-read c p P ⇌ *CONST read c CONST UNIV* (λ*p*. *P*)
-write c p P ⇌ *CONST write c p P*
-readX c p b P => *CONST read c* {*p*. *b*} (λ*p*. *P*)
-writeS a P ⇌ *CONST write0 a P*
-readS c p A P => *CONST read c A* (λ*p*. *P*)

Syntax Check:

term‹ $d?y \rightarrow c!x \rightarrow P = Q$›

lemma *read-cont*[*simp*]: $cont\ P \Longrightarrow cont\ (\lambda y.\ read\ c\ A\ (P\ y))$
unfolding *read-def o-def*
by (*rule Mprefix-cont*) (*rule cont2cont-fun*)

lemma *read-cont'*[*simp*]: $(\bigwedge x.\ cont\ (f\ x)) \Longrightarrow cont\ (\lambda y.\ c?x \rightarrow f\ x\ y)$ **by** *simp*

lemma *write-cont*[*simp*]: $cont\ (P::('b::cpo \Rightarrow 'a\ process)) \Longrightarrow cont(\lambda x.\ (c!a \rightarrow P\ x))$

by(*simp add*:*write-def*)

corollary *write0-cont-lub* : *contlub*(*Mprefix* {*a*})
using *contlub-Mprefix* **by** *blast*

lemma *write0-contlub* : *contlub*(*write0 a*)
unfolding *write0-def contlub-def*
proof (*auto*)
fix *Y* :: *nat* ⇒ *'a process*
assume *chain Y*
have ∗ : *chain* (λ*i*. (λ-. *Y i*)::*'b* ⇒ *'a process*)
by (*meson* ‹*chain Y*› *fun-below-iff po-class.chain-def*)
have ∗∗: (λ*x*. *Lub Y*) = *Lub* (λ*i*. (λ-. *Y i*))
by(*rule ext*,*metis* ∗ *ch2ch-fun limproc-is-thelub limproc-is-thelub-fun lub-eq*)
show *Mprefix* {*a*} (λ*x*. *Lub Y*) = (⨆*i*. *Mprefix* {*a*} (λ*x*. *Y i*))
apply(*subst* ∗∗, *subst contlubE*[*OF contlub-Mprefix*])
by (*simp-all add*: ∗)
qed

lemma *write0-cont*[*simp*]: *cont* (*P*::(*'b*::*cpo* ⇒ *'a process*)) ⟹ *cont*(λ*x*. (*a* → *P x*))
by(*simp add*:*write0-def*)

end

theory *Mndetprefix*
imports *Process Stop Mprefix Ndet*
begin

4.4 Multiple non deterministic operator

definition
Mndetprefix :: [*'α set*, *'α* ⇒ *'α process*] ⇒ *'α process*
where *Mndetprefix A P* ≡ *if A* = {}
then STOP
else Abs-process(⋃ *x*∈*A*. $\mathcal{F}$(*x* → *P x*),
⋃ *x*∈*A*. $\mathcal{D}$(*x* → *P x*))

syntax
-Mndetprefix :: *pttrn* ⇒ *'a set* ⇒ *'a process* ⇒ *'a process*
((*3*⊓-∈- → / -) [*0*, *0*, *70*] *70*)

translations
⊓*x*∈*A*→ *P* ⇌ *CONST Mndetprefix A* (λ*x*. *P*)

lemma *mt-Mndetprefix*[*simp*] : *Mndetprefix* {} *P* = *STOP*
unfolding *Mndetprefix-def* **by** *simp*

lemma *Mndetprefix-is-process* : $A \neq \{\} \Longrightarrow$ *is-process* ($\bigcup$ $x{\in}A$. $\mathcal{F}(x \rightarrow P\ x)$, $\bigcup$ $x{\in}A$. $\mathcal{D}(x \rightarrow P\ x)$)
unfolding *is-process-def FAILURES-def DIVERGENCES-def*
apply *auto*
using *is-processT1* **apply** *auto*[*1*]
using *is-processT2* **apply** *blast*
using *is-processT3-SR* **apply** *blast*
using *is-processT4* **apply** *blast*
using *is-processT5-S1* **apply** *blast*
using *is-processT6* **apply** *blast*
using *is-processT7* **apply** *blast*
using *NF-ND* **apply** *auto*[*1*]
using *is-processT9* **by** *blast*

lemma *T-Mndetprefix1* : $\mathcal{T}$ (*Mndetprefix* $\{\}$ P) = $\{[]\}$
unfolding *Mndetprefix-def* **by**(*simp add*: *T-STOP*)

lemma *rep-abs-Mndetprefix*[*simp*]: $A \neq \{\} \Longrightarrow$
(*Rep-process* (*Abs-process*($\bigcup x{\in}A$. $\mathcal{F}(x \rightarrow P\ x)$,$\bigcup x{\in}A$. $\mathcal{D}$ $(x \rightarrow P\ x)$))) =
($\bigcup x{\in}A$. $\mathcal{F}(x \rightarrow P\ x)$, $\bigcup x{\in}A$. $\mathcal{D}$ $(x \rightarrow P\ x)$)
apply(*subst Process.process.Abs-process-inverse*)
by(*auto intro*: *Mndetprefix-is-process*[*simplified*])

lemma *T-Mndetprefix*: $A{\neq}\{\} \Longrightarrow \mathcal{T}$ (*Mndetprefix* A P) = ($\bigcup$ $x{\in}A$. $\mathcal{T}$ $(x \rightarrow P\ x)$)
unfolding *Mndetprefix-def*
apply(*simp*, *subst Traces-def*, *simp add*: *TRACES-def FAILURES-def*)
apply(*auto intro*: *Mndetprefix-is-process*[*simplified*])
unfolding *TRACES-def FAILURES-def* **apply**(*cases* $A = \{\}$)
apply(*auto intro*: *F-T D-T simp*: *Nil-elem-T*)
using *NF-NT* **by** *blast*

lemma *F-Mndetprefix1* : $\mathcal{F}$ (*Mndetprefix* $\{\}$ P) = $\{(s, X).\ s = []\}$
unfolding *Mndetprefix-def* **by**(*simp add*: *F-STOP*)

lemma *F-Mndetprefix*: $A{\neq}\{\} \Longrightarrow \mathcal{F}$ (*Mndetprefix* A P) = ($\bigcup$ $x{\in}A$. $\mathcal{F}(x \rightarrow P\ x)$)
unfolding *Mndetprefix-def*
by(*simp*, *subst Failures-def*, *auto simp* : *FAILURES-def F-Mndetprefix1*)

lemma *D-Mndetprefix1* : $\mathcal{D}$ (*Mndetprefix* $\{\}$ P) = $\{\}$
unfolding *Mndetprefix-def* **by**(*simp add*: *D-STOP*)

lemma *D-Mndetprefix* : $A{\neq}\{\} \Longrightarrow \mathcal{D}$(*Mndetprefix* A P) = ($\bigcup$ $x{\in}A$. $\mathcal{D}$ $(x \rightarrow P\ x)$)
unfolding *Mndetprefix-def*
apply(*simp*, *subst D-def*, *subst Process.process.Abs-process-inverse*)
by(*auto intro*: *Mndetprefix-is-process*[*simplified*] *simp*: *DIVERGENCES-def*)

Thus we know now, that Mndetprefix yields processes. Direct consequences

are the following distributivities:

lemma *Mndetprefix-unit* : $(\sqcap\ x \in \{a\} \to P\ x)\ = (a \to P\ a)$
by(*auto simp* : *Process.Process-eq-spec F-Mndetprefix D-Mndetprefix*)

lemma *Mndetprefix-Un-distrib* : $A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow (\sqcap\ x \in A \cup B \to P\ x) = ((\sqcap\ x \in A \to P\ x) \sqcap (\sqcap\ x \in B \to P\ x))$
by(*auto simp* : *Process.Process-eq-spec F-Ndet D-Ndet F-Mndetprefix D-Mndetprefix*)

The two lemmas *Mndetprefix* $\{?a\}$ $?P = ?a \to ?P\ ?a$ and $[\![?A \neq \{\};\ ?B \neq \{\}]\!] \Longrightarrow$ *Mndetprefix* $(?A \cup ?B)$ $?P =$ *Mndetprefix* $?A$ $?P \sqcap$ *Mndetprefix* $?B$ $?P$ together give us that *Mndetprefix* can be represented by a fold in the finite case.

lemma *Mndetprefix-distrib-unit* : $A - \{a\} \neq \{\} \Longrightarrow (\sqcap\ x \in insert\ a\ A \to P\ x) = ((a \to P\ a) \sqcap (\sqcap\ x \in A - \{a\} \to P\ x))$
by (*metis Un-Diff-cancel insert-is-Un insert-not-empty Mndetprefix-Un-distrib Mndetprefix-unit*)

4.4.1 Finite case Continuity

This also implies that Mndetprefix is continuous for the finite A and an arbitrary body f:

lemma *Mndetprefix-cont-finite*[*simp*]:
assumes *finite A*
and $\bigwedge x.\ cont\ (f\ x)$
shows $cont\ (\lambda y.\ \sqcap\ z \in A \to f\ z\ y)$
proof(*rule Finite-Set.finite-induct*[*OF* ‹*finite A*›])
show $cont\ (\lambda y.\ \sqcap z \in \{\} \to f\ z\ y)$ **by** *auto*
next
fix *A* **fix** *a*
assume $cont\ (\lambda y.\ \sqcap z \in A \to f\ z\ y)$ **and** $a \notin A$
show $cont\ (\lambda y.\ \sqcap z \in insert\ a\ A \to f\ z\ y)$
proof(*cases* $A = \{\}$)
case *True*
then show *?thesis* **by**(*simp add*: *Mndetprefix-unit True* ‹$\bigwedge x.\ cont\ (f\ x)$›)
next
case *False*
have $*$: $A - \{a\} \neq \{\}$ **by** (*simp add*: *False* ‹$a \notin A$›)
have $**$: $A - \{a\} = A$ **by** (*simp add*: ‹$a \notin A$›)
show *?thesis*
apply(*simp only*: *Mndetprefix-distrib-unit*[*OF* $*$], *simp only*: $**$)
by (*simp add*: ‹$cont\ (\lambda y.\ \sqcap z \in A \to f\ z\ y)$› *assms*(*2*))
qed
qed

4.4.2 General case Continuity

lemma *mono-Mndetprefix*[*simp*] : *monofun* (*Mndetprefix* (*A*::*'a set*))
proof(*cases* $A = \{\}$)

```
  case True
  then show ?thesis by(auto simp: monofun-def)
next
  case False
  then show ?thesis apply(simp add: monofun-def, intro allI impI)
    unfolding le-approx-def
    proof(simp add:T-Mndetprefix F-Mndetprefix D-Mndetprefix, intro conjI)
      fix x::'a ⇒ 'a process and y::'a ⇒ 'a process
      assume A ≠ {} and x ⊑ y
      show (⋃x∈A. 𝒟 (x→ y x)) ⊆ (⋃xa∈A. 𝒟 (xa → x xa))
        by (metis (mono-tags, lifting) SUP-mono ‹x ⊑ y› fun-below-iff le-approx1
mono-Mprefix0 write0-def)
    next
      fix x::'a ⇒ 'a process and y::'a ⇒  'a process
      assume ∗:A ≠ {} and ∗∗:x ⊑ y
      have ∗∗∗ : ⋀z. z ∈ A ⟹ x z ⊑ y z  by (simp add: ‹x ⊑ y› fun-belowD)
      with ∗ show ∀ s. (∀ xa∈A. s ∉ 𝒟 (xa → x xa)) ⟶ Ra (Mndetprefix A x) s
= Ra (Mndetprefix A y) s
        unfolding Ra-def
        by (auto simp:proc-ord2a mono-Mprefix0 write0-def F-Mndetprefix)
          (meson le-approx2 mono-Mprefix0 write0-def)+
    next
      fix x::'a ⇒ 'a process and y::'a ⇒  'a process
      assume ∗:A ≠ {} and x ⊑ y
      have ∗∗∗ : ⋀z. z ∈ A ⟹ (z → x z) ⊑ (z → y z)
        by (metis ‹x ⊑ y› fun-below-iff mono-Mprefix0 write0-def)
      with ∗ show min-elems (⋃xa∈A. 𝒟 (xa → x xa)) ⊆ (⋃x∈A. 𝒯 (x → y x))
        unfolding min-elems-def apply auto
          by (metis Set.set-insert elem-min-elems insert-subset le-approx3 le-less
min-elems5)
    qed
qed

lemma Mndetprefix-chainpreserving: chain Y ⟹ chain (λi. (⊓ z ∈ A → Y i z))
  apply(rule chainI, rename-tac i)
  apply(frule-tac i=i in chainE)
  by (simp add: below-fun-def mono-Mprefix0 write0-def monofunE)

lemma contlub-Mndetprefix : contlub (Mndetprefix A)
proof(cases A={})
  case True
  then show ?thesis by(auto simp: contlub-def)
next
  case False
  show ?thesis
  proof(rule contlubI, rule proc-ord-proc-eq-spec)
    fix Y :: nat ⇒ 'a ⇒ 'a process
    assume a:chain Y
    show (⊓x∈A → (⨆i. Y i) x) ⊑ (⨆i. ⊓x∈A → Y i x)
```

```
    proof(simp add:le-approx-def, intro conjI allI impI)
      show 𝒟 (⨆ i. ⊓x∈A → Y i x) ⊆ 𝒟 (⊓x∈A → Lub Y x)
        using a False D-LUB[OF Mndetprefix-chainpreserving[OF a], of A]
              limproc-is-thelub[OF Mndetprefix-chainpreserving[OF a], of A]


        apply (auto simp add:write0-def D-Mprefix D-LUB[OF ch2ch-fun[OF a]]
                           limproc-is-thelub-fun[OF a] D-Mndetprefix)

        by (metis (mono-tags, opaque-lifting) event.inject)
    next
      fix s :: 'a event list
      assume s ∉ 𝒟 (⊓x∈A → Lub Y x)
      show Ra (⊓x∈A → Lub Y x) s = Ra (⨆ i. ⊓x∈A → Y i x) s
        unfolding Ra-def
        using a False F-LUB[OF Mndetprefix-chainpreserving[OF a], of A]
              limproc-is-thelub[OF Mndetprefix-chainpreserving[OF a], of A]
        apply (auto simp add:write0-def F-Mprefix F-LUB[OF ch2ch-fun[OF a]]
                           limproc-is-thelub-fun[OF a] F-Mndetprefix)
        by (metis (mono-tags, opaque-lifting) event.inject)
    next
      show min-elems (𝒟 (⊓x∈A → Lub Y x)) ⊆ 𝒯 (⨆ i. ⊓x∈A → Y i x)
        unfolding min-elems-def
       using a False limproc-is-thelub[OF Mndetprefix-chainpreserving[OF a], of A]
              D-LUB[OF Mndetprefix-chainpreserving[OF a], of A]
              F-LUB[OF Mndetprefix-chainpreserving[OF a], of A]
          by (auto simp add:D-T write0-def D-Mprefix F-Mprefix D-Mndetprefix
F-Mndetprefix
                        D-LUB[OF ch2ch-fun[OF a]] F-LUB[OF ch2ch-fun[OF a]]
                        limproc-is-thelub-fun[OF a])
    qed
  next
    fix Y :: nat ⇒ 'a ⇒ 'a process
    assume a:chain Y
    show 𝒟 (⊓x∈A → (⨆ i. Y i) x) ⊆ 𝒟 (⨆ i. ⊓x∈A → Y i x)
        using a False D-LUB[OF Mndetprefix-chainpreserving[OF a], of A]
              limproc-is-thelub[OF Mndetprefix-chainpreserving[OF a], of A]
      by (auto simp add:write0-def D-Mprefix D-Mndetprefix D-LUB[OF ch2ch-fun[OF
a]]
                        limproc-is-thelub-fun[OF a])
  qed
qed

lemma Mndetprefix-cont[simp]: (⋀x. cont (f x)) ⟹ cont (λy. (⊓ z ∈ A → (f z
y)))
  apply(rule-tac f = λz y. (f y z) in Cont.cont-compose, rule monocontlub2cont)
  by (auto intro: mono-Mndetprefix contlub-Mndetprefix Fun-Cpo.cont2cont-lambda)

end
```

Chapter 5

The "Laws" of CSP

theory *CSP-Laws*
imports *Bot Skip Stop Det Ndet Mprefix Mndetprefix Seq Hiding Sync HOL−Eisbach.Eisbach*
begin

method *exI* **for** $y::'a$ = (*rule-tac exI*[**where** $x = y$])

5.1 General Laws

lemma *SKIP-Neq-STOP*: $SKIP \neq STOP$
by (*auto simp*: *Process-eq-spec F-SKIP F-STOP D-SKIP D-STOP Un-def*)

lemma *BOT-less1*[*simp*]: $\bot \leq (X::'a\ process)$
by (*simp add*: *le-approx-implies-le-ref*)

lemma *BOT-less2*[*simp*]: $BOT \leq (X::'a\ process)$
by *simp*

5.2 Deterministic Choice Operator Laws

lemma *mono-Det-FD-onside*[*simp*]: $P \leq P' \Longrightarrow (P \ \Box\ S) \leq (P' \ \Box\ S)$
unfolding *le-ref-def F-Det D-Det* **using** *F-subset-imp-T-subset* **by** *blast*

lemma *mono-Det-FD*[*simp*]: $[\![P \leq P';\ S \leq S']\!] \Longrightarrow (P \ \Box\ S) \leq (P' \ \Box\ S')$
by (*metis Det-commute dual-order.trans mono-Det-FD-onside*)

lemma *mono-Det-ref*: $[\![P \sqsubseteq P';\ S \sqsubseteq S']\!] \Longrightarrow (P \ \Box\ S) \sqsubseteq (P' \ \Box\ S')$
using *below-trans mono-Det mono-Det-sym* **by** *blast*

lemma *Det-BOT* : $(P \ \Box\ \bot) = \bot$
by (*auto simp add*:*Process-eq-spec D-Det F-Det is-processT2 D-imp-front-tickFree F-UU D-UU*)

lemma *Det-STOP*: $(P \Box STOP) = P$
by (*auto simp add*: *Process-eq-spec D-Det F-Det D-STOP F-STOP T-STOP Un-def Sigma-def is-processT8 is-processT6-S2*)

lemma *Det-id*: $(P \Box P) = P$
by (*auto simp*: *Process-eq-spec D-Det F-Det Un-def Sigma-def is-processT8 is-processT6-S2*)

lemma *Det-assoc*: $((M \Box P) \Box Q) = (M \Box (P \Box Q))$
by (*auto simp add*: *Process-eq-spec D-Det F-Det Un-def Sigma-def T-Det*)

5.3 NonDeterministic Choice Operator Laws

lemma *mono-Ndet-FD*[*simp*]: $[\![P \leq P'; S \leq S']\!] \Longrightarrow (P \sqcap S) \leq (P' \sqcap S')$
by (*auto simp*: *le-ref-def F-Ndet D-Ndet*)

lemma *mono-Ndet-FD-left*[*simp*]: $P \leq Q \Longrightarrow (P \sqcap S) \leq Q$
by (*metis D-Ndet F-Ndet dual-order.trans le-ref-def le-sup-iff order-refl*)

lemma *mono-Ndet-FD-right*[*simp*]: $P \leq Q \Longrightarrow (S \sqcap P) \leq Q$
by (*metis D-Ndet F-Ndet dual-order.trans le-ref-def le-sup-iff order-refl*)

lemma *mono-Ndet-ref*: $[\![P \sqsubseteq P'; S \sqsubseteq S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq (P' \sqcap S')$
using *below-trans mono-Ndet mono-Ndet-sym* **by** *blast*

lemma *Ndet-BOT*: $(P \sqcap \bot) = \bot$
by (*auto simp*: *Process-eq-spec D-Ndet F-Ndet is-processT2 D-imp-front-tickFree F-UU D-UU*)

lemma *Ndet-id*: $(P \sqcap P) = P$
by (*simp-all add*: *F-Ndet D-Ndet Process-eq-spec*)

lemma *Ndet-assoc*: $((M \sqcap P) \sqcap Q) = (M \sqcap (P \sqcap Q))$
by (*simp-all add*: *F-Ndet D-Ndet Process-eq-spec Un-assoc*)

5.3.1 Multi-Operators laws

lemma *Det-distrib*: $(M \Box (P \sqcap Q)) = ((M \Box P) \sqcap (M \Box Q))$
by (*auto simp add*: *Process-eq-spec F-Det D-Det F-Ndet D-Ndet Un-def T-Ndet*)

lemma *Ndet-distrib*: $(M \sqcap (P \Box Q)) = ((M \sqcap P) \Box (M \sqcap Q))$
by (*auto simp add*: *Process-eq-spec F-Det D-Det F-Ndet D-Ndet Un-def T-Ndet is-processT8 is-processT6-S2*)

lemma *Ndet-FD-Det*: $(P \sqcap Q) \leq (P \Box Q)$
apply(*auto simp add*:*le-ref-def D-Ndet D-Det F-Ndet F-Det T-Ndet T-Det Ra-def min-elems-def*)
using *is-processT6-S2 NF-ND* **by** *blast+*

5.4 Sequence Operator Laws

5.4.1 Preliminaries

definition *F-minus-D-Seq* **where**
F-minus-D-Seq P Q $\equiv \{(t, X).\ (t, X \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ t\} \cup$
$\{(t, X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in \mathcal{T}\ P \wedge (t2, X) \in \mathcal{F}\ Q\}$

lemma *F-minus-D-Seq-opt*: $(a,b) \in \mathcal{F}(P\ ;\ Q) = (a \in \mathcal{D}(P\ ;\ Q) \vee (a,b) \in$ *F-minus-D-Seq P Q*)
by (*auto simp add*: *F-Seq D-Seq F-minus-D-Seq-def*)

lemma *Process-eq-spec-optimized-Seq* :
$((P\ ;\ Q) = (U\ ;\ S)) = (\mathcal{D}\ (P\ ;\ Q) = \mathcal{D}\ (U\ ;\ S) \wedge$
F-minus-D-Seq P Q $\subseteq \mathcal{F}\ (U\ ;\ S) \wedge$
F-minus-D-Seq U S $\subseteq \mathcal{F}\ (P\ ;\ Q))$
unfolding *Process-eq-spec-optimized*[*of* $(P\ ;\ Q)\ (U\ ;\ S)$]
by (*auto simp*:*F-minus-D-Seq-opt*)

5.4.2 Laws

lemma *mono-Seq-FD*[*simp*]: $[\![P \leq P';\ S \leq S']\!] \Longrightarrow (P\ ;\ S) \leq (P'\ ;\ S')$
apply (*auto simp*: *le-ref-def F-Seq D-Seq*)
by (*metis F-subset-imp-T-subset subsetCE*)+

lemma *mono-Seq-ref*: $[\![P \sqsubseteq P';\ S \sqsubseteq S']\!] \Longrightarrow (P\ ;\ S) \sqsubseteq (P'\ ;\ S')$
using *below-trans mono-Seq mono-Seq-sym* **by** *blast*

lemma *BOT-Seq*: $(\bot\ ;\ P) = \bot$
apply(*auto simp add*: *Process-eq-spec D-Seq F-Seq front-tickFree-append F-UU D-UU T-UU*)
using *front-tickFree-append front-tickFree-implies-tickFree is-processT2* **apply** *blast*
using *D-imp-front-tickFree front-tickFree-append front-tickFree-implies-tickFree* **apply** *blast*
using *front-tickFree-charn tickFree-Nil* **apply** *blast*
using *D-imp-front-tickFree front-tickFree-append front-tickFree-implies-tickFree* **apply** *blast*
using *front-tickFree-Nil tickFree-Nil* **by** *blast*

lemma *Seq-SKIP*: $(P\ ;\ SKIP) = P$
apply (*auto simp add*: *Process-eq-spec F-Seq D-Seq F-SKIP D-SKIP is-processT7 is-processT8-S*
T-F-spec is-processT6-S1)
apply (*meson insertI2 is-processT4 subsetI*)
apply (*meson append-T-imp-tickFree is-processT5-S7 non-tickFree-tick not-Cons-self2 tickFree-append*)
using *T-F-spec insert-absorb is-processT5-S2* **apply** *fastforce*
apply (*metis F-T is-processT nonTickFree-n-frontTickFree*)

by (*metis append-Nil2 front-tickFree-mono is-processT nonTickFree-n-frontTickFree not-Cons-self*)

lemma *SKIP-Seq*: $(SKIP\ ;\ P) = P$
by (*auto simp add*: *Process-eq-spec D-Seq T-SKIP F-Seq F-SKIP D-SKIP is-processT8-Pair*)

lemma *STOP-Seq*: $(STOP\ ;\ P) = STOP$
by (*auto simp*: *Process-eq-spec F-Seq D-Seq F-STOP D-STOP T-STOP Un-def*)

5.4.3 Multi-Operators laws

lemma *Seq-Ndet-distrR*: $((P \sqcap Q)\ ;\ S) = ((P\ ;\ S) \sqcap (Q\ ;\ S))$
by (*auto simp add*: *Process-eq-spec D-Seq D-Ndet T-Ndet Un-def F-Seq T-Seq F-Ndet*)

lemma *Seq-Ndet-distrL*: $(P\ ;\ (Q \sqcap S)) = ((P\ ;\ Q) \sqcap (P\ ;\ S))$
by (*auto simp add*: *Process-eq-spec D-Seq D-Ndet T-Ndet Un-def F-Seq F-Ndet*)

lemma *Seq-assoc-D*: $\mathcal{D}\ (P\ ;\ (Q\ ;\ S)) = \mathcal{D}\ ((P\ ;\ Q)\ ;\ S)$
proof(*safe*, *goal-cases*)
case (*1 x*)
then show *?case*
apply(*auto simp add*:*D-Seq T-Seq*)
using *front-tickFree-Nil* **apply** *blast*
apply (*metis append.assoc append-single-T-imp-tickFree tickFree-append*)
by (*metis append.assoc*)
next
case (*2 x*)
then show *?case*
proof(*auto simp add*:*D-Seq T-Seq*, *goal-cases*)
case (*1 t2 t1a t2a*)
then show *?case* **using** *front-tickFree-append* **by** *fastforce*
next
case (*2 t1 t2 t1a t2a*)
then obtain *t2b* **where** *t2a* = *t2b*@[*tick*]
by (*metis T-nonTickFree-imp-decomp append-single-T-imp-tickFree non-tickFree-tick tickFree-append*)
with *2* **show** *?case* **by** *auto*
next
case (*3 t1 t2 t1a t2a*)
then show *?case* **by** (*metis front-tickFree-implies-tickFree process-charn*)
next
case (*4 t1 t2 t1a t2a*)
then obtain *t2b* **where** *t2a* = *t2b*@[*tick*]
by (*metis D-T T-nonTickFree-imp-decomp append-single-T-imp-tickFree non-tickFree-tick tickFree-append*)
with *4* **show** *?case*
by (*metis D-imp-front-tickFree append.assoc butlast-snoc front-tickFree-implies-tickFree process-charn*)

```
  next
    case (5 t1 t2 t1a t2a)
    then show ?case by (metis front-tickFree-implies-tickFree process-charn)
  next
    case (6 t1 t2 t1a t2a)
    then obtain t2b where  t2a = t2b@[tick]
     by (metis D-T T-nonTickFree-imp-decomp append-single-T-imp-tickFree non-tickFree-tick
tickFree-append)
    with 6 show ?case
     by (metis D-imp-front-tickFree append.assoc butlast-snoc front-tickFree-implies-tickFree
process-charn)
  qed
qed

lemma Seq-assoc: (P ; (Q ; S)) = ((P ; Q) ; S)
proof (auto simp: Process-eq-spec-optimized-Seq Seq-assoc-D, goal-cases)
  case (1 a b)
  then show ?case
 proof(auto simp add:F-minus-D-Seq-def Seq-assoc-D F-minus-D-Seq-opt append-single-T-imp-tickFree
del:conjI,
        goal-cases 11 12)
    case (11 t1 t2)
      then show ?case by (metis (mono-tags, lifting) D-Seq Seq-assoc-D UnCI
mem-Collect-eq)
  next
    case (12 t1 t1a t2a)
    hence (t1 @ t1a) @ [tick] ∈ 𝒯 (P ; Q) by (auto simp:T-Seq)
    then show ?case
      using 12(2)[rule-format, of t1@t1a t2a] 12(4,5,6) by simp
  qed
next
  case (2 a b)
  then show ?case
 proof(auto simp add:F-minus-D-Seq-def Seq-assoc-D F-minus-D-Seq-opt append-single-T-imp-tickFree
T-Seq del:conjI,
      goal-cases 21 22 23 24 25 26)
    case 21
    then show ?case
        by (metis (mono-tags, lifting) D-Seq UnCI append-Nil2 front-tickFree-Nil
mem-Collect-eq)
  next
    case (22 t1 t2 t1a t2a)
    then obtain t2b where  t2a = t2b@[tick]
     by (metis T-nonTickFree-imp-decomp append-single-T-imp-tickFree non-tickFree-tick
tickFree-append)
    with 22 show ?case using append.assoc butlast-snoc by auto
  next
    case (23 t1 t2 t1a t2a)
    hence t1 ∈ 𝒟 (P ; (Q ; S))
```

by (*simp add*:*D-Seq*) (*metis append-Nil2 front-tickFree-implies-tickFree process-charn*)
with *23 Seq-assoc-D* **show** *?case* **by** (*metis front-tickFree-implies-tickFree process-charn*)
next
case (*24 t1 t2 t1a t2a*)
then obtain *t2b* **where** $t2a = t2b@[tick]$
by (*metis D-T T-nonTickFree-imp-decomp append-single-T-imp-tickFree non-tickFree-tick tickFree-append*)
with *24* **show** *?case* **by** (*metis D-T append.assoc butlast-snoc*)
next
case (*25 t1 t2 t1a t2a*)
hence $t1 \in \mathcal{D}\ (P\ ;\ (Q\ ;\ S))$
by (*simp add*:*D-Seq*) (*metis append-Nil2 front-tickFree-implies-tickFree process-charn*)
with *25 Seq-assoc-D* **show** *?case* **by** (*metis front-tickFree-implies-tickFree process-charn*)
next
case (*26 t1 t2 t1a t2a*)
then obtain *t2b* **where** $t2a = t2b@[tick]$
by (*metis D-T T-nonTickFree-imp-decomp append-single-T-imp-tickFree non-tickFree-tick tickFree-append*)
with *26* **show** *?case* **by** (*metis D-T append.assoc butlast-snoc*)
qed
qed

5.5 The Multi-Prefix Operator Laws

lemma *mono-Mprefix-FD*[*simp*]: $\forall x \in A.\ P\ x \leq P'\ x \Longrightarrow Mprefix\ A\ P \leq Mprefix\ A\ P'$
by (*auto simp*: *le-ref-def F-Mprefix D-Mprefix*) *blast*+

lemmas *mono-Mprefix-ref* = *mono-Mprefix0*

lemma *Mprefix-STOP*: $(Mprefix\ \{\}\ P) = STOP$
by (*auto simp*:*Process-eq-spec F-Mprefix D-Mprefix D-STOP F-STOP*)

5.5.1 Multi-Operators laws

lemma *Mprefix-Un-distrib*: $(Mprefix\ (A \cup B)\ P) = ((Mprefix\ A\ P) \Box (Mprefix\ B\ P))$
apply (*unfold Process-eq-spec*, *rule conjI*)
apply (*auto*, (*elim disjE conjE* | *simp-all add*: *F-Det F-Mprefix Un-def image-def*)+, *auto*)
by (*auto simp add*: *D-Det D-Mprefix Un-def*)

lemma *Mprefix-Seq*: $((Mprefix\ A\ P)\ ;\ Q) = (Mprefix\ A\ (\lambda x.\ (P\ x)\ ;\ Q))$
proof (*unfold Process-eq-spec*, *rule conjI*, *rule subset-antisym*, *goal-cases*)
case *1*

```
  then show ?case
    apply(unfold split-def F-Seq D-Seq F-Mprefix T-Mprefix D-Mprefix)
    apply(rule subsetI, simp-all, elim disjE conjE exE)
        apply(rule disjI1, simp, meson tickFree-tl)
       apply (rule disjI2, rule conjI, simp) apply auto[1]
      apply (auto simp add:hd-append)[1]
   using tickFree-tl apply fastforce
  by (auto simp add: hd-append)[1]
next
  case 2
  then show ?case
    apply(unfold split-def F-Seq D-Seq F-Mprefix T-Mprefix D-Mprefix)
    apply(rule subsetI, simp-all, elim disjE conjE exE)
        apply(rule disjI1, simp, blast)
       apply(rule disjI1, metis event.simps(3) list.exhaust-sel tickFree-Cons)
  proof(goal-cases)
    case (1 x a t1 t2)
    then show ?case
      apply(rule-tac disjI2, rule-tac disjI1, rule-tac x=(ev a)#t1 in exI)
      using hd-Cons-tl image-iff by fastforce
  next
    case (2 x a t1 t2)
    then show ?case
       apply(rule-tac disjI2, rule-tac disjI2, rule-tac disjI1, rule-tac x=(ev a)#t1
in exI)
      using hd-Cons-tl image-iff by fastforce
  next
    case (3 x a t1 t2)
    then show ?case
       apply(rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac x=(ev a)#t1
in exI)
      using hd-Cons-tl image-iff by fastforce
  qed
next
  case 3
  then show ?case
    apply (auto simp add: D-Mprefix D-Seq T-Mprefix)
        using tickFree-tl apply blast
       apply (metis event.distinct(1) hd-append image-iff list.sel(1))
      apply (metis event.distinct(1) hd-append list.sel(1) tl-append2)
   apply (metis (no-types, opaque-lifting) append-Cons event.distinct(1) image-eqI
list.distinct(1)
                    list.exhaust-sel list.sel(1) list.sel(3) tickFree-Cons)
   by (metis append-Cons list.exhaust-sel list.sel(1) list.sel(3))
qed
```

5.5.2 Derivative Operators laws

lemma *Mprefix-singl*: (*Mprefix* {*a*} *P*) = (*a* → (*P a*))

by (*simp add*:*write0-def Mprefix-def*, *rule arg-cong*[*of - - λx. Abs-process x*]) *fastforce*

lemma *mono-read-FD*: $(\bigwedge x.\ P\ x \leq Q\ x) \Longrightarrow\ (c?x \rightarrow (P\ x)) \leq (c?x \rightarrow (Q\ x))$
by (*simp add*: *read-def*)

lemma *mono-write-FD*: $(P \leq Q) \Longrightarrow\ (c!x \rightarrow P) \leq (c!x \rightarrow Q)$
by (*simp add*: *write-def*)

lemma *mono-write0-FD*: $P \leq Q \Longrightarrow (a \rightarrow P) \leq (a \rightarrow Q)$
by (*simp add*: *write0-def*)

lemma *mono-read-ref*: $(\bigwedge x.\ P\ x \sqsubseteq Q\ x) \Longrightarrow\ (c?x \rightarrow (P\ x)) \sqsubseteq (c?x \rightarrow (Q\ x))$
by (*simp add*: *mono-Mprefix0 read-def*)

lemma *mono-write-ref*: $(P \sqsubseteq Q) \Longrightarrow\ (c!x \rightarrow P) \sqsubseteq (c!x \rightarrow Q)$
by (*simp add*: *mono-Mprefix0 write-def*)

lemma *mono-write0-ref*: $P \sqsubseteq Q \Longrightarrow (a \rightarrow P) \sqsubseteq (a \rightarrow Q)$
by (*simp add*: *mono-Mprefix0 write0-def*)

lemma *write0-Ndet*: $(a \rightarrow (P \sqcap Q)) = ((a \rightarrow P) \sqcap (a \rightarrow Q))$
by (*auto simp*: *Process-eq-spec write0-def D-Ndet F-Ndet F-Mprefix D-Mprefix Un-def*)

lemma *write0-Det-Ndet*: $((a \rightarrow P) \Box (a \rightarrow Q)) = ((a \rightarrow P) \sqcap (a \rightarrow Q))$
by (*auto simp*: *Process-eq-spec write0-def D-Ndet F-Ndet F-Det D-Det*) (*simp add*: *F-Mprefix*)+

lemma *Mprefix-Det*: ‹$(\Box e{\in}A \rightarrow P\ e) \Box (\Box e{\in}A \rightarrow Q\ e) = \Box e{\in}A \rightarrow (P\ e \sqcap Q\ e)$›
by (*auto simp*: *Process-eq-spec F-Det D-Det*) (*auto simp*: *D-Ndet F-Ndet F-Mprefix D-Mprefix*)

5.6 The Hiding Operator Laws

5.6.1 Preliminaries

lemma *elemDIselemHD*: ‹$t \in \mathcal{D}\ P \Longrightarrow$ *trace-hide* t (ev ‘ A) $\in \mathcal{D}\ (P \setminus A)$›
proof (*cases tickFree t*)
case *True*
assume $t \in \mathcal{D}\ P$
with *True* **show** *?thesis* **by** (*simp add*:*D-Hiding*, *rule-tac x=t* **in** *exI*, *rule-tac x=[]* **in** *exI*, *simp*)
next
case *False*
assume *a*:$t \in \mathcal{D}\ P$
with *False* **obtain** t' **where** $t = t'@[tick]$ **using** *D-imp-front-tickFree nonTick-Free-n-frontTickFree* **by** *blast*
with *a* **show** *?thesis* **apply** (*auto simp add*:*D-Hiding*, *rule-tac x=t'* **in** *exI*,

rule-tac x=[tick] **in** *exI*)
by (*meson front-tickFree-implies-tickFree front-tickFree-single is-processT*)
qed

lemma *length-strict-mono*: *strict-mono* (*f*::*nat* ⇒ *'a list*) ⟹ *length* (*f i*) ≥ *i* + *length* (*f 0*)
apply(*induct i*, *simp*)
by (*metis dual-order.trans lessI less-length-mono not-less not-less-eq-eq plus-nat.simps*(*2*) *strict-mono-def*)

lemma *mono-trace-hide*: *a* ≤ *b* ⟹ *trace-hide a* (*ev* ‘ *A*) ≤ *trace-hide b* (*ev* ‘ *A*)
by (*metis filter-append le-list-def*)

lemma *mono-constant*:
assumes *mono* (*f*::*nat* ⇒ *'a event list*) **and** ∀ *i*. *f i* ≤ *a*
shows ∃ *i*. ∀ *j*≥*i*. *f j* = *f i*
proof −
from *assms*(*2*) **have** ∀ *i*. *length* (*f i*) ≤ *length a*
by (*simp add*: *le-length-mono*)
hence *aa*:*finite* {*length* (*f i*)|*i*. *True*}
using *finite-nat-set-iff-bounded-le* **by** *auto*
define *lm* **where** *l2*:*lm* = *Max* {*length* (*f i*)|*i*. *True*}
with *aa* **obtain** *im* **where** *length* (*f im*) = *lm* **using** *Max-in* **by** *fastforce*
with *l2 assms*(*1*) **show** *?thesis*
apply(*rule-tac x=im* **in** *exI*, *intro impI allI*)
by (*metis* (*mono-tags*, *lifting*) *Max-ge aa antisym le-length-mono le-neq-trans less-length-mono*
mem-Collect-eq mono-def order-less-irrefl)
qed

lemma *elemTIselemHT*: ‹*t* ∈ 𝒯 *P* ⟹ *trace-hide t* (*ev* ‘ *A*) ∈ 𝒯 (*P* \ *A*)›
proof (*cases tickFree t*)
case *True*
assume *a*:*t* ∈ 𝒯 *P*
with *True* **show** *?thesis*
proof (*cases* (∃ *ta*. *trace-hide t* (*ev* ‘ *A*) = *trace-hide ta* (*ev* ‘ *A*) ∧ (*ta*, *ev* ‘ *A*) ∈ ℱ *P*))
case *True*
thus *?thesis* **by** (*simp add*:*T-Hiding*)
next
case *False*
with *a False inf-hidden*[*of A t P*] **obtain** *f* **where** *isInfHiddenRun f P A* ∧ *t* ∈ *range f* **by** *auto*
with *True* **show** *?thesis*
by (*simp add*:*T-Hiding*, *rule-tac disjI2*, *rule-tac x=t* **in** *exI*, *rule-tac x=*[] **in** *exI*, *auto*)
qed
next
case *False*

```
  assume a:t ∈ 𝒯 P
  with False obtain t' where tt:t = t'@[tick] using T-nonTickFree-imp-decomp
by auto
  with a show ?thesis
  proof (cases (∃ ta. trace-hide t (ev ‘ A) = trace-hide ta (ev ‘ A) ∧ (ta, ev ‘ A) ∈
ℱ P))
    case True
    thus ?thesis by (simp add:T-Hiding)
  next
    case False
    assume t ∈ 𝒯 P
    with False inf-hidden[of A t P] obtain f where isInfHiddenRun f P A ∧ t ∈
range f by auto
    then show ?thesis
        apply (simp add:T-Hiding, rule-tac disjI2, rule-tac x=t' in exI, rule-tac
x=[tick] in exI, auto)
        apply (metis append-T-imp-tickFree list.distinct(1) tt)
       using tt apply force
         by (metis False append-T-imp-tickFree is-processT5-S7 non-tickFree-tick
not-Cons-self2 tickFree-append tt)
  qed
qed

lemma Hiding-Un-D1: ‹𝒟 (P \ (A ∪ B)) ⊆ 𝒟 ((P \ A) \ B)›
proof (simp add:conj-commute D-Hiding, intro conjI subset-antisym subsetI, simp-all,
goal-cases)
  case (1 x)
  then obtain t u where B1:x = trace-hide t (ev ‘ (A ∪ B)) @ u and B2:tickFree
t and B3:front-tickFree u and
            B4:(t ∈ 𝒟 P ∨ (∃ (f:: nat ⇒ 'a event list). isInfHiddenRun f P (A ∪ B)
∧ t ∈ range f)) by auto
  thus ?case
    apply(erule-tac  disjE)
    apply(rule-tac x=trace-hide t (ev ‘ A) in exI, rule-tac x=u in exI)
    apply(simp add:Hiding-tickFree tickFree-def)
    apply(rule disjI1, rule-tac x=t in exI, rule-tac x=[] in exI, simp)
  proof(goal-cases)
    case 1
    then obtain f n where fc:isInfHiddenRun f P (A ∪ B) ∧ t = f n by auto
    define ff where ff = (λi. take (i + length (f 0)) (f i))
    with fc have ffc:isInfHiddenRun ff P (A ∪ B) ∧ t ∈ range ff
    proof (auto, goal-cases)
      case 1
      { fix x
        from length-strict-mono[of f Suc x , OF 1(2)]
        have a:take (x + length (f 0)) (f (Suc x)) < take ((Suc x) + length (f 0))
(f (Suc x))
          by (simp add: take-Suc-conv-app-nth)
        from 1(2)[THEN strict-monoD, of x Suc x, simplified]
```

obtain *t* **where** *f* (*Suc x*) = (*f x* @ *t*) **by** (*metis le-list-def less-imp-le*)
with *length-strict-mono*[*of f x*, *OF 1*(*2*)]
have *take* (*x* + *length* (*f 0*)) (*f x*) = *take* (*x* + *length* (*f 0*)) (*f* (*Suc x*)) **by** *simp*
with *a* **have** *take* (*x* + *length* (*f 0*)) (*f x*) < *take* (*Suc x* + *length* (*f 0*)) (*f* (*Suc x*)) **by** *simp*
}
thus *?case* **by** (*meson lift-Suc-mono-less strict-mono-def*)
next
case (*2 i*)
have *take* (*i* + *length* (*f 0*)) (*f i*) ≤ *f i*
using *append-take-drop-id le-list-def* **by** *blast*
also have ∀ *x y*. *x* ≤ *y* ∧ *y* ∈ 𝒯 *P* ⟶ *x* ∈ 𝒯 *P* **using** *is-processT3-ST-pref* **by** *blast*
ultimately show *?case* **using** *2*(*3*) **by** *blast*
next
case (*3 i*)
hence (*f i*) ≥ (*f 0*) **using** *strict-mono-less-eq* **by** *blast*
hence *take* (*i* + *length* (*f 0*)) (*f i*) ≥ (*f 0*)
by (*metis add.commute append-eq-conv-conj le-list-def take-add*)
hence *a*:[*x*←*take* (*i* + *length* (*f 0*)) (*f i*) . *x* ∉ *ev* ‘ *A* ∧ *x* ∉ *ev* ‘ *B*] ≥ [*x*←*f 0* . *x* ∉ *ev* ‘ *A* ∧ *x* ∉ *ev* ‘ *B*]
by (*metis* (*no-types*, *lifting*) *filter-append le-list-def*)
have *take* (*i* + *length* (*f 0*)) (*f i*) ≤ *f i*
using *append-take-drop-id le-list-def* **by** *blast*
hence [*x*←*take* (*i* + *length* (*f 0*)) (*f i*) . *x* ∉ *ev* ‘ *A* ∧ *x* ∉ *ev* ‘ *B*] ≤ [*x*←*f i* . *x* ∉ *ev* ‘ *A* ∧ *x* ∉ *ev* ‘ *B*]
by (*metis* (*no-types*, *lifting*) *filter-append le-list-def*)
with *a 3*(*4*) **show** *?case* **by** (*metis* (*no-types*, *lifting*) *dual-order.antisym*)
next
case *4*
have *f* (*length* (*f n*) − *length* (*f 0*)) ≥ *f n*
by (*simp add*: *4*(*2*) *add-le-imp-le-diff length-strict-mono strict-mono-less-eq*)
hence *f n* = (λ*i*. *take* (*i* + *length* (*f 0*)) (*f i*)) (*length* (*f n*) − *length* (*f 0*))
by (*metis 4*(*2*) *add.commute append-eq-conv-conj diff-is-0-eq′*
le-add-diff-inverse le-list-def le-zero-eq nat-le-linear strict-mono-less-eq)
then show *?case* **by** *blast*
qed
thus *?case* **proof**(*cases* ∃ *m*. (∀ *i*>*m*. *last* (*ff i*) ∈ (*ev* ‘ *A*)))
case *True*
then obtain *m* **where** *mc*:∀ *i*>*m*. *last* (*ff i*) ∈ (*ev* ‘ *A*) **by** *blast*
hence *mc2*:*tickFree* (*ff m*)
by (*metis* (*no-types*, *lifting*) *B2 event.distinct*(*1*) *ffc*
image-iff mem-Collect-eq set-filter tickFree-def)
with *mc mc2 1 ffc* **show** *?thesis*
apply(*rule-tac x*=*trace-hide* (*ff m*) (*ev* ‘ *A*) **in** *exI*, *rule-tac x*=*u* **in** *exI*, *simp*, *intro conjI*)
apply (*metis* (*no-types*, *lifting*) *mem-Collect-eq set-filter tickFree-def*)
apply (*metis* (*no-types*, *lifting*) *rangeE*)

```
      apply(rule disjI1, rule-tac x=ff m in exI, rule-tac x=[] in exI, intro conjI,
simp-all)
      apply(rule disjI2, rule-tac x=λi. ff(i + m) in exI, intro conjI)
        apply (metis (no-types, lifting) add.commute add.right-neutral rangeI)
       apply (simp add: strict-mono-def)
      apply blast
      proof(rule allI, goal-cases)
       case (1 i)
        from ffc ff-def True have ∃ t. (ff (i + m)) = (ff m) @ t ∧ set t ⊆ (ev ‘
A)
       proof(induct i)
         case 0
         then show ?case by fastforce
       next
         case (Suc i)
        then obtain tt where tc:(ff (i + m)) = (ff m) @ tt ∧ set tt ⊆ (ev ‘ A)
by blast
         from ffc ff-def length-strict-mono[of ff] have lc:length (ff (Suc i + m))
= Suc (length (ff (i + m)))
           by (metis (no-types, lifting) add-Suc fc length-strict-mono length-take
min.absorb2)
        from True obtain l where lc2:l = last (ff (Suc i + m)) ∧ l ∈ (ev ‘ A)
          by (meson less-add-same-cancel2 mc zero-less-Suc)
        from ffc obtain r where rc:ff (Suc i + m) = ff (i + m)@r
              by (metis add.commute add-Suc-right le-list-def lessI less-imp-le
strict-mono-less-eq)
            with lc have length r = 1 by (metis Suc-eq-plus1 add-left-cancel
length-append)
        with rc lc2 have r = [l]
              by (metis (no-types, lifting) Nil-is-append-conv Suc-eq-plus1 ap-
pend-butlast-last-id
                        append-eq-append-conv append-eq-append-conv2 length-Cons
length-append)
         with Suc lc2 tc rc show ?case by (rule-tac x=tt@[l] in exI, auto)
       qed
       then show ?case using filter-empty-conv by fastforce
      qed
    next
     case False
     { fix i
      assume as:(i::nat) > 0
      with ffc obtain tt where ttc:ff i = ff 0 @ tt ∧ set tt ⊆ ev ‘ (A ∪ B)
        unfolding isInfHiddenRun-1 by blast
      with ffc as have tt ≠ [] using strict-mono-eq by fastforce
      with ttc have last (ff i) ∈ ev ‘ (A ∪ B) by auto
     }
     hence as2:∀ i. ∃ j>i. last(ff j) ∈ ((ev ‘ B) − (ev ‘ A))
      by (metis DiffI False UnE gr-implies-not-zero gr-zeroI image-Un)
     define ffb where ffb = rec-nat t (λi t. (let j = SOME j. ff j > t ∧
```

```
                    last(ff j) ∈ ((ev ‘ B) − (ev ‘ A)) in ff j))
    with as2 have ffbff:⋀n. ffb n ∈ range ff
    by (metis (no-types, lifting) ffc old.nat.exhaust old.nat.simps(6) old.nat.simps(7)
rangeI)
    from 1 ffb-def show ?thesis
      apply(rule-tac x=trace-hide t (ev ‘ A) in exI, rule-tac x=u in exI, simp,
intro conjI)
      apply (meson filter-is-subset set-rev-mp tickFree-def)
   proof(rule disjI2, rule-tac x=λi. trace-hide (ffb i) (ev ‘ A) in exI, intro conjI,
goal-cases)
      case 1
      then show ?case by (metis (no-types, lifting) old.nat.simps(6) rangeI)
    next
      case 2
      { fix n
        have a0:(ffb (Suc n)) = ff (SOME j. ff j > ffb n ∧ last(ff j) ∈ ((ev ‘ B)
− (ev ‘ A)))
          by (simp add: ffb-def)
        from ffbff obtain i where a1:ffb n = ff i by blast
        with as2 have ∃j. ff j > ffb n ∧ last(ff j) ∈ ((ev ‘ B) − (ev ‘ A))
          by (metis ffc strict-mono-def)
        with a0 a1 have a:(ffb (Suc n)) > (ffb n) ∧ last (ffb (Suc n)) ∉ (ev ‘ A)
          by (metis (no-types, lifting) Diff-iff tfl-some)
        then obtain r where ffb (Suc n) = (ffb n)@r ∧ last r ∉ (ev ‘ A)
          by (metis append-self-conv last-append le-list-def less-list-def)
        hence trace-hide (ffb (Suc n)) (ev ‘ A) > trace-hide (ffb n) (ev ‘ A)
       by (metis (no-types, lifting) a append-self-conv filter-append filter-empty-conv

                                last-in-set le-list-def less-list-def)
      }
         then show ?case by (metis (mono-tags, lifting) lift-Suc-mono-less-iff
strict-monoI)
    next
      case 3
      then show ?case by (metis (mono-tags) elemTIselemHT ffbff ffc rangeE)
    next
      case 4
      from ffbff ffc show ?case by (metis rangeE trace-hide-union)
    qed
  qed
 qed
qed

lemma Hiding-Un-D2: ‹finite A ⟹ 𝒟 ((P \ A) \ B) ⊆ 𝒟 (P \ (A ∪ B))›
proof (simp add:conj-commute D-Hiding, intro conjI subset-antisym subsetI, simp-all,
goal-cases)
 case (1 x)
 then obtain t u where B1:x = trace-hide t (ev ‘ B) @ u
              and B2:tickFree t
```

```
                and  B3:front-tickFree u
                and  B4:(t ∈ 𝒟 (P \ A) ∨
                        (∃(f:: nat ⇒ 'a event list). isInfHiddenRun f (P \ A) B ∧ t
∈ range f))
    by (simp add:D-Hiding) blast
  thus ?case
  proof(erule-tac disjE, auto simp add:D-Hiding, goal-cases)
    case (1 ta ua)
    then show ?case
      by (rule-tac x=ta in exI, rule-tac x = trace-hide ua (ev ‘ B) @ u in exI,
          auto simp add: front-tickFree-append tickFree-def)
  next
    case (2 ua f xa)
    then show ?case
      apply(rule-tac x=f xa in exI, rule-tac x=trace-hide ua (ev ‘ B) @ u in exI,
            intro conjI disjI2)
         apply(auto simp add: front-tickFree-append tickFree-def)
      by (rule-tac x=f in exI) (metis (no-types) filter-filter rangeI)
  next
    case (3 f xx)
    note 3a = 3
    then show ?case
    proof(cases ∃ i. f i ∈ 𝒟 (P \ A))
      case True
      with 3 show ?thesis
      proof(auto simp add:D-Hiding, goal-cases)
        case (1 i ta ua)
        then show ?case
        apply (rule-tac x=ta in exI, rule-tac x=trace-hide ua (ev ‘ B) @ u in exI,
intro conjI)
              apply (metis (full-types) front-tickFree-append Hiding-tickFree tick-
Free-append)
           apply(subgoal-tac trace-hide (f xx) (ev ‘ B) = trace-hide (f i) (ev ‘ B),
auto)
         apply (metis (full-types) filter-append filter-filter)
        by (metis (full-types) filter-append filter-filter)
      next
        case (2 i ua fa xa)
        hence trace-hide (f xx) (ev ‘ B) = trace-hide (f i) (ev ‘ B) by metis
        with 2 show ?case
          apply (rule-tac x=fa xa in exI, rule-tac x=trace-hide ua (ev ‘ B) @ u in
exI, intro conjI)
               apply (metis (full-types) front-tickFree-append Hiding-tickFree tick-
Free-append)
            apply (simp-all)
          apply(rule-tac disjI2, rule-tac x=fa in exI, auto)
          by (metis (no-types) filter-filter)
      qed
    next
```

case *False*
with *3* **have** *Falsebis*:$\forall i.\ (f\ i \in \mathcal{T}\ (P \setminus A) \wedge f\ i \notin \mathcal{D}\ (P \setminus A))$ **by** *blast*
with *T-Hiding*[*of P A*] *D-Hiding*[*of P A*]
have $\forall i.\ (f\ i \in \{$*trace-hide t* (*ev* ‘ *A*) $|t.\ (t,$ *ev* ‘ *A*$) \in \mathcal{F}\ P\})$
by (*metis* (*no-types*, *lifting*) *UnE*)
hence *ff0*:$\forall i.\ (\exists t.\ f\ i =$ *trace-hide t* (*ev* ‘ *A*) $\wedge\ t \in \mathcal{T}\ P)$ **using** *F-T* **by** *fastforce*
define *ff* **where** *ff1*:*ff* = ($\lambda i.$ *SOME t. f i = trace-hide t* (*ev* ‘ *A*) $\wedge\ t \in \mathcal{T}\ P$)
hence *inj ff* **unfolding** *inj-def* **by** (*metis* (*mono-tags*, *lifting*) *3*(*4*) *ff0 strict-mono-eq tfl-some*)
hence *ff2*:*infinite* (*range ff*) **using** *finite-imageD* **by** *blast*
{ fix *tt i*
assume *tt* $\in$ *range ff*
then obtain *k* **where** *ff k* = *tt* **using** *finite-nat-set-iff-bounded-le* **by** *blast*
hence *kk0*:*f k* = *trace-hide tt* (*ev* ‘ *A*) $\wedge$ *tt* $\in \mathcal{T}\ P$ **using** *ff1*
by (*metis* (*mono-tags*, *lifting*) *ff0 someI-ex*)
hence *set* (*take i tt*) $\subseteq$ *set* (*f i*) $\cup$ (*ev* ‘ *A*)
proof(*cases* $k \leq i$)
case *True*
hence *set* (*f k*) $\subseteq$ *set* (*f i*)
by (*metis 3*(*4*) *le-list-def set-append strict-mono-less-eq sup.cobounded1*)
moreover from *kk0* **have** *set* (*take i tt*) $\subseteq$ *set* (*f k*) $\cup$ (*ev* ‘ *A*) **using** *in-set-takeD* **by** *fastforce*
ultimately show *?thesis* **by** *blast*
next
case *False*
have *a*:*length* (*f i*) $\geq i$ **by** (*meson 3*(*4*) *dual-order.trans le-add1 length-strict-mono*)
have *b*:*f i* $\leq$ *f k* **using** *3*(*4*) *False nat-le-linear strict-mono-less-eq* **by** *blast*
with *a* **have** *c*:*take i* (*f k*) $\leq$ (*f i*)
by (*metis append-eq-conv-conj le-add-diff-inverse le-list-def take-add*)
from *kk0*[*THEN conjunct1*] **have** *c1*:*f k* = (*trace-hide* (*take i tt*) (*ev* ‘ *A*)) @
(*trace-hide* (*drop i tt*) (*ev* ‘ *A*))
by (*metis append-take-drop-id filter-append*)
have *length* (*trace-hide* (*take i tt*) (*ev* ‘ *A*)) $\leq i$
by (*metis length-filter-le length-take min.absorb2 nat-le-linear order.trans take-all*)
with *c1* **have** *take i* (*f k*) $\geq$ (*trace-hide* (*take i tt*) (*ev* ‘ *A*)) **by** (*simp add*: *le-list-def*)
with *c* **obtain** *t* **where** *d*:*f i* = (*trace-hide* (*take i tt*) (*ev* ‘ *A*))@*t*
by (*metis append.assoc le-list-def*)
then show *?thesis* **using** *in-set-takeD* **by** *fastforce*
qed
} note *ee*=*this*
{ fix *i*
have *finite* $\{$(*take i t*)$|t.\ t \in$ *range ff*$\}$
proof(*induct i*, *simp*)
case (*Suc i*)
have *ff*:$\{$*take* (*Suc i*) *t*$|t.\ t \in$ *range ff*$\} \subseteq \{$(*take i t*)$|t.\ t \in$ *range ff*$\} \cup$

```
                (⋃ e∈(set (f (Suc i)) ∪ (ev ‘ A)). {(take i t)@[e]|t. t ∈ range ff})
(is ?AA ⊆ ?BB)
          proof
            fix t
            assume t ∈ ?AA
            then obtain t′ where hh:t′ ∈ range ff ∧ t = take (Suc i) t′
              using finite-nat-set-iff-bounded-le by blast
            with ee[of t′] show t ∈ ?BB
              proof(cases length t′ > i)
                case True
                      hence ii:take (Suc i) t′ = take i t′ @ [t′!i] by (simp add:
take-Suc-conv-app-nth)
                with ee[of t′ Suc i] have t′!i ∈ set (f (Suc i)) ∪ (ev ‘ A) by (simp
add: hh)
                with ii hh show ?thesis by blast
              next
                case False
                hence take (Suc i) t′ = take i t′ by fastforce
                with hh show ?thesis by auto
              qed
          qed
        { fix e
          have {x @ [e] |x. ∃ t. x = take i t ∧ t ∈ range ff} = {take i t′ @ [e] |t′.
t′ ∈ range ff}
            by auto
          hence finite({(take i t′) @ [e] |t′. t′∈range ff})
            using finite-image-set[of - λt. t@[e], OF Suc] by auto
        } note gg=this
        have finite(set (f (Suc i)) ∪ (ev ‘ A)) using 1(1) by simp
       with ff gg Suc show ?case by (metis (no-types, lifting) finite-UN finite-Un
finite-subset)
      qed
    } note ff=this
    hence ∀ i. {take i t |t. t ∈ range ff} = {t. ∃ t′. t = take i (ff t′)} by auto
    with KoenigLemma[of range ff] ff ff2
    obtain f′ where gg:strict-mono (f′:: nat ⇒ ′a event list) ∧
                                    range f′ ⊆ {t. ∃ t′∈range ff. t ≤ t′} by auto
    { fix i
      from gg obtain n where aa:f′ i ≤ ff n by blast
      have ∃ t. f n = f 0 @ t by (metis 3a(4) le0 le-list-def strict-mono-less-eq)
       with mono-trace-hide[OF aa, of A] ff0 ff1 have ∃ t. trace-hide (f′ i) (ev ‘
A) ≤ f 0 @ t
        by (metis (mono-tags, lifting) someI-ex)
    } note zz=this
    {
      define ff′ where ff′ = (λi. trace-hide (f′ i) (ev ‘ A))
      with gg have mono ff′
        by (rule-tac monoI, simp add: mono-trace-hide strict-mono-less-eq)
      assume aa:∀ i. trace-hide (f′ i) (ev ‘ A) ≤ f 0
```

```
      with aa mono-constant have ∃ i. ∀ j≥i. ff' j = ff' i using ‹mono ff'› ff'-def
by blast
      then obtain m where bb:∀ j≥m. ff' j = ff' m by blast
      have ff' m ∈ 𝒟 (P \ A)
        proof(simp add:D-Hiding, rule-tac x=f' m in exI, rule-tac x=[] in exI,
intro conjI, simp, goal-cases)
        case 1
        from gg have f' m < f' (Suc m) by (meson lessI strict-monoD)
        moreover from gg obtain k where f' (Suc m) ≤ ff k by blast
       moreover have ff k ∈ 𝒯 P by (metis (mono-tags, lifting) ff0 ff1 someI-ex)
        ultimately show ?case
         by (metis NF-NT append-Nil2 front-tickFree-mono is-processT le-list-def
less-list-def)
      next
        case 2
        then show ?case unfolding ff'-def by simp
      next
        case 3
        then show ?case
         proof(rule disjI2, rule-tac x=λi. f' (m + i) in exI, simp-all, intro conjI
allI, goal-cases)
          case 1
          show ?case using gg[THEN conjunct1]
            by (rule-tac strict-monoI, simp add: strict-monoD)
        next
          case (2 i)
          from gg obtain k where f' (m + i) ≤ ff k by blast
          moreover from ff0 ff1 have ff k ∈ 𝒯 P by (metis (mono-tags, lifting)
someI-ex)
         ultimately have f' (m + i) ∈ 𝒯 P using is-processT3-ST-pref by blast
          then show ?case by (simp add: add.commute)
        next
          case (3 i)
           then show ?case using bb[THEN spec, of m+i, simplified] unfolding
ff'-def by assumption
        next
          case 4
          then show ?case by (metis Nat.add-0-right rangeI)
        qed
      qed
      with gg False have False
     by (metis (no-types, lifting) Falsebis aa append-Nil2 ff'-def front-tickFree-mono

                                  is-processT is-processT2-TR le-list-def)
    }
    with zz obtain m where hh:trace-hide (f' m) (ev ‘ A) ≥ f 0
      unfolding le-list-def by (metis append-eq-append-conv2)
    from ff0 ff1 gg show ?thesis
   proof(auto simp add:T-Hiding, rule-tac x=f' m in exI, rule-tac x=u in exI,
```

```
                    intro conjI, simp-all add:3(3), goal-cases)
      case 1
      hence f' m < f' (Suc m) by (meson lessI strict-monoD)
      moreover from gg obtain k where f' (Suc m) ≤ ff k by blast
      moreover have ff k ∈ 𝒯 P by (metis (mono-tags, lifting) ff0 ff1 someI-ex)
      ultimately show ?case
        by (metis NF-NT append-Nil2 front-tickFree-mono is-processT le-list-def
less-list-def)
    next
      case 2
      from gg obtain k where f' m ≤ ff k by blast
      with ff0 ff1 mono-trace-hide[of f' m] have trace-hide (f' m) (ev ‘ A) ≤ f k
        by (metis (mono-tags, lifting) someI-ex)
    with mono-trace-hide[OF this, of B] mono-trace-hide[OF hh, of B] 3(6)[THEN
spec, of k] 3(6)
      show ?case by (metis (full-types) dual-order.antisym filter-filter)
    next
      case 3 show ?case
       proof(rule disjI2, rule-tac x=λi. f' (m + i) in exI, simp-all, intro conjI
allI, goal-cases)
        case 1
        then show ?case by (metis Nat.add-0-right rangeI)
      next
        case 2 with 3(4) show ?case
          by (rule-tac strict-monoI, simp add: strict-monoD)
      next
        case (3 i)
        from gg obtain k where f' (m + i) ≤ ff k by blast
         moreover from ff0 ff1 have ff k ∈ 𝒯 P by (metis (mono-tags, lifting)
someI-ex)
        ultimately have f' (m + i) ∈ 𝒯 P using is-processT3-ST-pref by blast
        then show ?case by (simp add: add.commute)
      next
        case (4 i)
        from gg obtain k where f' (m + i) ≤ ff k by blast
        with ff0 ff1 mono-trace-hide[of f' (m + i)] have ll:trace-hide (f' (m + i))
(ev ‘ A) ≤ f k
          by (metis (mono-tags, lifting) someI-ex)
        { fix a b c assume (a::'a event list) ≤ b and b ≤ c and c ≤ a hence b
= c by auto}
        note jj=this
        from jj[OF mono-trace-hide[OF hh, of B],
                OF mono-trace-hide[THEN mono-trace-hide, of f' m f' (m + i) B
A,
                          OF gg[THEN conjunct1, THEN strict-mono-mono,
                          THEN monoD, of m m+i, simplified]]]
            mono-trace-hide[OF ll, of B]
        show ?case unfolding 3a(6) [THEN spec, of k] by simp
      qed
```

```
    qed
  qed
 qed
qed

lemma Hiding-Un-D: ‹finite A ⟹ 𝒟 ((P \ A) \ B) = 𝒟 (P \ (A ∪ B))›
 using Hiding-Un-D1 Hiding-Un-D2 by blast
```

5.6.2 Laws

```
lemma mono-Hiding-FD[simp]: ‹P ≤ Q ⟹ P \ A ≤ Q \ A›
 apply (auto simp: le-ref-def F-Hiding D-Hiding)
 using F-subset-imp-T-subset by blast+

lemmas mono-Hiding-ref = mono-Hiding

lemma Hiding-Un: ‹finite A ⟹ P \ (A ∪ B) = (P \ A) \ B›
proof (simp add:Process-eq-spec, intro conjI, unfold F-Hiding, goal-cases)
 case 1
 thus ?case (is {(s, X). ?A s X} ∪ {(s, X). ?B s} =
            {(s, X). (∃ t. (?C1 s t ∧ (t, X ∪ ev ‘ B) ∈ ?C2 ∪ ?C3))} ∪ {(s, X).
?D s})
 proof(unfold F-Hiding set-eq-subset Un-subset-iff, intro conjI, goal-cases)
   case 1
   then show ?case
   by (auto, metis (no-types) filter-filter image-Un sup-commute sup-left-commute)
 next
   case 2
   then show ?case
     by (rule-tac Un-mono[of {}, simplified], insert Hiding-Un-D[of A P B], simp
add: D-Hiding)
 next
   case 3
   have {(s, X). (∃ t. (?C1 s t ∧ (t, X ∪ ev ‘ B) ∈ ?C2))} ⊆ {(s, X). ?A s X}
   by (auto, metis (no-types) filter-filter image-Un sup-commute sup-left-commute)
   moreover have {(s, X). (∃ t. (?C1 s t ∧ (t, X ∪ ev ‘ B) ∈ ?C3))} ⊆ {(s, X).
?B s}
   proof(auto,goal-cases)
     case (1 ta u)
     then show ?case using Hiding-fronttickFree by blast
   next
     case (2 u f x)
     then show ?case
       apply(rule-tac x=f x in exI, rule-tac x=trace-hide u (ev ‘ B) in exI, auto)
        using Hiding-fronttickFree apply blast
       apply(erule-tac x=f in allE) by (metis (no-types) filter-filter rangeI)
   qed
   moreover have {(s, X). (∃ t. (?C1 s t ∧ (t, X ∪ ev ‘ B) ∈ ?C2 ∪ ?C3))} =
                          {(s, X). (∃ t. (?C1 s t ∧ (t, X ∪ ev ‘ B) ∈ ?C2 ))} ∪
```

```
                    {(s, X). (∃ t. (?C1 s t ∧ (t, X ∪ ev ‘ B) ∈ ?C3))} by
blast
    ultimately show ?case by (metis (no-types, lifting) Un-mono)
  next
    case 4
    then show ?case
      by (rule-tac Un-mono[of {}, simplified], insert Hiding-Un-D[of A P B], simp
add:D-Hiding)
  qed
next
  case 2
  then show ?case by (simp add: Hiding-Un-D)
qed

lemma Hiding-set-BOT: (⊥ \ A) = ⊥
  apply(auto simp add:Process-eq-spec D-Hiding F-Hiding F-UU D-UU)
        using Hiding-fronttickFree apply blast
       using front-tickFree-append Hiding-tickFree apply blast
      using front-tickFree-append Hiding-tickFree apply blast
    apply (metis (full-types) append-Nil filter.simps(1) tickFree-Nil tickFree-implies-front-tickFree)
    using front-tickFree-append Hiding-tickFree apply blast
   using front-tickFree-append Hiding-tickFree apply blast
  using tickFree-Nil by fastforce

lemma Hiding-set-STOP: (STOP \ A) = STOP
 apply(auto simp add:Process-eq-spec D-Hiding F-Hiding F-STOP D-STOP T-STOP)
  by (metis (full-types) lessI less-irrefl strict-mono-eq) +

lemma Hiding-set-SKIP: (SKIP \ A) = SKIP
 apply(auto simp add:Process-eq-spec D-Hiding F-Hiding F-SKIP D-SKIP T-SKIP
split:if-splits)
       apply (metis filter.simps(1) non-tickFree-tick)
     apply (metis (full-types) Hiding-tickFree n-not-Suc-n non-tickFree-tick strict-mono-eq)
    apply (metis (full-types) Hiding-tickFree n-not-Suc-n non-tickFree-tick strict-mono-eq)
    apply (metis event.distinct(1) filter.simps(1) imageE)
   apply (metis event.distinct(1) filter.simps(1) filter.simps(2) imageE)
  by (metis (full-types) Hiding-tickFree n-not-Suc-n non-tickFree-tick strict-mono-eq)

lemma Hiding-set-empty: (P \ {}) = P
 apply(auto simp add:Process-eq-spec D-Hiding F-Hiding is-processT7 is-processT8-S
strict-mono-eq)
 by (metis append-Nil2 front-tickFree-implies-tickFree front-tickFree-single is-processT

          nonTickFree-n-frontTickFree)
```

5.6.3 Multi-Operators laws

```
lemma Hiding-Ndet: (P ⊓ Q) \ A = ((P \ A) ⊓ (Q \ A))
proof(auto simp add:Process-eq-spec D-Hiding F-Hiding,
```

```
    simp-all add: F-Ndet T-Ndet D-Ndet D-Hiding F-Hiding, goal-cases)
  case (1 b t)
  then show ?case by blast
next
  case (2 b t u)
  then show ?case by blast
next
  case (3 b u f x)
  from 3(4) have A:infinite ({i. f i ∈ 𝒯 P}) ∨ infinite ({i. f i ∈ 𝒯 Q})
    using finite-nat-set-iff-bounded by auto
  hence (∀ i. f i ∈ 𝒯 P) ∨ (∀ i. f i ∈ 𝒯 Q)
    by (metis (no-types, lifting) 3(3) finite-nat-set-iff-bounded
          is-processT3-ST-pref mem-Collect-eq not-less strict-mono-less-eq)
  with A show ?case by (metis (no-types, lifting) 3(1,2,3,5) rangeI)
next
  case (4 a b)
  then show ?case by blast
next
  case (5 t u)
  then show ?case by blast
next
  case (6 u f x)
  from 6(4) have A:infinite ({i. f i ∈ 𝒯 P}) ∨ infinite ({i. f i ∈ 𝒯 Q})
    using finite-nat-set-iff-bounded by auto
  hence (∀ i. f i ∈ 𝒯 P) ∨ (∀ i. f i ∈ 𝒯 Q)
    by (metis (no-types, lifting) 6(3) finite-nat-set-iff-bounded
          is-processT3-ST-pref mem-Collect-eq not-less strict-mono-less-eq)
  with A show ?case by (metis (no-types, lifting) 6(1,2,3,5) rangeI)
next
  case (7 x)
  then show ?case by blast
qed

lemma Hiding-Mprefix-distr:
  (B ∩ A) = {} ⟹ ((Mprefix A P) \ B) = (Mprefix A (λx. ((P x) \ B)))
proof (auto simp add: Process-eq-spec,
    simp-all add: F-Mprefix T-Mprefix D-Mprefix D-Hiding F-Hiding,
    goal-cases)
  case (1 x b) then show ?case
  proof(elim exE disjE conjE, goal-cases)
    case (1 t)
    then show ?case by (simp add: inf-sup-distrib2)
  next
    case (2 t a)
    then show ?case
      by (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)
                                imageE list.sel(1) list.sel(3) list.collapse neq-Nil-conv)
  next
    case (3 t u a)
```

```
    hence hd t ∉ ev ‘ B by force
    with 3 havex = hd t # trace-hide (tl t) (ev ‘ B) @ u
      by (metis append-Cons filter.simps(2) list.exhaust-sel)
    with 3 show ?case by (metis list.distinct(1) list.sel(1) list.sel(3) tickFree-tl)
  next
    case (4 t u f)
    then obtain k where kk:t = f k by blast
    from 4 obtain a where f 1 ≠ [] ∧ ev a = hd (f 1) and aa:a ∈ A
      by (metis less-numeral-extra(1) nil-le not-less strict-mono-less-eq)
    with 4(1) 4(6)[THEN spec, of 0] 4(7)[THEN spec, of 1] have f 0 ≠ [] ∧ hd
(f 0) = ev a
      apply auto
        apply (metis (no-types, lifting) disjoint-iff-not-equal event.inject
                                    filter.simps(2) hd-Cons-tl imageE list.distinct(1))
     apply (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)

                                    hd-Cons-tl imageE list.distinct(1))
      by (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)
                                hd-Cons-tl imageE list.sel(1))
    with 4(1, 7) aa have nf: ∀ i. f i ≠ [] ∧ hd (f i) = ev a
     by (metis (mono-tags, opaque-lifting) 4(5) append-Cons le-list-def le-simps(2)
list.distinct(1)
          list.exhaust-sel list.sel(1) neq0-conv old.nat.distinct(1) strict-mono-less-eq)

   with 4(5) have sm:strict-mono (tl ∘ f) by (simp add: less-tail strict-mono-def)
    with 4 aa kk nf show ?case
      apply(rule-tac disjI2, intro conjI)
          apply (metis (no-types, lifting) Nil-is-append-conv disjoint-iff-not-equal
event.inject
                                    filter.simps(2) hd-Cons-tl imageE list.distinct(1))
     apply (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)

                                    hd-Cons-tl hd-append2 image-iff list.distinct(1)
list.sel(1))
      apply(rule-tac x=a in exI, intro conjI disjI2)
       apply (metis disjoint-iff-not-equal event.inject filter.simps(2)
                hd-Cons-tl hd-append2 image-iff list.distinct(1) list.sel(1))
     apply(rule-tac x=tl t in exI, rule-tac x=u in exI, intro conjI, simp-all)
        apply (metis tickFree-tl)
     apply (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)

                                    hd-Cons-tl imageE list.distinct(1) list.sel(3)
tl-append2)
      apply(subst disj-commute, rule-tac disjCI)
      apply(rule-tac x=tl ∘ f in exI, intro conjI)
         apply auto
      apply (metis (no-types, lifting) filter.simps(2) hd-Cons-tl list.sel(3))
      done
  qed
```

```
next
  case (2 x b)
  then show ?case proof(elim exE disjE conjE, goal-cases)
    case 1 then show ?case
      apply(rule-tac disjI1, rule-tac x=[] in exI)
      by (simp add: disjoint-iff-not-equal inf-sup-distrib2)
  next
    case (2 a t) then show ?case
      apply(rule-tac disjI1, rule-tac  x=(ev a)#t in exI)
      using list.collapse by fastforce
  next
    case (3 a t u)
    then show ?case
      apply(rule-tac disjI2, rule-tac  x=(ev a)#t in exI, rule-tac x=u in exI)
      using list.collapse by fastforce
  next
    case (4 a t u f)
    then show ?case
      apply(rule-tac disjI2)
      apply(rule-tac x=(ev a) # t in exI, rule-tac x=u in exI, intro conjI, simp)
        apply auto[1]
       using hd-Cons-tl apply fastforce
      apply(rule-tac disjI2, rule-tac x=λi. (ev a) # (f i) in exI)
      by (auto simp add: less-cons strict-mono-def)
  qed
next
  case (3 x) then show ?case
  proof(elim exE disjE conjE, goal-cases)
    case (1 t u a)
    hence aa: hd (trace-hide t (ev ‘ B)) = ev a ∧ trace-hide t (ev ‘ B) ≠ []
      by (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)
hd-Cons-tl
                                imageE list.distinct(1) list.sel(1))
    with 1 have  hd x = ev a ∧ x ≠ [] by simp
    with 1 show ?case
      apply(intro conjI, simp-all, rule-tac x=a in exI, simp)
      apply(rule-tac x=tl t in exI, rule-tac x=u in exI, intro conjI, simp-all)
       using tickFree-tl apply blast
        using aa by (metis (no-types, lifting) disjoint-iff-not-equal event.inject fil-
ter.simps(2)
                        hd-Cons-tl imageE list.sel(3) tl-append2)
  next
    case (2 t u f)
    then obtain k where kk:t = f k by blast
    from 2 obtain a where f 1 ≠ [] ∧ ev a = hd (f 1) and aa:a ∈ A
      by (metis less-numeral-extra(1) nil-le not-less strict-mono-less-eq)
    with 2(1) 2(6)[THEN spec, of 0] 2(7)[THEN spec, of 1] have f 0 ≠ [] ∧ hd
(f 0) = ev a
      apply auto
```

```
      apply (metis (no-types, lifting) disjoint-iff-not-equal event.inject
                                filter.simps(2) hd-Cons-tl imageE list.distinct(1))
    apply (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)

                                hd-Cons-tl imageE list.distinct(1))
     by (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)
                                hd-Cons-tl imageE list.sel(1))
   with 2(1, 7) aa have nf: ∀ i. f i ≠ [] ∧ hd (f i) = ev a
    by (metis (mono-tags, opaque-lifting) 2(5) append-Cons le-list-def le-simps(2)
list.distinct(1)
          list.exhaust-sel list.sel(1) neq0-conv old.nat.distinct(1) strict-mono-less-eq)

   with 2(5) have sm:strict-mono (tl ∘ f) by (simp add: less-tail strict-mono-def)
   from 2(1,4) nf aa kk have x1:x ≠ []
    by (metis Nil-is-append-conv disjoint-iff-not-equal event.inject filter.simps(2)
            hd-Cons-tl imageE list.distinct(1))
   with 2(1,4) nf aa kk have x2: hd (trace-hide t (ev ‘ B)) = ev a ∧ trace-hide
t (ev ‘ B) ≠ []
     by (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)
hd-Cons-tl
                                imageE list.distinct(1) list.sel(1))
   with 2(1,4) nf aa kk x1 have x3:hd x = ev a by simp
   with 2 aa kk nf sm x1  show ?case
     apply(intro conjI, simp-all)
    apply(rule-tac x=tl t in exI, rule-tac x=u in exI, intro conjI, simp-all)
       apply (metis tickFree-tl)
    apply (metis (no-types, lifting) disjoint-iff-not-equal event.inject filter.simps(2)

                                hd-Cons-tl imageE list.distinct(1) list.sel(3)
tl-append2)
     apply(subst disj-commute, rule-tac disjCI)
     apply(rule-tac x=tl ∘ f in exI, intro conjI)
        apply auto
      apply (metis (no-types, lifting) filter.simps(2) hd-Cons-tl list.sel(3))
     by (metis (no-types, lifting) filter.simps(2) hd-Cons-tl list.sel(3))
  qed
next
  case (4 x) then show ?case
  proof(elim exE disjE conjE, goal-cases)
    case (1 a t u)
    then show ?case
      apply(rule-tac  x=(ev a)#t in exI, rule-tac x=u in exI)
      using list.collapse by fastforce
  next
    case (2 a t u f)
    then show ?case
     apply(rule-tac x=(ev a) # t in exI, rule-tac x=u in exI, intro conjI, simp-all)
        apply auto[1]
       using hd-Cons-tl apply fastforce
```

```
    apply(rule-tac disjI2, rule-tac x=λi. (ev a) # (f i) in exI)
    by (auto simp add: less-cons strict-mono-def)
  qed
qed

lemma no-Hiding-read: (∀ y. c y ∉ B) ⟹ ((c?x → (P x)) \ B) = (c?x → ((P x)
\ B))
  by (simp add: read-def o-def, subst Hiding-Mprefix-distr, auto)

lemma no-Hiding-write0: a ∉ B ⟹ ((a → P) \ B) = (a → (P \ B))
  by (simp add: Hiding-Mprefix-distr write0-def)

lemma Hiding-write0: a ∈ B ⟹ ((a → P) \ B) = (P \ B)
proof (auto simp add: write0-def Process-eq-spec,
      simp-all add: F-Mprefix T-Mprefix D-Mprefix D-Hiding F-Hiding,
      goal-cases)
  case (1 x b)
  then show ?case
    apply(elim exE disjE conjE)
      apply (metis filter.simps(2) hd-Cons-tl image-eqI)
     apply (metis (no-types, lifting) filter.simps(2) image-eqI list.sel(1)
                                  list.sel(3) neq-Nil-conv tickFree-tl)
  proof(goal-cases)
    case (1 t u f)
   have fS: strict-mono (f ∘ Suc) by (metis 1(5) Suc-mono comp-def strict-mono-def)
    from 1 have aa:∀ i. f (Suc i) ≠ []
      by (metis (full-types) less-Suc-eq-le less-irrefl nil-le strict-mono-less-eq)
   with fS have sm:strict-mono (tl ∘ f ∘ Suc) by (simp add: less-tail strict-mono-def)
    with 1 aa show ?case
      apply(subst disj-commute, rule-tac disjCI, simp)
      apply(rule-tac x=tl (f 1) in exI, rule-tac x=u in exI, intro conjI, simp-all)
        apply (metis Hiding-tickFree imageE tickFree-tl)
      apply (metis (no-types, lifting) filter.simps(2) hd-Cons-tl image-eqI rangeE)
      apply(subst disj-commute, rule-tac disjCI)
      apply(rule-tac x=tl ∘ f ∘ Suc in exI, intro conjI)
         apply auto
         apply (metis (no-types, lifting) filter.simps(2) hd-Cons-tl list.sel(3))
      done
  qed
next
  case (2 aa b)
  then show ?case
    apply(elim exE disjE conjE)
      apply (metis (no-types, lifting) filter.simps(2) image-eqI list.distinct(1)
                                  list.sel(1) list.sel(3))
  proof(goal-cases)
    case (1 t u)
    then show ?case by (rule-tac disjI2, rule-tac x=(ev a)#t in exI, auto)
  next
```

```
    case (2 t u f)
    then show ?case
      apply(rule-tac disjI2)
     apply(rule-tac x=(ev a) # t in exI, rule-tac x=u in exI, intro conjI, simp-all)
      apply(rule-tac disjI2)
      apply(rule-tac x=λi. (ev a) # (f i) in exI, intro conjI)
      by (auto simp add: less-cons strict-mono-def)
  qed
next
  case (3 x)
  then show ?case
    apply(elim exE disjE conjE)
     apply(rule-tac x=tl t in exI, rule-tac x=u in exI, intro conjI, simp-all)
    using tickFree-tl apply blast
     apply (metis filter.simps(2) hd-Cons-tl image-eqI)
  proof(goal-cases)
    case (1 t u f)
   have fS: strict-mono (f ∘ Suc) by (metis 1(5) Suc-mono comp-def strict-mono-def)
    from  1 have aa:∀ i. f (Suc i) ≠ []
      by (metis (full-types) less-Suc-eq-le less-irrefl nil-le strict-mono-less-eq)
   with fS have sm:strict-mono (tl ∘ f ∘ Suc) by (simp add: less-tail strict-mono-def)
    with 1 aa show ?case
      apply(rule-tac x=tl (f 1) in exI, rule-tac x=u in exI, intro conjI, simp-all)
        apply (metis Hiding-tickFree imageE tickFree-tl)
      apply (metis (no-types, lifting) filter.simps(2) hd-Cons-tl image-eqI rangeE)
      apply(subst disj-commute, rule-tac disjCI)
      apply(rule-tac x=tl ∘ f ∘ Suc in exI, intro conjI)
         apply auto
         apply (metis (no-types, lifting) filter.simps(2) hd-Cons-tl list.sel(3))
      done
  qed
next
  case (4 x)
  then show ?case
    apply(elim exE disjE conjE)
  proof(goal-cases)
    case (1 t u)
    then show ?case by (rule-tac x=(ev a)#t in exI, auto)
  next
    case (2 t u f)
    then show ?case
     apply(rule-tac x=(ev a) # t in exI, rule-tac x=u in exI, intro conjI, simp-all)
      apply(rule-tac disjI2)
      apply(rule-tac x=λi. (ev a) # (f i) in exI, intro conjI)
      by (auto simp add: less-cons strict-mono-def)
  qed
qed

lemma no-Hiding-write: (∀ y. c y ∉ B) ⟹ ((c!a → P) \ B) = (c!a → (P \ B))
```

by(*simp add*: *write-def*, *subst Hiding-Mprefix-distr*, *auto*)

lemma *Hiding-write*: $(c\ a) \in B$ ==> $((c!a \rightarrow P) \setminus B) = (P \setminus B)$
by (*simp add*: *write-def Hiding-write0 Mprefix-singl*)

5.7 The Sync Operator Laws

5.7.1 Preliminaries

lemma *SyncTlEmpty*:*a setinterleaves* $((\text{[]}, u), A) \Longrightarrow$ *tl a setinterleaves* $((\text{[]}, tl\ u), A)$
by (*case-tac u*, *simp*, *case-tac a*, *simp-all split*:*if-splits*)

lemma *SyncHd-Tl*:
a setinterleaves $((t, u), A) \wedge hd\ t \in A \wedge hd\ u \notin A$
$\Longrightarrow hd\ a = hd\ u \wedge$ *tl a setinterleaves* $((t, tl\ u), A)$
by (*case-tac u*) (*case-tac t*, *auto split*:*if-splits*)+

lemma *SyncHdAddEmpty*:
$(tl\ a)$ *setinterleaves* $((\text{[]}, u), A) \wedge hd\ a \notin A \wedge a \neq \text{[]}$
$\Longrightarrow a$ *setinterleaves* $((\text{[]}, hd\ a \# u), A)$
using *hd-Cons-tl* **by** *fastforce*

lemma *SyncHdAdd*:
$(tl\ a)$ *setinterleaves* $((t, u), A) \wedge hd\ a \notin A \wedge hd\ t \in A \wedge a \neq \text{[]}$
$\Longrightarrow a$ *setinterleaves* $((t, hd\ a \# u), A)$
by (*case-tac a*, *simp*, *case-tac t*, *auto*)

lemmas *SyncHdAdd1* = *SyncHdAdd*[*of* $a\#r$, *simplified*] **for** *a r*

lemma *SyncSameHdTl*:
a setinterleaves $((t, u), A) \wedge hd\ t \in A \wedge hd\ u \in A$
$\Longrightarrow hd\ t = hd\ u \wedge hd\ a = hd\ t \wedge (tl\ a)$ *setinterleaves* $((tl\ t, tl\ u), A)$
by (*case-tac u*) (*case-tac t*, *auto split*:*if-splits*)+

lemma *SyncSingleHeadAdd*:
a setinterleaves $((t, u), A) \wedge h \notin A$
$\Longrightarrow (h\#a)$ *setinterleaves* $((h\#t, u), A)$
by (*case-tac u*, *auto split*:*if-splits*)

lemma *TickLeftSync*:
$tick \in A \wedge$ *front-tickFree* $t \wedge s$ *setinterleaves* $(([tick], t), A) \Longrightarrow s = t \wedge last\ t = tick$
proof(*induct t arbitrary*: *s*)
case *Nil*
then show *?case* **by** (*simp add*: *Nil.prems*)
next
case (*Cons a t*)
then show *?case*

```
    apply (auto split:if-splits)
      using emptyLeftProperty apply blast
    apply (metis last-ConsR last-snoc nonTickFree-n-frontTickFree tickFree-Cons)
    by (metis append-Cons append-Nil front-tickFree-mono)+
qed

lemma EmptyLeftSync:s setinterleaves (([], t), A) ⟹ s = t ∧ set t ∩ A = {}
  by (meson Int-emptyI emptyLeftNonSync emptyLeftProperty)

lemma tick-T-F:t@[tick] ∈ 𝒯 P ⟹ (t@[tick], X) ∈ ℱ P
  using append-T-imp-tickFree is-processT5-S7 by force

lemma event-set: (e::'a event) ∈ insert tick (range ev)
  by (metis event.exhaust insert-iff rangeI)

lemma Mprefix-Sync-distr1-D:
   A ⊆ S
    ⟹ B ⊆ S
    ⟹ 𝒟 (Mprefix A P ⟦S⟧ Mprefix B Q) = 𝒟 (□ x ∈ A ∩ B → (P x ⟦S⟧ Q
x))
  apply(auto,simp-all add:D-Sync F-Sync F-Mprefix T-Mprefix D-Mprefix)
   apply(elim exE disjE conjE)
          apply (metis (no-types, lifting) Sync.sym empty-iff image-iff insertI2
                    list.exhaust-sel setinterleaving.simps(2) subsetCE)
proof(goal-cases)
  case (1 x t u r v a aa)
  from 1(1,2,6,8,11,13) have aa1: hd t∈insert tick (ev ‘ S)∧hd u∈insert tick (ev
‘ S)
    by blast
  from 1(5,6,7,8,11,13) aa1 have aa2:  hd x ∈ ev ‘ (A ∩ B)
    by (metis (no-types, lifting) IntI empty-setinterleaving event.inject hd-append2
image-iff SyncSameHdTl)
  then show ?case
   using 1(3,4,5,6,7,9,10,13,14) by (metis (no-types, lifting)  Nil-is-append-conv
aa1
                                        empty-setinterleaving event.inject hd-append2
                                        SyncSameHdTl tickFree-tl tl-append2)
next
  case (2 x t u r v a)
  then show ?case
   by (metis (no-types, lifting) Sync.si-empty3 equals0D imageE image-eqI insertI2
list.exhaust-sel subsetCE)
next
  case (3 x t u r v a aa)
  from 3(1,2,6,8,11,13) have aa1: hd t∈insert tick (ev ‘ S)∧hd u∈insert tick (ev
‘ S)
    by blast
  from 3(5,6,7,8,11,13) aa1 have aa2:  hd x ∈ ev ‘ (A ∩ B)
    by (metis (no-types, lifting) IntI empty-setinterleaving event.inject hd-append2
```

image-iff SyncSameHdTl)
then show *?case*
using *3*(*3*,*4*,*5*,*6*,*7*,*9*,*10*,*13*,*14*)
by (*metis* (*no-types*, *lifting*) *Nil-is-append-conv aa1 empty-setinterleaving event.inject*
hd-append2 SyncSameHdTl tickFree-tl tl-append2)
next
case (*4 x t u r v a*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *Sync.si-empty3 equals0D imageE image-eqI insertI2 list.exhaust-sel subsetCE*)
next
case (*5 x t u r v a aa*)
from *5*(*1*,*2*,*6*,*8*,*11*,*13*) **have** *aa1*: *hd t*∈*insert tick* (*ev* ‘ *S*)∧*hd u*∈*insert tick* (*ev* ‘ *S*)
by *blast*
from *5*(*5*,*6*,*7*,*8*,*11*,*13*) *aa1* **have** *aa2*: *hd x* ∈ *ev* ‘ (*A* ∩ *B*)
by (*metis* (*no-types*, *lifting*) *IntI empty-setinterleaving event.inject hd-append2 image-iff SyncSameHdTl*)
then show *?case*
using *5*(*3*,*4*,*5*,*6*,*7*,*9*,*10*,*13*,*14*) **by** (*metis aa1 append-Nil2 empty-setinterleaving event.inject SyncSameHdTl*)
next
case (*6 x t u r v a*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *Sync.si-empty3 equals0D imageE image-eqI insertI2 list.exhaust-sel subsetCE*)
next
case (*7 x t u r v a aa*)
from *7*(*1*,*2*,*6*,*8*,*11*,*13*) **have** *aa1*: *hd t*∈*insert tick* (*ev* ‘ *S*)∧*hd u*∈*insert tick* (*ev* ‘ *S*)
by *blast*
from *7*(*5*,*6*,*7*,*8*,*11*,*13*) *aa1* **have** *aa2*: *hd x* ∈ *ev* ‘ (*A* ∩ *B*)
by (*metis* (*no-types*, *lifting*) *IntI empty-setinterleaving event.inject hd-append2 image-iff SyncSameHdTl*)
then show *?case*
using *7*(*3*,*4*,*5*,*6*,*7*,*9*,*10*,*13*,*14*) **by** (*metis aa1 append-Nil2 empty-setinterleaving event.inject SyncSameHdTl*)
next
case (*8 x*)
then show *?case*
apply(*elim exE disjE conjE*)
proof(*goal-cases*)
case (*1 a t u r v*)
obtain *r1 t1 u1* **where** *aa0*: *r1*=*hd x*#*r*∧*t1*=*hd x*#*t*∧*u1*=*hd x*#*u*
by *auto*
from *1*(*3*,*4*,*5*,*7*,*8*,*9*) **have** *aa1*: *tickFree r1*∧*x* = *r1* @ *v*
by (*metis Cons-eq-appendI aa0 event.distinct*(*1*) *list.exhaust-sel tickFree-Cons*)
from *1*(*2*,*4*,*9*) **have** *aa2*: *r1 setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *S*))∧*t1*

≠ []
using *aa0 subsetCE* **by** *auto*
from *1(4,5,10,11) aa0*
have *aa3*: (*tl t1*) ∈ 𝒟 (*P a*) ∧ (*tl u1*) ∈ 𝒯 (*Q a*) ∧ *ev a* = *hd t1* ∧ *ev a* = *hd u1* ∧
hd t1 ∈ *ev* ‘ *A*∧ *hd u1* ∈ *ev* ‘ *B*
by *auto*
then show *?case*
using *1(6) aa1 aa2* **by** *fastforce*
next
case (*2 a t u r v*)
obtain *r1 t1 u1* **where** *aa0*: *r1*=*hd x*#*r*∧*t1*=*hd x*#*t*∧*u1*=*hd x*#*u*
by *auto*
from *2(3,4,5,7,8,9)* **have** *aa1*: *tickFree r1*∧*x* = *r1* @ *v*
by (*metis Cons-eq-appendI aa0 event.distinct(1) list.exhaust-sel tickFree-Cons*)
from *2(2,4,9)* **have** *aa2*: *r1 setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *S*))∧*t1* ≠ []
using *aa0 subsetCE* **by** *auto*
from *2(4,5,10,11) aa0*
have *aa3*: (*tl t1*) ∈ 𝒟 (*Q a*) ∧ (*tl u1*) ∈ 𝒯 (*P a*) ∧ *ev a* = *hd t1* ∧ *ev a* = *hd u1* ∧
hd t1 ∈ *ev* ‘ *A*∧ *hd u1* ∈ *ev* ‘ *B*
by *auto*
then show *?case*
using *2(6) aa1 aa2* **by** *fastforce*
next
case (*3 a t u r v*)
obtain *r1 t1 u1* **where** *aa0*: *r1*=*hd x*#*r*∧*t1*=*hd x*#*t*∧*u1*=*hd x*#*u*
by *auto*
from *3(2,4,9)* **have** *aa2*: *r1 setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *S*))∧*t1* ≠ []
using *aa0 subsetCE* **by** *auto*
from *3(3,4,5,7,8,10,11) aa0*
have *aa3*: (*tl t1*) ∈ 𝒟(*P a*)∧(*tl u1*) ∈ 𝒯 (*Q a*)∧ *ev a* = *hd t1*∧*ev a* = *hd u1* ∧
hd t1 ∈ *ev* ‘ *A*∧ *hd u1* ∈ *ev* ‘ *B*∧*x* = *r1* @ *v*
by (*metis* (*no-types*, *lifting*) *Int-lower1 Int-lower2 append-Nil2 imageE image-eqI*
list.exhaust-sel list.sel(1) list.sel(3) subsetCE)
then show *?case*
by (*metis* (*no-types*, *lifting*) *3(6) 3(7) aa2 event.inject imageE*)
next
case (*4 a t u r v*)
obtain *r1 t1 u1* **where** *aa0*: *r1*=*hd x*#*r*∧*t1*=*hd x*#*t*∧*u1*=*hd x*#*u*
by *auto*
from *4(2,4,9)* **have** *aa2*: *r1 setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *S*))∧*t1* ≠ []
using *aa0 subsetCE* **by** *auto*
from *4(3,4,5,7,8,10,11) aa0*
have *aa3*: (*tl t1*) ∈ 𝒟 (*Q a*)∧(*tl u1*) ∈ 𝒯 (*P a*) ∧ *ev a* = *hd t1*∧*ev a* = *hd u1*∧

```
            hd t1 ∈ ev ‘ A∧ hd u1 ∈ ev ‘ B∧x = r1 @ v
      by (metis (no-types, lifting)  Int-lower1 Int-lower2 append-Nil2 imageE im-
age-eqI
            list.exhaust-sel list.sel(1) list.sel(3) subsetCE)
    then show ?case
      by (metis (no-types, lifting) 4(6) 4(7) aa2 event.inject imageE)
  qed
qed

lemma Hiding-interleave:
  A ∩ C = {}
  ⟹ r setinterleaves ((t, u), C)
  ⟹ (trace-hide r A) setinterleaves ((trace-hide t A, trace-hide u A), C)
proof(induct r arbitrary:t u)
  case Nil
  then show ?case
    using EmptyLeftSync empty-setinterleaving by fastforce
next
  case (Cons a r)
  then show ?case
    apply(cases t) using EmptyLeftSync apply fastforce
    apply(cases u) apply fastforce
  proof(simp split:if-splits, goal-cases)
    case (1 a list lista)
    then show ?case by fastforce
  next
    case (2 a list aa lista)
    then show ?case
      apply(erule-tac disjE)
       using Cons(1)[of list u]
       apply (metis (no-types, lifting) filter.simps(2) SyncSingleHeadAdd)
      using Cons(1)[of t lista]
      by (metis (no-types, lifting) Sync.sym filter.simps(2) SyncSingleHeadAdd)
  next
    case (3 a list lista)
    then show ?case by fastforce
  qed
qed

lemma non-Sync-interleaving:
  (set t ∪ set u) ∩ C = {} ⟹ setinterleaving (t, C, u) ≠ {}
proof(induct t arbitrary:u)
  case Nil
  then show ?case
   by (metis Un-empty-left disjoint-iff-not-equal emptyLeftSelf empty-iff empty-set)
next
  case (Cons a t)
  then show ?case
```

```
  proof(induct u)
    case Nil
    then show ?case using Sync.sym by auto
  next
    case (Cons a u)
    then show ?case by auto
  qed
qed

lemma interleave-Hiding:
  A ∩ C = {}
  ⟹ ra setinterleaves ((trace-hide t A, trace-hide u A), C)
  ⟹ ∃ r. ra = trace-hide r A ∧ r setinterleaves ((t, u), C)
proof(induct length t + length u arbitrary:ra t u rule:nat-less-induct)
  case Ind:1
  then show ?case
  proof (cases t)
    case Nilt: Nil
    from Ind(2) have a:set (trace-hide u A) ∩ C = {} ⟹ set u ∩ C = {} by
auto
    hence b: u setinterleaves ((t, u), C)
      by (metis (no-types, lifting) Ind(3) EmptyLeftSync Nilt disjoint-iff-not-equal
            emptyLeftSelf filter.simps(1))
    then show ?thesis
      apply(rule-tac x=u in exI)
      using EmptyLeftSync[of ra C trace-hide u A] a b Cons Nilt Ind(3) by auto
  next
    case Const: (Cons ta tlist)
    then show ?thesis
    proof(cases u)
      case Nilu: Nil
      from Ind(2) have a:set (trace-hide t A) ∩ C = {} ⟹ set t ∩ C = {} by
auto
      hence b: t setinterleaves ((t, u), C)
        by (metis Ind(3) Nilu EmptyLeftSync disjoint-iff-not-equal emptyLeftSelf
filter.simps(1) Sync.sym)
      then show ?thesis
        apply(rule-tac x=t in exI)
       using EmptyLeftSync[of ra C trace-hide t A] a b Ind Nilu by (metis Sync.sym
filter.simps(1))
    next
      case Consu: (Cons ua ulist)
      with Const Ind(2,3) show ?thesis
      proof(auto split:if-splits simp del:setinterleaving.simps, goal-cases)
        case 1
        then show ?case
        proof (auto, goal-cases)
          case (1 raa)
            with Ind(1)[THEN spec, of length tlist + length u, simplified Const,
```

simplified,
THEN spec, *THEN spec*, *of tlist u*, *simplified Ind*, *simplified*,
THEN spec, *of raa*] **obtain** *r* **where**
raa = *trace-hide r A* ∧ *r setinterleaves* ((*tlist*, *u*), *C*) **by** *auto*
then show *?case* **using** *1*(*4*) *Consu* **by** *force*
next
case (*2 raa*)
with *Ind*(*1*)[*THEN spec*, *of length t* + *length ulist*, *simplified Consu*, *simplified*,
THEN spec, *THEN spec*, *of t ulist*, *simplified Ind*, *simplified*,
THEN spec, *of raa*] **obtain** *r* **where**
raa = *trace-hide r A* ∧ *r setinterleaves* ((*t*, *ulist*), *C*) **by** *auto*
then show *?case* **using** *2*(*2*) *2*(*4*) **by** *force*
next
case (*3 raa*)
with *Ind*(*1*)[*THEN spec*, *of length tlist* + *length ulist*, *simplified Const Consu*, *simplified*,
THEN spec, *THEN spec*, *of tlist ulist*, *simplified Ind*, *simplified*,
THEN spec, *of raa*] **obtain** *r* **where**
raa = *trace-hide r A* ∧ *r setinterleaves* ((*tlist*, *ulist*), *C*) **by** *auto*
then show *?case* **using** *3*(*4*) **by** *force*
next
case (*4 raa*)
with *Ind*(*1*)[*THEN spec*, *of length t* + *length ulist*, *simplified Consu*, *simplified*,
THEN spec, *THEN spec*, *of t ulist*, *simplified Ind*, *simplified*,
THEN spec, *of raa*] **obtain** *r* **where**
raa = *trace-hide r A* ∧ *r setinterleaves* ((*t*, *ulist*), *C*) **by** *auto*
then show *?case* **using** *4*(*5*) *Const* **by** *force*
next
case (*5 raa*)
with *Ind*(*1*)[*THEN spec*, *of length tlist* + *length u*, *simplified Const*, *simplified*,
THEN spec, *THEN spec*, *of tlist u*, *simplified Ind*, *simplified*,
THEN spec, *of raa*] **obtain** *r* **where**
raa = *trace-hide r A* ∧ *r setinterleaves* ((*tlist*, *u*), *C*) **by** *auto*
then show *?case* **using** *5*(*4*) *Consu* **by** *force*
next
case (*6 raa*)
with *Ind*(*1*)[*THEN spec*, *of length t* + *length ulist*, *simplified Consu*, *simplified*,
THEN spec, *THEN spec*, *of t ulist*, *simplified Ind*, *simplified*,
THEN spec, *of raa*] **obtain** *r* **where**
raa = *trace-hide r A* ∧ *r setinterleaves* ((*t*, *ulist*), *C*) **by** *auto*
then show *?case* **using** *6*(*5*) *Const* **by** *force*
next
case (*7 raa*)
with *Ind*(*1*)[*THEN spec*, *of length tlist* + *length u*, *simplified Const*, *simplified*,

```
                THEN spec, THEN spec, of tlist u, simplified Ind, simplified,
                THEN spec, of raa] obtain r where
        raa = trace-hide r A ∧ r setinterleaves ((tlist, u), C) by auto
      then show ?case using 7(4) Consu by force
    qed
  next
    case 2
      with Ind(1)[THEN spec, of length t + length ulist, simplified Consu,
simplified,
                THEN spec, THEN spec, of t ulist, simplified Ind, simplified,
                THEN spec, of ra] obtain r where
      ra = trace-hide r A ∧ r setinterleaves ((t, ulist), C) by auto
    then show ?case
      apply(rule-tac x=ua#r in exI)
      using 2(5) Const Ind.prems(1) by auto
  next
    case 3
  with Ind(1)[THEN spec, of length tlist + length u, simplified Const, simplified,

                THEN spec, THEN spec, of tlist u, simplified Ind, simplified,
                THEN spec, of ra] obtain r where
      ra = trace-hide r A ∧ r setinterleaves ((tlist, u), C) by auto
    then show ?case
      apply(rule-tac x=ta#r in exI)
      using 3(4) Consu Ind.prems(1) by auto
  next
    case 4
      with Ind(1)[THEN spec, of length tlist + length ulist, simplified Const
Consu, simplified,
                THEN spec, THEN spec, of tlist ulist, simplified Ind, simplified,
                THEN spec, of ra] obtain r where
      ra = trace-hide r A ∧ r setinterleaves ((tlist, ulist), C) by auto
    then show ?case
      apply(rule-tac x=ta#ua#r in exI)
      using 4(4,5) Consu Const Ind.prems(1) apply auto
       using SyncSingleHeadAdd apply blast
      by (metis Sync.sym SyncSingleHeadAdd)
  qed
 qed
qed
qed

lemma interleave-size:
  s setinterleaves ((t,u), C) ⟹ length s = length t + length u − length(filter (λx.
x∈C) t)
proof(induct s arbitrary:t u)
  case Nil
  then show ?case using EmptyLeftSync empty-setinterleaving by fastforce
next
```

```
  case (Cons a list)
  then show ?case
    apply(cases t) using emptyLeftProperty apply fastforce
    apply(cases u)
     apply (metis (no-types, lifting) Sync.sym add-diff-cancel-right' emptyLeftNon-Sync
                emptyLeftProperty filter-empty-conv)
    proof(auto split:if-splits, goal-cases 11 12 13 14 15)
      case (11 tlist ulist)
      then show ?case
        by (metis (no-types, lifting) Suc-diff-le le-add1 length-filter-le order-trans)
    next
      case (12 ta tlist ulist)
      with 12(3)[rule-format, of ta#tlist ulist] show ?case
        by simp (metis Suc-diff-le le-add1 length-filter-le order-trans)
    next
      case (13 tlist ua ulist)
      with 13(3)[rule-format, of tlist ua#ulist] show ?case
       by simp (metis Suc-diff-le le-less-trans length-filter-le less-SucI less-Suc-eq-le
less-add-Suc1)
    next
      case (14 ta tlist ulist)
      with 14(3)[rule-format, of ta#tlist ulist] show ?case
       by simp (metis Suc-diff-le le-less-trans length-filter-le less-SucI less-Suc-eq-le
less-add-Suc1)
    next
      case (15 tlist ua ulist)
      with 15(3)[rule-format, of tlist ua#ulist] show ?case
       by simp (metis Suc-diff-le le-less-trans length-filter-le less-SucI less-Suc-eq-le
less-add-Suc1)
    qed
qed

lemma interleave-eq-size:
  s setinterleaves ((t,u), C) ⟹ s' setinterleaves ((t,u), C) ⟹ length s = length
s'
  by (simp add: interleave-size)

lemma interleave-set: s setinterleaves ((t,u), C) ⟹ set t ∪ set u ⊆ set s
proof(induct s arbitrary: t u)
  case Nil
  then show ?case using EmptyLeftSync empty-setinterleaving by blast
next
  case (Cons a s)
  then show ?case
    apply(cases t) using emptyLeftProperty apply fastforce
      apply(cases u) apply (metis Sync.sym Un-empty-right emptyLeftProperty
empty-set eq-refl)
    by (auto simp add: subset-iff split:if-splits)
```

qed

lemma *interleave-order*: *s setinterleaves* ((*t1*@*t2*,*u*), *C*) $\Longrightarrow$ *set*(*t2*) $\subseteq$ *set*(*drop* (*length t1*) *s*)
proof(*induct s arbitrary*: *t1 t2 u*)
  case *Nil*
  then show *?case* **using** *empty-setinterleaving* **by** *auto*
next
  case (*Cons a s*)
  then show *?case*
  apply(*cases t1*) **using** *append-self-conv2 interleave-set* **apply** *fastforce*
  apply(*cases u*) **apply** (*metis EmptyLeftSync Sync.sym append-eq-conv-conj order-refl*)
  proof (*auto simp add*: *subset-iff split*:*if-splits*, *goal-cases*)
    case (*1 a list lista t*)
    then show *?case* **using** *Cons*(*1*)[*of a#list t2 lista*, *simplified*, *OF 1*(*6*)]
     by (*meson Suc-n-not-le-n contra-subsetD nat-le-linear set-drop-subset-set-drop*)
  next
    case (*2 a list lista t*)
    then show *?case* **using** *Cons*(*1*)[*of a#list t2 lista*, *simplified*, *OF 2*(*7*)]
     by (*meson Suc-n-not-le-n contra-subsetD nat-le-linear set-drop-subset-set-drop*)
  qed
qed

lemma *interleave-append*:
  s setinterleaves ((*t1*@*t2*,*u*), *C*)
   $\Longrightarrow \exists$ *u1 u2 s1 s2*. *u* = *u1*@*u2* $\wedge$ *s* = *s1*@*s2* $\wedge$
                s1 setinterleaves ((*t1*,*u1*), *C*) $\wedge$ *s2 setinterleaves* ((*t2*,*u2*), *C*)
proof(*induct s arbitrary*: *t1 t2 u*)
  case *Nil*
  then show *?case* **using** *empty-setinterleaving* **by** *fastforce*
next
  case (*Cons a s*)
  then show *?case*
  apply(*cases t1*) **using** *append-self-conv2 interleave-set* **apply** *fastforce*
  apply(*cases u*) **apply**(*exI* [], *exI* []) **apply** *auto*[*1*]
    apply (*metis* (*no-types*, *opaque-lifting*) *Nil-is-append-conv append-Cons*)
  proof (*auto simp add*: *subset-iff split*:*if-splits*, *goal-cases*)
    case (*1 list lista*)
    with *Cons*(*1*)[*of list t2 lista*, *simplified*, *OF 1*(*5*)] **show** *?case*
    proof(*elim exE conjE*, *goal-cases*)
      case (*1 u1 u2 s1 s2*)
      then show *?case*
        by (*exI a#u1*, *exI u2*, *simp*) (*metis append-Cons*)
    qed
  next
    case (*2 aa list lista*)
    with *Cons*(*1*)[*of aa#list t2 lista*, *simplified*, *OF 2*(*6*)] **show** *?case*
    proof(*elim exE conjE*, *goal-cases*)

```
      case (1 u1 u2 s1 s2)
      then show ?case
        by (exI a#u1, exI u2, simp) (metis append-Cons)
    qed
  next
    case (3 list aa lista)
    with Cons(1)[of list t2 aa#lista, simplified, OF 3(6)] show ?case
    proof(elim exE conjE, goal-cases)
      case (1 u1 u2 s1 s2)
      then show ?case
        apply (exI u1, exI u2, simp) by (metis append-Cons SyncSingleHeadAdd)
    qed
  next
    case (4 aa list lista)
    with Cons(1)[of aa#list t2 lista, simplified, OF 4(6)] show ?case
    proof(elim exE conjE, goal-cases)
      case (1 u1 u2 s1 s2)
      then show ?case
        by (exI a#u1, exI u2, simp) (metis append-Cons)
    qed
  next
    case (5 list aa lista)
    with Cons(1)[of list t2 aa#lista, simplified, OF 5(6)] show ?case
    proof(elim exE conjE, goal-cases)
      case (1 u1 u2 s1 s2)
      then show ?case
        apply (exI u1, exI u2, simp) by (metis append-Cons SyncSingleHeadAdd)
    qed
  qed
qed

lemma interleave-append-sym:
  s setinterleaves ((t,u1@u2), C)
   ⟹ ∃ t1 t2 s1 s2. t = t1@t2 ∧ s = s1@s2 ∧
                    s1 setinterleaves ((t1,u1), C) ∧ s2 setinterleaves ((t2,u2), C)
  by (metis (no-types) Sync.sym interleave-append)

lemma interleave-append-tail:
   s setinterleaves ((t1,u), C) ⟹ (set t2) ∩ C = {} ⟹ (s@t2) setinterleaves
((t1@t2,u), C)
proof(induct s arbitrary: t1 t2 u)
  case Nil
  then show ?case by (metis Set.set-insert Sync.sym append-Nil disjoint-insert(2)

                          emptyLeftSelf empty-setinterleaving)
next
  case (Cons a s)
  then show ?case
    apply(cases u) using EmptyLeftSync Sync.sym apply fastforce
```

```
    apply(cases t1, cases t2) apply simp apply fastforce
  proof(auto, goal-cases)
    case (1 list lista)
    with 1(1)[OF 1(7) 1(2)] show ?case by simp
  next
    case (2 list aa lista)
    with 2(1)[OF 2(2)] show ?case by simp
  next
    case (3 list aa lista)
    with 3(1)[OF 3(9) 3(2)] show ?case by simp
  next
    case (4 list aa lista)
    with 4(1)[OF 4(9) 4(2)] show ?case by simp
  qed
qed

lemma interleave-append-tail-sym:
  s setinterleaves ((t,u1), C) ⟹ (set u2) ∩ C = {} ⟹ (s@u2) setinterleaves
((t,u1@u2), C)
  by (metis (no-types) Sync.sym interleave-append-tail)

lemma interleave-assoc-1:
  tu setinterleaves ((t, u), A)
  ⟹ tuv setinterleaves ((tu, v), A)
  ⟹ ∃ uv. uv setinterleaves ((u, v), A) ∧ tuv setinterleaves ((t, uv), A)
proof(induct tuv arbitrary: t tu u v)
  case Nil
  then show ?case using EmptyLeftSync empty-setinterleaving by blast
next
  case Cons-tuv:(Cons a tuv)
  then show ?case
  proof(cases t)
    case Nil-t:Nil
    with Cons-tuv(2) have *:tu = u using EmptyLeftSync by blast
    show ?thesis
    proof(cases u)
      case Nil-u:Nil
      with Nil-t Cons-tuv show ?thesis using EmptyLeftSync by fastforce
    next
      case Cons-u:(Cons ua ulist)
      with Nil-t Cons-tuv(2) have ua ∉ A by force
      show ?thesis
      proof(cases v)
        case Nil-v:Nil
        with * Nil-t Cons-tuv(2,3) Cons-u show ?thesis using Sync.sym by blast
      next
        case Cons-v:(Cons va vlist)
         with * Nil-t Cons-tuv(2,3) Cons-u show ?thesis apply(exI a#tuv, auto
split:if-splits)
```

```
        using Cons-tuv.hyps Cons-tuv.prems(1) emptyLeftProperty by blast+
      qed
    qed
  next
    case Cons-t:(Cons ta tlist)
    show ?thesis
    proof(cases u)
      case Nil-u:Nil
      with Cons-t Cons-tuv show ?thesis
        by (metis Sync.sym emptyLeftProperty emptyLeftSelf empty-set equals0D
ftf-Sync21)
    next
      case Cons-u:(Cons ua ulist)
      show ?thesis
      proof(cases v)
        case Nil-v:Nil
        with Cons-tuv(3) have a # tuv = tu by (simp add: Nil-v EmptyLeftSync
Sync.sym)
          with Nil-v Cons-u Cons-t Cons-tuv show ?thesis apply(exI u, auto
split:if-splits)
          apply (metis Cons-t Nil-v Sync.sym emptyLeftNonSync list.set-intros(1))
          using Cons-tuv(1)[of tuv tlist u]
          apply(metis (no-types, lifting) Sync.sym emptyLeftNonSync emptyLeftSelf
list.sel(3) SyncTlEmpty)
        by (metis Cons-t Sync.sym emptyLeftProperty) fastforce+
      next
        case Cons-v:(Cons va vlist)
        with Cons-t Cons-u Cons-tuv(2,3) show ?thesis
        proof(auto split:if-splits, goal-cases)
          case (1 tulist)
          from Cons-tuv(1)[OF 1(5) 1(10)] obtain uvlist
            where uvlist setinterleaves ((ulist, vlist), A) ∧ tuv setinterleaves ((tlist,
uvlist), A) by blast
          with 1 show ?case by(exI va#uvlist, simp)
        next
          case (2 u)
           then show ?case by (metis Cons-tuv.hyps Cons-tuv.prems(1) Sync.sym
SyncSingleHeadAdd)
        next
          case (3 u)
         then show ?case by (metis Cons-tuv.hyps Sync.sym SyncSingleHeadAdd)

        next
          case (4 u)
           then show ?case by (metis Cons-tuv.hyps Cons-tuv.prems(1) Sync.sym
SyncSingleHeadAdd)
        next
          case (5 u)
         then show ?case by (metis Cons-tuv.hyps Sync.sym SyncSingleHeadAdd)
```

```
next
  case (6 u)
   then show ?case by (metis Cons-tuv.hyps Cons-tuv.prems(1) Sync.sym
SyncSingleHeadAdd)
next
  case (7 u)
  then show ?case by (metis Cons-tuv.hyps Sync.sym SyncSingleHeadAdd)
next
  case (8 tulist)
  from Cons-tuv(1)[OF 8(5) 8(9)] obtain uvlist
    where uvlist setinterleaves ((va#ulist, va#vlist), A) ∧
          tuv setinterleaves ((tlist, uvlist), A) by blast
  with 8 show ?case using SyncSingleHeadAdd by (exI uvlist, simp) blast
next
  case (9 u)
   then show ?case by (metis Cons-tuv.hyps Cons-tuv.prems(1) Sync.sym
SyncSingleHeadAdd)
next
  case (10 tulist)
  from Cons-tuv(1)[OF 10(6) 10(10)] obtain uvlist
    where uvlist setinterleaves ((ua#ulist, va#vlist), A) ∧
          tuv setinterleaves ((tlist, uvlist), A) by blast
  with 10 show ?case using SyncSingleHeadAdd by (exI uvlist, simp) blast
next
  case (11 u)
   then show ?case by (metis Cons-tuv.hyps Cons-tuv.prems(1) Sync.sym
SyncSingleHeadAdd)
next
  case (12 u)
  then show ?case using Cons-tuv.hyps by fastforce
next
  case (13 u)
  then show ?case by (metis Cons-tuv.hyps Sync.sym SyncSingleHeadAdd)
next
  case (14 u)
  then show ?case by (metis Cons-tuv.hyps Sync.sym SyncSingleHeadAdd)
next
  case (15 u)
  then show ?case by (metis Cons-tuv.hyps Sync.sym SyncSingleHeadAdd)
next
  case (16 u)
   then show ?case by (metis Cons-tuv.hyps Cons-tuv.prems(1) Sync.sym
SyncSingleHeadAdd)
next
  case (17 u)
  then show ?case by (metis Cons-tuv.hyps Sync.sym SyncSingleHeadAdd)
next
  case (18 u)
  then show ?case using Cons-tuv.hyps by fastforce
```

```
      next
        case (19 u)
        then show ?case using Cons-tuv.hyps by fastforce
      next
        case (20 u)
        then show ?case by (metis Cons-tuv.hyps Cons-tuv.prems(1) Sync.sym
SyncSingleHeadAdd)
      next
        case (21 u)
        then show ?case using Cons-tuv.hyps by fastforce
      qed
    qed
  qed
 qed
qed
```

lemma *interleave-assoc-2*:
assumes $*$:*uv setinterleaves* $((u, v), A)$ **and**
$**$:*tuv setinterleaves* $((t, uv), A)$
shows $\exists$ *tu. tu setinterleaves* $((t, u), A) \wedge$ *tuv setinterleaves* $((tu, v), A)$
using $*$ $**$ *Sync.sym interleave-assoc-1* **by** *blast*

5.7.2 Laws

lemma *Sync-commute*: $(P [\![S]\!] Q) = (Q [\![S]\!] P)$
by (*simp add*: *Process-eq-spec mono-D-Sync mono-F-Sync*)

lemma *mono-Sync-FD-oneside*[*simp*]: $P \leq P' \Longrightarrow (P [\![S]\!] Q) \leq (P' [\![S]\!] Q)$
apply(*auto simp*:*le-ref-def F-Sync D-Sync*)
using *F-subset-imp-T-subset front-tickFree-Nil* **by** *blast+*

lemma *mono-Sync-FD*[*simp*]: $[\![P \leq P'; Q \leq Q']\!] \Longrightarrow (P [\![S]\!] Q) \leq (P' [\![S]\!] Q')$
using *mono-Sync-FD-oneside*[*of P P′ S Q*] *mono-Sync-FD-oneside*[*of Q Q′ S P*]
by (*simp add*: *order-trans Sync-commute*)

lemma *mono-Sync-ref*: $[\![P \sqsubseteq P'; Q \sqsubseteq Q']\!] \Longrightarrow (P [\![S]\!] Q) \sqsubseteq (P' [\![S]\!] Q')$
using *mono-Sync-sym*[*of Q Q′ P S*] *mono-Sync*[*of P P′ S Q′*] *below-trans* **by** *blast*

lemma *Sync-BOT*: $(P [\![S]\!] \bot) = \bot$
apply(*auto simp add*:*Process-eq-spec*, *simp-all add*:*D-Sync F-Sync F-UU D-UU*)
apply (*metis D-imp-front-tickFree append-Nil2 front-tickFree-append ftf-Sync is-processT is-processT2-TR*)
apply (*metis Nil-elem-T Sync.si-empty1 append-Nil front-tickFree-Nil insertI1 non-tickFree-implies-nonMt*)
apply (*metis D-imp-front-tickFree append-Nil2 front-tickFree-append ftf-Sync is-processT2-TR*)
by (*metis Nil-elem-T Sync.si-empty1 append-Nil front-tickFree-Nil insertI1 non-tickFree-implies-nonMt*)

lemma *Sync-SKIP-SKIP*: $(SKIP \llbracket S \rrbracket SKIP) = SKIP$
apply(*auto simp add*:*Process-eq-spec*, *simp-all add*:*D-Sync F-Sync F-SKIP D-SKIP*)
apply(*elim exE conjE disjE*, *auto*)
apply (*metis Sync.si-empty1 inf.idem insertI1 sup-commute sup-inf-absorb*)
apply(*exI* [*tick*], *exI* [*tick*], *simp*)
by *blast*

lemma *Sync-SKIP-STOP*: $(SKIP \llbracket S \rrbracket STOP) = STOP$
proof(*auto simp add*:*Process-eq-spec*, *simp-all add*:*D-Sync F-Sync F-SKIP D-SKIP F-STOP D-STOP*, *goal-cases*)
case (*1 a b*)
then show *?case* **by** (*elim exE conjE disjE*, *auto*)
next
case (*2 a b*)
then show *?case*
apply(*rule-tac x*=[] **in** *exI*, *simp*, *rule-tac x*=*b*−{*tick*} **in** *exI*, *simp*, *rule-tac x*=*b* **in** *exI*)
by *blast*
qed

5.7.3 Multi-Operators laws

lemma *Mprefix-Sync-distr*:
$\llbracket A \cap S = \{\}; A' \subseteq S; B \cap S = \{\}; B' \subseteq S \rrbracket \Longrightarrow$
$(\Box\, x \in A \cup A' \rightarrow P\, x) \llbracket S \rrbracket (\Box\, y \in B \cup B' \rightarrow Q\, y) = (\Box\, x \in A \rightarrow (P\, x \llbracket S \rrbracket (\Box\, y \in B \cup B' \rightarrow Q\, y)))$
$\Box\, (\Box\, y \in B \rightarrow ((\Box\, x \in A \cup A' \rightarrow P\, x) \llbracket S \rrbracket Q\, y))$
$\Box\, (\Box\, x \in A' \cap B' \rightarrow (P\, x \llbracket S \rrbracket Q\, x))$
proof(*auto simp add*:*Process-eq-spec*, *simp-all add*:*D-Sync F-Sync D-Mprefix F-Mprefix T-Mprefix D-Det F-Det T-Det*,
elim exE disjE conjE, *goal-cases*)
case (*1 a b t u X Y*)
then show *?case* **by** *auto*
next
case (*2 a b t u X Y aa*)
have *21*:‹*a* = *u*›
using *2*(*5*,*7*) *EmptyLeftSync* **by** *blast*
hence *22*:‹*hd a* ∈ *ev* ‘ *B*›
using *emptyLeftNonSync*[*rule-format*, *of a* ‹*insert tick* (*ev* ‘ *S*)› *u* ‹*hd u*›, *simplified*]
2(*10*,*4*,*5*,*7*,*9*) **by** *auto*
with *2* **show** *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *simp add*:*21*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *simp add*:*21*)
apply(*intro disjI2 conjI*, *simp-all add*:*21 22*)
apply(*exI aa*, *simp add*:*21*)
apply(*rule-tac disjI1*, *exI t*, *exI* ‹*tl u*›, *exI X*, *rule-tac conjI*, *simp*, *exI Y*,

```
simp)
    by (meson SyncTlEmpty)
next
  case (3 a b t u X Y aa)
  have 31:‹a = t›
     using 3(6) 3(8) EmptyLeftSync Sync.sym by blast
  hence 32:‹hd a ∈ ev ‘ A›
    using emptyLeftNonSync[rule-format, of a ‹insert tick (ev ‘ S)› t ‹hd t›]
          3(10,2,5,6,8) Sync.sym by auto blast
  with 3 show ?case
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp add:31)
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp add:31)
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp-all add:31 32)
    apply(exI aa, simp)
   apply(rule-tac disjI1, exI ‹tl t›, exI u, exI X, rule-tac conjI, simp, exI Y, simp)

    by (simp add: Sync.sym SyncTlEmpty)
next
  case (4 a b t u X Y aa ab)
  hence ‹   (hd a = ev aa ∧ hd a ∈ ev ‘ A ∧ tl a setinterleaves ((tl t, u), insert
tick (ev ‘ S)))
            ∨ (hd a = ev ab ∧ hd a ∈ ev ‘ B ∧ tl a setinterleaves ((t, tl u), insert
tick (ev ‘ S)))
            ∨ (hd a = ev ab ∧ aa = ab ∧ hd a ∈ ev ‘ (A′ ∩ B′) ∧ tl a setinterleaves
((tl t, tl u), insert tick (ev ‘ S)))›
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
    apply (simp del:si-neq split:if-splits)
   apply (metis (no-types, lifting) IntI UnE disjoint-iff event.simps(1) event.simps(3)
imageE imageI list.sel(1) list.sel(3))
   by(metis (no-types, opaque-lifting) UnE image-Un list.sel(1) list.sel(3) subset-eq
subset-image-iff)+
  with 4 show ?case
  proof(elim disjE conjE, goal-cases)
    case 41:1
    then show ?case
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis empty-setinterleaving)
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis empty-setinterleaving)

    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis empty-setinterleaving,
simp-all)
      apply(exI aa, simp)
       apply(rule-tac disjI1, exI ‹tl t›, exI u, exI X, rule-tac conjI, simp, exI Y,
simp)
      by blast
  next
    case 42:2
    then show ?case
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis empty-setinterleaving)
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis empty-setinterleaving)
```

```
      apply(rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac conjI, metis
empty-setinterleaving, simp-all)
     apply(exI ab, simp)
     apply(rule-tac disjI1, exI t, exI ‹tl u›, exI X, rule-tac conjI)
     by blast+
  next
    case 43:3
    then show ?case
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis empty-setinterleaving)
     apply(rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac
conjI, metis empty-setinterleaving, simp)
      by(exI aa, simp, rule-tac disjI1, exI ‹tl t›, exI ‹tl u›, exI X, rule-tac conjI,
simp, exI Y, simp)
  qed
next
  case (5 a b t u r v aa)
  have 51:‹a = t@v›
     using 5(7,8,11) EmptyLeftSync Sync.sym by blast
  hence 52:‹hd a ∈ ev ‘ A›
    using emptyLeftNonSync[rule-format, of r ‹insert tick (ev ‘ S)› t ‹hd t›]
          5(10,11,2,8,9) Sync.sym by auto blast
  with 5 show ?case
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp add:51)
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp-all add:51)
    apply(rule-tac disjI1, exI aa, simp)
    apply(rule-tac disjI2, exI ‹tl t›, exI u, exI ‹tl r›, exI v, simp)
    by (simp add: Sync.sym SyncTlEmpty tickFree-tl)
next
  case (6 a b t u r v aa ab)
  hence ‹(hd r = ev aa ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)))
         ∨ (hd r = ev ab ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S)))›
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
    apply(simp del:si-neq split:if-splits, blast, metis empty-iff list.collapse)
     apply(metis (no-types, lifting) event.simps(3) image-eqI insert-iff SyncHd-Tl)
     defer
     apply blast

  proof −
    assume a1: ‹hd t ∉ ev ‘ S›
       and a2: ‹setinterleaving (t, insert tick (ev ‘ S), ev ab # tl u) =
              {hd t # ua |ua. ua setinterleaves ((tl t, ev ab # tl u), insert tick (ev ‘
S))}
              ∪ {ev ab # ua |ua. ua setinterleaves ((t, tl u), insert tick (ev ‘ S))}›
    have f3: ev ab ∈ ev ‘ B
      using 6(13) by blast
    have f4: hd t ∈ ev ‘ A
```

```
      using a1 6(10) 6(2) by blast
    show ‹hd r = hd t ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)) ∨
          hd r = ev ab ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick (ev
‘ S))›
      apply auto
      using 6(8) a2 6(14) 6(15) hd-Cons-tl apply fastforce
      using 6(8) a2 6(14) 6(15) f4 hd-Cons-tl apply fastforce
      using 6(8) a2 6(14) 6(15) hd-Cons-tl apply fastforce
      using 6(8) a2 6(14) 6(15) f3 hd-Cons-tl apply fastforce
      using 6(8) a2 6(14) 6(15) f3 hd-Cons-tl apply fastforce
      using 6(8) a2 6(14) 6(15) f4 f3 hd-Cons-tl apply fastforce
      using 6(13) 6(8) a2 6(14) 6(15) hd-Cons-tl apply fastforce
      using 6(8) a2 6(14) 6(15) f4 hd-Cons-tl apply fastforce
      using 6(8) a2 6(14) 6(15) hd-Cons-tl by fastforce
  qed
  with 6 show ?case
  proof(elim disjE conjE, goal-cases)
    case 61:1
    then show ?case
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving, simp-all)
      apply(rule-tac conjI, metis empty-setinterleaving hd-append2)
      apply(exI aa, rule-tac conjI, metis empty-setinterleaving hd-append2)
      apply(rule-tac disjI2, exI ‹tl t›, exI u, exI ‹tl r›, exI v)
      by (metis empty-setinterleaving tickFree-tl tl-append2)
  next
    case 62:2
    then show ?case
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
      apply(rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac conjI, metis
append-is-Nil-conv empty-setinterleaving)
      apply(rule-tac conjI, metis empty-setinterleaving hd-append2)
      apply(exI ab, rule-tac conjI, metis empty-setinterleaving hd-append2)
      apply(rule-tac disjI2, exI t, exI ‹tl u›, exI ‹tl r›, exI v)
      by (metis empty-setinterleaving tickFree-tl tl-append2)
  qed
next
  case (7 a b t u r v aa ab)
  hence ‹   (hd r = ev aa ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert
tick (ev ‘ S)))
            ∨ (hd r = ev ab ∧ aa = ab ∧ hd r ∈ ev ‘ (A′ ∩ B′) ∧ tl r setinterleaves
```

((*tl t*, *tl u*), *insert tick* (*ev* ‘ *S*)))›
using *si-neq*[*of* ‹*hd t*› ‹*tl t*› ‹*insert tick* (*ev* ‘ *S*)› ‹*hd u*› ‹*tl u*›]
apply (*simp del*:*si-neq split*:*if-splits*)
apply (*metis* (*no-types*, *opaque-lifting*) *IntI UnE empty-iff event.inject imageE imageI insert-iff SyncSameHdTl*)
apply (*metis empty-iff hd-Cons-tl*, *blast*, *blast*)
proof
assume *a1*: *ev ab* ∈ *ev* ‘ *S*
assume *a2*: *hd t* ∉ *ev* ‘ *S*
assume *a3*: *hd t* ≠ *tick*
assume *a4*: *hd u* = *ev ab*
assume *a5*: *hd t* ∈ *ev* ‘ (*A* ∪ *A′*)
assume *a7*: *A′* ⊆ *S*
show ‹*hd r* = *hd t* ∧ *hd r* ∈ *ev* ‘ *A* ∧ *tl r setinterleaves* ((*tl t*, *u*), *insert tick* (*ev* ‘ *S*))›
apply *safe*
apply (*metis 7*(*8*) *Sync.sym a1 a2 a3 a4 insertCI insertE SyncHd-Tl*)
apply (*metis 7*(*8*) *Sync.sym Un-iff a1 a2 a4 a5 a7 image-Un insert-iff sup.absorb-iff2 SyncHd-Tl SyncSameHdTl*)
by (*metis 7*(*8*) *Sync.sym a1 a2 a3 a4 insertCI insertE SyncHd-Tl*)
qed
with *7* **show** *?case*
proof(*elim disjE conjE*, *goal-cases*)
case *71*:*1*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*, *simp-all*)
apply(*rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply(*exI aa*, *rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply(*rule-tac disjI2*, *exI* ‹*tl t*›, *exI u*, *exI* ‹*tl r*›, *exI v*)
by (*metis empty-setinterleaving tickFree-tl tl-append2*)
next
case *72*:*2*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply (*exI aa*, *rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply (*rule-tac disjI2*, *exI* ‹*tl t*›, *exI* ‹*tl u*›, *exI* ‹*tl r*›, *exI v*)
by (*metis empty-setinterleaving tickFree-tl tl-append2*)
qed
next
case (*8 a b u t r v aa*)

```
  have 81:‹r = u›
    using 8 EmptyLeftSync Sync.sym by blast
  hence 82:‹hd r ∈ ev ‘ B›
   using emptyLeftNonSync[rule-format, of r ‹insert tick (ev ‘ S)› u ‹hd u›]
        8 Sync.sym by auto blast
  with 8 show ?case
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp add:81)
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
    apply(intro disjI2 conjI, simp-all add:81 82)
    apply(exI aa, simp add:81)
    apply(rule-tac disjI2,  exI ‹tl u›, exI t, exI ‹tl r›, exI v)
   by (simp add: 81 Sync.sym SyncTlEmpty tickFree-tl)
next
  case (9 a b u t r v ab aa)
  hence ‹(hd r = ev aa ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)))
        ∨ (hd r = ev ab ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S)))›
   using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
    apply(simp del:si-neq split:if-splits, metis disjoint-iff event.inject image-iff,
blast, blast)
   apply (simp add: Sync.sym image-Un list.exhaust-sel rev-image-eqI sup.absorb-iff2)
    defer
   apply (metis (no-types, lifting) Sync.sym event.simps(3) imageI insertE insertI2
SyncHd-Tl)
  proof −
   assume a1: hd u ∈ ev ‘ B ∨ hd u ∈ ev ‘ B′
   assume a2: r setinterleaves ((t, u), insert tick (ev ‘ S))
   assume a3: setinterleaving (u, insert tick (ev ‘ S), ev aa # tl t) =
              {ev aa # ua |ua. ua setinterleaves ((u, tl t), insert tick (ev ‘ S))} ∪
              {hd u # ua |ua. ua setinterleaves ((tl u, ev aa # tl t), insert tick (ev
‘ S))}
   assume a4: hd t = ev aa
   assume a5: t ≠ []
   assume a6: aa ∈ A
   assume a7: hd u ∉ ev ‘ S
   assume a8: B′ ∪ S = S
   obtain ees :: ′a event list ⇒ ′a event list where
     f9: ((∄ es. r = hd u # es ∧ es setinterleaves ((tl u, ev aa # tl t), insert tick
(ev ‘ S))) ∨
          r = hd u # ees r ∧ ees r setinterleaves ((tl u, ev aa # tl t), insert tick
(ev ‘ S))) ∧
          ((∃ es. r = hd u # es ∧ es setinterleaves ((tl u, ev aa # tl t), insert tick
(ev ‘ S))) ∨
          (∀ es. r ≠ hd u # es ∨ es ∉ setinterleaving (tl u, insert tick (ev ‘ S), ev
aa # tl t)))
     by fastforce
   have f10: ev aa # tl t = t
```

using *a5 a4 hd-Cons-tl* **by** *fastforce*
then have *f11*: $r \in \{ev\ aa\ \#\ es\ |es.\ es\ setinterleaves\ ((u,\ tl\ t),\ insert\ tick\ (ev\ `\ S))\} \cup$
$\{hd\ u\ \#\ es\ |es.\ es\ setinterleaves\ ((tl\ u,\ ev\ aa\ \#\ tl\ t),\ insert\ tick\ (ev\ `\ S))\}$
using *a3 a2* **by** (*simp add*: *Sync.sym*)
have *f12*: $\forall e\ f\ a\ A.\ (e::'a\ event) \neq f\ (a::'a) \vee a \notin A \vee e \in f\ `\ A$
by (*meson image-eqI*)
moreover
{ assume $r \neq hd\ u\ \#\ ees\ r \vee ees\ r \notin setinterleaving\ (tl\ u,\ insert\ tick\ (ev\ `\ S),\ ev\ aa\ \#\ tl\ t)$
then obtain $eesa :: 'a\ event\ list \Rightarrow 'a\ event\ list$ **where**
$r = ev\ aa\ \#\ eesa\ r \wedge eesa\ r\ setinterleaves\ ((u,\ tl\ t),\ insert\ tick\ (ev\ `\ S))$
using *f11 f9* **by** *fast*
then have $hd\ r = ev\ aa \wedge hd\ r \in ev\ `\ A \wedge tl\ r\ setinterleaves\ ((u,\ tl\ t),\ insert\ tick\ (ev\ `\ S)) \vee hd\ r = hd\ u \wedge hd\ r \in ev\ `\ B \wedge tl\ r\ setinterleaves\ ((t,\ tl\ u),\ insert\ tick\ (ev\ `\ S))$
using *f12 a6* **by** (*metis list.sel*(*1*) *list.sel*(*3*)) **}**
ultimately show $hd\ r = ev\ aa \wedge hd\ r \in ev\ `\ A \wedge tl\ r\ setinterleaves\ ((u,\ tl\ t),\ insert\ tick\ (ev\ `\ S)) \vee hd\ r = hd\ u \wedge hd\ r \in ev\ `\ B \wedge tl\ r\ setinterleaves\ ((t,\ tl\ u),\ insert\ tick\ (ev\ `\ S))$
using *f10 a8 a7 a1* **by** (*metis Sync.sym UnI1 image-Un list.sel*(*1*) *list.sel*(*3*))
qed
with *9* **show** *?case*
proof(*elim disjE conjE*, *goal-cases*)
case *91*:*1*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*, *simp*)
apply(*rule-tac conjI*, *metis empty-setinterleaving hd-append image-eqI*)
apply(*exI aa*, *rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply(*rule-tac disjI2*, *exI u*, *exI* ‹*tl t*›, *exI* ‹*tl r*›, *exI v*, *simp*)
by (*metis Sync.sym empty-setinterleaving tickFree-tl tl-append2*)
next
case *92*:*2*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*, *simp*)
apply(*rule-tac conjI*, *metis empty-setinterleaving hd-append*)
apply(*exI ab*, *rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply(*rule-tac disjI2*, *exI* ‹*tl u*›, *exI* ‹*t*›, *exI* ‹*tl r*›, *exI v*, *simp*)

```
    by (metis Sync.sym empty-setinterleaving tickFree-tl tl-append2)
  qed
next
  case (10 a b u t r v ab aa)
  hence ‹   (hd r = ev ab ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert
tick (ev ‘ S)))
            ∨ (hd r = ev ab ∧ aa = ab ∧ hd r ∈ ev ‘ (A′ ∩ B′) ∧ tl r setinterleaves
((tl t, tl u), insert tick (ev ‘ S)))›
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
    apply (simp del:si-neq split:if-splits)
   apply (metis (no-types, opaque-lifting) IntI Sync.sym UnE empty-iff event.inject
image-iff insert-iff SyncSameHdTl)
      apply (metis insert-iff SyncSameHdTl)
    defer
     apply (metis imageI subsetD)
    apply (metis imageI subsetD)
  proof
    assume a1: hd u ≠ tick ∧ hd u ∉ ev ‘ S
    assume a2: ev aa ∈ ev ‘ S
    assume a3: hd t = ev aa
    assume a4: r setinterleaves ((u, t), insert tick (ev ‘ S))
    assume a5: B′ ⊆ S
    show ‹hd r = hd u ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S))›
     apply safe
       apply (metis 10(8) Sync.sym a1 a2 a3 insertCI insertE SyncHd-Tl)
      apply (metis (no-types, opaque-lifting) 10(10) 10(8) Sync.sym UnE a1 a2
a3 a5 image-Un insert-iff subset-eq subset-image-iff SyncHd-Tl)
     by (metis Sync.sym a1 a2 a3 a4 insertCI insertE SyncHd-Tl)
  qed
  with 10 show ?case
  proof(elim disjE conjE, goal-cases)
    case 101:1
    then show ?case
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
      apply(rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac conjI, metis
append-is-Nil-conv empty-setinterleaving, simp)
     apply(rule-tac conjI, metis empty-setinterleaving hd-append)
     apply(exI ab, rule-tac conjI, metis empty-setinterleaving hd-append2)
     apply(rule-tac disjI2, exI ‹tl u›, exI ‹t›, exI ‹tl r›, exI v, simp)
     by (metis Sync.sym empty-setinterleaving tickFree-tl tl-append2)
  next
    case 102:2
    then show ?case
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
```

```
    apply(rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac
conjI, metis append-is-Nil-conv empty-setinterleaving)
    apply(rule-tac conjI, metis empty-setinterleaving hd-append2)
    apply (exI aa, rule-tac conjI, metis empty-setinterleaving hd-append2)
    apply (rule-tac disjI2, exI ‹tl u›, exI ‹tl t›, exI ‹tl r›, exI v, simp)
    by (metis Sync.sym empty-setinterleaving tickFree-tl tl-append2)
  qed
next
  case (11 a b t u r v aa)
  have 111:‹a = t@v›
    using 11(7,8,11) EmptyLeftSync Sync.sym by blast
  hence 112:‹hd a ∈ ev ‘ A›
    using emptyLeftNonSync[rule-format, of r ‹insert tick (ev ‘ S)› t ‹hd t›]
        11(10,11,2,8,9) Sync.sym by auto blast
  with 11 show ?case
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp add:111)
    apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp-all add:111)
    apply(rule-tac disjI1, exI aa, simp)
    apply(rule-tac disjI2, exI ‹tl t›, exI u, exI ‹tl r›, exI v, simp)
    by (simp add: Sync.sym SyncTlEmpty tickFree-tl)
next
  case (12 a b t u r v aa ab)
  hence ‹(hd r = ev aa ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)))
        ∨ (hd r = ev ab ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S)))›
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
     apply(simp del:si-neq split:if-splits, metis disjoint-iff event.inject image-iff,
blast)
     apply (metis (no-types, lifting) event.simps(3) imageI insert-iff SyncHd-Tl)
    defer apply blast
  proof −
    assume a1: ab ∈ B
    assume a2: r setinterleaves ((t, u), insert tick (ev ‘ S))
    assume a3: setinterleaving (t, insert tick (ev ‘ S), ev ab # tl u) =
            {hd t # ua |ua. ua setinterleaves ((tl t, ev ab # tl u), insert tick (ev ‘
S))} ∪
            {ev ab # ua |ua. ua setinterleaves ((t, tl u), insert tick (ev ‘ S))}
    assume a4: hd u = ev ab
    assume a5: u ≠ []
    assume a6: hd t ∈ ev ‘ (A ∪ A′)
    assume a7: A′ ⊆ S
    assume a8: hd t ∉ ev ‘ S
    have f9: hd t ∈ ev ‘ A
      using a8 a7 a6 by blast
    show hd r = hd t ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick (ev
‘ S)) ∨
        hd r = ev ab ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick (ev
‘ S))
```

```
    proof −
      obtain ees :: 'a event list ⇒ 'a event list where
        f1: ((∄ es. r = hd t # es ∧ es setinterleaves ((tl t, ev ab # tl u), insert tick
(ev ‘ S))) ∨
              r = hd t # ees r ∧ ees r setinterleaves ((tl t, ev ab # tl u), insert tick
(ev ‘ S))) ∧
             ((∃ es. r = hd t # es ∧ es setinterleaves ((tl t, ev ab # tl u), insert tick
(ev ‘ S))) ∨
              (∀ es. r ≠ hd t # es ∨ es ∉ setinterleaving (tl t, insert tick (ev ‘ S), ev
ab # tl u)))
        by meson
      have f2: ev ab # tl u = u
        using a5 by (subst a4[symmetric], rule list.collapse)
      moreover
      { assume r ≠ hd t # ees r ∨ ees r ∉ setinterleaving (tl t, insert tick (ev ‘
S), ev ab # tl u)
        then obtain eesa :: 'a event list ⇒ 'a event list where
          r = ev ab # eesa r ∧ eesa r setinterleaves ((t, tl u), insert tick (ev ‘ S))
          using f2 f1 a2 a3 by fastforce
        then have ?thesis
          by (metis a1 imageI list.sel(1) list.sel(3)) }
      ultimately show ?thesis
        by (metis (no-types) f9 list.sel(1) list.sel(3))
    qed
  qed
  with 12 show ?case
  proof(elim disjE conjE, goal-cases)
    case 121:1
    then show ?case
      apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
      apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
      apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving, simp)
      apply(exI aa, rule-tac conjI, simp)
      by (rule-tac disjI2, exI ‹tl t›, exI u, exI ‹tl r›, exI v, simp)
  next
    case 122:2
    then show ?case
      apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
      apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
      apply(rule-tac disjI2, rule-tac disjI2, rule-tac disjI2, rule-tac conjI, metis
append-is-Nil-conv empty-setinterleaving, simp)
      by (rule-tac disjI2, exI ‹t›, exI ‹tl u›, exI ‹tl r›, exI v, simp)  blast
  qed
next
```

case (*13 a b t u r v aa ab*)
hence ‹ (*hd r* = *ev aa* ∧ *hd r* ∈ *ev* ‘ *A* ∧ *tl r setinterleaves* ((*tl t*, *u*), *insert tick* (*ev* ‘ *S*)))
∨ (*hd r* = *ev ab* ∧ *aa* = *ab* ∧ *hd r* ∈ *ev* ‘ (*A*′ ∩ *B*′) ∧ *tl r setinterleaves* ((*tl t*, *tl u*), *insert tick* (*ev* ‘ *S*)))›
using *si-neq*[*of* ‹*hd t*› ‹*tl t*› ‹*insert tick* (*ev* ‘ *S*)› ‹*hd u*› ‹*tl u*›]
apply (*simp del*:*si-neq split*:*if-splits*)
apply (*metis* (*no-types*, *opaque-lifting*) *IntI UnE empty-iff event.inject image-iff insert-iff SyncSameHdTl*)
apply (*metis empty-iff hd-Cons-tl*, *blast*, *blast*)
proof
assume *a1*: ‹*r setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *S*))›
assume *a2*: ‹*hd t* ∈ *ev* ‘ (*A* ∪ *A*′)›
assume *a3*: ‹*hd t* ≠ *tick*›
assume *a4*: ‹*hd t* ∉ *ev* ‘ *S*›
assume *a5*: ‹*ev ab* ∈ *ev* ‘ *S*›
assume *a6*: ‹*setinterleaving* (*t*, *insert tick* (*ev* ‘ *S*), *ev ab* # *tl u*) = {*hd t* # *ua* |*ua*. *ua setinterleaves* ((*tl t*, *ev ab* # *tl u*), *insert tick* (*ev* ‘ *S*))}›
show ‹*hd r* = *hd t* ∧ *hd r* ∈ *ev* ‘ *A* ∧ *tl r setinterleaves* ((*tl t*, *u*), *insert tick* (*ev* ‘ *S*))›
apply *safe*
using *13*(*14*) *13*(*15*) *13*(*8*) *a6 list.collapse* **apply** *fastforce*
using *13*(*14*) *13*(*15*) *13*(*2*) *a1 a2 a4 a6 list.collapse* **apply** *fastforce*
by (*metis 13*(*15*) *Sync.sym a5 a3 a4 a1 insertCI insertE SyncHd-Tl*)
qed
with *13* **show** *?case*
proof(*elim disjE conjE*, *goal-cases*)
case *131*:*1*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*, *simp-all*)
apply(*exI aa*, *rule-tac conjI*, *simp*)
apply(*rule-tac disjI2*, *exI* ‹*tl t*›, *exI u*, *exI* ‹*tl r*›, *exI v*)
using *131*(*5*) **by** *blast*
next
case *132*:*2*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply (*exI aa*, *rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply (*rule-tac disjI2*, *exI* ‹*tl t*›, *exI* ‹*tl u*›, *exI* ‹*tl r*›, *exI v*)
by (*metis empty-setinterleaving tl-append2*)

```
  qed
next
  case (14 a b u t r v aa)
  have 141:‹r = u›
     using 14 EmptyLeftSync Sync.sym by blast
  hence 142:‹hd r ∈ ev ‘ B›
    using emptyLeftNonSync[rule-format, of r ‹insert tick (ev ‘ S)› u ‹hd u›]
          14 Sync.sym by auto blast
  with 14 show ?case
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, simp add:141)
     apply(rule-tac disjI2, rule-tac disjI1, rule-tac conjI, metis append-is-Nil-conv
empty-setinterleaving)
     apply(intro disjI2 conjI, simp-all add:141 142)
     apply(exI aa, simp add:141)
     apply(rule-tac disjI2,  exI ‹tl u›, exI t, exI ‹tl r›, exI v)
    by (simp add: 141 Sync.sym SyncTlEmpty tickFree-tl)
next
  case (15 a b u t r v ab aa)
  hence ‹(hd r = ev aa ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)))
         ∨ (hd r = ev ab ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S)))›
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
     apply(simp del:si-neq split:if-splits, metis disjoint-iff event.inject image-iff,
blast, blast)
    defer
     apply (metis (no-types, lifting) Sync.sym event.simps(3) imageI insertE in-
sertI2 SyncHd-Tl)
  proof −
    assume a1: hd u ∈ ev ‘ (B ∪ B′)
    assume a2: B′ ⊆ S
    assume a3: hd u ∉ ev ‘ S
    assume a4: r setinterleaves ((u, t), insert tick (ev ‘ S))
    assume a5: setinterleaving (ev aa # tl t, insert tick (ev ‘ S), u) =
               {ev aa # ua |ua. ua setinterleaves ((tl t, u), insert tick (ev ‘ S))} ∪
               {hd u # ua |ua. ua setinterleaves ((ev aa # tl t, tl u), insert tick (ev
‘ S))}
    assume a6: hd t = ev aa
    assume a7: t ≠ []
    assume a8: aa ∈ A
    have f9: hd u ∈ ev ‘ B
      using a3 a2 a1 by fast
    have f10: ev aa # tl t = t
      using a7 a6 hd-Cons-tl by fastforce
   then have r ∈ {ev aa # es |es. es setinterleaves ((tl t, u), insert tick (ev ‘ S))}
∪
                {hd u # es |es. es setinterleaves ((ev aa # tl t, tl u), insert tick (ev
‘ S))}
     using a5 a4 by (simp add: Sync.sym)
```

then show *hd r* = *ev aa* ∧ *hd r* ∈ *ev* ‘ *A* ∧ *tl r setinterleaves* ((*tl t*, *u*), *insert tick* (*ev* ‘ *S*)) ∨
hd r = *hd u* ∧ *hd r* ∈ *ev* ‘ *B* ∧ *tl r setinterleaves* ((*t*, *tl u*), *insert tick* (*ev* ‘ *S*))
using *f10 f9 a8* **by** *fastforce*
qed
with *15* **show** *?case*
proof(*elim disjE conjE*, *goal-cases*)
case *151*:*1*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*, *simp*)
apply(*rule-tac disjI2*, *exI u*, *exI* ‹*tl t*›, *exI* ‹*tl r*›, *exI v*, *simp*)
using *Sync.sym* **by** *blast*
next
case *152*:*2*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*, *simp*)
apply(*exI ab*, *rule-tac conjI*, *simp*)
apply(*rule-tac disjI2*, *exI* ‹*tl u*›, *exI* ‹*t*›, *exI* ‹*tl r*›, *exI v*, *simp*)
by (*metis Sync.sym*)
qed
next
case (*16 a b u t r v ab aa*)
hence ‹ (*hd r* = *ev ab* ∧ *hd r* ∈ *ev* ‘ *B* ∧ *tl r setinterleaves* ((*t*, *tl u*), *insert tick* (*ev* ‘ *S*)))
∨ (*hd r* = *ev ab* ∧ *aa* = *ab* ∧ *hd r* ∈ *ev* ‘ (*A*′ ∩ *B*′) ∧ *tl r setinterleaves* ((*tl t*, *tl u*), *insert tick* (*ev* ‘ *S*)))›
using *si-neq*[*of* ‹*hd t*› ‹*tl t*› ‹*insert tick* (*ev* ‘ *S*)› ‹*hd u*› ‹*tl u*›]
apply (*simp del*:*si-neq split*:*if-splits*)
apply (*metis* (*no-types*, *opaque-lifting*) *16*(*11*) *16*(*3*) *Int-iff Sync.sym UnE empty-iff event.inject imageE imageI insertCI SyncSameHdTl*)
apply (*metis insert-iff SyncSameHdTl*)
defer
apply (*metis imageI subsetD*)
apply (*metis imageI subsetD*)
proof −
assume *a1*: *hd u* ≠ *tick* ∧ *hd u* ∉ *ev* ‘ *S*
assume *a2*: *B*′ ⊆ *S*
assume *a3*: *hd u* ∈ *ev* ‘ (*B* ∪ *B*′)

assume *a4*: *r setinterleaves* ((*u*, *t*), *insert tick* (*ev* ‘ *S*))
assume *a5*: *setinterleaving* (*ev aa* # *tl t*, *insert tick* (*ev* ‘ *S*), *u*) =
{*hd u* # *ua* |*ua*. *ua setinterleaves* ((*ev aa* # *tl t*, *tl u*), *insert tick* (*ev* ‘ *S*))}
assume *a6*: *hd t* = *ev aa*
assume *a7*: *t* ≠ []
have *f8*: *B′* ∪ *S* = *S*
using *a2* **by** *blast*
have *f9*: *ev aa* # *tl t* = *t*
using *a7 a6 hd-Cons-tl* **by** *fastforce*
then have ∃ *es*. *r* = *hd u* # *es* ∧ *es setinterleaves* ((*ev aa* # *tl t*, *tl u*), *insert tick* (*ev* ‘ *S*))
using *a5 a4* **by** (*simp add*: *Sync.sym*)
then show *hd r* = *hd u* ∧ *hd r* ∈ *ev* ‘ *B* ∧ *tl r setinterleaves* ((*t*, *tl u*), *insert tick* (*ev* ‘ *S*)) ∨
hd r = *hd u* ∧ *aa* = *ab* ∧ *hd r* ∈ *ev* ‘ (*A′* ∩ *B′*) ∧ *tl r setinterleaves* ((*tl t*, *tl u*), *insert tick* (*ev* ‘ *S*))
using *f9 f8 a3 a1* **by** (*metis Un-iff image-Un list.sel*(*1*) *list.sel*(*3*))
qed
with *16* **show** *?case*
proof(*elim disjE conjE*, *goal-cases*)
case *161*:*1*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*, *simp*)
apply(*exI ab*, *rule-tac conjI*, *simp*)
apply(*rule-tac disjI2*, *exI* ‹*tl u*›, *exI* ‹*t*›, *exI* ‹*tl r*›, *exI v*, *simp*)
by (*metis Sync.sym*)
next
case *162*:*2*
then show *?case*
apply(*rule-tac disjI2*, *rule-tac disjI1*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac disjI2*, *rule-tac conjI*, *metis append-is-Nil-conv empty-setinterleaving*)
apply(*rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply (*exI aa*, *rule-tac conjI*, *metis empty-setinterleaving hd-append2*)
apply (*rule-tac disjI2*, *exI* ‹*tl u*›, *exI* ‹*tl t*›, *exI* ‹*tl r*›, *exI v*, *simp*)
by (*metis Sync.sym*)
qed
next
case (*17 aA bB*)
have *tact*:‹⋀*P Q*. ∄ *t u r v*. *P t u r v* ⟹ *Q* ∨ (∃ *t u r v*. *P t u r v*) ⟹ *Q*›
by *fastforce*
from *17* **show** *?case*

```
proof (safe, goal-cases)
  case 1
  obtain X where 11: (X::'a event set)= bB − ev ‘ A' by auto
  obtain Y where 12: (Y::'a event set)= bB − ev ‘ B' by auto
  from 1 11 12 have 13: bB = (X ∪ Y) ∩ insert tick (ev ‘ S) ∪ X ∩ Y
    by auto
  with 11 12 1 show ?case
    apply(exI ‹[]›, exI ‹[]›, exI X, rule-tac conjI)
     apply safe
          apply (metis disjoint-iff-not-equal imageI)
         apply (metis image-iff)
        apply (metis disjoint-iff imageI)
       apply (metis IntI image-eqI)
      apply (metis image-eqI)
     apply (metis rev-image-eqI)
    by (exI Y) (simp add: disjoint-iff image-Un)
next
  case (2 x a t u X Y)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI1, exI ‹ev x#t›, exI ‹[]›, exI X, rule conjI, simp)
    by (exI Y, simp, metis disjoint-iff event.inject hd-Cons-tl imageE)
next
  case (3 x a t u X Y xa aa)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI1, exI ‹ev x#t›, exI ‹u›, exI X, rule conjI, simp)
    apply (exI Y, simp)
    using hd-Cons-tl SyncSingleHeadAdd by fastforce+
next
  case (4 x a t u X Y xa aa)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI1, exI ‹ev x#t›, exI ‹u›, exI X, rule conjI, simp)
    apply (exI Y, simp)
    using hd-Cons-tl SyncSingleHeadAdd by fastforce+
next
  case (5 x a t u r v)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹[]›, exI ‹ev x#r›, exI v, simp)
    by (metis disjoint-iff event.inject hd-Cons-tl imageE)
next
  case (6 x a t u r v aa)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹u›, exI ‹ev x#r›, exI v, simp)
    using hd-Cons-tl SyncSingleHeadAdd by fastforce+
next
```

```
    case (7 x a t u r v aa)
    then show ?case
      apply(erule-tac tact)
      apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹u›, exI ‹ev x#r›, exI v, simp)
      using hd-Cons-tl SyncSingleHeadAdd by fastforce+
  next
    case (8 x a u t r v xa aa)
    then show ?case
      apply(erule-tac tact)
       apply(rule-tac disjI2, exI ‹u›, exI ‹ev x#t›, exI ‹ev x#r›, exI ‹v›, simp)

      using si-neq[of ‹ev x› ‹t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
      apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast)
     by (metis Sync.sym SyncSingleHeadAdd event.distinct(1) insert-iff list.collapse)
blast
  next
    case (9 x a u t r v xa aa)
    then show ?case
      apply(erule-tac tact)
       apply(rule-tac disjI2, exI ‹u›, exI ‹ev x#t›, exI ‹ev x#r›, exI ‹v›, simp)

      using si-neq[of ‹ev x› ‹t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
      apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast, blast)
      by (metis SyncHdAdd1 event.distinct(1) insert-iff list.collapse)
  next
    case (10 x a t u r v)
    then show ?case
      apply(erule-tac tact)
     apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹[]›, exI ‹ev x#r›, exI ‹[]›, simp)
      by (metis disjoint-iff event.inject hd-Cons-tl imageE)
  next
    case (11 x a t u r v aa)
    then show ?case
      apply(erule-tac tact)
      apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹u›, exI ‹ev x#r›, exI ‹[]›, simp)
      using SyncSingleHeadAdd list.collapse by fastforce
  next
    case (12 x a t u r v aa)
    then show ?case
      apply(erule-tac tact)
      apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹u›, exI ‹ev x#r›, exI ‹[]›, simp)
      using SyncSingleHeadAdd list.collapse by fastforce
  next
    case (13 x a u t r v xa aa)
    then show ?case
      apply(erule-tac tact)
     apply(rule-tac disjI2, exI ‹u›, exI ‹ev x#t›, exI ‹ev x#r›, exI ‹[]›, simp)
      using si-neq[of ‹ev x› ‹t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
      apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast)
```

```
    by (metis Sync.sym SyncSingleHeadAdd event.distinct(1) insert-iff list.collapse)
blast
  next
    case (14 x a u t r v xa aa)
    then show ?case
      apply(erule-tac tact)
      apply(rule-tac disjI2, exI ‹u›, exI ‹ev x#t›, exI ‹ev x#r›, exI ‹[]›, simp)
      using si-neq[of ‹ev x› ‹t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
      apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast, blast)
      by (metis SyncHdAdd1 event.distinct(1) insert-iff list.collapse)
  next
    case (15 x a t u X Y)
    then show ?case
      apply(erule-tac tact)
      apply(rule-tac disjI1, exI ‹[]›, exI ‹ev x#u›, exI X, rule conjI, simp)
      by (exI Y, simp, metis disjoint-iff event.inject hd-Cons-tl imageE)
  next
    case (16 x a t u X Y xa aa)
    then show ?case
      apply(erule-tac tact)
      apply(rule-tac disjI1, exI ‹t›, exI ‹ev x#u›, exI X, rule conjI, simp, exI Y,
simp)
      using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹tl u›]
      apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast, blast)
    by (metis Sync.sym SyncSingleHeadAdd event.distinct(1) insert-iff list.collapse)
blast
  next
    case (17 x a t u X Y xa aa)
    then show ?case
      apply(erule-tac tact)
      apply(rule-tac disjI1, exI ‹t›, exI ‹ev x#u›, exI X, rule conjI, simp)
      apply (exI Y, simp)
      using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹tl u›]
     apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast) sledgeham-
mer
      using SyncHdAdd by fastforce blast+
  next
    case (18 x a t u r v xa aa)
    then show ?case
      apply(erule-tac tact)
       apply(rule-tac disjI2, exI ‹t›, exI ‹ev x#u›, exI ‹ev x#r›, exI ‹v›, simp)

      using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹u›]
      apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast, blast)
    by (metis Sync.sym SyncSingleHeadAdd event.distinct(1) insert-iff list.collapse)
blast
  next
    case (19 x a t u r v xa aa)
    then show ?case
```

```
    apply(erule-tac tact)
     apply(rule-tac disjI2, exI ‹t›, exI ‹ev x#u›, exI ‹ev x#r›, exI ‹v›, simp)

    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹u›]
    apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast)
    by (metis SyncHdAdd1 event.distinct(1) insert-iff list.collapse) blast+
  next
   case (20 x a u t r v)
   then show ?case
    apply(erule-tac tact)
     apply(rule-tac disjI2, exI ‹ev x#u›, exI ‹[]›, exI ‹ev x#r›, exI ‹v›, simp)

    by (metis disjoint-iff event.inject hd-Cons-tl imageE)
  next
   case (21 x a u t r v aa)
   then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#u›, exI ‹t›, exI ‹ev x#r›, exI ‹v›, simp)
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹u›]
    apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast, blast)
    by (metis list.collapse) blast
  next
   case (22 x a u t r v aa)
   then show ?case
    apply(erule-tac tact)
   apply(rule-tac disjI2, exI ‹ev x#u›, exI ‹t›, exI ‹ev x#r›, exI ‹v›, simp)
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹u›]
    apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast)
    by (metis list.exhaust-sel) blast+
  next
   case (23 x a t u r v xa aa)
   then show ?case
    apply(erule-tac tact)
     apply(rule-tac disjI2, exI ‹t›, exI ‹ev x#u›, exI ‹ev x#r›, exI ‹[]›, simp)

    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹u›]
    apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast, blast)
   by (metis Sync.sym SyncSingleHeadAdd event.distinct(1) insert-iff list.collapse)
blast
  next
   case (24 x a t u r v xa aa)
   then show ?case
    apply(erule-tac tact)
     apply(rule-tac disjI2, exI ‹t›, exI ‹ev x#u›, exI ‹ev x#r›, exI ‹[]›, simp)

    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹u›]
   apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast) sledgeham-
mer
    using SyncHdAdd1 list.collapse by fastforce blast+
```

```
next
  case (25 x a t u r v)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹[]›, exI ‹ev x#r›, exI ‹[]›, simp)
    by (metis disjoint-iff event.inject hd-Cons-tl imageE)
next
  case (26 x a u t r v aa)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#u›, exI ‹t›, exI ‹ev x#r›, exI ‹[]›, simp)
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹u›]
    apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast, blast)
    by (metis list.collapse) blast
next
  case (27 x a u t r v aa)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#u›, exI ‹t›, exI ‹ev x#r›, exI ‹[]›, simp)
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹ev x› ‹u›]
    apply (simp add: SyncSingleHeadAdd split: if-splits, blast, blast) sledgeham-
mer
    by (metis list.collapse) blast+
next
  case (28 x a t u X Y)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI1, exI ‹ev x#t›, exI ‹ev x#u›, exI X, rule conjI, simp)
    by (exI Y, simp, metis hd-Cons-tl image-eqI in-mono)
next
  case (29 x a t u r v)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹ev x#u›, exI ‹ev x#r›, exI ‹v›,
simp)
    by (metis image-eqI list.exhaust-sel subsetD)
next
  case (30 x a t u r v)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹ev x#u›, exI ‹ev x#r›, exI ‹v›,
simp)
    by (metis image-eqI list.exhaust-sel subsetD)
next
  case (31 x a t u r v)
  then show ?case
    apply(erule-tac tact)
    apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹ev x#u›, exI ‹ev x#r›, exI ‹[]›,
simp)
```

```
      by (metis image-eqI list.exhaust-sel subsetD)
  next
    case (32 x a t u r v)
    then show ?case
      apply(erule-tac tact)
       apply(rule-tac disjI2, exI ‹ev x#t›, exI ‹ev x#u›, exI ‹ev x#r›, exI ‹[]›,
simp)
      by (metis image-eqI list.exhaust-sel subsetD)
  qed
next
  case (18 aA)
  then show ?case
  proof(elim exE disjE conjE, goal-cases)
    case (1 t u r v a)
      have 11:‹r = t›
        using 1 EmptyLeftSync Sync.sym by blast
      hence 12:‹hd aA ∈ ev ‘ A›
        using emptyLeftNonSync[rule-format, of r ‹insert tick (ev ‘ S)› t ‹hd t›]
          1 Sync.sym by auto blast
    with 1 11 show ?case
      apply(rule-tac disjI1, simp)
      by (metis Sync.sym SyncTlEmpty tickFree-tl)
  next
    case (2 t u r v a aa)
    hence ‹(hd r = ev a ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)))
        ∨ (hd r = ev aa ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S)))›
      using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
      apply(simp del:si-neq split:if-splits, blast , metis empty-iff hd-Cons-tl)
       apply (metis (no-types, lifting) event.simps(3) imageI insert-iff SyncHd-Tl)

       defer
       apply blast
    proof(safe, simp-all add: in-mono, goal-cases)
      case (1 x)
      show ?case
        using 1(18) 1(21) 2(14) 2(15) 2(8) list.collapse by fastforce
    next
      case (2 x)
      show ?case
        using 2(13) 2(14) 2(18) 2(20) 2(21) 2(8) list.collapse by fastforce
    next
      case (3 x)
      show ?case
        using 2(14) 2(15) 2(8) 3(18) 3(21) list.collapse by fastforce
    next
      case (4 x)
      show ?case
```

```
      using 2(13) 2(14) 2(15) 2(8) 4(18) 4(21) list.collapse by fastforce
  next
    case (5 x)
    show ?case
      using 2(14) 2(15) 2(8) 5(18) 5(21) list.collapse by fastforce
  next
    case (6 x)
    show ?case
      using 6(13) 6(14) 6(18) 6(8) hd-Cons-tl 2(13) 6(20) 6(21) by fastforce
  next
    case (7 x)
    show ?case
      using 2(13) 2(14) 2(15) 2(8) 7(18) 7(21) list.collapse by fastforce
  next
    case (8 x)
    show ?case
      using 8(20) 8(21) 8(13) 8(14) 8(18) 8(8) hd-Cons-tl by fastforce
  next
    case (9 x)
    show ?case
      using 9(21) 9(13) 9(14) 9(13) 9(14) 9(18) 9(8) hd-Cons-tl by fastforce
  qed
  with 2 show ?case
    by (simp add: hd-append) (metis empty-setinterleaving tickFree-tl)
 next
  case (3 t u r v a aa)
  hence ‹   (hd r = ev a ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert
tick (ev ‘ S)))
            ∨ (hd r = ev aa ∧ aa = a ∧ hd r ∈ ev ‘ (A′ ∩ B′) ∧ tl r setinterleaves
((tl t, tl u), insert tick (ev ‘ S)))›
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
    apply (simp del:si-neq split:if-splits)
        apply (simp add:  UnE disjoint-iff imageE insertI2 rev-image-eqI)
       apply (metis (no-types, lifting) Int-iff Un-iff event.inject imageE image-eqI
insertCI SyncSameHdTl)
       apply (metis insert-iff SyncSameHdTl)
      apply (metis imageI subsetD)
     apply (metis imageI subsetD)
     by (metis (no-types, opaque-lifting) 3(10) 3(2) Sync.sym Un-iff image-Un
subset-insertI sup.orderE SyncHd-Tl SyncSameHdTl)
  with 3 show ?case
    by (simp add: hd-append) (metis Sync.sym empty-setinterleaving tickFree-tl)
 next
  case (4 u t r v a)
  have 41:‹r = u›
    using 4 EmptyLeftSync Sync.sym by blast
  hence 42:‹hd aA ∈ ev ‘ B›
    using emptyLeftNonSync[rule-format, of r ‹insert tick (ev ‘ S)› u ‹hd u›]
        4 Sync.sym by auto blast
```

```
    with 4 41 show ?case
      apply(rule-tac disjI2, rule-tac disjI1, simp)
      by (metis Sync.sym SyncTlEmpty tickFree-tl)
  next
    case (5 u t r v aa a)
    hence ‹(hd r = ev a ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)))
        ∨ (hd r = ev aa ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S)))›
      using si-neq[of ‹hd u› ‹tl u› ‹insert tick (ev ‘ S)› ‹hd t› ‹tl t› ]
      apply(simp del:si-neq split:if-splits, blast, blast)
        apply (metis (no-types, lifting) Sync.sym event.simps(3) imageI insert-iff
SyncHd-Tl)
       defer
       apply blast
      proof(safe, simp-all add: in-mono, goal-cases)
      case (1 x)
      show ?case
        using 1(13) 1(14) 1(18) 1(21) 5(8) list.collapse by fastforce
    next
      case (2 x)
      show ?case
        using 5(14) 5(15) 2(18) 5(8) 2(21) 5(13) list.collapse by fastforce
    next
      case (3 x)
      show ?case
        using 3(18) 3(21) 5(14) 5(15) 5(8) Sync.sym hd-Cons-tl by fastforce
    next
      case (4 x)
      show ?case
        using 4(18) 4(20) 4(21) 5(14) 5(15) 5(8) list.collapse by fastforce
    next
      case (5 x)
      show ?case
        using 5(13) 5(14) 5(18) 5(21) 5(8) Sync.sym hd-Cons-tl by fastforce
    next
      case (6 x)
      show ?case
       using 6(20) 6(21) 6(13) 6(14) 6(18) 6(8) 6(12) Sync.sym hd-Cons-tl by
fastforce
    next
      case (7 x)
      show ?case
      using 7(18) 7(13) 7(14) 7(8) 7(20) 7(21) Sync.sym hd-Cons-tl by fastforce
    next
      case (8 x)
      show ?case
      using 8(13) 8(14) hd-Cons-tl Sync.sym 8(18) 8(8) 8(21) 8(12) by fastforce
    next
```

```
      case (9 x)
      show ?case
        using 9(14) 9(13) list.collapse 9(21) Sync.sym 9(18) 9(8) by fastforce
    qed
    with 5 show ?case
      by (simp add: hd-append) (metis Sync.sym empty-setinterleaving tickFree-tl)
  next
    case (6 u t r v aa a)
      hence ‹   (hd r = ev aa ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert
tick (ev ‘ S)))
              ∨ (hd r = ev aa ∧ aa = a ∧ hd r ∈ ev ‘ (A′ ∩ B′) ∧ tl r setinterleaves
((tl t, tl u), insert tick (ev ‘ S)))›
        using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
        apply (simp del:si-neq split:if-splits)
            apply (simp add:  UnE disjoint-iff imageE insertI2 rev-image-eqI)
          apply (metis (no-types, lifting) Int-iff Sync.sym Un-iff event.inject imageE
image-eqI insertCI SyncSameHdTl)
            apply (metis insert-iff SyncSameHdTl)
          apply (metis (no-types, opaque-lifting) Sync.sym UnE image-Un insert-iff
subset-eq subset-image-iff SyncHd-Tl SyncSameHdTl)
         apply (metis imageI subsetD)
        by (metis image-eqI subsetD)
      with 6 show ?case
       by (simp add: hd-append) (metis Sync.sym empty-setinterleaving tickFree-tl)
  next
    case (7 t u r v a)
    have 71:‹r = t›
      using 7 EmptyLeftSync Sync.sym by blast
    hence 72:‹hd aA ∈ ev ‘ A›
      using emptyLeftNonSync[rule-format, of r ‹insert tick (ev ‘ S)› t ‹hd t›]
        7 Sync.sym by auto blast
    with 7 71 show ?case
      apply(rule-tac disjI1, simp)
      by (metis Sync.sym append-self-conv front-tickFree-Nil SyncTlEmpty)
  next
    case (8 t u r v a aa)
    hence ‹(hd r = ev a ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)))
          ∨ (hd r = ev aa ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S)))›
      using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
      apply(simp del:si-neq split:if-splits, blast , metis empty-iff hd-Cons-tl)
       apply (metis (no-types, lifting) event.simps(3) imageI insert-iff SyncHd-Tl)

       defer
       apply blast
    proof(safe, simp-all add: in-mono, goal-cases)
      case (1 x)
      show ?case
```

```
      using 1(17) 1(20) 8(14) 8(15) 8(8) list.collapse by fastforce
  next
    case (2 x)
    show ?case
      using 2(12) 2(13) hd-Cons-tl 2(11) 8(8) 2(17) 2(20) 2(19) by fastforce
  next
    case (3 x)
    show ?case
     using 3(12) 3(13) 3(17) 3(20) 3(7) 3(8) Sync.sym hd-Cons-tl by fastforce
  next
    case (4 x)
    show ?case
      using 4(11) 4(17) 4(20) 8(14) 8(15) 8(8) list.collapse by fastforce
  next
    case (5 x)
    show ?case
     using 5(17) 5(12) 5(13) 5(7) 5(8) 5(20) list.collapse Sync.sym hd-Cons-tl
by fastforce
  next
    case (6 x)
    show ?case
     using 6(11) 6(12) 6(13) 6(17) 6(19) 6(20) 6(7) list.collapse by fastforce
  next
    case (7 x)
    show ?case
      using 7(17) 7(20) 8(13) 8(14) 8(15) 8(8) list.collapse by fastforce
  next
    case (8 x)
    show ?case
      using 8(12) 8(13) 8(17) 8(19) 8(20) 8(7) list.collapse by fastforce
  next
    case (9 x)
    show ?case
     using 9(20) Sync.sym 9(12) 9(13) list.exhaust-sel 9(12) 9(13) 9(17) 9(7)
Sync.sym hd-Cons-tl by fastforce
  qed
  with 8 show ?case
    by (metis empty-setinterleaving self-append-conv)
 next
  case (9 t u r v a aa)
    hence ‹   (hd r = ev a ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert
tick (ev ‘ S)))
            ∨ (hd r = ev aa ∧ aa = a ∧ hd r ∈ ev ‘ (A′ ∩ B′) ∧ tl r setinterleaves
((tl t, tl u), insert tick (ev ‘ S)))›
      using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
      apply (simp del:si-neq split:if-splits)
         apply (simp add: disjoint-iff)
       apply (metis (no-types, lifting) Int-iff Un-iff event.inject imageE image-eqI
insertCI SyncSameHdTl)
```

```
        apply (metis insert-iff SyncSameHdTl)
       apply (metis rev-image-eqI subsetD)
      apply (metis imageI subsetD)
       by (metis (no-types, opaque-lifting) Sync.sym Un-iff image-Un insert-iff
sup.absorb-iff2 SyncHd-Tl)
    with 9 show ?case
      by (metis empty-setinterleaving self-append-conv)
  next
    case (10 u t r v a)
    have 101:‹r = u›
      using 10 EmptyLeftSync Sync.sym by blast
    hence 102:‹hd aA ∈ ev ‘ B›
      using emptyLeftNonSync[rule-format, of r ‹insert tick (ev ‘ S)› u ‹hd u›]
          10 Sync.sym by auto blast
    with 10 101 show ?case
      apply(rule-tac disjI2, rule-tac disjI1, simp)
     by (metis Sync.sym append.right-neutral front-tickFree-Nil SyncTlEmpty)
  next
    case (11 u t r v aa a)
    hence ‹(hd r = ev a ∧ hd r ∈ ev ‘ A ∧ tl r setinterleaves ((tl t, u), insert tick
(ev ‘ S)))
        ∨ (hd r = ev aa ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert tick
(ev ‘ S)))›
      using si-neq[of ‹hd u› ‹tl u› ‹insert tick (ev ‘ S)› ‹hd t› ‹tl t› ]
      apply(simp del:si-neq split:if-splits, blast, blast)
       apply (metis (no-types, lifting) Sync.sym event.simps(3) imageI insert-iff
SyncHd-Tl)
       defer
       apply blast
    proof(safe, simp-all add: in-mono, goal-cases)
      case (1 x)
      show ?case
        using 1(17) 1(20) 11(14) 11(15) 11(8) list.collapse by fastforce
    next
      case (2 x)
      show ?case
        using 2(12) 2(13) hd-Cons-tl 2(11) 11(8) 2(17) 2(20) by fastforce
    next
      case (3 x)
      show ?case
          using 3(12) 3(13) 3(17) 3(20) 3(7) 3(8) Sync.sym hd-Cons-tl by
fastforce
    next
      case (4 x)
      show ?case
        using 11(14) 11(15) 11(8) 4(17) 4(19) 4(20) list.collapse by fastforce
    next
      case (5 x)
      show ?case
```

```
      using 5(17) 5(12) 5(13) 5(7) 5(8) 5(20) list.collapse Sync.sym hd-Cons-tl
by fastforce
    next
      case (6 x)
      show ?case
      using 6(11) 6(12) 6(13) 6(17) 6(19) 6(20) 6(7) list.collapse by fastforce
    next
      case (7 x)
      show ?case
        using 7(12) 7(13) 7(17) 7(19) 7(20) 7(7) Sync.sym list.collapse by
fastforce
    next
      case (8 x)
      show ?case
      using 8(20) 8(12) 8(11) 8(13) list.collapse Sync.sym 8(12) 8(13) 8(17)
8(7) hd-Cons-tl by fastforce
    next
      case (9 x)
      show ?case
        using 9(20) Sync.sym 9(12) 9(13) list.exhaust-sel 9(12) 9(13) 9(17)
9(7) Sync.sym hd-Cons-tl by fastforce
    qed
    with 11 show ?case
      by (metis Sync.sym append.right-neutral empty-setinterleaving)
  next
    case (12 u t r v aa a)
    hence ‹  (hd r = ev aa ∧ hd r ∈ ev ‘ B ∧ tl r setinterleaves ((t, tl u), insert
tick (ev ‘ S)))
          ∨ (hd r = ev aa ∧ aa = a ∧ hd r ∈ ev ‘ (A′ ∩ B′) ∧ tl r setinterleaves
((tl t, tl u), insert tick (ev ‘ S)))›
    using si-neq[of ‹hd t› ‹tl t› ‹insert tick (ev ‘ S)› ‹hd u› ‹tl u›]
    apply (simp del:si-neq split:if-splits)
        apply (simp add: UnE disjoint-iff imageE insertI2 rev-image-eqI)
       apply (metis (no-types, lifting) Int-iff Sync.sym Un-iff event.inject imageE
image-eqI insertCI SyncSameHdTl)
        apply (metis insert-iff SyncSameHdTl)
       apply (metis (no-types, opaque-lifting) Sync.sym UnE image-Un insert-iff
subset-eq subset-image-iff SyncHd-Tl SyncSameHdTl)
      apply (metis imageI subsetD)
      by (metis image-eqI subsetD)
  with 12 show ?case
    by (metis (no-types, lifting) Sync.sym append-Nil2 empty-setinterleaving)
 qed
next
 case (19 x)
 then show ?case
 proof(elim exE disjE conjE, goal-cases)
   case (1 a t u r v)
   then show ?case
```

```
    apply(exI ‹ev a#t›, exI u, exI ‹ev a#r›, exI v, simp)
    apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
     by (metis list.exhaust-sel)
  next
    case (2 a t u r v aa)
    then show ?case
      apply(exI ‹ev a#t›, exI u, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
       by (metis list.exhaust-sel)
  next
    case (3 a t u r v aa)
    then show ?case
      apply(exI ‹ev a#t›, exI u, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
       by (metis list.exhaust-sel)
  next
    case (4 a t u r v aa)
    then show ?case
      apply(exI ‹t›, exI  ‹ev a#u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff in-mono SyncHdAdd1 SyncS-
ingleHeadAdd)
        apply (metis list.exhaust-sel)
      apply (metis (no-types, opaque-lifting) Sync.sym event.inject event.simps(3)
imageE insertE SyncSingleHeadAdd)
       by (metis list.exhaust-sel)
  next
    case (5 a t u r v)
    then show ?case
      apply(exI ‹ev a#t›, exI u, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all)
       apply blast
      by (metis list.exhaust-sel)
  next
    case (6 a t u r v aa)
    then show ?case
      apply(exI ‹ev a#t›, exI u, exI ‹ev a#r›, exI v, simp)
     apply (safe, simp-all add: disjoint-iff-not-equal image-iff SyncSingleHeadAdd)
       by (metis list.exhaust-sel)
  next
    case (7 a t u r v aa)
    then show ?case
      apply(exI ‹ev a#t›, exI u, exI ‹ev a#r›, exI v, simp)
     apply (safe, simp-all add: disjoint-iff-not-equal image-iff SyncSingleHeadAdd)
       by (metis list.exhaust-sel)
  next
    case (8 a t u r v aa)
    then show ?case
      apply(exI ‹t›, exI  ‹ev a#u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff in-mono SyncHdAdd1)
```

```
      apply (metis list.exhaust-sel)
      apply (metis (no-types, lifting) Sync.sym event.distinct(1) event.inject im-
ageE insertE SyncSingleHeadAdd)
    by (metis list.collapse)
  next
    case (9 a t u r v aa)
    then show ?case
      apply(exI ‹t›, exI ‹ev a#u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff in-mono SyncHdAdd1)
        apply (metis list.exhaust-sel)
      apply (metis (no-types, lifting) Sync.sym event.distinct(1) event.inject im-
ageE insertE SyncSingleHeadAdd)
    by (metis list.collapse)
  next
    case (10 a t u r v)
    then show ?case
      apply(exI ‹ev a#t›, exI u, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
      by (metis list.exhaust-sel)
  next
    case (11 a t u r v aa)
    then show ?case
      apply(exI ‹ev a#t›, exI ‹u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
      by (metis list.exhaust-sel)
  next
    case (12 a t u r v aa)
    then show ?case
      apply(exI ‹ev a#t›, exI ‹u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
      by (metis list.exhaust-sel)
  next
    case (13 a t u r v aa)
    then show ?case
      apply(exI ‹t›, exI ‹ev a#u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, auto simp add: disjoint-iff image-iff subset-eq SyncHdAdd1)
        apply (metis list.exhaust-sel)
      apply (metis (no-types, lifting) Sync.sym event.inject event.simps(3) imageE
insertE SyncSingleHeadAdd)
      by (metis hd-Cons-tl)
  next
    case (14 a t u r v)
    then show ?case
      apply(exI ‹ev a#t›, exI u, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
      by (metis list.collapse)
  next
    case (15 a t u r v aa)
    then show ?case
```

```
    apply(exI ‹ev a#t›, exI ‹u›, exI ‹ev a#r›, exI v, simp)
    apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
    by (metis list.exhaust-sel)
  next
    case (16 a t u r v aa)
    then show ?case
      apply(exI ‹ev a#t›, exI ‹u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all add: disjoint-iff image-iff SyncSingleHeadAdd)
      by (metis list.exhaust-sel)
  next
    case (17 a t u r v)
    then show ?case
      apply(exI ‹ev a#t›, exI ‹ev a#u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all)
        apply (metis list.exhaust-sel)
      by blast+
  next
    case (18 a t u r v)
    then show ?case
      apply(exI ‹ev a#t›, exI ‹ev a#u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all)
        apply (metis list.exhaust-sel)
      by blast+
  next
    case (19 a t u r v)
    then show ?case
      apply(exI ‹ev a#t›, exI ‹ev a#u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all)
        apply (metis list.exhaust-sel)
      by blast+
  next
    case (20 a t u r v)
    then show ?case
      apply(exI ‹ev a#t›, exI ‹ev a#u›, exI ‹ev a#r›, exI v, simp)
      apply (safe, simp-all)
        apply (metis list.exhaust-sel)
      by blast+
  qed
qed
```

```
lemma Mprefix-Sync-distr-bis:
  ‹(Mprefix A P) ⟦S⟧ (Mprefix B Q) =
   (□x∈A − S → (P x ⟦S⟧ Mprefix B Q)) □ (□y∈B − S → (Mprefix A P ⟦S⟧ Q
y)) □ (□x∈A ∩ B ∩ S → (P x ⟦S⟧ Q x))›
  by (subst Mprefix-Sync-distr[of ‹A − S› S ‹A ∩ S› ‹B − S› ‹B ∩ S›, simplified
Un-Diff-Int])
     (simp-all add: Int-commute inf-left-commute)
```

lemmas *Mprefix-Sync-distr-subset* = *Mprefix-Sync-distr*[**where** *A* = ‹{}› **and** *B* = ‹{}›, *simplified*, *simplified Mprefix-STOP Det-STOP trans*[*OF Det-commute Det-STOP*]]
and *Mprefix-Sync-distr-indep* = *Mprefix-Sync-distr*[**where** *A′*= ‹{}› **and** *B′*= ‹{}›, *simplified*, *simplified Mprefix-STOP Det-STOP*]
and *Mprefix-Sync-distr-right* = *Mprefix-Sync-distr*[**where** *A* = ‹{}› **and** *B′*= ‹{}›, *simplified*, *simplified Mprefix-STOP Det-STOP trans*[*OF Det-commute Det-STOP*], *rotated*]
and *Mprefix-Sync-distr-left* = *Mprefix-Sync-distr*[**where** *A′*= ‹{}› **and** *B* = ‹{}›, *simplified*, *simplified Mprefix-STOP Det-STOP trans*[*OF Det-commute Det-STOP*]]

lemma *Mprefix-Sync-SKIP*: (*Mprefix B P* ⟦ *S* ⟧ *SKIP*) = (□ *x* ∈ *B* − *S* → (*P x* ⟦ *S* ⟧ *SKIP*))
proof (*auto simp add*:*Process-eq-spec-optimized*, *goal-cases*)
case (*1 x*)
then show *?case*
proof(*simp-all add*:*D-Sync D-Mprefix D-SKIP T-SKIP*, *elim exE conjE*, *goal-cases*)
case (*1 t u r v a*)
then show *?case*
apply(*intro conjI*)
using *empty-setinterleaving* **apply** *blast*
apply(*elim disjE*)
apply (*metis* (*no-types*, *lifting*) *DiffI Sync.sym emptyLeftProperty empty-iff hd-append2 image-diff-subset insertI2 list.exhaust-sel setinterleaving.simps*(*2*) *subsetCE*)
apply (*metis D-imp-front-tickFree Sync.sym TickLeftSync empty-iff empty-setinterleaving event.distinct*(*1*) *ftf-Sync21 insertI1 list.expand list.sel*(*1*) *list.set-intros*(*1*) *SyncHd-Tl SyncSameHdTl tickFree-def*)
using *Sync.sym emptyLeftNonSync emptyLeftProperty hd-in-set* **apply** *fastforce*
apply (*metis* (*no-types*, *lifting*) *DiffI Sync.sym append-Nil2 event.distinct*(*1*) *image-diff-subset insertI1 insertI2 list.sel*(*1*) *subsetCE SyncHd-Tl SyncSameHdTl*)
apply(*rule-tac x=a* **in** *exI*, *intro conjI*)
apply (*metis Sync.sym emptyLeftProperty empty-setinterleaving hd-append2 insertI1 list.sel*(*1*) *SyncHd-Tl SyncSameHdTl*)
apply(*rule-tac x=tl t* **in** *exI*, *rule-tac x=u* **in** *exI*, *rule-tac x=tl r* **in** *exI*, *rule-tac x=v* **in** *exI*)
by (*metis* (*no-types*, *lifting*) *Sync.sym empty-setinterleaving event.distinct*(*1*) *insertI1 list.sel*(*1*) *SyncHd-Tl SyncSameHdTl SyncTlEmpty tickFree-tl tl-append2*)
qed
next
case (*2 x*)
then show *?case*
proof(*simp-all add*:*D-Sync D-Mprefix D-SKIP T-SKIP*, *elim exE conjE*, *goal-cases*)
case (*1 a t u r v*)
then show *?case*

```
        apply(rule-tac x=hd x#t in exI, rule-tac x=u in exI, rule-tac x=hd x#r
in exI,
              rule-tac x=v in exI, auto)
       using hd-Cons-tl list.exhaust-sel by force+
    qed
next
  case (3 a b)
  then show ?case
  proof(simp add:F-Sync F-Mprefix F-SKIP, elim exE disjE, simp-all, goal-cases)
    case (1 t u X)
    { fix X Y A B
      assume b = (X ∪ Y) ∩ insert tick (ev ‘ B) ∪ X ∩ Y and X ∩ ev ‘ A = {}
and tick ∉ Y
      hence b ∩ ev ‘ (A − B) = {} by (rule-tac equals0I, auto)
    } note l1 = this
    from 1 show ?case
      apply(elim exE conjE disjE)
          using l1 apply simp
         apply(rule disjI2, simp)
        apply(intro disjI2 conjI)
          using empty-setinterleaving apply blast
        apply (metis (no-types, opaque-lifting) DiffI Sync.sym UnI2 contra-subsetD
emptyLeftNonSync
                emptyLeftProperty hd-in-set image-diff-subset insert-is-Un)
       apply (metis Sync.sym Un-commute emptyLeftProperty list.sel(1) list.sel(3)
neq-Nil-conv SyncTlEmpty)
       apply(intro disjI2 conjI)
         using empty-setinterleaving apply blast
          apply (metis (no-types, opaque-lifting) DiffI Sync.sym event.distinct(1)
event.inject
                          imageE image-eqI insertI1 insertI2 list.sel(1) SyncHd-Tl
SyncSameHdTl)
      apply (metis Sync.sym event.distinct(1) insertI1 list.sel(1) SyncHd-Tl Sync-
SameHdTl)
       done
  next
    case (2 t u r v)
    from 2(2) have a ∈ 𝒟 (Mprefix B P ⟦S⟧ SKIP)
      apply(simp add:D-Sync) using T-SKIP by blast
    with 2(3) have a ∈ 𝒟 (□ x ∈  B − S → (P x ⟦ S ⟧ SKIP)) by simp
    with 2(2) show ?case
      by (simp add:D-Sync D-Mprefix, rule-tac disjI2, metis)
  qed
next
  case (4 a b)
  then show ?case
  proof(simp add: F-Sync F-Mprefix F-SKIP, elim exE disjE, simp-all, goal-cases)
    case 1
    then show ?case
```

```
      apply(rule-tac disjI1, rule-tac x=[] in exI, simp, rule-tac x=[] in exI, simp)
      apply(rule-tac x=b − (ev ‘ B) in exI, rule-tac conjI)
       by (blast, rule-tac x=b − {tick} in exI, blast)
  next
    case 2
    then show ?case
    proof (elim conjE exE, elim disjE exE, simp-all, goal-cases)
      case (1 aa t u X)
      then show ?case
        apply(erule-tac conjE, erule-tac exE, simp-all)
        apply(rule-tac disjI1, rule-tac x=(ev aa)#t in exI, rule-tac x=u in exI,
              rule-tac x=X in exI, intro conjI, simp-all)
        apply auto[1]
        by (metis (no-types, lifting) DiffE event.distinct(1) event.inject
            imageE insertE list.exhaust-sel SyncSingleHeadAdd)
    next
      case (2 aa t u r v)
      from 2(2) have a ∈ 𝒟 (□ x ∈  B − S → (P x ⟦ S ⟧ SKIP))
        apply(simp add:D-Mprefix D-Sync) by (metis 2(1) 2(3) 2(4))
      with 2(5) have a ∈ 𝒟 (Mprefix B P ⟦S⟧ SKIP) by simp
      with 2(2) show ?case by (simp add:D-Sync D-Mprefix)
    qed
  qed
qed
```

lemma *Sync-Ndet-left-distrib*: $((P \sqcap Q) \llbracket S \rrbracket M) = ((P \llbracket S \rrbracket M) \sqcap (Q \llbracket S \rrbracket M))$
by (*auto simp*: *Process-eq-spec*, *simp-all add*: *D-Sync F-Sync D-Ndet F-Ndet T-Ndet*) *blast+*

lemma *Sync-Ndet-right-distrib*: $(M \llbracket S \rrbracket (P \sqcap Q)) = ((M \llbracket S \rrbracket P) \sqcap (M \llbracket S \rrbracket Q))$
by (*metis Sync-commute Sync-Ndet-left-distrib*)

lemma *prefix-Sync-SKIP1*: $a \notin S \Longrightarrow (a \rightarrow P \llbracket S \rrbracket SKIP) = (a \rightarrow (P \llbracket S \rrbracket SKIP))$
by (*metis* (*no-types*, *lifting*) *write0-def empty-Diff insert-Diff-if Mprefix-Sync-SKIP Mprefix-singl*)

lemma *prefix-Sync-SKIP2*: $a \in S \Longrightarrow (a \rightarrow P \llbracket S \rrbracket SKIP) = STOP$
by (*simp add*: *Mprefix-STOP Mprefix-Sync-SKIP write0-def*)

lemma *prefix-Sync-SKIP*: $(a \rightarrow P \llbracket S \rrbracket SKIP) = (\mathit{if}\ a \in S\ \mathit{then}\ STOP\ \mathit{else}\ (a \rightarrow (P \llbracket S \rrbracket SKIP)))$
by (*auto simp add*: *prefix-Sync-SKIP2 prefix-Sync-SKIP1*)

lemma *prefix-Sync1*: $\llbracket a \in S;\ b \in S;\ a \neq b \rrbracket \Longrightarrow (a \rightarrow P \llbracket S \rrbracket (b \rightarrow Q)) = STOP$
using *Mprefix-Sync-distr-subset*[*of* $\{a\}$ S $\{b\}$ $\lambda x.\ P\ \lambda x.\ Q$, *simplified*]
by (*auto*, *simp add*: *Mprefix-STOP Mprefix-singl*)

lemma *prefix-Sync2*: $a \in S \Longrightarrow ((a \rightarrow P) \llbracket S \rrbracket (a \rightarrow Q)) = (a \rightarrow (P \llbracket S \rrbracket Q))$

by (*simp add*:*write0-def Mprefix-Sync-distr-subset*[*of* {*a*} *S* {*a*} λ*x*. *P* λ*x*. *Q*, *simplified*])

lemma *prefix-Sync3*: ⟦*a* ∈ *S*; *b* ∉ *S*⟧ ⟹ (*a* → *P* ⟦*S*⟧ (*b* → *Q*)) = (*b* → ((*a* → *P*) ⟦*S*⟧ *Q*))
using *Mprefix-Sync-distr-right*[*of* {*b*} *S* {*a*} λ*x*. *P* λ*x*. *Q*, *simplified*]
by (*auto simp add*:*write0-def Sync-commute*)

lemma *Hiding-Sync-D1*: ‹*finite A* ⟹ *A* ∩ *S* = {} ⟹ 𝒟 ((*P* ⟦*S*⟧ *Q*) \ *A*) ⊆ 𝒟 ((*P* \ *A*) ⟦*S*⟧ (*Q* \ *A*))›
proof (*simp only*:*D-Sync T-Sync D-Hiding T-Hiding*, *intro subsetI CollectI*,
elim CollectE exE conjE, *elim exE CollectE disjE*, *goal-cases*)
case (*1 x t u ta ua r v*)
then show *?case* (**is** ∃ *t ua ra va*. *?P t ua ra va*)
apply (*subgoal-tac ?P* (*trace-hide ta* (*ev* ‘ *A*)) (*trace-hide ua* (*ev* ‘ *A*))
(*trace-hide r* (*ev* ‘ *A*)) ((*trace-hide v* (*ev* ‘ *A*))@*u*))
apply (*metis* (*no-types*, *lifting*))
apply(*simp-all add*:*front-tickFree-append Hiding-tickFree tickFree-def disjoint-iff-not-equal*
Hiding-interleave[*of ev* ‘ *A insert tick* (*ev* ‘ *S*) *r ta ua*])
apply(*elim conjE*, *erule-tac disjE*, *rule-tac disjI1*, *rule-tac conjI*, *case-tac tickFree ta*)
apply(*meson front-tickFree-charn self-append-conv tickFree-def*)
apply (*metis D-imp-front-tickFree emptyLeftNonSync ftf-Sync21 ftf-Sync32 in-set-conv-decomp-last*
insertI1 is-processT2-TR nonTickFree-n-frontTickFree tickFree-Nil tickFree-def)
apply(*subst disj-commute*, *rule-tac disjCI*, *simp*)
subgoal proof(*goal-cases*)
case *1*
hence *a*:*tickFree ua* **by** (*metis front-tickFree-implies-tickFree is-processT2-TR is-processT5-S7*)
from *1*(*11*) *1*(*10*) *inf-hidden*[*of A ua Q*] **obtain** *ff* **where** *isInfHiddenRun ff Q A* ∧ *ua* ∈ *range ff* **by** *auto*
with *1*(*2*,*3*,*4*,*7*) *a* **show** *?case*
apply(*rule-tac x*=*ua* **in** *exI*, *rule-tac x*=[] **in** *exI*)
using *front-tickFree-Nil tickFree-def* **by** *blast*
qed
apply(*rule-tac disjI2*, *rule-tac conjI*, *case-tac tickFree ta*)
apply(*meson front-tickFree-charn self-append-conv tickFree-def*)
apply (*metis D-imp-front-tickFree emptyLeftNonSync ftf-Sync21 ftf-Sync32 in-set-conv-decomp-last*
insertI1 is-processT2-TR nonTickFree-n-frontTickFree tickFree-Nil tickFree-def)
apply(*subst disj-commute*, *rule-tac disjCI*, *simp*)
proof(*goal-cases*)
case *1*
hence *a*:*tickFree ua* **by** (*metis front-tickFree-implies-tickFree is-processT2-TR is-processT5-S7*)
from *1*(*10*) *1*(*11*) *inf-hidden*[*of A ua P*] **obtain** *ff* **where** *isInfHiddenRun ff*

```
P A ∧ ua ∈ range ff by auto
      with 1(2,3,4,7) a show ?case
        apply(rule-tac x=ua in exI, rule-tac x=[] in exI)
        using front-tickFree-Nil tickFree-def by blast
    qed
next
  case (2 x t u f)
  then show ?case (is ∃ t ua ra va. ?P t ua ra va)
  proof(elim UnE exE conjE rangeE CollectE, goal-cases 21)
    case (21 xa)
    note aa = 21(7)[rule-format, of xa]
    with 21 show ?case
    proof(elim UnE exE conjE rangeE CollectE, goal-cases 211 212)
      case (211 taa uaa)
      then show ?case (is ∃ t ua ra va. ?P t ua ra va)
       proof (cases ∀ i. f i ∈ {s. ∃ t u. t ∈ 𝒯 P ∧ u ∈ 𝒯 Q ∧ s setinterleaves ((t,
u), ev ‘ S ∪ {tick})})
        case True
        define ftu where ftu ≡ λi. (SOME (t,u). t ∈ 𝒯 P ∧ u ∈ 𝒯 Q ∧
                                  (f i) setinterleaves ((t, u), ev ‘ S ∪ {tick}))
        define ft fu where ft ≡ fst ∘ ftu and fu ≡ snd ∘ ftu
        have inj ftu
        proof
          fix x y
          assume ftuxy: ftu x = ftu y
          obtain t u where tu:(t, u) = ftu x by (metis surj-pair)
         have ∃ t u. t ∈ 𝒯 P ∧ u ∈ 𝒯 Q ∧ f x setinterleaves ((t, u), ev ‘ S ∪ {tick})

            using True[rule-format, of x] by simp
          with tu have a:(f x) setinterleaves ((t, u), ev ‘ S ∪ {tick})
           unfolding ftu-def by (metis (mono-tags, lifting) exE-some old.prod.case
tu)
         have ∃ t u. t ∈ 𝒯 P ∧ u ∈ 𝒯 Q ∧ f y setinterleaves ((t, u), ev ‘ S ∪ {tick})

            using True[rule-format, of y] by simp
          with ftuxy tu have b:(f y) setinterleaves ((t, u), ev ‘ S ∪ {tick})
           unfolding ftu-def by (metis (mono-tags, lifting) exE-some old.prod.case
tu)
             from interleave-eq-size[OF a b] have length (f x) = length (f y) by
assumption
          with 211(6) show x = y
          by (metis add.left-neutral less-length-mono linorder-neqE-nat not-add-less2
strict-mono-def)
        qed
        hence inf-ftu: infinite (range ftu) using finite-imageD by blast
        have range ftu ⊆ range ft × range fu
          by (clarify, metis comp-apply fst-conv ft-def fu-def rangeI snd-conv)
        with inf-ftu have inf-ft-fu: infinite (range ft) ∨ infinite (range fu)
          by (meson finite-SigmaI infinite-super)
```

have *a*:*isInfHiddenRun f* (*P* ⟦*S*⟧ *Q*) *A*
using *2*(*6*) *T-Sync* **by** *blast*
{ fix *i*
from *a* **obtain** *t* **where** *f i* = *f 0* @ *t* ∧ *set t* ⊆ (*ev* ‘ *A*)
unfolding *isInfHiddenRun-1* **by** *blast*
hence *set* (*f i*) ⊆ *set* (*f 0*) ∪ (*ev* ‘ *A*) **by** *auto*
} note *b* = *this*
{ fix *x*
obtain *t u* **where** *tu*:(*t*, *u*) = *ftu x* **by** (*metis surj-pair*)
have ∃ *t u*. *t* ∈ 𝒯 *P* ∧ *u* ∈ 𝒯 *Q* ∧ *f x setinterleaves* ((*t*, *u*), *ev* ‘ *S* ∪ {*tick*})

using *True*[*rule-format*, *of x*] **by** *simp*
with *tu* **have** *a*:*t* ∈ 𝒯 *P* ∧ *u* ∈ 𝒯 *Q* ∧ (*f x*) *setinterleaves* ((*t*, *u*), *ev* ‘ *S* ∪ {*tick*})
unfolding *ftu-def* **by** (*metis* (*mono-tags*, *lifting*) *exE-some old.prod.case tu*)
hence *ft x* ∈ 𝒯 *P* ∧ *fu x* ∈ 𝒯 *Q* ∧ (*f x*) *setinterleaves* ((*ft x*, *fu x*), *ev* ‘ *S* ∪ {*tick*})
by (*metis comp-apply fst-conv ft-def fu-def snd-conv tu*)
hence *ft x* ∈ 𝒯 *P* ∧ *fu x* ∈ 𝒯 *Q* ∧ *set* (*ft x*) ∪ *set* (*fu x*) ⊆ *set* (*f x*)
∧ (*f x*) *setinterleaves* ((*ft x*, *fu x*), *ev* ‘ *S* ∪ {*tick*}) **using** *interleave-set* **by** *blast*
} note *ft-fu-f* = *this*
with *b* **have** *c*:*ft i* ∈ 𝒯 *P* ∧ *fu i* ∈ 𝒯 *Q* ∧ *set* (*ft i*) ∪ *set* (*fu i*) ⊆ *set* (*f 0*) ∪ *ev* ‘ *A* **for** *i* **by** *blast*
with *inf-ft-fu* **show** *?thesis*
proof(*elim disjE*, *goal-cases 2111 2112*)
case *2111*
have ∃ *t*′∈*range ft*. *t* = *take i t*′ ⟶ (*set t* ⊆ *set* (*f 0*) ∪ *ev* ‘ *A* ∧ *length t* ≤ *i*) **for** *i*
by *simp* (*metis 211*(*9*) *b*)
hence {*t*. ∃ *t*′∈*range ft*. *t* = *take i t*′} ⊆ {*t*. *set t* ⊆ *set* (*f 0*) ∪ *ev* ‘ *A* ∧ *length t* ≤ *i*} **for** *i*
by *auto* (*meson UnE c in-set-takeD subsetCE sup.boundedE*)
with *2*(*1*) *finite-lists-length-le*[*of set* (*f 0*) ∪ *ev* ‘ *A*] **have** *finite* {*t*. ∃ *t*′∈*range ft*. *t* = *take i t*′} **for** *i*
by (*meson List.finite-set finite-Un finite-imageI finite-subset*)
from *KoenigLemma*[*of range ft*, *OF 2111*(*2*), *rule-format*, *OF this*] **obtain** *ftf*::*nat* ⇒ *′a event list* **where**
d:*strict-mono ftf* ∧ *range ftf* ⊆ {*t*. ∃ *t*′∈*range ft*. *t* ≤ *t*′} **by** *blast*
with *c d*[*THEN conjunct2*] **have** *e*:*range ftf* ⊆ 𝒯 *P* **using** *is-processT3-ST-pref* **by** *blast*
define *ftfs* **where** *f*:*ftfs* = *ftf* ∘ (λ*n*. *n* + *length* (*f 0*))
from *e d*[*THEN conjunct1*] *strict-mono-def* **have** *f1*:*range ftfs* ⊆ 𝒯 *P* **and** *f2*:*strict-mono ftfs*
by (*auto simp add*: *strict-mono-def f*)
{ fix *i*
have *t0*:*length* (*ftfs 0*) ≥ *length* (*f 0*)
by (*metis d*[*THEN conjunct1*] *add-leE comp-def f length-strict-mono*)

```
        obtain tt where t1:ftfs i = (ftfs 0) @ tt
          by (metis f2 le0 le-list-def strict-mono-less-eq)
          from d[THEN conjunct2] f obtain j where t2:ftfs i ≤ ft j by simp
blast
        obtain tt1 where t3: ft j = (ftfs 0) @ tt @ tt1
          by (metis append.assoc le-list-def t1 t2)
        with t0 interleave-order[of f j ftfs 0 tt@tt1 ev ‘ S ∪ {tick} fu j] ft-fu-f
          have t4:set tt ⊆ set (drop (length (f 0)) (f j))
        by (metis Un-subset-iff set-append set-drop-subset-set-drop subset-Un-eq)
        with 21 isInfHiddenRun-1[of f - A] have t5: set (drop (length (f 0)) (f
j)) ⊆ ev ‘ A
          by (metis (full-types) a append-eq-conv-conj)
        with t4 have set tt ⊆ ev ‘ A by simp
        with t1 have trace-hide (ftfs i) (ev ‘ A) = trace-hide (ftfs 0) (ev ‘ A)
          by (simp add: subset-eq)
      } note f3 = this
      from d[THEN conjunct2] f obtain i where f4:ftfs 0 ≤ ft i by simp blast
      show ?case
        apply(rule-tac x=trace-hide (ft i) (ev ‘ A) in exI, rule-tac x=trace-hide
(fu i) (ev ‘ A) in exI)
        apply(rule-tac x=trace-hide (f i) (ev ‘ A) in exI, rule-tac x=u in exI)
      proof (intro conjI, goal-cases 21111 21112 21113 21114 21115)
        case 21111
        then show ?case using 2 by (simp add: 211(3))
      next
        case 21112
        then show ?case by (metis 211(4) 211(9) a Hiding-tickFree)
      next
        case 21113
        then show ?case by (metis 211(5) 211(8) 211(9))
      next
        case 21114
        then show ?case
          apply(rule Hiding-interleave)
           using 211(2) apply blast
          using ft-fu-f by simp
      next
        case 21115
        from f4 obtain u where f5:ft i = (ftfs 0) @ u by (metis le-list-def)
        have tickFree (f i) by (metis 211(4) 211(8) 211(9) Hiding-tickFree)
         with ft-fu-f have f6:tickFree (ft i) by (meson subsetCE sup.boundedE
tickFree-def)
        show ?case
          apply (rule disjI1, rule conjI, simp)
            apply(rule-tac x=ftfs 0 in exI, rule-tac x=trace-hide u (ev ‘ A) in
exI, intro conjI)
              apply (metis f5 front-tickFree-Nil front-tickFree-mono ft-fu-f Hid-
ing-fronttickFree is-processT2-TR)
          apply (metis f5 f6 tickFree-append)
```

```
            apply (simp add: f5, rule disjI2, rule-tac x=ftfs in exI)
           using f1 f2 f3 apply blast
          using elemTIselemHT[of fu i Q A, simplified T-Hiding] ft-fu-f by blast
        qed
      next
        case 2112
        have ∃t'∈range fu. t = take i t' ⟶ (set t ⊆ set (f 0) ∪ ev ‘ A ∧ length t
≤ i) for i
          by simp (metis 211(9) b)
        hence {t. ∃t'∈range fu. t = take i t'} ⊆ {t. set t ⊆ set (f 0) ∪ ev ‘ A ∧
length t ≤ i}  for i
          by auto (meson UnE c in-set-takeD subsetCE sup.boundedE)
           with 2(1) finite-lists-length-le[of set (f 0) ∪ ev ‘ A] have finite {t.
∃t'∈range fu. t = take i t'} for i
          by (meson List.finite-set finite-Un finite-imageI finite-subset)
       from KoenigLemma[of range fu, OF 2112(2), rule-format, OF this] obtain
fuf::nat ⇒ 'a event list where
          d:strict-mono fuf ∧ range fuf ⊆ {t. ∃t'∈range fu. t ≤ t'} by blast
      with c d[THEN conjunct2] have e:range fuf ⊆ 𝒯 Q using is-processT3-ST-pref
by blast
        define fufs where f:fufs = fuf ∘ (λn. n + length (f 0))
         from e d[THEN conjunct1] strict-mono-def have f1:range fufs ⊆ 𝒯 Q
and f2:strict-mono fufs
          by (auto simp add: strict-mono-def f)
        { fix i
          have t0:length (fufs 0) ≥ length (f 0)
            by (metis d[THEN conjunct1] add-leE comp-def f length-strict-mono)
          obtain tt where t1:fufs i = (fufs 0) @ tt
            by (metis f2 le0 le-list-def strict-mono-less-eq)
           from d[THEN conjunct2] f obtain j where t2:fufs i ≤ fu j by simp
blast
         obtain tt1 where t3: fu j = (fufs 0) @ tt @ tt1 by (metis append.assoc
le-list-def t1 t2)
          with t0 interleave-order[of f j fufs 0 tt@tt1 ev ‘ S ∪ {tick} ft j] ft-fu-f
            have t4:set tt ⊆ set (drop (length (f 0)) (f j))
             by (metis (no-types, opaque-lifting) Sync.sym Un-subset-iff order-trans

                                        set-append set-drop-subset-set-drop)
          with 21 isInfHiddenRun-1[of f - A] have t5: set (drop (length (f 0)) (f
j)) ⊆ ev ‘ A
            by (metis (full-types) a append-eq-conv-conj)
          with t4 have set tt ⊆ ev ‘ A by simp
          with t1 have trace-hide (fufs i) (ev ‘ A) = trace-hide (fufs 0) (ev ‘ A)
            by (simp add: subset-eq)
        } note f3 = this
       from d[THEN conjunct2] f obtain i where f4:fufs 0 ≤ fu i by simp blast
        show ?case
          apply(rule-tac x=trace-hide (fu i) (ev ‘ A) in exI,
                rule-tac x=trace-hide (ft i) (ev ‘ A) in exI)
```

```
      apply(rule-tac x=trace-hide (f i) (ev ‘ A) in exI, rule-tac x=u in exI)
    proof (intro conjI, goal-cases 21111 21112 21113 21114 21115)
      case 21111
      then show ?case using 2 by (simp add: 211(3))
    next
      case 21112
      then show ?case by (metis 211(4) 211(9) a Hiding-tickFree)
    next
      case 21113
      then show ?case by (metis 211(5) 211(8) 211(9))
    next
      case 21114
      then show ?case
        apply(rule Hiding-interleave)
         using 211(2) apply blast
        using ft-fu-f using Sync.sym by blast
    next
      case 21115
      from f4 obtain u where f5:fu i = (fufs 0) @ u by (metis le-list-def)
      have tickFree (f i) by (metis 211(4) 211(8) 211(9) Hiding-tickFree)
       with ft-fu-f have f6:tickFree (fu i) by (meson subsetCE sup.boundedE
tickFree-def)
      show ?case
        apply (rule disjI2, rule conjI, simp)
          apply(rule-tac x=fufs 0 in exI, rule-tac x=trace-hide u (ev ‘ A) in
exI, intro conjI)
            apply (metis f5 front-tickFree-Nil front-tickFree-mono ft-fu-f Hid-
ing-fronttickFree is-processT2-TR)
        apply (metis f5 f6 tickFree-append)
          apply (simp add: f5, rule disjI2, rule-tac x=fufs in exI)
         using f1 f2 f3 apply blast
        using elemTIselemHT[of ft i P A, simplified T-Hiding] ft-fu-f by blast

    qed
  qed
next
  case False
  then obtain xaa
    where f xaa ∉ {s. ∃ t u. t ∈ 𝒯 P ∧ u ∈ 𝒯 Q ∧ s setinterleaves ((t, u), ev
‘ S ∪ {tick})}
      by blast
  with 211(7)[rule-format, of xaa] obtain ta ua ra va
    where bb:front-tickFree va ∧ (tickFree ra ∨ va = []) ∧
            f xaa = ra @ va ∧ ra setinterleaves ((ta, ua), ev ‘ S ∪ {tick}) ∧
            (ta ∈ 𝒟 P ∧ ua ∈ 𝒯 Q ∨ ta ∈ 𝒟 Q ∧ ua ∈ 𝒯 P) by blast
  from 211 have cc:x = trace-hide (f xaa) (ev ‘ A) @ u by (metis (no-types,
lifting))
  from bb 211(9) 211(8) 21(4) have tickFree ra ∧ tickFree va
    by (metis Hiding-tickFree tickFree-append)
```

```
      with cc bb 211 show ?thesis
        apply(subgoal-tac ?P (trace-hide ta (ev ‘ A)) (trace-hide ua (ev ‘ A))
                        (trace-hide ra (ev ‘ A))  ((trace-hide va (ev ‘ A))@u))
         apply (metis (no-types, lifting))
           apply(simp-all add:front-tickFree-append Hiding-tickFree tickFree-def
disjoint-iff-not-equal
                                Hiding-interleave[of ev ‘ A insert tick (ev ‘ S) ra ta
ua])
          apply(elim conjE disjE, rule-tac disjI1, rule-tac conjI, case-tac tickFree
ta)
            apply(meson front-tickFree-charn self-append-conv tickFree-def)
                 apply (metis D-imp-front-tickFree emptyLeftNonSync ftf-Sync21
ftf-Sync32 in-set-conv-decomp-last
                              insertI1 is-processT2-TR nonTickFree-n-frontTickFree
tickFree-Nil tickFree-def)
       apply(subst disj-commute, rule-tac disjCI, simp)
        subgoal proof(goal-cases 2111)
        case 2111
      hence a:tickFree ua by (metis front-tickFree-implies-tickFree is-processT2-TR
is-processT5-S7)
        from 2111(20) 2111(21) inf-hidden[of A ua Q] obtain ff
          where isInfHiddenRun ff Q A ∧ ua ∈ range ff by auto
        with 2111(2,3,4,7) a show ?case
          apply(rule-tac x=ua in exI, rule-tac x=[] in exI)
          using front-tickFree-Nil tickFree-def by blast
        qed
        apply(rule-tac disjI2, rule-tac conjI, case-tac tickFree ta)
          apply(meson front-tickFree-charn self-append-conv tickFree-def)
       apply (metis D-imp-front-tickFree emptyLeftNonSync ftf-Sync21 ftf-Sync32
in-set-conv-decomp-last
                 insertI1 is-processT2-TR nonTickFree-n-frontTickFree tickFree-Nil
tickFree-def)
        apply(subst disj-commute, rule-tac disjCI, simp)
        proof(goal-cases 2111)
          case 2111
       hence a:tickFree ua by (metis front-tickFree-implies-tickFree is-processT2-TR
is-processT5-S7)
          from 2111(20) 2111(21) inf-hidden[of A ua P] obtain ff
            where isInfHiddenRun ff P A ∧ ua ∈ range ff by auto
          with 2111(2,3,4,7) a show ?case
            apply(rule-tac x=ua in exI, rule-tac x=[] in exI)
            using front-tickFree-Nil tickFree-def by blast
        qed
    qed
  next
    case (212 ta ua r v)
    then show ?case (is ∃ t ua ra va. ?P t ua ra va)
      apply (subgoal-tac ?P (trace-hide ta (ev ‘ A)) (trace-hide ua (ev ‘ A))
                        (trace-hide r (ev ‘ A)) ((trace-hide v (ev ‘ A))@u))
```

```
      apply (metis (no-types, lifting))
       apply(simp-all add:front-tickFree-append Hiding-tickFree tickFree-def dis-
joint-iff-not-equal
                    Hiding-interleave[of ev ‘ A insert tick (ev ‘ S) r ta ua])
     apply(erule-tac disjE, rule-tac disjI1, rule-tac conjI, case-tac tickFree ta)
       apply(meson front-tickFree-charn self-append-conv tickFree-def)
       apply (metis D-imp-front-tickFree emptyLeftNonSync ftf-Sync21 ftf-Sync32
in-set-conv-decomp-last
                 insertI1 is-processT2-TR nonTickFree-n-frontTickFree tickFree-Nil
tickFree-def)
     apply(subst disj-commute, rule-tac disjCI, simp)
     subgoal proof(goal-cases 2121)
       case 2121
     hence a:tickFree ua by (metis front-tickFree-implies-tickFree is-processT2-TR
is-processT5-S7)
       from 2121(14) 2121(13) inf-hidden[of A ua Q] obtain ff
         where isInfHiddenRun ff Q A ∧ ua ∈ range ff by auto
       with 2121(2,3,4,7) a show ?case
         apply(rule-tac x=ua in exI, rule-tac x=[] in exI)
         using front-tickFree-Nil tickFree-def by blast
     qed
     apply(rule-tac disjI2, rule-tac conjI, case-tac tickFree ta)
       apply(meson front-tickFree-charn self-append-conv tickFree-def)
       apply (metis D-imp-front-tickFree emptyLeftNonSync ftf-Sync21 ftf-Sync32
in-set-conv-decomp-last
                 insertI1 is-processT2-TR nonTickFree-n-frontTickFree tickFree-Nil
tickFree-def)
     apply(subst disj-commute, rule-tac disjCI, simp)
     proof(goal-cases 2121)
       case 2121
     hence a:tickFree ua by (metis front-tickFree-implies-tickFree is-processT2-TR
is-processT5-S7)
       from 2121(14) 2121(13) inf-hidden[of A ua P] obtain ff
         where isInfHiddenRun ff P A ∧ ua ∈ range ff by auto
       with 2121(2,3,4,7) a show ?case
         apply(rule-tac x=ua in exI, rule-tac x=[] in exI)
         using front-tickFree-Nil tickFree-def by blast
     qed
   qed
  qed
qed

lemma Hiding-Sync-D2:
  assumes *:A ∩ S = {}
  shows     ‹𝒟 ((P \ A) ⟦S⟧ (Q \ A)) ⊆ 𝒟 ((P ⟦S⟧ Q) \ A)›
proof −
  { fix P Q
    have {s. ∃ t u r v. front-tickFree v ∧ (tickFree r ∨ v = []) ∧
          s = r @ v ∧ r setinterleaves ((t, u), insert tick (ev ‘ S)) ∧
```

```
        (t ∈ 𝒟 (P \ A) ∧ u ∈ 𝒯 (Q \ A))} ⊆ 𝒟 ((P ⟦S⟧ Q) \ A)
  proof(simp add:D-Hiding D-Sync, intro subsetI CollectI, elim exE conjE Col-
lectE, goal-cases a)
    case (a x t u r v ta1 ta2)
    from a(1,3−9) show ?case
    proof(erule-tac disjE, goal-cases)
      case 1
       with interleave-append[of r trace-hide ta1 (ev ‘ A) ta2 insert tick (ev ‘ S)
u]
      obtain u1 u2 r1 r2
        where a1:u = u1 @ u2 and a2:r = r1 @ r2 and
            a3:r1 setinterleaves ((trace-hide ta1 (ev ‘ A), u1), insert tick (ev ‘ S))
and
             a4:r2 setinterleaves ((ta2, u2), insert tick (ev ‘ S)) by blast
      with 1 show ?case
      proof(auto simp only:T-Hiding, goal-cases 11 12 13)
        case (11 ua)
        from trace-hide-append[OF 11(12)]
       obtain ua1 where a5:u1 = trace-hide ua1 (ev ‘ A) ∧ ua1 ≤ ua ∧ ua1 ∈
𝒯 Q
          using 11(13) F-T is-processT3-ST-pref le-list-def by blast
        from interleave-Hiding[OF - 11(10)[simplified a5]] obtain ra1
          where a6:r1 = trace-hide ra1 (ev ‘ A) ∧ ra1 setinterleaves ((ta1, ua1),
insert tick (ev ‘ S))
          using ∗ by blast
        from a(2) 11 a5 a6 show ?case
          apply(exI ra1, exI r2@v, intro conjI, simp-all (asm-lr))
         apply (metis a(5) append-Nil2 front-tickFree-append front-tickFree-mono
ftf-Sync is-processT2-TR)
           apply (metis equals0D ftf-Sync1 ftf-Sync21 insertI1 tickFree-def)
          using front-tickFree-Nil by blast
      next
        case (12 ua1 ua2)
        then show ?case
        proof(cases u1 ≤ trace-hide ua1 (ev ‘ A))
          case True
            then obtain uu1 where u1@uu1 = trace-hide ua1 (ev ‘ A) using
le-list-def by blast
          with trace-hide-append[OF this] obtain uaa1
           where a5:u1 = trace-hide uaa1 (ev ‘ A) ∧ uaa1 ≤ ua1 ∧ uaa1 ∈ 𝒯 Q
            using 12(15) NT-ND is-processT3-ST-pref le-list-def by blast
          from interleave-Hiding[OF - 12(10)[simplified a5]] obtain rr1
            where a6:r1 = trace-hide rr1 (ev ‘ A) ∧
                    rr1 setinterleaves ((ta1, uaa1), insert tick (ev ‘ S))
            using ∗ by blast
          from a(2) 12 a5 a6 show ?thesis
          apply(exI rr1, exI r2@v, intro conjI, simp-all (asm-lr))
         apply (metis a(5) append-Nil2 front-tickFree-append front-tickFree-mono
ftf-Sync is-processT2-TR)
```

```
          apply (metis equals0D ftf-Sync1 ftf-Sync21 insertI1 tickFree-def)
         using front-tickFree-Nil by blast
       next
         case False
         with 12(14) obtain uaa1 where u1 = trace-hide ua1 (ev ‘ A)@uaa1
           by (metis append-eq-append-conv2 le-list-def)
          with a3 interleave-append-sym[of r1 trace-hide ta1 (ev ‘ A) insert tick
(ev ‘ S)
              trace-hide ua1 (ev ‘ A) uaa1]
         obtain taa1 taa2 ra1 ra2
           where aa1:trace-hide ta1 (ev ‘ A) = taa1 @ taa2 and aa2:r1 = ra1
@ ra2 and
             aa3:ra1 setinterleaves ((taa1, trace-hide ua1 (ev ‘ A)), insert tick (ev
‘ S)) and
             aa4:ra2 setinterleaves ((taa2,uaa1), insert tick (ev ‘ S)) by blast
         with trace-hide-append[OF aa1[symmetric]]
        obtain taaa1 where a5:taa1 = trace-hide taaa1 (ev ‘ A) ∧ taaa1 ≤ ta1
∧ taaa1 ∈ 𝒯 P
           using 12(7) D-T is-processT3-ST-pref le-list-def by blast
         with interleave-Hiding[OF - aa3[simplified a5]] obtain rr1
           where a6:ra1 = trace-hide rr1 (ev ‘ A) ∧ rr1 setinterleaves ((taaa1,
ua1), insert tick (ev ‘ S))
           using ∗ by blast
         from a(2) 12 a5 show ?thesis
           apply(exI rr1, exI ra2@r2@v, intro conjI, simp-all (asm-lr))
             apply (metis aa2 front-tickFree-append front-tickFree-mono ftf-Sync
                        Hiding-tickFree self-append-conv tickFree-append)
             apply (metis a6 ftf-Sync1 le-list-def tickFree-append tickFree-def)
            using a6 aa2 apply blast
           using Sync.sym a6 front-tickFree-Nil by blast
       qed
      next
       case (13 ua fa xa)
        with isInfHiddenRun-1[of fa Q] have a0:∀ i. ∃ t. fa i = fa 0 @ t ∧ set t
⊆ (ev ‘ A) by simp
       define tlf where tlf ≡ λi. drop (length (fa 0)) (fa i)
      have tlf-def2:(fa x) = (fa 0) @ (tlf x) for x by (metis a0 append-eq-conv-conj
tlf-def)
       { fix x y
         assume ∗:(x::nat) < y
         hence fa x = fa 0 @ tlf x ∧ fa y = fa 0 @ tlf y by (metis tlf-def2)
         hence  tlf x < tlf y
           using strict-monoD[OF 13(16), of x y] by (simp add: A ∗ le-list-def)
       } note tlf-strict-mono = this
       have tlf-range:set (tlf i) ⊆ (ev ‘ A) for i
         by (metis a0 append-eq-conv-conj tlf-def)
       from 13 show ?case
       proof(cases u1 ≤ trace-hide (fa xa) (ev ‘ A))
         case True
```

then obtain *uu1* **where** *u1@uu1 = trace-hide (fa xa) (ev ‘ A)* **using** *le-list-def* **by** *blast*
with *trace-hide-append[OF this]*
obtain *uaa1* **where** *a5:u1 = trace-hide uaa1 (ev ‘ A) ∧ uaa1 ≤ (fa xa) ∧ uaa1 ∈ 𝒯 Q*
by *(metis 13(17) is-processT3-ST le-list-def)*
from *interleave-Hiding[OF - 13(10)[simplified a5]]* **obtain** *rr1*
where *a6:r1 = trace-hide rr1 (ev ‘ A) ∧ rr1 setinterleaves ((ta1, uaa1), insert tick (ev ‘ S))*
using ∗ **by** *blast*
from *a(2) 13 a5 a6* **show** *?thesis*
apply(*exI rr1, exI r2@v, intro conjI, simp-all (asm-lr)*)
apply *(metis a(5) append-Nil2 front-tickFree-append front-tickFree-mono ftf-Sync is-processT2-TR)*
apply *(metis equals0D ftf-Sync1 ftf-Sync21 insertI1 tickFree-def)*
using *front-tickFree-Nil* **by** *blast*
next
case *False*
with *13(14)* **obtain** *uaa1* **where** *u1 = trace-hide (fa 0) (ev ‘ A)@uaa1*
by *(metis 13(18) append-eq-append-conv2 le-list-def)*
with *a3 interleave-append-sym[of r1 trace-hide ta1 (ev ‘ A) insert tick (ev ‘ S)*
trace-hide (fa 0) (ev ‘ A) uaa1]
obtain *taa1 taa2 ra1 ra2*
where *aa1:trace-hide ta1 (ev ‘ A) = taa1 @ taa2* **and** *aa2:r1 = ra1 @ ra2* **and**
aa3:ra1 setinterleaves ((taa1, trace-hide (fa 0) (ev ‘ A)), insert tick (ev ‘ S)) **and**
aa4:ra2 setinterleaves ((taa2,uaa1), insert tick (ev ‘ S)) **by** *blast*
with *trace-hide-append[OF aa1[symmetric]]*
obtain *taaa1* **where** *a5:taa1 = trace-hide taaa1 (ev ‘ A) ∧ taaa1 ≤ ta1 ∧ taaa1 ∈ 𝒯 P*
using *13(7) D-T is-processT3-ST-pref le-list-def* **by** *blast*
with *interleave-Hiding[OF - aa3[simplified a5]]* **obtain** *rr1*
where *a6:ra1 = trace-hide rr1 (ev ‘ A) ∧*
rr1 setinterleaves ((taaa1, (fa 0)), insert tick (ev ‘ S))
using ∗ **by** *blast*
from *a(2) 13 a0 a5 a6 aa1 aa2 aa3* **show** *?thesis*
apply(*exI rr1, exI ra2@r2@v, intro conjI, simp-all (asm-lr)*)
apply *(metis front-tickFree-append front-tickFree-mono ftf-Sync*
Hiding-tickFree self-append-conv tickFree-append)
apply *(metis a6 ftf-Sync1 le-list-def tickFree-append tickFree-def)*
proof *(rule-tac disjI2, exI λi. rr1@(tlf i), intro conjI, goal-cases 131 132 133 134)*
case *131*
then show *?case*
apply(*rule-tac strict-monoI, drule-tac tlf-strict-mono,rule-tac less-append*) **by** *assumption*
next

```
            case 132
           have (rr1@tlf i) setinterleaves ((taaa1, fa i),insert tick (ev ‘ S)) for i
              apply(subst tlf-def2, rule interleave-append-tail-sym[OF a6[THEN
conjunct2], of tlf i])
              using ∗ tlf-range by blast
            then show ?case
              unfolding T-Sync using a5 13(17) by auto
          next
            case 133
            from tlf-range have trace-hide (y @ tlf i) (ev ‘ A) = trace-hide y (ev
‘ A) for i y
              apply(induct y) using filter-empty-conv apply fastforce by simp
            then show ?case by simp
          next
            case 134
            have tlf 0 = []
              by (simp add: tlf-def)
            then show ?case by simp
          qed
        qed
      qed
    next
      case 2
      from 2(8) obtain nta::nat and ff::nat ⇒ ′a event list
        where strict-mono ff and ff:(∀ i. ff i ∈ 𝒯 P) ∧ ta1 = ff nta ∧
              (∀ i. trace-hide (ff i) (ev ‘ A) = trace-hide (ff 0) (ev ‘ A)) by blast
      define f where f ≡ (λi. ff (i + nta))
       with ff have f1:strict-mono f and f2:(∀ i. f i ∈ 𝒯 P) and f3:ta1 = f 0
and
              f4:(∀ i. trace-hide (f i) (ev ‘ A) = trace-hide ta1 (ev ‘ A))
     apply auto apply(rule strict-monoI, rule ‹strict-mono ff›[THEN strict-monoD])
        by simp metis
      with isInfHiddenRun-1[of f P] have a0:∀ i. ∃ t. f i = f 0 @ t ∧ set t ⊆ (ev
‘ A) by simp
      define tlf where tlf ≡ λi. drop (length (f 0)) (f i)
     have tlf-def2:(f x) = ta1 @ (tlf x) for x by (metis a0 f3 append-eq-conv-conj
tlf-def)
      { fix x y
        assume ∗:(x::nat) < y
        hence f x = f 0 @ tlf x ∧ f y = f 0 @ tlf y by (metis f3 tlf-def2)
        hence  tlf x < tlf y
          using strict-monoD[OF f1, of x y] by (simp add: A ∗ le-list-def)
      } note tlf-strict-mono = this
      have tlf-range:set (tlf i) ⊆ (ev ‘ A) for i
        by (metis a0 append-eq-conv-conj tlf-def)
      with 2 interleave-append[of r trace-hide ta1 (ev ‘ A) ta2 insert tick (ev ‘ S)
u]
      obtain u1 u2 r1 r2
        where a1:u = u1 @ u2 and a2:r = r1 @ r2 and
```

```
            a3:r1 setinterleaves ((trace-hide ta1 (ev ‘ A), u1), insert tick (ev ‘ S))
and
            a4:r2 setinterleaves ((ta2, u2), insert tick (ev ‘ S)) by blast
      with 2(1−6) f1 f2 f3 f4 show ?case
      proof(auto simp only:T-Hiding, goal-cases 21 22 23)
        case (21 ua)
        from trace-hide-append[OF 21(14)] obtain ua1
          where a5:u1 = trace-hide ua1 (ev ‘ A) ∧ ua1 ≤ ua ∧ ua1 ∈ 𝒯 Q
          using 21(15) F-T is-processT3-ST-pref le-list-def by blast
        from interleave-Hiding[OF - a3[simplified a5]] obtain ra1
          where a6:r1 = trace-hide ra1 (ev ‘ A) ∧ ra1 setinterleaves ((ta1, ua1),
insert tick (ev ‘ S))
          using ∗ by blast
        from a(2) 21 a5 a6 show ?case
          apply(exI ra1, exI r2@v, intro conjI, simp-all (asm-lr))
         apply (metis a(5) append-Nil2 front-tickFree-append front-tickFree-mono
ftf-Sync is-processT2-TR)
          apply (metis equals0D ftf-Sync1 ftf-Sync21 insertI1 tickFree-def)
           proof (rule-tac disjI2, exI λi. ra1@(tlf i), intro conjI, goal-cases 211
212 213 214)
            case 211
            then show ?case
                    by(rule-tac strict-monoI, drule-tac tlf-strict-mono, rule-tac
less-append)(assumption)
          next
            case 212
            have (ra1@tlf i) setinterleaves ((f i, ua1),insert tick (ev ‘ S)) for i
                  apply(subst tlf-def2, rule interleave-append-tail[OF a6[THEN
conjunct2], of tlf i])
              using ∗ tlf-range by blast
            then show ?case
              unfolding T-Sync using a5 21(15) f2 by auto
          next
            case 213
            from tlf-range have trace-hide (y @ tlf i) (ev ‘ A) = trace-hide y (ev
‘ A) for i y
              apply(induct y) using filter-empty-conv apply fastforce by simp
            then show ?case by simp
          next
            case 214
            have tlf 0 = []
              by (simp add: tlf-def)
            then show ?case by simp
          qed
      next
        case (22 ua1 ua2)
        then show ?case
        proof(cases u1 ≤ trace-hide ua1 (ev ‘ A))
          case True
```

```
          then obtain uu1 where  u1@uu1 = trace-hide ua1 (ev ‘ A) using
le-list-def by blast
        from trace-hide-append[OF this]
        obtain uaa1 where a5:u1 = trace-hide uaa1 (ev ‘ A) ∧ uaa1 ≤ ua1 ∧
uaa1 ∈ 𝒯 Q
          using 22(17) D-T is-processT3-ST le-list-def by blast
        from interleave-Hiding[OF - a3[simplified a5]]
        obtain ra1
            where a6:r1 = trace-hide ra1 (ev ‘ A) ∧ ra1 setinterleaves ((ta1,
uaa1), insert tick (ev ‘ S))
          using ∗ by blast
        from a(2) 22 a5 a6 show ?thesis
          apply(exI ra1, exI r2@v, intro conjI, simp-all (asm-lr))
        apply (metis a(5) append-Nil2 front-tickFree-append front-tickFree-mono
ftf-Sync is-processT2-TR)
          apply (metis equals0D ftf-Sync1 ftf-Sync21 insertI1 tickFree-def)
          proof (rule-tac disjI2, exI λi. ra1@(tlf i), intro conjI, goal-cases 211
212 213 214)
          case 211
        then show ?case apply(rule-tac strict-monoI, drule-tac tlf-strict-mono,
rule-tac less-append) by assumption
          next
          case 212
          have (ra1@tlf i) setinterleaves ((f i, uaa1),insert tick (ev ‘ S)) for i
                apply(subst tlf-def2, rule interleave-append-tail[OF a6[THEN
conjunct2], of tlf i])
            using ∗ tlf-range by blast
          then show ?case
            unfolding T-Sync using a5 22(15) f2 by auto
          next
          case 213
          from tlf-range have trace-hide (y @ tlf i) (ev ‘ A) = trace-hide y (ev
‘ A) for i y
            apply(induct y) using filter-empty-conv apply fastforce by simp
          then show ?case by simp
          next
          case 214
          have tlf 0 = []
            by (simp add: tlf-def)
          then show ?case by simp
          qed
      next
        case False
        with 22(16) obtain uaa1 where u1 = trace-hide ua1 (ev ‘ A)@uaa1
          by (metis append-eq-append-conv2 le-list-def)
          with a3 interleave-append-sym[of r1 trace-hide ta1 (ev ‘ A) insert tick
(ev ‘ S)
            trace-hide ua1 (ev ‘ A) uaa1]
        obtain taa1 taa2 ra1 ra2
```

where *aa1*:*trace-hide ta1* (*ev* ‘ *A*) = *taa1* @ *taa2* **and** *aa2*:*r1* = *ra1* @ *ra2* **and**
aa3:*ra1 setinterleaves* ((*taa1*, *trace-hide ua1* (*ev* ‘ *A*)), *insert tick* (*ev* ‘ *S*)) **and**
aa4:*ra2 setinterleaves* ((*taa2*,*uaa1*), *insert tick* (*ev* ‘ *S*)) **by** *blast*
with *trace-hide-append*[*OF aa1*[*symmetric*]]
obtain *taaa1* **where** *a5*:*taa1* = *trace-hide taaa1* (*ev* ‘ *A*) ∧ *taaa1* ≤ *ta1* ∧ *taaa1* ∈ $\mathcal{T}$ *P*
using *f2 f3 is-processT3-ST-pref le-list-def* **by** *metis*
with *interleave-Hiding*[*OF - aa3*[*simplified a5*]] **obtain** *rr1*
where *a6*:*ra1* = *trace-hide rr1* (*ev* ‘ *A*) ∧
rr1 setinterleaves ((*taaa1*, *ua1*), *insert tick* (*ev* ‘ *S*))
using ∗ **by** *blast*
from *a*(*2*) *22 a5* **show** *?thesis*
apply(*exI rr1*, *exI ra2@r2@v*, *intro conjI*, *simp-all* (*asm-lr*))
apply (*metis a*(*5*) *a*(*8*) *aa2 front-tickFree-append front-tickFree-mono ftf-Sync Hiding-tickFree*
is-processT2-TR self-append-conv tickFree-append)
apply (*metis a6 ftf-Sync1 le-list-def tickFree-append tickFree-def*)
using *a6 aa2* **apply** *blast*
using *Sync.sym a6*[*THEN conjunct2*] *front-tickFree-Nil* **by** (*metis append-Nil2*)
qed
next
case (*23 ua fa xa*)
with *isInfHiddenRun-1*[*of fa Q*] **have** *a0*:∀ *i*. ∃ *t*. *fa i* = *fa 0* @ *t* ∧ *set t* ⊆ (*ev* ‘ *A*) **by** *simp*
define *tlfa* **where** *tlfa* ≡ λ*i*. *drop* (*length* (*fa 0*)) (*fa i*)
have *tlfa-def2*:(*fa x*) = (*fa 0*) @ (*tlfa x*) **for** *x* **by** (*metis a0 append-eq-conv-conj tlfa-def*)
{ fix *x y*
assume ∗:(*x*::*nat*) < *y*
hence *fa x* = *fa 0* @ *tlfa x* ∧ *fa y* = *fa 0* @ *tlfa y* **by** (*metis tlfa-def2*)
hence *tlfa x* < *tlfa y*
using *strict-monoD*[*OF 23*(*18*), *of x y*] **by** (*simp add*: *A* ∗ *le-list-def*)
} note *tlfa-strict-mono* = *this*
have *tlfa-range*:*set* (*tlfa i*) ⊆ (*ev* ‘ *A*) **for** *i*
by (*metis a0 append-eq-conv-conj tlfa-def*)
from *23* **show** *?case*
proof(*cases u1* ≤ *trace-hide* (*fa xa*) (*ev* ‘ *A*))
case *True*
then obtain *uu1* **where** *u1@uu1* = *trace-hide* (*fa xa*) (*ev* ‘ *A*) **using** *le-list-def* **by** *blast*
with *trace-hide-append*[*OF this*]
obtain *uaa1* **where** *a5*:*u1* = *trace-hide uaa1* (*ev* ‘ *A*) ∧ *uaa1* ≤ (*fa xa*) ∧ *uaa1* ∈ $\mathcal{T}$ *Q*
by (*metis 23*(*19*) *is-processT3-ST le-list-def*)
from *interleave-Hiding*[*OF - a3*[*simplified a5*]]
obtain *rr1*

where *a6*:*r1* = *trace-hide rr1* (*ev* ‘ *A*) ∧ *rr1 setinterleaves* ((*ta1*, *uaa1*), *insert tick* (*ev* ‘ *S*))
using ∗ **by** *blast*
from *a*(*2*) *23 a5 a6* **show** *?thesis*
apply(*exI rr1*, *exI r2@v*, *intro conjI*, *simp-all* (*asm-lr*))
apply (*metis a*(*5*) *append-Nil2 front-tickFree-append front-tickFree-mono ftf-Sync is-processT2-TR*)
apply (*metis equals0D ftf-Sync1 ftf-Sync21 insertI1 tickFree-def*)
proof (*rule-tac disjI2*, *exI* λ*i*. *rr1*@(*tlf i*), *intro conjI*, *goal-cases 211 212 213 214*)
case *211*
then show *?case* **apply**(*rule-tac strict-monoI*, *drule-tac tlf-strict-mono*, *rule-tac less-append*) **by** *assumption*
next
case *212*
have (*rr1@tlf i*) *setinterleaves* ((*f i*, *uaa1*),*insert tick* (*ev* ‘ *S*)) **for** *i*
apply(*subst tlf-def2*, *rule interleave-append-tail*[*OF a6*[*THEN conjunct2*], *of tlf i*])
using ∗ *tlf-range* **by** *blast*
then show *?case*
unfolding *T-Sync* **using** *a5 23*(*17*) *f2* **by** *auto*
next
case *213*
from *tlf-range* **have** *trace-hide* (*y* @ *tlf i*) (*ev* ‘ *A*) = *trace-hide y* (*ev* ‘ *A*) **for** *i y*
apply(*induct y*) **using** *filter-empty-conv* **apply** *fastforce* **by** *simp*
then show *?case* **by** *simp*
next
case *214*
have *tlf 0* = []
by (*simp add*: *tlf-def*)
then show *?case* **by** *simp*
qed
next
case *False*
with *23*(*16*) **obtain** *uaa1* **where** *u1* = *trace-hide* (*fa 0*) (*ev* ‘ *A*)@*uaa1*
by (*metis 23*(*20*) *append-eq-append-conv2 le-list-def*)
with *a3 interleave-append-sym*[*of r1 trace-hide ta1* (*ev* ‘ *A*) *insert tick* (*ev* ‘ *S*) *trace-hide* (*fa 0*) (*ev* ‘ *A*) *uaa1*]
obtain *taa1 taa2 ra1 ra2*
where *aa1*:*trace-hide ta1* (*ev* ‘ *A*) = *taa1* @ *taa2* **and** *aa2*:*r1* = *ra1* @ *ra2* **and**
aa3:*ra1 setinterleaves* ((*taa1*, *trace-hide* (*fa 0*) (*ev* ‘ *A*)), *insert tick* (*ev* ‘ *S*)) **and**
aa4:*ra2 setinterleaves* ((*taa2*,*uaa1*), *insert tick* (*ev* ‘ *S*)) **by** *blast*
with *trace-hide-append*[*OF aa1*[*symmetric*]]
obtain *taaa1* **where** *a5*:*taa1* = *trace-hide taaa1* (*ev* ‘ *A*) ∧ *taaa1* ≤ *ta1* ∧ *taaa1* ∈ $\mathcal{T}$ *P*
by (*metis f2 f3 is-processT3-ST le-list-def*)

```
          with interleave-Hiding[OF - aa3[simplified a5]] obtain rr1
            where a6:ra1 = trace-hide rr1 (ev ‘ A) ∧
                    rr1 setinterleaves ((taaa1, (fa 0)), insert tick (ev ‘ S))
            using ∗ by blast
          from a(2) 23 a0 a5 a6 aa1 aa2 aa3 show ?thesis
            apply(exI rr1, exI ra2@r2@v, intro conjI, simp-all (asm-lr))
                apply (metis a(5) a(8) front-tickFree-append front-tickFree-mono
ftf-Sync
                                    Hiding-tickFree is-processT2-TR self-append-conv)
             apply (metis a6 ftf-Sync1 le-list-def tickFree-append tickFree-def)
           proof (rule-tac disjI2, exI λi. rr1@(tlfa i), intro conjI, goal-cases 131
132 133 134)
             case 131
             then show ?case
                    by(rule-tac strict-monoI, drule-tac tlfa-strict-mono, rule-tac
less-append) assumption
           next
             case 132
            have (rr1@tlfa i) setinterleaves ((taaa1, fa i),insert tick (ev ‘ S)) for i
               apply(subst tlfa-def2, rule interleave-append-tail-sym[OF a6[THEN
conjunct2], of tlfa i])
               using ∗ tlfa-range by blast
             then show ?case
               unfolding T-Sync using a5 23(19) by auto
           next
             case 133
              from tlfa-range have trace-hide (y @ tlfa i) (ev ‘ A) = trace-hide y
(ev ‘ A) for i y
               apply(induct y) using filter-empty-conv apply fastforce by simp
             then show ?case by simp
           next
             case 134
             have tlfa 0 = []
               by (simp add: tlfa-def)
             then show ?case by simp
           qed
         qed
       qed
     qed
   qed
  }
  from this[of P Q] this[of Q P] Sync-commute[of P S Q] show ?thesis
    by (simp add:D-Sync T-Sync) blast
qed

lemma Hiding-Sync:
  ‹finite A ⟹ A ∩ S = {} ⟹ ((P ⟦S⟧ Q) \ A) = ((P \ A) ⟦S⟧ (Q \ A))›
proof(auto simp add: Process-eq-spec-optimized subset-antisym[OF Hiding-Sync-D1
Hiding-Sync-D2],
```

```
    simp-all add: F-Sync F-Hiding,
    goal-cases)
  case (1 a b)
  then show ?case
  proof(elim disjE, goal-cases 11 12)
    case 11
    then show ?case
    proof(elim exE conjE, elim disjE, goal-cases 111 112)
      case (111 c)
      from 111(7) obtain t u X Y where
        a: (t, X) ∈ ℱ P and
        b: (u, Y) ∈ ℱ Q and
        c: c setinterleaves ((t, u), insert tick (ev ‘ S)) and
        d: b ∪ ev ‘ A = (X ∪ Y) ∩ insert tick (ev ‘ S) ∪ X ∩ Y by auto
      from 1(2) d have e:ev ‘ A ⊆ X ∩ Y by blast
      from a b c d e 111(2,3) show ?case
        apply(rule-tac disjI1)
        apply(rule-tac x=trace-hide t (ev ‘ A) in exI, rule-tac x=trace-hide u (ev ‘
A) in exI)
        apply(rule-tac x=X − ((ev ‘ A) − b) in exI, intro conjI disjI1, rule-tac
x=t in exI)
        apply (metis Int-subset-iff Un-Diff-Int Un-least is-processT4 sup-ge1)
        apply(rule-tac x=Y − ((ev ‘ A) − b) in exI, intro conjI disjI1, rule-tac
x=u in exI)
        apply (metis Int-subset-iff Un-Diff-Int Un-least is-processT4 sup-ge1)
        using Hiding-interleave[of ev ‘ A insert tick (ev ‘ S) c t u] by blast+
    next
      case (112 t)
      from 112(7) have a:front-tickFree t
        by (metis (mono-tags, opaque-lifting) D-imp-front-tickFree append-Nil2
              front-tickFree-append ftf-Sync is-processT2-TR)
      from 112 have t ∈ 𝒟 (P ⟦S⟧ Q) by (simp add:D-Sync)
      with 112(1,2,3) have a ∈ 𝒟 ((P ⟦S⟧ Q) \ A)
        apply (simp add:D-Hiding)
        proof(case-tac tickFree t, goal-cases 1121 1122)
          case 1121
          then show ?case
            by (rule-tac x=t in exI, rule-tac x=[] in exI, simp)
        next
          case 1122
          with a obtain t′ where t = t′@[tick] using nonTickFree-n-frontTickFree
by blast
          with a show ?case
            apply (rule-tac x=t′ in exI, rule-tac x=[tick] in exI, auto)
             using front-tickFree-mono apply blast
            using 1122(4) is-processT9 by blast
        qed
      then show ?case
        apply(rule-tac disjI2)
```

```
      using Hiding-Sync-D1[of A S P Q, OF 112(1,2)] D-Sync[of P \ A S Q \
A] by auto
    qed
  next
    case 12
    hence a ∈ 𝒟 ((P ⟦S⟧ Q) \ A) by (simp add:D-Hiding)
    then show ?case
      apply(rule-tac disjI2)
      using Hiding-Sync-D1[of A S P Q, OF 12(1,2)] D-Sync[of P \ A S Q \ A]
by auto
  qed
next
  case (2 a b)
  then show ?case
  proof(elim disjE, goal-cases 21 22)
    case 21
    then show ?case
    proof(elim exE conjE, elim disjE, goal-cases 211 212 213 214)
      case (211 t u X Y)
      then obtain ta ua where t211: t = trace-hide ta (ev ‘ A) ∧ (ta, X ∪ ev ‘
A) ∈ ℱ P and
                                u211: u = trace-hide ua (ev ‘ A) ∧ (ua, Y ∪ ev ‘ A) ∈ ℱ
Q by blast
      from interleave-Hiding[of (ev ‘ A) insert tick (ev ‘ S) a ta ua] 211(1,2,3)
t211 u211
      obtain aa where a211: a = trace-hide aa (ev ‘ A) ∧
                                aa setinterleaves ((ta, ua), insert tick (ev ‘ S)) by blast
      with 211(1,2,3,4) t211 u211 show ?case
        apply(rule-tac disjI1, rule-tac x=aa in exI, simp)
        apply(rule-tac disjI1, rule-tac x=ta in exI, rule-tac x=ua in exI)
        apply(rule-tac x=X ∪ ev ‘ A in exI, rule conjI) using is-processT4 apply
blast
        apply(rule-tac x=Y ∪ ev ‘ A in exI, rule conjI) using is-processT4 by
blast+
    next
      case (212 t u X Y)
      hence u ∈ 𝒟 (Q \ A) and t ∈ 𝒯 (P \ A)
        apply (simp add:D-Hiding)
        using 212(8) F-T elemTIselemHT by blast
      with 212(3,8) have a ∈ 𝒟 (((P \ A) ⟦S⟧ (Q \ A)))
        apply (simp add:D-Sync)
        using Sync.sym front-tickFree-charn by blast
      then show ?case
        apply (rule-tac disjI2)
          using Hiding-Sync-D2[of A S P Q, OF 2(2)] D-Hiding[of (P ⟦S⟧ Q),
simplified] by auto
    next
      case (213 t u X Y)
      hence u ∈ 𝒯 (Q \ A) and t ∈ 𝒟 (P \ A)
```

```
      by (simp-all add:D-Hiding, metis F-T elemTIselemHT)
    with 213(3,8) have a ∈ 𝒟 (((P \ A) ⟦S⟧ (Q \ A)))
      apply (simp add:D-Sync)
      using Sync.sym front-tickFree-charn by blast
    then show ?case
      apply (rule-tac disjI2)
        using Hiding-Sync-D2[of A S P Q, OF 2(2)] D-Hiding[of (P ⟦S⟧ Q),
simplified] by auto
  next
    case (214 t u X Y)
    hence u ∈ 𝒯 (Q \ A) and t ∈ 𝒟 (P \ A)
      by (simp-all add:D-Hiding T-Hiding)
    with 214(3,8) have a ∈ 𝒟 (((P \ A) ⟦S⟧ (Q \ A)))
      apply (simp add:D-Sync)
      using Sync.sym front-tickFree-charn by blast
    then show ?case
      apply (rule-tac disjI2)
        using Hiding-Sync-D2[of A S P Q, OF 2(2)] D-Hiding[of (P ⟦S⟧ Q),
simplified] by auto
  qed
next
  case 22
  hence ‹a ∈ 𝒟 (((P \ A) ⟦S⟧ (Q \ A)))› by (simp add:D-Sync)
  then show ?case
    apply (rule-tac disjI2)
    using Hiding-Sync-D2[of A S P Q, OF 2(2)] D-Hiding[of (P ⟦S⟧ Q), simpli-
fied] by auto
qed
qed

lemma Sync-assoc-oneside-D: 𝒟 (P ⟦S⟧ (Q ⟦S⟧ T)) ⊆ 𝒟 (P ⟦S⟧ Q ⟦S⟧ T)
proof(auto simp add:D-Sync T-Sync del:disjE, goal-cases a)
  case (a tt uu rr vv)
  from a(3,4) show ?case
  proof(auto, goal-cases)
    case (1 t u)
    with interleave-assoc-2[OF 1(5) 1(1)] obtain tu
      where *:tu setinterleaves ((tt, t), insert tick (ev ‘ S))
        and **:rr setinterleaves ((tu, u), insert tick (ev ‘ S))
      by blast
   with a(1,2) 1 show ?case by (metis append.right-neutral front-tickFree-charn)

  next
    case (2 t u u1 u2)
    with interleave-append-sym [OF 2(1)] obtain t1 t2 r1 r2
      where a1:tt = t1 @ t2 and a2:rr = r1 @ r2
        and a3:r1 setinterleaves ((t1, u1), insert tick (ev ‘ S))
        and a4:r2 setinterleaves ((t2, u2), insert tick (ev ‘ S)) by blast
    with interleave-assoc-2[OF 2(5) a3] obtain tu
```

```
      where *:tu setinterleaves ((t1, t), insert tick (ev ‘ S))
       and **:r1 setinterleaves ((tu, u), insert tick (ev ‘ S))
      by blast
    with a1 a2 a3 a(1,2) 2 show ?case
      apply(exI tu, exI u, exI r1, exI r2@vv, intro conjI, simp-all)
      apply (metis D-imp-front-tickFree append-Nil2 front-tickFree-append front-tickFree-mono
ftf-Sync)
      using D-imp-front-tickFree front-tickFree-append front-tickFree-mono ftf-Sync
apply blast
      by (metis NT-ND Sync.sym append-Nil2 front-tickFree-Nil is-processT3-ST)
  next
    case (3 v u u1 u2)
    with interleave-append-sym [OF 3(1)] obtain t1 t2 r1 r2
      where a1:tt = t1 @ t2 and a2:rr = r1 @ r2
       and  a3:r1 setinterleaves ((t1, u1), insert tick (ev ‘ S))
       and  a4:r2 setinterleaves ((t2, u2), insert tick (ev ‘ S)) by blast
    with interleave-assoc-2[OF - a3, of u v] obtain tu
      where *:tu setinterleaves ((t1, u), insert tick (ev ‘ S))
       and **:r1 setinterleaves ((tu, v), insert tick (ev ‘ S))
      using Sync.sym 3(5) by blast
    with a1 a2 a3 a(1,2) 3 show ?case
      apply(exI v, exI tu, exI r1, exI r2@vv, intro conjI, simp-all)
      apply (metis D-imp-front-tickFree append-Nil2 front-tickFree-append front-tickFree-mono
ftf-Sync)
      using D-imp-front-tickFree front-tickFree-append front-tickFree-mono ftf-Sync
apply blast
      using Sync.sym apply blast by (meson D-T is-processT3-ST)
  next
    case (4 t u)
    with interleave-assoc-2[OF 4(3) 4(1)] obtain tu
      where *:tu setinterleaves ((tt, t), insert tick (ev ‘ S))
       and **:rr setinterleaves ((tu, u), insert tick (ev ‘ S))
      by blast
    with a(1,2) 4 show ?case
      apply(exI tu, exI u, exI rr, exI vv, simp) using NT-ND front-tickFree-Nil by
blast
  next
    case (5 t u)
    with interleave-assoc-2[OF - 5(1), of u t] obtain tu
      where *:tu setinterleaves ((tt, u), insert tick (ev ‘ S))
       and **:rr setinterleaves ((tu, t), insert tick (ev ‘ S))
      using Sync.sym by blast
    with a(1,2) 5 show ?case
      apply(exI tu, exI t, exI rr, exI vv, simp) using NT-ND front-tickFree-Nil by
blast
  next
    case (6 t u u1 u2)
    with interleave-append [OF 6(1)] obtain t1 t2 r1 r2
      where a1:uu = t1 @ t2 and a2:rr = r1 @ r2
```

```
      and a3:r1 setinterleaves ((t1, u1), insert tick (ev ‘ S))
      and a4:r2 setinterleaves ((t2, u2), insert tick (ev ‘ S)) using Sync.sym by
blast
   with interleave-assoc-2[OF 6(5) a3] obtain tu
     where *:tu setinterleaves ((t1, t), insert tick (ev ‘ S))
      and **:r1 setinterleaves ((tu, u), insert tick (ev ‘ S))
     by blast
   with a1 a2 a3 a(1,2) 6 show ?case
     apply(exI tu, exI u, exI r1, exI r2@vv, intro conjI, simp-all)
      apply (metis append-Nil2 front-tickFree-append front-tickFree-mono ftf-Sync
is-processT2-TR)
    apply (meson front-tickFree-append front-tickFree-mono ftf-Sync is-processT2-TR)

     by (metis Sync.sym append-Nil2 front-tickFree-Nil is-processT3-ST)
  next
   case (7 v u t1 t2)
   with interleave-append [OF 7(1)] obtain u1 u2 r1 r2
     where a1:uu = u1 @ u2 and a2:rr = r1 @ r2
      and  a3:r1 setinterleaves ((t1, u1), insert tick (ev ‘ S))
      and  a4:r2 setinterleaves ((t2, u2), insert tick (ev ‘ S)) by blast
   with interleave-assoc-2[of t1 u - v r1 u1] obtain tu
     where *:tu setinterleaves ((u1, u), insert tick (ev ‘ S))
      and **:r1 setinterleaves ((tu, v), insert tick (ev ‘ S))
     using 7(5) Sync.sym by blast
   with a1 a2 a3 a(1,2) 7 show ?case
     apply(exI v, exI tu, exI r1, exI r2@vv, intro conjI, simp-all)
      apply (metis append-Nil2 front-tickFree-append front-tickFree-mono ftf-Sync
is-processT2-TR)
    apply (meson front-tickFree-append front-tickFree-mono ftf-Sync is-processT2-TR)

     using Sync.sym apply blast by (meson D-T is-processT3-ST)
  next
   case (8 t u)
   with interleave-assoc-2[OF 8(3), of rr uu] obtain tu
     where *:tu setinterleaves ((uu, t), insert tick (ev ‘ S))
      and **:rr setinterleaves ((tu, u), insert tick (ev ‘ S))
     using Sync.sym by blast
   with a(1,2) 8 show ?case
     apply(exI tu, exI u, exI rr, exI vv, simp) using Sync.sym front-tickFree-Nil
by blast
  next
   case (9 t u)
   with interleave-assoc-1[OF 9(3), of rr uu] obtain uv
     where *:uv setinterleaves ((u, uu), insert tick (ev ‘ S))
      and **:rr setinterleaves ((t, uv), insert tick (ev ‘ S))
     by blast
   with a(1,2) 9 show ?case
     apply(exI t, exI uv, exI rr, exI vv, simp) using Sync.sym front-tickFree-Nil
by blast
```

```
  qed
qed

lemma Sync-assoc-oneside: ((P ⟦S⟧ Q) ⟦S⟧ T) ≤ (P ⟦S⟧ (Q ⟦S⟧ T))
proof(unfold le-ref-def, rule conjI, goal-cases D F)
  case D
  then show ?case using Sync-assoc-oneside-D by assumption
next
  case F
  then show ?case
  unfolding F-Sync proof(rule Un-least, goal-cases FF DD)
    case (FF)
    then show ?case
     proof(rule subsetI, simp add:D-Sync T-Sync split-paired-all, elim exE conjE
disjE, goal-cases)
      case (1 r Xtuv t uv Xt Xuv u v Xu Xv)
      with interleave-assoc-2[OF 1(6), OF 1(2)] obtain tu
        where *:tu setinterleaves ((t, u), insert tick (ev ‘ S))
        and **:r setinterleaves ((tu, v), insert tick (ev ‘ S)) by blast
      with 1(1,3,4,5,7) show ?case
        apply(rule-tac disjI1, exI tu, exI v, exI (Xt ∪ Xu) ∩ insert tick (ev ‘ S) ∪
Xt ∩ Xu,
              intro conjI disjI1, simp-all)
        by blast+
    next
      case (2 r Xtuv t uv Xt Xuv u v uv1 uv2)
      with interleave-append-sym [of r t - uv1 uv2] obtain t1 t2 r1 r2
        where a1:t = t1 @ t2 and a2:r = r1 @ r2
          and a3:r1 setinterleaves ((t1, uv1), insert tick (ev ‘ S))
          and a4:r2 setinterleaves ((t2, uv2), insert tick (ev ‘ S)) by metis
      with interleave-assoc-2[OF 2(6), OF a3] obtain tu
        where *:tu setinterleaves ((t1, u), insert tick (ev ‘ S))
         and **:r1 setinterleaves ((tu, v), insert tick (ev ‘ S)) by blast
     from 2(1) a1 obtain Xt1 where ***:(t1, Xt1) ∈ ℱ P using is-processT3-SR
by blast
      from 2 a1 a2 a3 a4 * ** *** show ?case
          apply(rule-tac disjI2, exI tu, exI v, exI r1, exI r2, intro conjI disjI1,
simp-all)
        apply (metis front-tickFree-charn front-tickFree-mono ftf-Sync is-processT2)

         apply (metis equals0D ftf-Sync1 ftf-Sync21 insertI1 tickFree-def)
        using F-T Sync.sym front-tickFree-Nil by blast
    next
      case (3 r Xtuv t uv Xt Xuv v u uv1 uv2)
      with interleave-append-sym [of r t - uv1 uv2] obtain t1 t2 r1 r2
        where a1:t = t1 @ t2 and a2:r = r1 @ r2
         and  a3:r1 setinterleaves ((t1, uv1), insert tick (ev ‘ S))
         and  a4:r2 setinterleaves ((t2, uv2), insert tick (ev ‘ S)) by metis
      with interleave-assoc-2[OF - a3, of u v] obtain tu
```

```
      where *:tu setinterleaves ((t1, u), insert tick (ev ‘ S))
       and **:r1 setinterleaves ((tu, v), insert tick (ev ‘ S)) using 3(6) Sync.sym
by blast
    from 3(1) a1 obtain Xt1
      where ***:(t1, Xt1) ∈ ℱ P using is-processT3-SR by blast
    from 3 a1 a2 a3 a4 * ** *** show ?case
      apply(rule-tac disjI2, exI v, exI tu, exI r1, exI r2, intro conjI, simp-all)
      apply (metis front-tickFree-charn front-tickFree-mono ftf-Sync is-processT2)

       apply (metis equals0D ftf-Sync1 ftf-Sync21 insertI1 tickFree-def)
      using Sync.sym F-T by blast+
  next
    case (4 r Xtuv t uv Xt Xuv u v uv1 uv2)
    with interleave-assoc-2[OF 4(6), of r t] obtain tu
      where *:tu setinterleaves ((t, u), insert tick (ev ‘ S))
       and **:r setinterleaves ((tu, v), insert tick (ev ‘ S)) by auto
    from 4 * **  show ?case
      apply(rule-tac disjI2, exI tu, exI v, exI r, exI [], intro conjI, simp-all)
      by (metis 4(4) F-T Sync.sym append.right-neutral)
  next
    case (5 r Xtuv t uv Xt Xuv v u uv1 uv2)
    from 5(2,7,5,6)  interleave-assoc-2[of uv u - v r t] obtain tu
      where *:tu setinterleaves ((t, u), insert tick (ev ‘ S))
       and **:r setinterleaves ((tu, v), insert tick (ev ‘ S)) by (metis Sync.sym
append-Nil2)
    from 5 * **  show ?case
      apply(rule-tac disjI2, exI v, exI tu, exI r, exI [], intro conjI, simp-all)
      using Sync.sym F-T by blast+
  qed
 next
  case (DD)
  then show ?case
   using Sync-assoc-oneside-D[of P S Q T] by (simp add:D-Sync Collect-mono-iff
sup.coboundedI2)
 qed
qed

lemma Sync-assoc: ((P ⟦S⟧ Q) ⟦S⟧ T) = (P ⟦S⟧ (Q ⟦S⟧ T))
 by (metis antisym-conv Sync-assoc-oneside Sync-commute)
```

5.7.4 Derivative Operators laws

```
lemmas Par-BOT = Sync-BOT[where S = ‹UNIV›]
  and Par-SKIP-SKIP = Sync-SKIP-SKIP[where S = ‹UNIV›]
  and Par-SKIP-STOP = Sync-SKIP-STOP[where S = ‹UNIV›]
  and Par-commute = Sync-commute[where S = ‹UNIV›]
  and Par-assoc = Sync-assoc[where S = ‹UNIV›]
  and Par-Ndet-right-distrib = Sync-Ndet-right-distrib[where S = ‹UNIV›]
  and Par-Ndet-left-distrib = Sync-Ndet-left-distrib[where S = ‹UNIV›]
```

and *Mprefix-Par-SKIP* = *Mprefix-Sync-SKIP*[**where** *S* = ‹*UNIV*›]
and *prefix-Par-SKIP* = *prefix-Sync-SKIP*[**where** *S* = ‹*UNIV*›, *simplified*]
and *mono-Par-FD* = *mono-Sync-FD*[**where** *S* = ‹*UNIV*›]
and *mono-Par-ref* = *mono-Sync-ref*[**where** *S* = ‹*UNIV*›]
and *prefix-Par1* = *prefix-Sync1*[**where** *S* = ‹*UNIV*›, *simplified*]
and *prefix-Par2* = *prefix-Sync2*[**where** *S* = ‹*UNIV*›, *simplified*]
and *Mprefix-Par-distr* = *Mprefix-Sync-distr-subset*[**where** *S* = ‹*UNIV*›, *simplified*]

and *Inter-BOT* = *Sync-BOT*[**where** *S* = ‹{}›]
and *Inter-SKIP-SKIP* = *Sync-SKIP-SKIP*[**where** *S* = ‹{}›]
and *Inter-SKIP-STOP* = *Sync-SKIP-STOP*[**where** *S* = ‹{}›]
and *Inter-commute* = *Sync-commute*[**where** *S* = ‹{}›]
and *Inter-assoc* = *Sync-assoc*[**where** *S* = ‹{}›]
and *Inter-Ndet-right-distrib* = *Sync-Ndet-right-distrib*[**where** *S* = ‹{}›]
and *Inter-Ndet-left-distrib* = *Sync-Ndet-left-distrib*[**where** *S* = ‹{}›]
and *Mprefix-Inter-SKIP* = *Mprefix-Sync-SKIP*[**where** *S* = ‹{}›, *simplified*, *simplified Mprefix-STOP*]
and *prefix-Inter-SKIP* = *prefix-Sync-SKIP*[**where** *S* = ‹{}›, *simplified*]
and *mono-Inter-FD* = *mono-Sync-FD*[**where** *S* = ‹{}›]
and *mono-Inter-ref* = *mono-Sync-ref*[**where** *S* = ‹{}›]
and *Hiding-Inter* = *Hiding-Sync*[**where** *S* = ‹{}›, *simplified*]
and *Mprefix-Inter-distr* = *Mprefix-Sync-distr-indep*[**where** *S* = ‹{}›, *simplified*]
and *prefix-Inter* = *Mprefix-Sync-distr-indep*[*of* ‹{*a*}› ‹{}› ‹{*b*}› ‹λ*x*. *P*› ‹λ*x*. *Q*›,
simplified, *folded write0-def*] **for** *a P b Q*

lemma *Inter-SKIP*: (*P* ||| *SKIP*) = *P*
proof(*auto simp*:*Process-eq-spec D-Sync F-Sync F-SKIP D-SKIP T-SKIP*, *goal-cases*)
case (*1 a t X Y*)
then have *aa*:(*X* ∪ *Y*) ∩ {*tick*} ∪ *X* ∩ *Y* ⊆ *X* **by** (*simp add*: *Int-Un-distrib2*)
have *front-tickFree t* **using** *1*(*1*) *is-processT2* **by** *blast*
with *1 EmptyLeftSync*[*of a* {*tick*} *t*] **have** *bb*:*a*=*t*
using *Sync.sym* **by** *blast*
from *1*(*1*) *aa bb* **show** *?case* **using** *is-processT4* **by** *blast*
next
case (*2 a t X Y*)
have *front-tickFree t* **using** *2*(*1*) *is-processT2* **by** *blast*
with *2 TickLeftSync*[*of* {*tick*} *t a*] **have** *bb*:*a*=*t* ∧ *last t*=*tick*
using *Sync.sym* **by** *blast*
with *2* **have** *a* ∈ 𝒯 *P* ∧ *last a* = *tick* **using** *F-T* **by** *blast*
with *tick-T-F*[*of butlast a P* (*X* ∪ *Y*) ∩ {*tick*} ∪ *X* ∩ *Y*] **show** *?case*
by (*metis 2*(*2*) *EmptyLeftSync bb self-append-conv2 snoc-eq-iff-butlast*)
next
case (*3 b t r v*)
with *3 EmptyLeftSync*[*of r* {*tick*} *t*] **have** *bb*:*r*=*t*
using *Sync.sym* **by** *blast*

```
  with 3 show ?case using is-processT by blast
next
  case (4 b t r v)
  with TickLeftSync[of {tick} t r, simplified] have bb:r=t
    using D-imp-front-tickFree Sync.sym by blast
  with 4 show ?case using is-processT by blast
next
  case (5 b t r)
  with EmptyLeftSync[of r {tick} t] have bb:r=t
    using Sync.sym by blast
  with 5 show ?case using is-processT by blast
next
  case (6 b t r)
  with TickLeftSync[of {tick} t r, simplified] have bb:r=t
    using D-imp-front-tickFree Sync.sym by blast
  with 6 show ?case using is-processT by blast
next
  case (7 a b)
  then show ?case
    apply(cases tickFree a)
     apply(rule-tac x=a in exI, rule-tac x=[] in exI, rule-tac x=b in exI, simp)
     apply(rule-tac x=b − {tick} in exI, simp, intro conjI)
      using emptyLeftSelf[of a {tick}] Sync.sym tickFree-def apply fastforce
     apply blast
    apply(rule-tac x=a in exI, rule-tac x=[tick] in exI, rule-tac x=b in exI, simp,
rule conjI)
    subgoal proof −
      assume a1: ¬ tickFree a
      assume a2: (a, b) ∈ ℱ P
      then obtain ees :: 'a event list ⇒ 'a event list where
        f3: a = ees a @ [tick]
        using a1 by (meson is-processT2 nonTickFree-n-frontTickFree)
      then have ees a setinterleaves (([], ees a), {tick})
        using a2 by (metis (no-types) DiffD2 Diff-insert-absorb emptyLeftSelf
                              front-tickFree-implies-tickFree is-processT2 tickFree-def)
      then show a setinterleaves ((a, [tick]), {tick})
        using f3 by (metis (no-types) Sync.sym addSyncEmpty insertI1)
    qed
    apply(rule-tac x=b − {tick} in exI) by blast
next
  case (8 t r v)
  with EmptyLeftSync[of r {tick} t] have bb:r=t
    using Sync.sym by blast
  with 8 show ?case using is-processT by blast
next
  case (9 t r v)
  with TickLeftSync[of {tick} t r, simplified] have bb:r=t
    using D-imp-front-tickFree Sync.sym by blast
  with 9 show ?case using is-processT by blast
```

```
next
  case (10 t r)
  with EmptyLeftSync[of r {tick} t] have bb:r=t
    using Sync.sym by blast
  with 10 show ?case by simp
next
  case (11 t r)
  with TickLeftSync[of {tick} t r, simplified] have bb:r=t
    using D-imp-front-tickFree Sync.sym by blast
  with 11 show ?case by simp
next
  case (12 x)
  then show ?case
  proof(cases tickFree x)
    case True
    with 12 show ?thesis
      apply(rule-tac x=x in exI, rule-tac x=[] in exI,
            rule-tac x=x in exI, rule-tac x=[] in exI, simp)
      by (metis Sync.sym emptyLeftSelf singletonD tickFree-def)
  next
    case False
    with 12 show ?thesis
      apply(rule-tac x=butlast x in exI, rule-tac x=[] in exI,
            rule-tac x=butlast x in exI, rule-tac x=[tick] in exI, simp, intro conjI)
       apply (metis D-imp-front-tickFree append-butlast-last-id front-tickFree-mono

                    list.distinct(1) tickFree-Nil)
      using NT-ND T-nonTickFree-imp-decomp apply fastforce
       apply (metis NT-ND Sync.sym append-T-imp-tickFree append-butlast-last-id
emptyLeftSelf
                    list.distinct(1) singletonD tickFree-Nil tickFree-def)
     using D-imp-front-tickFree is-processT9 nonTickFree-n-frontTickFree by fast-
force
  qed
qed

lemma Inter-STOP-Seq-STOP: (P ||| STOP) = (P ; STOP)
proof (auto simp add:Process-eq-spec D-Sync F-Sync F-STOP D-STOP T-STOP
F-Seq D-Seq, goal-cases)
  case (1 a t X Y)
  have insert tick ((X ∪ Y) ∩ {tick} ∪ X ∩ Y) ⊆ X ∪ {tick} by blast
  with 1 show ?case
    by (metis (mono-tags, lifting) EmptyLeftSync Sync.sym is-processT
                                   is-processT5-S2 no-Trace-implies-no-Failure)
next
  case (2 a t X)
  then show ?case using EmptyLeftSync Sync.sym tickFree-def by fastforce
next
  case (3 b t r v)
```

```
  then show ?case using EmptyLeftSync Sync.sym tickFree-def by blast
next
  case (4 t r v)
  then show ?case using EmptyLeftSync Sync.sym by blast
next
  case (5 b t r)
  then show ?case by (metis EmptyLeftSync Sync.sym is-processT)
next
  case (6 t r)
  then show ?case using EmptyLeftSync Sync.sym tickFree-def by fastforce
next
  case (7 a b)
  then show ?case
    apply(rule-tac x=a in exI, rule-tac x=b in exI, intro conjI)
      apply (meson insert-iff is-processT subsetI)
     using emptyLeftSelf[of a {tick}] Sync.sym tickFree-def apply fastforce
    by blast
next
  case (8 a b)
  then show ?case
    apply(rule-tac x=a in exI, rule-tac x=b−{tick} in exI, intro conjI)
      apply (meson NF-NT is-processT6)
     using emptyLeftSelf[of a {tick}] Sync.sym tickFree-def
     apply (metis append-T-imp-tickFree list.distinct(1) singletonD)
    by blast
next
  case (9 b t1 t2)
  hence t1 setinterleaves ((t1, []), {tick})
    using emptyLeftSelf[of t1 {tick}] Sync.sym tickFree-def by fastforce
  with 9(1)[THEN spec, THEN spec, of t1 t1, simplified] have False
    using 9(2) 9(3) 9(4) by blast
  then show ?case ..
next
  case (10 t r v)
  then show ?case
    apply(rule-tac x=r in exI, rule-tac x=v in exI, auto)
    using EmptyLeftSync Sync.sym by blast
next
  case (11 t r)
  then show ?case
    apply(rule-tac x=r in exI, rule-tac x=[] in exI, simp)
    using EmptyLeftSync Sync.sym tickFree-def by fastforce
next
  case (12 t1 t2)
  then show ?case
    apply(rule-tac x=t1 in exI, rule-tac x=t1 in exI, rule-tac x=t2 in exI, simp)
    by (metis Sync.sym emptyLeftSelf singletonD tickFree-def)
qed
```

```
lemma Par-STOP: P ≠ ⊥ ⟹ (P || STOP) = STOP
proof(auto simp add:Process-eq-spec,
    simp-all add:D-Sync F-Sync F-STOP D-STOP T-STOP is-processT2 D-imp-front-tickFree
F-UU D-UU,
      goal-cases)
  case (1 a b aa ba)
  then show ?case
  proof(elim exE conjE disjE, goal-cases)
    case (1 t X Y)
    then show ?case
      proof(cases t)
        case Nil then show ?thesis using 1(4) EmptyLeftSync by blast
      next
        case (Cons a list)
        with 1 show ?thesis by (simp split:if-splits, cases a, simp-all)
      qed
  next
    case (2 t r v)
    hence [] ∈ 𝒟 P
    proof(cases t)
      case Nil with 2 show ?thesis by simp
    next
      case (Cons a list)
      with 2 show ?thesis by (simp split:if-splits, cases a, simp-all)
    qed
    with 2(1,2) show ?case using NF-ND is-processT7 by fastforce
  next
    case (3 t r v)
    hence [] ∈ 𝒟 P
    proof(cases t)
      case Nil with 3 show ?thesis by simp
    next
      case (Cons a list)
      with 3 show ?thesis by (simp split:if-splits, cases a, simp-all)
    qed
    with 3(1,2) show ?case using NF-ND is-processT7 by fastforce
  qed
next
  case (2 a b aa ba)
  then show ?case
   apply(rule-tac disjI1, rule-tac x=[] in exI, rule-tac x={} in exI, simp, rule-tac
conjI)
    using is-processT apply blast
    apply(rule-tac x=ba in exI, rule-tac set-eqI, rule-tac iffI)
    using event-set apply blast
    by fast
next
  case (3 a b x)
```

```
  then show ?case
  proof(elim exE conjE disjE, goal-cases)
    case (1 t r v)
    hence [] ∈ 𝒟 P
    proof(cases t)
      case Nil with 1 show ?thesis by simp
    next
      case (Cons a list)
      with 1 show ?thesis by (simp split:if-splits, cases a, simp-all)
    qed
    with 3(1,2) show ?case using NF-ND is-processT7 by fastforce
  next
    case (2 t r v)
    hence [] ∈ 𝒟 P
    proof(cases t)
      case Nil with 2 show ?thesis by simp
    next
      case (Cons a list)
      with 2 show ?thesis by (simp split:if-splits, cases a, simp-all)
    qed
    with 3(1,2) show ?case using NF-ND is-processT7 by fastforce
  qed
next
  case (4 x a b)
  then show ?case
  proof(elim exE conjE disjE, goal-cases)
    case (1 t X Y)
    then show ?case
      proof(cases t)
        case Nil then show ?thesis using 1(4) EmptyLeftSync by blast
      next
        case (Cons a list)
        with 1 show ?thesis by (simp split:if-splits, cases a, simp-all)
      qed
  next
    case (2 t r v)
    hence [] ∈ 𝒟 P
    proof(cases t)
      case Nil with 2 show ?thesis by simp
    next
      case (Cons a list)
      with 2 show ?thesis by (simp split:if-splits, cases a, simp-all)
    qed
    with 2(1,2) show ?case using NF-ND is-processT7 by fastforce
  next
    case (3 t r v)
    hence [] ∈ 𝒟 P
    proof(cases t)
      case Nil with 3 show ?thesis by simp
```

```
    next
      case (Cons a list)
      with 3 show ?thesis by (simp split:if-splits, cases a, simp-all)
    qed
  with 3(1,2) show ?case using NF-ND is-processT7 by fastforce
  qed
next
  case (5 x a b)
  then show ?case
  apply(rule-tac disjI1, rule-tac x=[] in exI, rule-tac x={} in exI, simp, rule-tac
conjI)
  using is-processT apply blast
  apply(rule-tac x=b in exI, rule-tac set-eqI, rule-tac iffI)
  using event-set apply blast
  by fast
next
  case (6 x xa)
  then show ?case
  proof(elim exE conjE disjE, goal-cases)
    case (1 t r v)
    hence [] ∈ 𝒟 P
    proof(cases t)
      case Nil with 1 show ?thesis by simp
    next
      case (Cons a list)
      with 1 show ?thesis by (simp split:if-splits, cases a, simp-all)
    qed
    with 6(1,2) show ?case using NF-ND is-processT7 by fastforce
  next
    case (2 t r v)
    hence [] ∈ 𝒟 P
    proof(cases t)
      case Nil with 2 show ?thesis by simp
    next
      case (Cons a list)
      with 2 show ?thesis by (simp split:if-splits, cases a, simp-all)
    qed
    with 6(1,2) show ?case using NF-ND is-processT7 by fastforce
  qed
qed
```

5.8 Multiple Non Deterministic Operator Laws

```
lemma Mprefix-refines-Mndetprefix: (⊓ x ∈ A → P x) ≤ (□ x ∈ A → P x)
proof(cases A = {})
  case True
  then show ?thesis by (simp add: Mprefix-STOP)
next
  case False
```

then show *?thesis*
by (*auto simp add*:*le-ref-def D-Mprefix write0-def F-Mprefix F-Mndetprefix D-Mndetprefix*)
qed

lemma *Mndetprefix-refine-FD*: $a \in A \Longrightarrow (\sqcap\ x \in A \rightarrow (P\ x)) \leq (a \rightarrow (P\ a))$
by(*subgoal-tac* $A \neq \{\}$, *auto simp*:*le-ref-def D-Mndetprefix F-Mndetprefix*)

lemma *Mndetprefix-subset-FD*:$A \neq \{\} \Longrightarrow (\sqcap xa \in A \cup B \rightarrow\ P) \leq (\sqcap xa \in A \rightarrow P)$
by (*simp add*: *D-Mndetprefix F-Mndetprefix le-ref-def*)

lemma *mono-Mndetprefix-FD*[*simp*]: $\forall x \in A.\ P\ x \leq P'\ x \Longrightarrow (\sqcap\ x \in A \rightarrow (P\ x)) \leq (\sqcap\ x \in A \rightarrow (P'\ x))$
apply(*cases* $A \neq \{\}$, *auto simp*:*le-ref-def D-Mndetprefix F-Mndetprefix*)
by(*meson contra-subsetD le-ref-def mono-write0-FD*)+

lemma *Mndetprefix-FD-subset* : $A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow ((\sqcap\ x \in B \rightarrow P\ x)\ \leq (\sqcap\ x \in A \rightarrow P\ x))$
by (*metis bot.extremum-uniqueI Mndetprefix-Un-distrib mono-Ndet-FD-right order-refl sup.absorb-iff1*)

5.9 Infra-structure for Communication Primitives

lemma *read-read-Sync*:
assumes *contained*: $(\bigwedge y.\ c\ y \in S)$
shows $((c?x \rightarrow P\ x)\ [\![S]\!]\ (c?x \rightarrow Q\ x)) = (c?x \rightarrow ((P\ x)\ [\![S]\!]\ (Q\ x)))$
proof −
have *A*: *range* $c \subseteq S$ **by** (*insert contained*, *auto*)
show *?thesis* **by** (*auto simp*: *read-def o-def Set.Int-absorb2 Mprefix-Sync-distr-subset A*)
qed

lemma *read-read-non-Sync*:
$[\![\bigwedge y.\ c\ y \notin S;\ \bigwedge y.\ d\ y \in S]\!] \Longrightarrow (c?x \rightarrow (P\ x))\ [\![S]\!]\ (d?x \rightarrow (Q\ x)) = c?x \rightarrow ((P\ x)\ [\![S]\!]\ (d?x \rightarrow (Q\ x)))$
apply (*auto simp add*: *read-def o-def*)
using *Mprefix-Sync-distr-right*[*of range c S range d* $\lambda x.\ Q$ (*inv d x*) $\lambda x.\ P$ (*inv c x*)]
apply (*subst* (*1 2*) *Sync-commute*)
by *auto*

lemma *write-read-Sync*:
assumes *contained*: $\bigwedge y.\ c\ y \in S$
and *is-construct*: *inj c*
shows $(c!a \rightarrow P)\ [\![S]\!]\ (c?x \rightarrow Q\ x) = (c!a \rightarrow (P\ [\![S]\!]\ Q\ a))$
proof −
have *A* : *range* $c \subseteq S$ **by** (*insert contained*, *auto*)
have *B* : $\{c\ a\} \cap$ *range* $c \cap S = \{c\ a\}$ **by** (*insert contained*, *auto*)

show *?thesis*
apply (*simp add*: *read-def write-def o-def*, *subst Mprefix-Sync-distr-subset*)
by (*simp-all add*: *A B contained Mprefix-singl is-construct*)
qed

lemma *write-read-non-Sync*:
$[\![d\ a \notin S;\ \bigwedge y.\ c\ y \in S\ ]\!] \Longrightarrow (d!a \to P)\ [\![S]\!]\ (c?x \to Q\ x) = d!a \to (P\ [\![S]\!]\ (c?x \to Q\ x))$
apply (*simp add*: *read-def o-def write-def*)
using *Mprefix-Sync-distr-right*[*of* $\{d\ a\}$ *S range c* $\lambda x.\ (Q\ (inv\ c\ x))\ \lambda x.\ P$]
apply (*subst* (*1 2*) *Sync-commute*)
by *auto*

lemma *write0-read-non-Sync*:
$[\![d \in S;\ \bigwedge y.\ c\ y \notin S]\!] \Longrightarrow (d \to P)\ [\![S]\!]\ (c?x \to Q\ x) = c?x \to ((d \to P)\ [\![S]\!]\ Q\ x)$
apply (*simp add*: *read-def o-def*)
apply (*subst* (*1 2*) *Mprefix-singl*[*symmetric*])
apply (*subst Mprefix-Sync-distr-right*)
by *auto*

lemma *write0-write-non-Sync*:
$[\![d\ a \notin C;\ c \in C]\!] \Longrightarrow (c \to Q)\ [\![C]\!]\ (d!a \to P) = d!a \to ((c \to Q)\ [\![C]\!]\ P)$
apply(*simp add*: *write-def*)
apply(*subst* (*1 2*) *Mprefix-singl*[*symmetric*])
by (*auto simp*: *Mprefix-Sync-distr-right*)

lemmas *Sync-rules* = *prefix-Sync2*
read-read-Sync read-read-non-Sync
write-read-Sync write-read-non-Sync
write0-read-non-Sync
write0-write-non-Sync

lemmas *Hiding-rules* = *no-Hiding-read no-Hiding-write Hiding-write Hiding-write0*

lemmas *mono-rules* = *mono-read-ref mono-write-ref mono-write0-ref*

end

Chapter 6

Refinements

theory *Process-Order*
imports *Process Stop*
begin

definition *trace-refine* :: ‹$'a$ *process* $\Rightarrow$ $'a$ *process* $\Rightarrow$ *bool*› (**infix** ‹$\sqsubseteq_T$› *60*)
where ‹$P \sqsubseteq_T Q \equiv \mathcal{T}\ Q \subseteq \mathcal{T}\ P$›

definition *failure-refine* :: ‹$'a$ *process* $\Rightarrow$ $'a$ *process* $\Rightarrow$ *bool*› (**infix** ‹$\sqsubseteq_F$› *60*)
where ‹$P \sqsubseteq_F Q \equiv \mathcal{F}\ Q \subseteq \mathcal{F}\ P$›

definition *divergence-refine* :: ‹$'a$ *process* $\Rightarrow$ $'a$ *process* $\Rightarrow$ *bool*› (**infix** ‹$\sqsubseteq_D$› *53*)
where ‹$P \sqsubseteq_D Q \equiv \mathcal{D}\ Q \subseteq \mathcal{D}\ P$›

definition *failure-divergence-refine* :: ‹$'a$ *process* $\Rightarrow$ $'a$ *process* $\Rightarrow$ *bool*› (**infix** ‹$\sqsubseteq_{FD}$› *60*)
where ‹$P \sqsubseteq_{FD} Q \equiv P \leq Q$›

definition *trace-divergence-refine* :: ‹$'a$ *process* $\Rightarrow$ $'a$ *process* $\Rightarrow$ *bool*› (**infix** ‹$\sqsubseteq_{DT}$› *53*)

where ‹$P \sqsubseteq_{DT} Q \equiv P \sqsubseteq_T Q \wedge P \sqsubseteq_D Q$›

6.1 Idempotency

lemma *idem-F*[*simp*]: ‹$P \sqsubseteq_F P$›
and *idem-D*[*simp*]: ‹$P \sqsubseteq_D P$›
and *idem-T*[*simp*]: ‹$P \sqsubseteq_T P$›
and *idem-FD*[*simp*]: ‹$P \sqsubseteq_{FD} P$›
and *idem-DT*[*simp*]: ‹$P \sqsubseteq_{DT} P$›

by (*simp-all add*: *failure-refine-def divergence-refine-def trace-refine-def failure-divergence-refine-def trace-divergence-refine-def*)

6.2 Some obvious refinements

lemma *BOT-leF*[*simp*]: ‹$\bot \sqsubseteq_F Q$›
and *BOT-leD*[*simp*]: ‹$\bot \sqsubseteq_D Q$›
and *BOT-leT*[*simp*]: ‹$\bot \sqsubseteq_T Q$›
by (*simp-all add*: *failure-refine-def le-approx-lemma-F trace-refine-def le-approx1 divergence-refine-def le-approx-lemma-T*)

6.3 Antisymmetry

lemma *FD-antisym*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow Q \sqsubseteq_{FD} P \Longrightarrow P = Q$›
by (*simp add*: *failure-divergence-refine-def*)

lemma *DT-antisym*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow Q \sqsubseteq_{DT} P \Longrightarrow P = Q$›
apply (*simp add*: *trace-divergence-refine-def*)
oops

6.4 Transitivity

lemma *trans-F*: ‹$P \sqsubseteq_F Q \Longrightarrow Q \sqsubseteq_F S \Longrightarrow P \sqsubseteq_F S$›
and *trans-D*: ‹$P \sqsubseteq_D Q \Longrightarrow Q \sqsubseteq_D S \Longrightarrow P \sqsubseteq_D S$›
and *trans-T*: ‹$P \sqsubseteq_T Q \Longrightarrow Q \sqsubseteq_T S \Longrightarrow P \sqsubseteq_T S$›
and *trans-FD*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow Q \sqsubseteq_{FD} S \Longrightarrow P \sqsubseteq_{FD} S$›
and *trans-DT*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow Q \sqsubseteq_{DT} S \Longrightarrow P \sqsubseteq_{DT} S$›
by (*meson failure-refine-def order-trans*, *meson divergence-refine-def order-trans*, *meson trace-refine-def order-trans*, *meson failure-divergence-refine-def order-trans*,
meson divergence-refine-def order-trans trace-divergence-refine-def trace-refine-def)

6.5 Relations between refinements

lemma *leF-imp-leT*: ‹$P \sqsubseteq_F Q \Longrightarrow P \sqsubseteq_T Q$›
and *leFD-imp-leF*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow P \sqsubseteq_F Q$›
and *leFD-imp-leD*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow P \sqsubseteq_D Q$›
and *leDT-imp-leD*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow P \sqsubseteq_D Q$›
and *leDT-imp-leT*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow P \sqsubseteq_T Q$›
and *leF-leD-imp-leFD*: ‹$P \sqsubseteq_F Q \Longrightarrow P \sqsubseteq_D Q \Longrightarrow P \sqsubseteq_{FD} Q$›
and *leD-leT-imp-leDT*: ‹$P \sqsubseteq_D Q \Longrightarrow P \sqsubseteq_T Q \Longrightarrow P \sqsubseteq_{DT} Q$›
by (*simp-all add*: *F-subset-imp-T-subset failure-refine-def trace-refine-def divergence-refine-def*
trace-divergence-refine-def failure-divergence-refine-def le-ref-def)

6.6 More obvious refinements

lemma *BOT-leFD*[*simp*]: ‹$\bot \sqsubseteq_{FD} Q$›
and *BOT-leDT*[*simp*]: ‹$\bot \sqsubseteq_{DT} Q$›
by (*simp-all add*: *leF-leD-imp-leFD leD-leT-imp-leDT*)

lemma *leDT-STOP*[*simp*]: ‹$P \sqsubseteq_{DT} STOP$›
by (*simp add*: *D-STOP leD-leT-imp-leDT Nil-elem-T T-STOP divergence-refine-def trace-refine-def*)

lemma *leD-STOP*[*simp*]: ‹$P \sqsubseteq_{D} STOP$›
and *leT-STOP*[*simp*]: ‹$P \sqsubseteq_{T} STOP$›
by (*simp-all add*: *leDT-imp-leD leDT-imp-leT*)

6.7 Admissibility

lemma *le-F-adm*[*simp*]: ‹$cont\ (u::('a::cpo) \Rightarrow 'b\ process) \Longrightarrow monofun\ v \Longrightarrow adm(\lambda x.\ u\ x \sqsubseteq_F v\ x)$›
proof(*auto simp add*:*cont2contlubE adm-def failure-refine-def*)
fix *Y a b*
assume *1*: ‹*cont u*›
and *2*: ‹*monofun v*›
and *3*: ‹*chain Y*›
and *4*: ‹$\forall i.\ \mathcal{F}\ (v\ (Y\ i)) \subseteq \mathcal{F}\ (u\ (Y\ i))$›
and *5*: ‹$(a,\ b) \in \mathcal{F}\ (v\ (\bigsqcup x.\ Y\ x))$›
hence ‹$v\ (Y\ i)\ \sqsubseteq v\ (\bigsqcup i.\ Y\ i)$› **for** *i* **by** (*simp add*: *is-ub-thelub monofunE*)
hence ‹$\mathcal{F}\ (v\ (\bigsqcup i.\ Y\ i)) \subseteq \mathcal{F}\ (u\ (Y\ i))$› **for** *i* **using** *4 le-approx-lemma-F* **by** *blast*
then show ‹$(a,\ b) \in \mathcal{F}\ (\bigsqcup i.\ u\ (Y\ i))$›
using *F-LUB*[*OF ch2ch-cont*[*OF 1 3*]] *limproc-is-thelub*[*OF ch2ch-cont*[*OF 1 3*]] *5* **by** *force*
qed

lemma *le-T-adm*[*simp*]: ‹$cont\ (u::('a::cpo) \Rightarrow 'b\ process) \Longrightarrow monofun\ v \Longrightarrow adm(\lambda x.\ u\ x \sqsubseteq_T v\ x)$›
proof(*auto simp add*:*cont2contlubE adm-def trace-refine-def*)
fix *Y x*
assume *1*: ‹*cont u*›
and *2*: ‹*monofun v*›
and *3*: ‹*chain Y*›
and *4*: ‹$\forall i.\ \mathcal{T}\ (v\ (Y\ i)) \subseteq \mathcal{T}\ (u\ (Y\ i))$›
and *5*: ‹$x \in \mathcal{T}\ (v\ (\bigsqcup i.\ Y\ i))$›
hence ‹$v\ (Y\ i)\ \sqsubseteq v\ (\bigsqcup i.\ Y\ i)$› **for** *i* **by** (*simp add*: *is-ub-thelub monofunE*)
hence ‹$\mathcal{T}\ (v\ (\bigsqcup i.\ Y\ i)) \subseteq \mathcal{T}\ (u\ (Y\ i))$› **for** *i* **using** *4 le-approx-lemma-T* **by** *blast*
then show ‹$x \in \mathcal{T}\ (\bigsqcup i.\ u\ (Y\ i))$›
using *T-LUB*[*OF ch2ch-cont*[*OF 1 3*]] *limproc-is-thelub*[*OF ch2ch-cont*[*OF 1 3*]] *5* **by** *force*
qed

lemma *le-D-adm*[*simp*]: ‹*cont* (*u*::('*a*::*cpo*) ⇒ '*b process*) ⟹ *monofun v* ⟹ *adm*(λ*x*. *u x* $\sqsubseteq_D$ *v x*)›
proof(*auto simp add*:*cont2contlubE adm-def divergence-refine-def*)
  fix *Y x*
  assume *1*: ‹*cont u*›
    and *2*: ‹*monofun v*›
    and *3*: ‹*chain Y*›
    and *4*: ‹∀ *i*. $\mathcal{D}$ (*v* (*Y i*)) ⊆ $\mathcal{D}$ (*u* (*Y i*))›
    and *5*: ‹*x* ∈ $\mathcal{D}$ (*v* (⨆ *i*. *Y i*))›
  hence ‹*v* (*Y i*) ⊑ *v* (⨆ *i*. *Y i*)› **for** *i* **by** (*simp add*: *is-ub-thelub monofunE*)
  hence ‹$\mathcal{D}$ (*v* (⨆ *i*. *Y i*)) ⊆ $\mathcal{D}$ (*u* (*Y i*))› **for** *i* **using** *4 le-approx1* **by** *blast*
  then show ‹*x* ∈ $\mathcal{D}$ (⨆ *i*. *u* (*Y i*))›
    using *D-LUB*[*OF ch2ch-cont*[*OF 1 3*]] *limproc-is-thelub*[*OF ch2ch-cont*[*OF 1 3*]] *5* **by** *force*
qed

lemmas *le-FD-adm*[*simp*] = *le-adm*[*folded failure-divergence-refine-def*]

lemma *le-DT-adm*[*simp*]: ‹*cont* (*u*::('*a*::*cpo*) ⇒ '*b process*) ⟹ *monofun v* ⟹ *adm*(λ*x*. *u x* $\sqsubseteq_{DT}$ *v x*)›
  using *adm-conj*[*OF le-T-adm*[*of u v*] *le-D-adm*[*of u v*]] **by** (*simp add*: *trace-divergence-refine-def*)

end

Chapter 7

Generalisation of Normalisation of Non Deterministic CSP Processes

theory *CSP*
imports *CSP-Laws Process-Order*
begin

7.1 Monotonicity

lemma *mono-Det-D*[*simp*]: ‹$[\![P \sqsubseteq_D P'; S \sqsubseteq_D S']\!] \Longrightarrow (P \,\Box\, S) \sqsubseteq_D (P' \,\Box\, S')$›
by (*metis D-Det Un-mono divergence-refine-def*)

lemma *mono-Det-T*[*simp*]: ‹$[\![P \sqsubseteq_T P'; S \sqsubseteq_T S']\!] \Longrightarrow (P \,\Box\, S) \sqsubseteq_T (P' \,\Box\, S')$›
by (*metis T-Det Un-mono trace-refine-def*)

corollary *mono-Det-DT*[*simp*]: ‹$[\![P \sqsubseteq_{DT} P'; S \sqsubseteq_{DT} S']\!] \Longrightarrow (P \,\Box\, S) \sqsubseteq_{DT} (P' \,\Box\, S')$›
by (*simp-all add*: *trace-divergence-refine-def*)

lemmas *mono-Det-FD*[*simp*]= *mono-Det-FD*[*folded failure-divergence-refine-def*]

— Deterministic choice monotony doesn't hold for $\sqsubseteq_F$

lemma *mono-Ndet-F-left*[*simp*]: ‹$P \sqsubseteq_F Q \Longrightarrow (P \sqcap S) \sqsubseteq_F Q$›
by (*simp add*: *F-Ndet failure-refine-def order-trans*)

lemma *mono-Ndet-D-left*[*simp*]: ‹$P \sqsubseteq_D Q \Longrightarrow (P \sqcap S) \sqsubseteq_D Q$›
by (*simp add*: *D-Ndet divergence-refine-def order-trans*)

lemma *mono-Ndet-T-left*[*simp*]: ‹$P \sqsubseteq_T Q \Longrightarrow (P \sqcap S) \sqsubseteq_T Q$›

by (*simp add*: *T-Ndet trace-refine-def order-trans*)

7.2 Monotonicity

lemma *mono-Ndet-F*[*simp*]: ‹$[\![P \sqsubseteq_F P'; S \sqsubseteq_F S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq_F (P' \sqcap S')$›
by (*metis F-Ndet Un-mono failure-refine-def*)

lemma *mono-Ndet-D*[*simp*]: ‹$[\![P \sqsubseteq_D P'; S \sqsubseteq_D S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq_D (P' \sqcap S')$›
by (*metis D-Ndet Un-mono divergence-refine-def*)

lemma *mono-Ndet-T*[*simp*]: ‹$[\![P \sqsubseteq_T P'; S \sqsubseteq_T S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq_T (P' \sqcap S')$›
by (*metis T-Ndet Un-mono trace-refine-def*)

corollary *mono-Ndet-DT*[*simp*]: ‹$[\![P \sqsubseteq_{DT} P'; S \sqsubseteq_{DT} S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq_{DT} (P' \sqcap S')$›
by (*auto simp add*: *trace-divergence-refine-def*)

lemmas *mono-Ndet-FD*[*simp*]= *mono-Ndet-FD*[*folded failure-divergence-refine-def*]

corollary *mono-Ndet-DT-left*[*simp*]: ‹$P \sqsubseteq_{DT} Q \Longrightarrow (P \sqcap S) \sqsubseteq_{DT} Q$›
and *mono-Ndet-F-right*[*simp*]: ‹$P \sqsubseteq_F Q \Longrightarrow (S \sqcap P) \sqsubseteq_F Q$›
and *mono-Ndet-D-right*[*simp*]: ‹$P \sqsubseteq_D Q \Longrightarrow (S \sqcap P) \sqsubseteq_D Q$›
and *mono-Ndet-T-right*[*simp*]: ‹$P \sqsubseteq_T Q \Longrightarrow (S \sqcap P) \sqsubseteq_T Q$›
and *mono-Ndet-DT-right*[*simp*]: ‹$P \sqsubseteq_{DT} Q \Longrightarrow (S \sqcap P) \sqsubseteq_{DT} Q$›
by (*simp-all add*: *trace-divergence-refine-def Ndet-commute*)

lemmas *mono-Ndet-FD-left*[*simp*] = *mono-Ndet-FD-left*[*folded failure-divergence-refine-def*]
and *mono-Ndet-FD-right*[*simp*] = *mono-Ndet-FD-right*[*folded failure-divergence-refine-def*]

lemma *mono-Ndet-Det-FD*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_{FD} (P \sqsupset S)$›
by (*metis Det-id failure-divergence-refine-def mono-Det-FD mono-Ndet-FD-left mono-Ndet-FD-right order-refl*)

corollary *mono-Ndet-Det-F*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_F (P \sqsupset S)$›
and *mono-Ndet-Det-D*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_D (P \sqsupset S)$›
and *mono-Ndet-Det-T*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_T (P \sqsupset S)$›
and *mono-Ndet-Det-DT*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_{DT} (P \sqsupset S)$›
by (*simp-all add*: *leFD-imp-leF leFD-imp-leD leF-imp-leT leD-leT-imp-leDT*)

lemma *mono-Seq-F-right*[*simp*]: ‹$S \sqsubseteq_F S' \Longrightarrow (P ; S) \sqsubseteq_F (P ; S')$›
apply (*auto simp*: *failure-refine-def F-Seq append-single-T-imp-tickFree*)
using *NF-ND* **by** *fastforce*+

lemma *mono-Seq-D-right*[*simp*]: ‹$S \sqsubseteq_D S' \Longrightarrow (P ; S) \sqsubseteq_D (P ; S')$›

by (*auto simp*: *divergence-refine-def D-Seq*)

lemma *mono-Seq-T-right*[*simp*]: ‹$S \sqsubseteq_T S' \Longrightarrow (P\ ;\ S) \sqsubseteq_T (P\ ;\ S')$›
apply (*auto simp*: *trace-refine-def T-Seq append-single-T-imp-tickFree*)
using *D-T* **by** *fastforce+*

corollary *mono-Seq-DT-right*[*simp*]: ‹$S \sqsubseteq_{DT} S' \Longrightarrow (P\ ;\ S) \sqsubseteq_{DT} (P\ ;\ S')$›
by (*simp add*: *trace-divergence-refine-def*)

lemma *mono-Seq-DT-left*[*simp*]: ‹$P \sqsubseteq_{DT} P' \Longrightarrow (P\ ;\ S) \sqsubseteq_{DT} (P'\ ;\ S)$›
apply (*auto simp*: *trace-divergence-refine-def divergence-refine-def trace-refine-def D-Seq T-Seq*)
by (*metis* (*no-types*, *lifting*) *Nil-elem-T Process.F-T T-F append.right-neutral is-processT5-S3 subset-eq*)

— left Sequence monotony doesn't hold for $\sqsubseteq_F$, $\sqsubseteq_D$ and $\sqsubseteq_T$

corollary *mono-Seq-DT*[*simp*]: ‹$P \sqsubseteq_{DT} P' \Longrightarrow S \sqsubseteq_{DT} S' \Longrightarrow (P\ ;\ S) \sqsubseteq_{DT} (P'\ ;\ S')$›
using *mono-Seq-DT-left mono-Seq-DT-right trans-DT* **by** *blast*

lemmas *mono-Seq-FD*[*simp*]= *mono-Seq-FD*[*folded failure-divergence-refine-def*]

lemma *mono-Mprefix-F*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq_F Mprefix\ A\ Q$›
by (*auto simp*: *failure-refine-def F-Mprefix*) *blast+*

lemma *mono-Mprefix-D*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq_D Mprefix\ A\ Q$›
by (*auto simp*: *divergence-refine-def D-Mprefix*) *blast+*

lemma *mono-Mprefix-T*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq_T Mprefix\ A\ Q$›
by (*auto simp*: *trace-refine-def T-Mprefix*)

corollary *mono-Mprefix-DT*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq_{DT} Mprefix\ A\ Q$›
by (*simp add*: *trace-divergence-refine-def*)

lemmas *mono-Mprefix-FD*[*simp*] = *mono-Mprefix-FD*[*folded failure-divergence-refine-def*]

lemma *mono-Mprefix-DT-set*[*simp*]: ‹$A \subseteq B \Longrightarrow Mprefix\ B\ P \sqsubseteq_{DT} Mprefix\ A\ P$›

by (*auto simp add*:*T-Mprefix D-Mprefix trace-divergence-refine-def trace-refine-def divergence-refine-def*)

corollary *mono-Mprefix-D-set*[*simp*]: ‹$A \subseteq B \Longrightarrow Mprefix\ B\ P \sqsubseteq_D Mprefix\ A\ P$›
and *mono-Mprefix-T-set*[*simp*]: ‹$A \subseteq B \Longrightarrow Mprefix\ B\ P \sqsubseteq_T Mprefix\ A\ P$›

by (*simp-all add*: *leDT-imp-leD leDT-imp-leT*)

— Mprefix set monotony doesn't hold for $\sqsubseteq_F$ and $\sqsubseteq_{FD}$ where it holds for deterministic choice

lemma *mono-Hiding-F*[*simp*]: ‹$P \sqsubseteq_F Q \Longrightarrow (P \setminus A) \sqsubseteq_F (Q \setminus A)$›
apply(*cases* ‹$A = \{\}$›, *simp-all add*: *Hiding-set-empty failure-refine-def F-Hiding*, *intro conjI*, *blast*)
proof(*subst* (*2*) *Un-commute*, *rule subsetI*, *rule UnCI*, *auto*, *goal-cases*)
case (*1 b t u*)
then obtain *a* **where** *a*:‹$a \in A$› **by** *blast*
define *f* **where** *A*: ‹f = *rec-nat t* (λi *t*. *t* @ [*ev a*])›
hence *AA*: ‹*f* (*Suc i*) = (*f i*) @ [*ev a*]› **for** *i* **by** *simp*
hence *B*: ‹*strict-mono f*› **by** (*simp add*:*strict-mono-def lift-Suc-mono-less-iff*)
from *A* **have** *C*: ‹$t \in$ *range f*› **by** (*metis* (*mono-tags*, *lifting*) *old.nat.simps*(*6*) *range-eqI*)
{ fix *i*
have ‹$f\ i \in \mathcal{D}\ Q \wedge$ *tickFree* (*f i*) $\wedge$ *trace-hide* (*f i*) (*ev* ‘ *A*) = (*trace-hide t* (*ev* ‘ *A*))›
proof(*induct i*)
case *0*
then show *?case* **by** (*simp add*: *1*(*4*) *1*(*7*) *A*)
next
case (*Suc i*)
then show *?case*
apply (*simp add*:*AA a*)
using *is-processT7*[*rule-format*, *of* ‹*f i*› *Q* ‹[*ev a*]›] *front-tickFree-single* **by** *blast*
qed
}
with *B C* **have** ‹*isInfHiddenRun f P A* $\wedge$ $t \in$ *range f*›
by (*metis 1*(*1*) *D-T F-subset-imp-T-subset subsetD*)
with *1* **show** *?case* **by** *metis*
next
case (*2 b u f x*)
then show *?case* **using** *F-subset-imp-T-subset* **by** *blast*
qed

lemma *mono-Hiding-T*[*simp*]: ‹$P \sqsubseteq_T Q \Longrightarrow (P \setminus A) \sqsubseteq_T (Q \setminus A)$›
apply(*cases* ‹A={}›, *simp add*: *Hiding-set-empty*,
simp add:*trace-refine-def T-Hiding*, *intro conjI*)
proof(*goal-cases*)
case *1*
then show *?case*
proof(*subst Un-commute*, *rule-tac subsetI*, *rule-tac UnCI*, *auto*, *goal-cases*)
case *11*:(*1 t a*)
hence *tt*: ‹$t \in \mathcal{T}\ P$› **by** (*meson Process.F-T subset-eq*)
with *11*(*1*) *inf-hidden*[*of A t P*] **obtain** *f* **where** ‹*isInfHiddenRun f P A* $\wedge$ *t*

```
∈ range f› by auto
    with 11(4)[rule-format, of t ‹[]›] show ?case
      by (metis 11(1) tt append-Nil2 front-tickFree-Nil is-processT2-TR
              nonTickFree-n-frontTickFree tick-T-F)
  qed
next
  case 2
  then show ?case
  proof(subst Un-commute, auto, goal-cases)
    case 21:(1 t u a)
    define f where A: ‹f = rec-nat t (λi t. t @ [ev a])›
    hence AA: ‹f (Suc i) = (f i) @ [ev a]› for i by simp
    hence B: ‹strict-mono f› by (simp add:strict-mono-def lift-Suc-mono-less-iff)
    from A have C: ‹t ∈ range f›
      by (metis (mono-tags, lifting) old.nat.simps(6) range-eqI)
    { fix i
     have ‹f i ∈ 𝒟 Q ∧ tickFree (f i) ∧ trace-hide (f i) (ev ‘ A) = (trace-hide t (ev
‘ A))›
      proof(induct i)
        case 0
        then show ?case by (simp add: 21(4) 21(7) A)
      next
        case (Suc i)
        then show ?case
          apply (simp add:AA 21(6))
         using is-processT7[rule-format, of ‹f i› Q ‹[ev a]›] front-tickFree-single by
blast
      qed
    }
    with B C have ‹isInfHiddenRun f P A ∧ t ∈ range f› by (metis 21(1) D-T
subsetD)
    with 21 show ?case by metis
  next
    case 22:(2 b u f x)
    then show ?case by blast
  qed
qed

lemma mono-Hiding-DT[simp]: ‹P ⊑DT Q ⟹ (P \ A) ⊑DT (Q \ A)›
proof −
  assume as: ‹P ⊑DT Q›
  then have ‹(P \ A) ⊑D (Q \ A)›
    apply(auto simp:trace-divergence-refine-def divergence-refine-def
                trace-refine-def D-Hiding T-Hiding)
    by blast+
  with leDT-imp-leT[THEN mono-Hiding-T, OF as] show ?thesis by (simp add:
trace-divergence-refine-def)
qed
```

lemmas *mono-Hiding-FD*[*simp*] = *mono-Hiding-FD*[*folded failure-divergence-refine-def*]

— Obviously, Hide monotony doesn't hold for $\sqsubseteq_D$

lemma *mono-Sync-DT*[*simp*]: ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow (P \llbracket A \rrbracket Q) \sqsubseteq_{DT} (P' \llbracket A \rrbracket Q')$›
by (*simp add*:*trace-divergence-refine-def divergence-refine-def trace-refine-def T-Sync D-Sync*)
blast

lemmas *mono-Sync-FD*[*simp*] = *mono-Sync-FD*[*folded failure-divergence-refine-def*]

— Synchronization monotony doesn't hold for $\sqsubseteq_F$, $\sqsubseteq_D$ and $\sqsubseteq_T$

lemma *mono-Mndetprefix-F*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow$ *Mndetprefix* $A\ P \sqsubseteq_F$ *Mndetprefix* $A\ Q$›
by (*cases* ‹$A = \{\}$›, *auto simp*: *failure-refine-def F-Mndetprefix write0-def F-Mprefix*)

lemma *mono-Mndetprefix-D*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow$ *Mndetprefix* $A\ P \sqsubseteq_D$ *Mndetprefix* $A\ Q$›
by (*cases* ‹$A = \{\}$›, *auto simp*: *divergence-refine-def D-Mndetprefix write0-def D-Mprefix*)

lemma *mono-Mndetprefix-T*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *Mndetprefix* $A\ P \sqsubseteq_T$ *Mndetprefix* $A\ Q$›
by (*cases* ‹$A = \{\}$›, *auto simp*: *trace-refine-def T-Mndetprefix write0-def T-Mprefix*)

corollary *mono-Mndetprefix-DT*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow$ *Mndetprefix* $A\ P \sqsubseteq_{DT}$ *Mndetprefix* $A\ Q$›
by (*simp add*: *trace-divergence-refine-def*)

lemmas *mono-Mndetprefix-FD*[*simp*] = *mono-Mndetprefix-FD*[*folded failure-divergence-refine-def*]

lemmas *mono-Mndetprefix-FD-set*[*simp*] = *Mndetprefix-FD-subset*[*folded failure-divergence-refine-def*]

corollary *mono-Mndetprefix-F-set*[*simp*] : ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow$ *Mndetprefix* $B\ P \sqsubseteq_F$ *Mndetprefix* $A\ P$›

and *mono-Mndetprefix-D-set*[*simp*] : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix* $B\ P \sqsubseteq_D$ *Mndetprefix* $A\ P$›
and *mono-Mndetprefix-T-set*[*simp*] : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix* $B\ P \sqsubseteq_T$ *Mndetprefix* $A\ P$›
and *mono-Mndetprefix-DT-set*[*simp*]: ‹$A \subseteq B \Longrightarrow$ *Mndetprefix* $B\ P \sqsubseteq_{DT}$ *Mndetprefix* $A\ P$›
by (*cases* ‹$A = \{\}$›, *simp-all add*: *leFD-imp-leF leFD-imp-leD leF-imp-leT leD-leT-imp-leDT*)+

lemmas *Mprefix-refines-Mndetprefix-FD*[*simp*] =
Mprefix-refines-Mndetprefix[*folded failure-divergence-refine-def*]

corollary *Mprefix-refines-Mndetprefix-F*[*simp*] : ‹*Mndetprefix A P* $\sqsubseteq_F$ *Mprefix A P*›
and *Mprefix-refines-Mndetprefix-D*[*simp*] : ‹*Mndetprefix A P* $\sqsubseteq_D$ *Mprefix A P*›
and *Mprefix-refines-Mndetprefix-T*[*simp*] : ‹*Mndetprefix A P* $\sqsubseteq_T$ *Mprefix A P*›
and *Mprefix-refines-Mndetprefix-DT*[*simp*]: ‹*Mndetprefix A P* $\sqsubseteq_{DT}$ *Mprefix A P*›
by (*simp-all add*: *leFD-imp-leF leFD-imp-leD leF-imp-leT leD-leT-imp-leDT*)

lemma *mono-read-FD*[*simp*, *elim*]: $(\bigwedge x.\ P\ x \sqsubseteq_{FD} Q\ x) \Longrightarrow (c?x \to (P\ x)) \sqsubseteq_{FD} (c?x \to (Q\ x))$
by (*simp add*: *read-def*)

lemma *mono-write-FD*[*simp*, *elim*]: $(P \sqsubseteq_{FD} Q) \Longrightarrow (c!x \to P) \sqsubseteq_{FD} (c!x \to Q)$
by (*simp add*: *write-def*)

lemma *mono-write0-FD*[*simp*, *elim*]: $P \sqsubseteq_{FD} Q \Longrightarrow (a \to P) \sqsubseteq_{FD} (a \to Q)$
by (*simp add*: *write0-def*)

lemma *mono-read-DT*[*simp*, *elim*]: $(\bigwedge x.\ P\ x \sqsubseteq_{DT} Q\ x) \Longrightarrow (c?x \to (P\ x)) \sqsubseteq_{DT} (c?x \to (Q\ x))$
by (*simp add*: *read-def*)

lemma *mono-write-DT*[*simp*, *elim*]: $(P \sqsubseteq_{DT} Q) \Longrightarrow (c!x \to P) \sqsubseteq_{DT} (c!x \to Q)$
by (*simp add*: *write-def*)

lemma *mono-write0-DT*[*simp*, *elim*]: $P \sqsubseteq_{DT} Q \Longrightarrow (a \to P) \sqsubseteq_{DT} (a \to Q)$
by (*simp add*: *write0-def*)

— The following result is very useful, but we are not sure about where is its place

lemma *cont-process-rec*: ‹$P = (\mu\ X.\ f\ X) \Longrightarrow cont\ f \Longrightarrow P = f\ P$› **by** (*simp add*: *def-cont-fix-eq*)

end

theory *Assertions*
imports *CSP Process-Order*
begin

7.3 CSP Assertions

definition $DF :: 'a\ set \Rightarrow 'a\ process$
where $DF\ A \equiv \mu\ X.\ \sqcap\ x \in A \rightarrow X$

lemma *DF-unfold* : $DF\ A = (\sqcap\ z \in A \rightarrow DF\ A)$
by(*simp add*: *DF-def*, *rule trans*, *rule fix-eq*, *simp*)

definition *deadlock-free* $:: 'a\ process \Rightarrow bool$
where $deadlock\text{-}free\ P \equiv DF\ UNIV \sqsubseteq_{FD} P$

definition $DF_{SKIP} :: 'a\ set \Rightarrow 'a\ process$
where $DF_{SKIP}\ A \equiv \mu\ X.\ ((\sqcap\ x \in A \rightarrow X) \sqcap SKIP)$

definition *deadlock-free-v2* $:: 'a\ process \Rightarrow bool$
where $deadlock\text{-}free\text{-}v2\ P \equiv DF_{SKIP}\ UNIV \sqsubseteq_{F} P$

7.4 Some deadlock freeness laws

lemma *DF-subset*: $A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow DF\ B \sqsubseteq_{FD} DF\ A$
apply(*subst DF-def*, *rule fix-ind*)
apply(*rule le-FD-adm*, *simp-all add*: *monofunI*, *subst DF-unfold*)
by (*meson mono-Mndetprefix-FD mono-Mndetprefix-FD-set trans-FD*)

lemma *DF-Univ-freeness*: $A \neq \{\} \Longrightarrow (DF\ A) \sqsubseteq_{FD} P \Longrightarrow deadlock\text{-}free\ P$
by (*meson deadlock-free-def DF-subset failure-divergence-refine-def order.trans subset-UNIV*)

lemma *deadlock-free-Ndet*: $deadlock\text{-}free\ P \Longrightarrow deadlock\text{-}free\ Q \Longrightarrow deadlock\text{-}free\ (P \sqcap Q)$
by (*simp add*: *D-Ndet F-Ndet deadlock-free-def failure-divergence-refine-def le-ref-def*)

7.5 Preliminaries

lemma DF_{SKIP}*-unfold* : $DF_{SKIP}\ A = ((\sqcap\ z \in A \rightarrow DF_{SKIP}\ A) \sqcap SKIP)$
by(*simp add*: DF_{SKIP}*-def*, *rule trans*, *rule fix-eq*, *simp*)

7.6 Deadlock Free

lemma *div-free-*DF_{SKIP}: $\mathcal{D}(DF_{SKIP}\ A) = \{\}$

proof(*simp add*:DF_{SKIP}*-def fix-def*)
define Y **where** $Y \equiv \lambda i.\ iterate\ i\cdot(\Lambda\ x.\ (\sqcap xa{\in}A \rightarrow\ x) \sqcap SKIP)\cdot\bot$
hence *a*:*chain Y* **by** *simp*
have $\mathcal{D}\ (Y\ (Suc\ i)) = \{d.\ d \neq [] \wedge hd\ d\ \in (ev\ `\ A) \wedge tl\ d \in \mathcal{D}(Y\ i)\}$ **for** *i*
by (*cases A*={}, *auto simp add*:*Y-def D-STOP D-SKIP D-Mndetprefix write0-def D-Mprefix D-Ndet*)
hence *b*:$d \in \mathcal{D}(Y\ i) \Longrightarrow length\ d \geq i$ **for** *d i*
by (*induct i arbitrary*:*d*, *simp-all add*: *Nitpick.size-list-simp*(*2*))
{ fix *x*
assume *c*:$\forall\ i.\ x \in \mathcal{D}\ (Y\ i)$
from *b* **have** $x \notin \mathcal{D}\ (Y\ (Suc\ (length\ x)))$ **using** *Suc-n-not-le-n* **by** *blast*
with *c* **have** *False* **by** *simp*
}
with *a* **show** $\mathcal{D}\ (\bigsqcup i.\ Y\ i) = \{\}$
using *D-LUB*[*OF a*] *limproc-is-thelub*[*OF a*] **by** *auto*
qed

lemma *div-free-DF*: $\mathcal{D}(DF\ A) = \{\}$
proof −
have $DF_{SKIP}\ A\ \sqsubseteq_D DF\ A$
apply (*simp add*:DF_{SKIP}*-def*)
by(*rule fix-ind*, *simp-all add*: *monofunI*, *subst DF-unfold*, *simp*)
then show *?thesis* **using** *divergence-refine-def div-free-*DF_{SKIP} **by** *blast*
qed

lemma *deadlock-free-implies-div-free*: *deadlock-free* $P \Longrightarrow \mathcal{D}\ P = \{\}$
by (*simp add*: *deadlock-free-def div-free-DF failure-divergence-refine-def le-ref-def*)

7.7 Run

definition $RUN :: 'a\ set \Rightarrow 'a\ process$
where $RUN\ A \equiv \mu\ X.\ \Box\ x \in A \rightarrow X$

definition $CHAOS :: 'a\ set \Rightarrow 'a\ process$
where $CHAOS\ A \equiv \mu\ X.\ (STOP \sqcap (\Box\ x \in A \rightarrow X))$

definition *lifelock-free* :: $'a\ process \Rightarrow bool$
where *lifelock-free* $P \equiv CHAOS\ UNIV \sqsubseteq_{FD} P$

7.8 Reference processes and their unfolding rules

definition $CHAOS_{SKIP} :: 'a\ set \Rightarrow 'a\ process$
where $CHAOS_{SKIP}\ A \equiv \mu\ X.\ (SKIP \sqcap STOP \sqcap (\Box\ x \in A \rightarrow X))$

lemma *RUN-unfold* : $RUN\ A = (\Box\ z \in A \rightarrow RUN\ A)$
by(*simp add*: *RUN-def*, *rule trans*, *rule fix-eq*, *simp*)

lemma *CHAOS-unfold* : $CHAOS\ A = (STOP \sqcap (\Box\ z \in A \rightarrow CHAOS\ A))$
by(*simp add*: *CHAOS-def*, *rule trans*, *rule fix-eq*, *simp*)

lemma $CHAOS_{SKIP}$*-unfold*: $CHAOS_{SKIP}\ A \equiv SKIP \sqcap STOP \sqcap (\Box\ x \in A \rightarrow CHAOS_{SKIP}\ A)$
unfolding $CHAOS_{SKIP}$*-def* **using** *fix-eq*[*of* $\Lambda\ X.\ (SKIP \sqcap STOP \sqcap (\Box\ x \in A \rightarrow X))$] **by** *simp*

7.9 Process events and reference processes events

definition *events-of* **where** *events-of* $P \equiv (\bigcup t{\in}\mathcal{T}\ P.\ \{a.\ ev\ a \in set\ t\})$

lemma *events-DF*: *events-of* $(DF\ A) = A$
proof(*auto simp add*:*events-of-def*)
fix $x\ t$
show $t \in \mathcal{T}\ (DF\ A) \Longrightarrow ev\ x \in set\ t \Longrightarrow x \in A$
proof(*induct t*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a t*)
from *Cons*(*2*) **have** $a\ \#\ t \in \mathcal{T}\ (\sqcap z{\in}A \rightarrow\ DF\ A)$ **using** *DF-unfold* **by** *metis*
then obtain *aa* **where** $a = ev\ aa \wedge aa \in A \wedge t \in \mathcal{T}\ (DF\ A)$
by (*cases A*={}, *auto simp add*: *T-Mndetprefix write0-def T-Mprefix T-STOP*)
with *Cons* **show** *?case* **by** *auto*
qed
next
fix x
show $x \in A \Longrightarrow \exists\, xa{\in}\mathcal{T}\ (DF\ A).\ ev\ x \in set\ xa$
apply(*subst DF-unfold*, *cases A*={}, *auto simp add*:*T-Mndetprefix write0-def T-Mprefix*)
by (*metis Nil-elem-T list.sel*(*1*) *list.sel*(*3*) *list.set-intros*(*1*))
qed

lemma *events-*DF_{SKIP}: *events-of* $(DF_{SKIP}\ A) = A$
proof(*auto simp add*:*events-of-def*)
fix $x\ t$
show $t \in \mathcal{T}\ (DF_{SKIP}\ A) \Longrightarrow ev\ x \in set\ t \Longrightarrow x \in A$
proof(*induct t*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a t*)
from *Cons*(*2*) **have** $a\ \#\ t \in \mathcal{T}\ ((\sqcap z{\in}A \rightarrow\ DF_{SKIP}\ A) \sqcap SKIP)$ **using** DF_{SKIP}*-unfold* **by** *metis*
with *Cons* **obtain** *aa* **where** $a = ev\ aa \wedge aa \in A \wedge t \in \mathcal{T}\ (DF_{SKIP}\ A)$
by (*cases A*={}, *auto simp add*:*T-Mndetprefix write0-def T-Mprefix T-STOP T-SKIP T-Ndet*)
with *Cons* **show** *?case* **by** *auto*

```
  qed
next
  fix x
  show x ∈ A ⟹ ∃xa∈𝒯 (DF_SKIP A). ev x ∈ set xa
    apply(subst DF_SKIP-unfold, cases A={})
     apply(auto simp add:T-Mndetprefix write0-def T-Mprefix T-SKIP T-Ndet)
    by (metis Nil-elem-T list.sel(1) list.sel(3) list.set-intros(1))
qed

lemma events-RUN: events-of (RUN A) = A
proof(auto simp add:events-of-def)
  fix x t
  show t ∈ 𝒯 (RUN A) ⟹ ev x ∈ set t ⟹ x ∈ A
  proof(induct t)
    case Nil
    then show ?case by simp
  next
    case (Cons a t)
     from Cons(2) have a # t ∈ 𝒯 (□z∈A →  RUN A) using RUN-unfold by
metis
    then obtain aa where a = ev aa ∧ aa ∈ A ∧ t ∈ 𝒯 (RUN A) by (auto simp
add:T-Mprefix)
    with Cons show ?case by auto
  qed
next
  fix x
  show x ∈ A ⟹ ∃xa ∈ 𝒯(RUN A). ev x ∈ set xa
    apply(subst RUN-unfold, simp add: T-Mprefix)
    by (metis Nil-elem-T list.sel(1) list.sel(3) list.set-intros(1))
qed

lemma events-CHAOS: events-of (CHAOS A) = A
proof(auto simp add:events-of-def)
  fix x t
  show t ∈ 𝒯 (CHAOS A) ⟹ ev x ∈ set t ⟹ x ∈ A
  proof(induct t)
    case Nil
    then show ?case by simp
  next
    case (Cons a t)
     from Cons(2) have a # t ∈ 𝒯 (STOP ⊓ (□ z ∈ A → CHAOS A)) using
CHAOS-unfold by metis
    then obtain aa where a = ev aa ∧ aa ∈ A ∧ t ∈ 𝒯 (CHAOS A)
      by (auto simp add:T-Ndet T-Mprefix T-STOP)
    with Cons show ?case by auto
  qed
next
  fix x
  show x ∈ A ⟹ ∃xa∈𝒯 (CHAOS A). ev x ∈ set xa
```

```
    apply(subst CHAOS-unfold, simp add:T-Ndet T-Mprefix T-STOP)
    by (metis Nil-elem-T list.sel(1) list.sel(3) list.set-intros(1))
qed

lemma events-CHAOS_SKIP: events-of (CHAOS_SKIP A) = A
proof(auto simp add:events-of-def)
  fix x t
  show t ∈ 𝒯 (CHAOS_SKIP A) ⟹ ev x ∈ set t ⟹ x ∈ A
  proof(induct t)
    case Nil
    then show ?case by simp
  next
    case (Cons a t)
    from Cons(2) have a # t ∈ 𝒯 (SKIP ⊓ STOP ⊓ (□ z ∈ A → CHAOS_SKIP
A))
      using CHAOS_SKIP-unfold by metis
    with Cons obtain aa where a = ev aa ∧ aa ∈ A ∧ t ∈ 𝒯 (CHAOS_SKIP A)
      by (auto simp add:T-Ndet T-Mprefix T-STOP T-SKIP)
    with Cons show ?case by auto
  qed
next
  fix x
  show x ∈ A ⟹ ∃xa∈𝒯 (CHAOS_SKIP A). ev x ∈ set xa
   apply(subst CHAOS_SKIP-unfold, simp add:T-Ndet T-Mprefix T-STOP T-SKIP)
    by (metis Nil-elem-T list.sel(1) list.sel(3) list.set-intros(1))
qed

lemma events-div: 𝒟(P) ≠ {} ⟹  events-of (P) = UNIV
proof(auto simp add:events-of-def)
  fix x xa
  assume 1: x ∈ 𝒟 P
  show ∃x∈𝒯 P. ev xa ∈ set x
  proof(cases tickFree x)
    case True
    hence x@[ev xa] ∈ 𝒯 P
      using 1 NT-ND front-tickFree-single is-processT7 by blast
    then show ?thesis by force
  next
    case False
    hence (butlast x)@[ev xa] ∈ 𝒯 P
    by (metis 1 D-T D-imp-front-tickFree append-single-T-imp-tickFree butlast-snoc

                front-tickFree-single nonTickFree-n-frontTickFree process-charn)
    then show ?thesis by force
  qed
qed
```

lemma *DF_{SKIP}-subset-FD*: $A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow DF_{SKIP}\ B \sqsubseteq_{FD} DF_{SKIP}\ A$
apply(*subst DF_{SKIP}-def*, *rule fix-ind*, *rule le-FD-adm*, *simp-all add*:*monofunI*, *subst DF_{SKIP}-unfold*)
by (*rule mono-Ndet-FD*, *simp-all*) (*meson mono-Mndetprefix-FD mono-Mndetprefix-FD-set trans-FD*)

lemma *RUN-subset-DT*: $A \subseteq B \Longrightarrow RUN\ B \sqsubseteq_{DT} RUN\ A$
apply(*subst RUN-def*, *rule fix-ind*, *rule le-DT-adm*, *simp-all add*:*monofunI*, *subst RUN-unfold*)
by (*meson mono-Mprefix-DT mono-Mprefix-DT-set trans-DT*)

lemma *CHAOS-subset-FD*: $A \subseteq B \Longrightarrow CHAOS\ B \sqsubseteq_{FD} CHAOS\ A$
apply(*subst CHAOS-def*, *rule fix-ind*, *rule le-FD-adm*, *simp-all add*:*monofunI*, *subst CHAOS-unfold*)
by (*auto simp add*: *failure-divergence-refine-def le-ref-def*
D-Mprefix D-Ndet F-STOP F-Mprefix F-Ndet)

lemma *$CHAOS_{SKIP}$-subset-FD*: $A \subseteq B \Longrightarrow CHAOS_{SKIP}\ B \sqsubseteq_{FD} CHAOS_{SKIP}\ A$
apply(*subst $CHAOS_{SKIP}$-def*, *rule fix-ind*, *rule le-FD-adm*)
apply(*simp-all add*:*monofunI*, *subst $CHAOS_{SKIP}$-unfold*)
by (*auto simp add*: *failure-divergence-refine-def le-ref-def*
D-Mprefix D-Ndet F-STOP F-Mprefix F-Ndet)

7.10 Relations between refinements on reference processes

lemma *CHAOS-has-all-tickFree-failures*:
tickFree a $\Longrightarrow \{x.\ ev\ x \in set\ a\} \subseteq A \Longrightarrow (a,b) \in \mathcal{F}\ (CHAOS\ A)$
proof(*induct a*)
case *Nil*
then show *?case*
by (*subst CHAOS-unfold*, *simp add*:*F-Ndet F-STOP*)
next
case (*Cons a0 al*)
hence *tickFree al*
by (*metis append.left-neutral append-Cons front-tickFree-charn front-tickFree-mono*)
with *Cons* **show** *?case*
apply (*subst CHAOS-unfold*, *simp add*:*F-Ndet F-STOP F-Mprefix events-of-def*)
using *event-set* **by** *blast*
qed

lemma *$CHAOS_{SKIP}$-has-all-failures*:
assumes *as*:(*events-of P*) $\subseteq A$
shows $CHAOS_{SKIP}\ A \sqsubseteq_F P$
proof −
have *front-tickFree a* $\Longrightarrow set\ a \subseteq (\bigcup t{\in}\mathcal{T}\ P.\ set\ t) \Longrightarrow (a,b) \in \mathcal{F}\ (CHAOS_{SKIP}$

A) **for** *a b*
proof(*induct a*)
case *Nil*
then show *?case*
by (*subst* $CHAOS_{SKIP}$*-unfold*, *simp add*:*F-Ndet F-STOP*)
next
case (*Cons a0 al*)
hence *front-tickFree al*
by (*metis append.left-neutral append-Cons front-tickFree-charn front-tickFree-mono*)
with *Cons* **show** *?case*
apply (*subst* $CHAOS_{SKIP}$*-unfold*, *simp add*:*F-Ndet F-STOP F-SKIP F-Mprefix events-of-def as*)
apply(*cases a0=tick*)
apply (*metis append.simps*(*2*) *front-tickFree-charn front-tickFree-mono list.distinct*(*1*) *tickFree-Cons*)
using *event-set image-iff as*[*simplified events-of-def*] **by** *fastforce*
qed
thus *?thesis*
by (*simp add*: *F-T SUP-upper failure-refine-def process-charn subrelI*)
qed

corollary $CHAOS_{SKIP}$*-has-all-failures-ev*: $CHAOS_{SKIP}$ (*events-of P*) $\sqsubseteq_F$ *P*
and $CHAOS_{SKIP}$*-has-all-failures-Un*: $CHAOS_{SKIP}$ *UNIV* $\sqsubseteq_F$ *P*
by (*simp-all add*: $CHAOS_{SKIP}$*-has-all-failures*)

lemma DF_{SKIP}*-DF-refine-F*: DF_{SKIP} *A* $\sqsubseteq_F$ *DF A*
by (*simp add*:DF_{SKIP}*-def*, *rule fix-ind*, *simp-all add*: *monofunI*, *subst DF-unfold*, *simp*)

lemma *DF-RUN-refine-F*: *DF A* $\sqsubseteq_F$ *RUN A*
apply (*simp add*:*DF-def*, *rule fix-ind*, *simp-all add*: *monofunI*, *subst RUN-unfold*)
by (*meson Mprefix-refines-Mndetprefix-F mono-Mndetprefix-F trans-F*)

lemma *CHAOS-DF-refine-F*: *CHAOS A* $\sqsubseteq_F$ *DF A*
apply (*simp add*:*CHAOS-def DF-def*, *rule parallel-fix-ind*, *simp-all add*: *monofunI*)
apply (*rule le-F-adm*, *simp-all add*: *monofun-snd*)
by (*cases A={}*, *auto simp add*:*adm-def failure-refine-def F-Mndetprefix F-Mprefix write0-def F-Ndet F-STOP*)

corollary $CHAOS_{SKIP}$*-CHAOS-refine-F*: $CHAOS_{SKIP}$ *A* $\sqsubseteq_F$ *CHAOS A*
and $CHAOS_{SKIP}$*-*DF_{SKIP}*-refine-F*: $CHAOS_{SKIP}$ *A* $\sqsubseteq_F$ DF_{SKIP} *A*
by (*simp-all add*: $CHAOS_{SKIP}$*-has-all-failures events-CHAOS events-*DF_{SKIP} *trans-F*[*OF CHAOS-DF-refine-F DF-RUN-refine-F*])

lemma *div-free-CHAOS*$_{SKIP}$: $\mathcal{D}$ (*CHAOS*$_{SKIP}$ *A*) = {}
proof −
have *DF*$_{SKIP}$ *A* $\sqsubseteq_D$ *CHAOS*$_{SKIP}$ *A*
proof (*simp add*:*DF*$_{SKIP}$*-def*, *rule fix-ind*, *simp-all add*: *monofunI*, *subst CHAOS*$_{SKIP}$*-unfold*)
fix *x*
assume *1*:*x* $\sqsubseteq_D$ *CHAOS*$_{SKIP}$ *A*
have *a*:((⊓*xa*∈*A* → *x*) ⊓ *SKIP*) = (*SKIP* ⊓ *SKIP* ⊓ (⊓*xa*∈*A* → *x*))
by (*simp add*: *Ndet-commute Ndet-id*)
from *1* **have** *b*:(*SKIP* ⊓ *SKIP* ⊓ (⊓*xa*∈*A* → *x*)) $\sqsubseteq_D$ (*SKIP* ⊓ *STOP* ⊓ (□*xa*∈*A* → *CHAOS*$_{SKIP}$ *A*))
by (*meson Mprefix-refines-Mndetprefix-D idem-D leD-STOP mono-Mprefix-D mono-Ndet-D trans-D*)
from *a b* **show** ((⊓*xa*∈*A* → *x*) |−| *SKIP*) $\sqsubseteq_D$ (*SKIP* |−| *STOP* |−| *Mprefix A* (λ*x*. *CHAOS*$_{SKIP}$ *A*))
by *simp*
qed
then show *?thesis* **using** *divergence-refine-def div-free-DF*$_{SKIP}$ **by** *blast*
qed

lemma *div-free-CHAOS*: $\mathcal{D}$(*CHAOS A*) = {}
proof −
have *CHAOS*$_{SKIP}$ *A* $\sqsubseteq_D$ *CHAOS A*
apply (*simp add*:*CHAOS*$_{SKIP}$*-def*, *rule fix-ind*)
by (*simp-all add*: *monofunI*, *subst CHAOS-unfold*, *simp*)
then show *?thesis* **using** *divergence-refine-def div-free-CHAOS*$_{SKIP}$ **by** *blast*
qed

lemma *div-free-RUN*: $\mathcal{D}$(*RUN A*) = {}
proof −
have *CHAOS A* $\sqsubseteq_D$ *RUN A*
by (*simp add*:*CHAOS-def*, *rule fix-ind*, *simp-all add*: *monofunI*, *subst RUN-unfold*, *simp*)
then show *?thesis* **using** *divergence-refine-def div-free-CHAOS* **by** *blast*
qed

corollary *DF*$_{SKIP}$*-DF-refine-FD*: *DF*$_{SKIP}$ *A* $\sqsubseteq_{FD}$ *DF A*
and *DF-RUN-refine-FD*: *DF A* $\sqsubseteq_{FD}$ *RUN A*
and *CHAOS-DF-refine-FD*: *CHAOS A* $\sqsubseteq_{FD}$ *DF A*
and *CHAOS*$_{SKIP}$*-CHAOS-refine-FD*: *CHAOS*$_{SKIP}$ *A* $\sqsubseteq_{FD}$ *CHAOS A*
and *CHAOS*$_{SKIP}$*-DF*$_{SKIP}$*-refine-FD*: *CHAOS*$_{SKIP}$ *A* $\sqsubseteq_{FD}$ *DF*$_{SKIP}$ *A*
using *div-free-DF*$_{SKIP}$[*of A*] *div-free-CHAOS*$_{SKIP}$[*of A*] *div-free-DF*[*of A*] *div-free-RUN*[*of A*]
div-free-CHAOS[*of A*]
leF-leD-imp-leFD[*OF DF*$_{SKIP}$*-DF-refine-F*, *of A*] *leF-leD-imp-leFD*[*OF DF-RUN-refine-F*, *of A*]
leF-leD-imp-leFD[*OF CHAOS-DF-refine-F*, *of A*] *leF-leD-imp-leFD*[*OF CHAOS*$_{SKIP}$*-CHAOS-refine-F*, *of A*]
leF-leD-imp-leFD[*OF CHAOS*$_{SKIP}$*-DF*$_{SKIP}$*-refine-F*, *of A*]
by (*auto simp add*:*divergence-refine-def*)

lemma *traces-CHAOS-sub*: $\mathcal{T}(CHAOS\ A) \subseteq \{s.\ set\ s \subseteq ev\ `\ A\}$
proof(*auto*)
  fix *s sa*
  assume $s \in \mathcal{T}\ (CHAOS\ A)$ **and** $sa \in set\ s$
  then show $sa \in ev\ `\ A$
    apply (*induct s*, *simp*)
   by (*subst* (*asm*) (*2*) *CHAOS-unfold*, *cases A*={}, *auto simp add*:*T-Ndet T-STOP T-Mprefix*)
qed

lemma *traces-RUN-sub*: $\{s.\ set\ s \subseteq ev\ `\ A\} \subseteq \mathcal{T}(RUN\ A)$
proof(*auto*)
  fix *s*
  assume $set\ s \subseteq ev\ `\ A$
  then show $s \in \mathcal{T}\ (RUN\ A)$
   by (*induct s*, *simp add*: *Nil-elem-T*) (*subst RUN-unfold*, *auto simp add*:*T-Mprefix*)
qed

corollary *RUN-all-tickfree-traces1*: $\mathcal{T}(RUN\ A) = \{s.\ set\ s \subseteq ev\ `\ A\}$
    and *DF-all-tickfree-traces1*: $\mathcal{T}(DF\ A) = \{s.\ set\ s \subseteq ev\ `\ A\}$
    and *CHAOS-all-tickfree-traces1*: $\mathcal{T}(CHAOS\ A) = \{s.\ set\ s \subseteq ev\ `\ A\}$
  using *DF-RUN-refine-F*[*THEN leF-imp-leT*, *simplified trace-refine-def*]
      CHAOS-DF-refine-F[*THEN leF-imp-leT*,*simplified trace-refine-def*]
      traces-CHAOS-sub traces-RUN-sub **by** *blast*+

corollary *RUN-all-tickfree-traces2*: $tickFree\ s \Longrightarrow s \in \mathcal{T}(RUN\ UNIV)$
    and *DF-all-tickfree-traces2*: $tickFree\ s \Longrightarrow s \in \mathcal{T}(DF\ UNIV)$
    and *CHAOS-all-tickfree-trace2*: $tickFree\ s \Longrightarrow s \in \mathcal{T}(CHAOS\ UNIV)$
   apply(*simp-all add*:*tickFree-def RUN-all-tickfree-traces1*
                DF-all-tickfree-traces1 CHAOS-all-tickfree-traces1)
  by (*metis event-set insertE subsetI*)+

lemma *traces-*$CHAOS_{SKIP}$*-sub*: $\mathcal{T}(CHAOS_{SKIP}\ A) \subseteq \{s.\ front\text{-}tickFree\ s \wedge set\ s \subseteq (ev\ `\ A \cup \{tick\})\}$
proof(*auto simp add*:*is-processT2-TR*)
  fix *s sa*
  assume $s \in \mathcal{T}\ (CHAOS_{SKIP}\ A)$ **and** $sa \in set\ s$ **and** $sa \notin ev\ `\ A$
  then show $sa = tick$
   apply (*induct s*, *simp*)
   by (*subst* (*asm*) (*2*) $CHAOS_{SKIP}$*-unfold*, *cases A*={}, *auto simp add*:*T-Ndet T-STOP T-SKIP T-Mprefix*)
qed

lemma *traces-*DF_{SKIP}*-sub*:
  $\{s.\ front\text{-}tickFree\ s \wedge set\ s \subseteq (ev\ `\ A \cup \{tick\})\} \subseteq \mathcal{T}(DF_{SKIP}\ A::'a\ process)$
proof(*auto*)
  fix *s*

assume *a:front-tickFree s* **and** *b:set s* $\subseteq$ *insert tick* (*ev* ‘ *A*)
have *c:front-tickFree* ((*tick*::*'a event*) # *s*) $\Longrightarrow$ *s* = [] **for** *s*
by (*metis butlast.simps*(*2*) *butlast-snoc front-tickFree-charn list.distinct*(*1*) *tick-Free-Cons*)
with *a b* **show** *s* $\in \mathcal{T}$ (DF_{SKIP} *A*)
apply (*induct s*, *simp add*: *Nil-elem-T*, *subst* DF_{SKIP}*-unfold*, *cases A*={})
apply (*subst* DF_{SKIP}*-unfold*, *cases A*={})
apply(*auto simp add*:*T-Mprefix T-Mndetprefix write0-def T-SKIP T-Ndet T-STOP*)
apply (*metis append-Cons append-Nil front-tickFree-charn front-tickFree-mono*)
by (*metis append-Cons append-Nil front-tickFree-mono*)
qed

corollary DF_{SKIP}*-all-front-tickfree-traces1*:
$\mathcal{T}$(DF_{SKIP} *A*) = {*s*. *front-tickFree s* $\wedge$ *set s* $\subseteq$ (*ev* ‘ *A* $\cup$ {*tick*})}
and $CHAOS_{SKIP}$*-all-front-tickfree-traces1*:
$\mathcal{T}$($CHAOS_{SKIP}$ *A*) = {*s*. *front-tickFree s* $\wedge$ *set s* $\subseteq$ (*ev* ‘ *A* $\cup$ {*tick*})}
using $CHAOS_{SKIP}$*-*DF_{SKIP}*-refine-F*[*THEN leF-imp-leT*, *simplified trace-refine-def*]
traces-$CHAOS_{SKIP}$*-sub traces-*DF_{SKIP}*-sub* **by** *blast+*

corollary DF_{SKIP}*-all-front-tickfree-traces2*: *front-tickFree s* $\Longrightarrow$ *s* $\in \mathcal{T}$(DF_{SKIP} *UNIV*)
and $CHAOS_{SKIP}$*-all-front-tickfree-traces2*: *front-tickFree s* $\Longrightarrow$ *s* $\in \mathcal{T}$($CHAOS_{SKIP}$ *UNIV*)
apply(*simp-all add*:*tickFree-def* DF_{SKIP}*-all-front-tickfree-traces1*
$CHAOS_{SKIP}$*-all-front-tickfree-traces1*)
by (*metis event-set subsetI*)+

corollary DF_{SKIP}*-has-all-traces*: DF_{SKIP} *UNIV* $\sqsubseteq_T$ *P*
and $CHAOS_{SKIP}$*-has-all-traces*: $CHAOS_{SKIP}$ *UNIV* $\sqsubseteq_T$ *P*
apply (*simp add*:*trace-refine-def* DF_{SKIP}*-all-front-tickfree-traces2 is-processT2-TR subsetI*)
by (*simp add*:*trace-refine-def* $CHAOS_{SKIP}$*-all-front-tickfree-traces2 is-processT2-TR subsetI*)

lemma *deadlock-free-implies-non-terminating*:
deadlock-free (*P*::*'a process*) $\Longrightarrow \forall$ *s*$\in\mathcal{T}$ *P*. *tickFree s*
unfolding *deadlock-free-def* **apply**(*drule leFD-imp-leF*, *drule leF-imp-leT*) **unfolding** *trace-refine-def*
using *DF-all-tickfree-traces1*[*of* (*UNIV*::*'a set*)] *tickFree-def* **by** *fastforce*

lemma *deadlock-free-v2-is-right*:
deadlock-free-v2 (*P*::*'a process*) $\longleftrightarrow$ ($\forall$ *s*$\in\mathcal{T}$ *P*. *tickFree s* $\longrightarrow$ (*s*, *UNIV*::*'a event set*) $\notin \mathcal{F}$ *P*)
proof

assume *a*:*deadlock-free-v2 P*
have *tickFree s* $\longrightarrow$ (*s*, *UNIV*) $\notin \mathcal{F}$ (DF_{SKIP} *UNIV*) **for** *s*::'*a event list*
proof(*induct s*)
case *Nil*
then show *?case* **by** (*subst* DF_{SKIP}*-unfold*, *simp add*:*F-Mndetprefix write0-def F-Mprefix F-Ndet F-SKIP*)
next
case (*Cons a s*)
then show *?case*
by (*subst* DF_{SKIP}*-unfold*, *simp add*:*F-Mndetprefix write0-def F-Mprefix F-Ndet F-SKIP*)
qed
with *a* **show** $\forall$ *s*$\in\mathcal{T}$ *P*. *tickFree s* $\longrightarrow$ (*s*, *UNIV*) $\notin \mathcal{F}$ *P*
using *deadlock-free-v2-def failure-refine-def* **by** *blast*
next
assume *as1*:$\forall$ *s*$\in\mathcal{T}$ *P*. *tickFree s* $\longrightarrow$ (*s*, *UNIV*) $\notin \mathcal{F}$ *P*
have *as2*:*front-tickFree s* $\Longrightarrow$ ($\exists$ *aa* $\in$ *UNIV*. *ev aa* $\notin$ *b*) $\Longrightarrow$ (*s*, *b*) $\in \mathcal{F}$ (DF_{SKIP} (*UNIV*::'*a set*))
for *s b*
proof(*induct s*)
case *Nil*
then show *?case*
by (*subst* DF_{SKIP}*-unfold*, *auto simp add*:*F-Mndetprefix write0-def F-Mprefix F-Ndet F-SKIP*)
next
case (*Cons hda tla*)
then show *?case*
proof(*simp add*:DF_{SKIP}*-def fix-def*)
define *Y* **where** *Y* $\equiv$ λ*i*. *iterate i*$\cdot$(Λ *x*. ($\sqcap$*xa*$\in$(*UNIV*::'*a set*) $\rightarrow$ *x*) $\sqcap$ *SKIP*)$\cdot\bot$
assume *a*:*front-tickFree* (*hda* # *tla*) **and** *b*:*front-tickFree tla* $\Longrightarrow$ (*tla*, *b*) $\in \mathcal{F}$ ($\bigsqcup$ *i*. *Y i*)
and *c*:$\exists$ *aa*. *ev aa* $\notin$ *b*
from *Y-def* **have** *cc*:*chain Y* **by** *simp*
from *b* **have** *d*:*front-tickFree tla* $\Longrightarrow$ $\exists$ *aa*$\in$*UNIV*. *ev aa* $\notin$ *b* $\Longrightarrow$(*tla*, *b*) $\in \mathcal{F}$ (*Y i*) **for** *i*
using *F-LUB*[*OF cc*] *limproc-is-thelub*[*OF cc*] **by** *simp*
from *Y-def* **have** *e*:$\mathcal{F}$(*Mndetprefix UNIV* (λ*x*. *Y i*) $\sqcap$ *SKIP*) $\subseteq \mathcal{F}$ (*Y* (*Suc i*)) **for** *i* **by**(*simp*)
from *a* **have** *f*:*tla* $\neq$ [] $\Longrightarrow$ *hda* $\neq$ *tick front-tickFree tla*
apply (*metis butlast.simps*(*2*) *butlast-snoc front-tickFree-charn*
list.distinct(*1*) *tickFree-Cons*)
by (*metis a append-Cons append-Nil front-tickFree-Nil front-tickFree-mono*)
have *g*:(*hda*#*tla*, *b*) $\in \mathcal{F}$ (*Y* (*Suc i*)) **for** *i*
using *f c e*[*of i*] *d*[*of i*]
by (*auto simp*: *F-Mndetprefix write0-def F-Mprefix Y-def F-Ndet F-SKIP*) (*metis event.exhaust*)+
have *h*:(*hda*#*tla*, *b*) $\in \mathcal{F}$ (*Y 0*)
using *NF-ND cc g po-class.chainE proc-ord2a* **by** *blast*

```
      from a b c show (hda#tla, b) ∈ ℱ (⨆ i. Y i)
        using F-LUB[OF cc] is-ub-thelub[OF cc]
         by (metis D-LUB-2 cc g limproc-is-thelub po-class.chainE proc-ord2a pro-
cess-charn)
    qed
  qed
  show deadlock-free-v2 P
  proof(auto simp add:deadlock-free-v2-def failure-refine-def)
    fix s b
    assume as3:(s, b) ∈ ℱ P
    hence a1:s ∈ 𝒯 P front-tickFree s
       using F-T apply blast
      using as3 is-processT2 by blast
    show (s, b) ∈ ℱ (DF_SKIP UNIV)
    proof(cases tickFree s)
      case FT-True:True
      hence a2:(s, UNIV) ∉ ℱ P
        using a1 as1 by blast
      then show ?thesis
        by (metis FT-True UNIV-I UNIV-eq-I a1(2) as2 as3 emptyE event.exhaust

                  is-processT6-S1 tickFree-implies-front-tickFree-single)
    next
      case FT-False: False
      then show ?thesis
        by (meson T-F-spec UNIV-witness a1(2) append-single-T-imp-tickFree
                  as2 emptyE is-processT5-S7)
    qed
  qed
qed

lemma deadlock-free-v2-implies-div-free: deadlock-free-v2 P ⟹ 𝒟 P = {}
  by (metis F-T append-single-T-imp-tickFree deadlock-free-v2-is-right ex-in-conv
          nonTickFree-n-frontTickFree process-charn)

corollary deadlock-free-v2-FD: deadlock-free-v2 P = DF_SKIP UNIV ⊑_FD P
  unfolding deadlock-free-v2-def
  using deadlock-free-v2-implies-div-free leFD-imp-leF leF-leD-imp-leFD
        deadlock-free-v2-def divergence-refine-def
  by fastforce

lemma all-events-refusal:
   (s, {tick} ∪ ev ‘ (events-of P)) ∈ ℱ P ⟹ (s, UNIV::'a event set) ∈ ℱ P
proof −
  assume a1:(s, {tick} ∪ ev ‘ events-of P) ∈ ℱ P
  { assume (s, UNIV) ∉ ℱ P
    then obtain c where c ∉ {tick} ∪ ev ‘ events-of P ∧ s @ [c] ∈ 𝒯 P
      using is-processT5-S1[of s {tick} ∪ ev ‘ events-of P P
          UNIV − ({tick} ∪ ev ‘ events-of P), simplified] F-T a1 by auto
```

hence *False* **by** (*simp add*:*events-of-def*, *cases c*) *fastforce*+
}
with *a1* **show** $(s, UNIV) \in \mathcal{F}\ P$ **by** *blast*
qed

corollary *deadlock-free-v2-is-right-wrt-events*:
deadlock-free-v2 (P::'*a process*) $\longleftrightarrow$
$(\forall\, s{\in}\mathcal{T}\ P.\ tickFree\ s \longrightarrow (s, \{tick\} \cup ev\ `\ (events\text{-}of\ P)) \notin \mathcal{F}\ P)$
unfolding *deadlock-free-v2-is-right* **using** *all-events-refusal* **apply** *auto*
using *is-processT4* **by** *blast*

lemma *deadlock-free-is-deadlock-free-v2*:
deadlock-free P $\Longrightarrow$ *deadlock-free-v2 P*
using DF_{SKIP}*-DF-refine-FD deadlock-free-def deadlock-free-v2-FD trans-FD* **by** *blast*

7.11 *deadlock-free* and *deadlock-free-v2* with *SKIP* and *STOP*

lemma *deadlock-free-v2-SKIP*: *deadlock-free-v2 SKIP*
unfolding *deadlock-free-v2-FD* **by** (*subst* DF_{SKIP}*-unfold*) *simp*

lemma *non-deadlock-free-SKIP*: ‹¬ *deadlock-free SKIP*›
by (*metis T-SKIP deadlock-free-implies-non-terminating insertCI non-tickFree-tick*)

lemma *non-deadlock-free-v2-STOP*: ‹¬ *deadlock-free-v2 STOP*›
by (*simp add*: *F-STOP Nil-elem-T deadlock-free-v2-is-right*)

lemma *non-deadlock-free-STOP*: ‹¬ *deadlock-free STOP*›
using *deadlock-free-is-deadlock-free-v2 non-deadlock-free-v2-STOP* **by** *blast*

end

Chapter 8

Conclusion

8.1 Related Work

As mentioned earlier, this work has its very ancient roots in a first formalization of A. Camilieri in the early 90s in HOL. This work was reformulated and substantially extended in HOL-CSP 1.0 published in 1997. In 2005, Roggenbach and Isobe published CSP-Prover, a formal theory of a (fragment of) the Failures model of CSP. This work led to a couple of publications culminating in [6]; emphasis was put on actually completing the CSP theory up to the point where it is sufficiently tactically supported to serve as a kind of tool. This theory is still maintained and last releases (the latest one was released on 18 February 2019) can be found under https://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html. This theory represents the first half of Roscoes theory of a Failures/Divergence model, i.e. the Failures part. More recently, Pasquale Noce [9, 11, 10] developed a theory of non-interference notions based on an abstract denotational model fragment of the Failure/Divergence Model of CSP (without continuity and algebraic laws); this theory could probably be rebuilt on top of our work.

The present work could be another, more "classic" foundation of test-generation techniques of this kind paving the way to an interaction with FDR and its possibility to generate labelled transition systems as output that could drive specialized tactics in HOL-CSP 2.0.

8.2 Lessons learned

We have ported a first formalization in Isabelle/HOL on the Failure/Divergence model of CSP, done with Isabelle93-7 in 1997, to a modern Isabelle version. Particularly, we use the modern declarative proof style available in Isabelle/Isar instead of imperative proof one, the latter being used in the old version. On the one hand, it is worth noting that some of the old theories

still have a surprisingly high value: Actually it took time to develop the right granularity of abstraction in lemmas, which is thus still quite helpful and valuable to reconstruct the theory in the new version. If a substantially large body of lemmas is available, the degree of automation tends to increase. On the other hand, redevelopment from scratch is unavoidable in parts where basic libraries change. For example, this was a necessary consequence of our decision to base HOL-CSP 2.0 on HOLCF instead of continuing the development of an older fixed-point theory; nearly all continuity proofs had to be redeveloped. Moreover, a fresh look on old proof-obligations may incite unexpected generalizations and some newly proved lemmas that cannot be constructed in the old version even with several attempts. The influence of the chosen strategy (from scratch or refactoring) on the proof length is inconclusive.

Note that our data does not allow to make a prediction on the length of a porting project — the effort was distributed over a too long period and performed by a team with initially very different knowledge about CSP and interactive theorem proving.

8.3 A Summary on New Results

Compared to the original version of HOL-CSP 1.0, the present theory is complete relative to Roscoe's Book[12]. It contains a number of new theorems and some interesting (and unexpected) generalizations:

1. $?P \sqsubseteq ?Q \Longrightarrow ?P \setminus ?A \sqsubseteq ?Q \setminus ?A$ is now also valid for the infinite case (arbitrary hide-set A).

2. $P \setminus A \cup B = P \setminus A \setminus B$ is true for *finite* A (see *finite* $?A \Longrightarrow ?P \setminus ?A \cup ?B = ?P \setminus ?A \setminus ?B$); this was not even proven in HOL-CSP 1.0 for the singleton case! It can be considered as the most complex theorem of this theory.

3. distribution laws of hiding over synchronisation $[\![$*finite* $?A$; $?A \cap ?S = \{\}]\!] \Longrightarrow ?P \; [\![?S]\!] \; ?Q \setminus ?A = (?P \setminus ?A) \; [\![?S]\!] \; (?Q \setminus ?A)$; however, this works only in the finite case. A true monster proof.

4. the synchronization operator is associative $?P \; [\![?S]\!] \; ?Q \; [\![?S]\!] \; ?T = ?P \; [\![?S]\!] \; (?Q \; [\![?S]\!] \; ?T)$. (In HOL-CSP 1.0, this had only be shown for special cases like $?P||?Q||?T = ?P||(?Q||?T)$).

5. the generalized non-deterministic choice operator — relevant for proofs of deadlock-freeness — has been added to the theory *Mndetprefix* $?A$ $?P \equiv$ *if* $?A = \{\}$ *then STOP else Abs-process* $(\bigcup_{x \in ?A} \mathcal{F}\ (x \rightarrow ?P\ x), \bigcup_{x \in ?A} \mathcal{D}\ (x \rightarrow ?P\ x))$; it is proven monotone in the general case and

continuous for the special case $(\bigwedge x.\ cont\ (?f\ x)) \Longrightarrow cont\ (\lambda y.\ \sqcap z{\in}?A \rightarrow\ ?f\ z\ y)$ relevant for the definition of the deadlock reference processes $DF\ ?A \equiv \mu\ x.\ \sqcap xa{\in}?A \rightarrow\ x$ and $DF_{SKIP}\ ?A \equiv \mu\ x.\ (\sqcap xa{\in}?A \rightarrow x) \sqcap SKIP$.

end

Chapter 9

Annex: Refinement Example with Buffer over infinite Alphabet

theory *CopyBuffer*
imports *Assertions*
begin

9.1 Defining the Copy-Buffer Example

datatype $'a$ *channel* = *left* $'a$ | *right* $'a$ | *mid* $'a$ | *ack*

definition *SYN* :: ($'a$ *channel*) *set*
where *SYN* $\equiv$ (*range mid*) $\cup$ {*ack*}

definition *COPY* :: ($'a$ *channel*) *process*
where *COPY* $\equiv$ (μ *COPY*. *left*?x $\rightarrow$ (*right*!x $\rightarrow$ *COPY*))

definition *SEND* :: ($'a$ *channel*) *process*
where *SEND* $\equiv$ (μ *SEND*. *left*?x $\rightarrow$ (*mid*!x $\rightarrow$(*ack* $\rightarrow$ *SEND*)))

definition *REC* :: ($'a$ *channel*) *process*
where *REC* $\equiv$ (μ *REC*. *mid*?x $\rightarrow$ (*right*!x $\rightarrow$ (*ack* $\rightarrow$ *REC*)))

definition *SYSTEM* :: ($'a$ *channel*) *process*
where ‹*SYSTEM* $\equiv$ (*SEND* $[\![$ *SYN* $]\!]$ *REC*) $\setminus$ *SYN*›

thm *SYSTEM-def*

9.2 The Standard Proof

9.2.1 Channels and Synchronization Sets

First part: abstract properties for these events to SYN. This kind of stuff could be automated easily by some extra-syntax for channels and SYN-sets.

lemma [*simp*]: *left x* $\notin$ *SYN*
by(*auto simp*: *SYN-def*)

lemma [*simp*]: *right x* $\notin$ *SYN*
by(*auto simp*: *SYN-def*)

lemma [*simp*]: *ack* $\in$ *SYN*
by(*auto simp*: *SYN-def*)

lemma [*simp*]: *mid x* $\in$ *SYN*
by(*auto simp*: *SYN-def*)

lemma [*simp*]: *inj mid*
by(*auto simp*: *inj-on-def*)

lemma *finite* (*SYN*:: *'a channel set*) $\Longrightarrow$ *finite* $\{(t::'a).\ True\}$
by (*metis* (*no-types*) *SYN-def UNIV-def channel.inject*(*3*) *finite-Un finite-imageD inj-on-def*)

9.2.2 Definitions by Recursors

Second part: Derive recursive process equations, which are easier to handle in proofs. This part IS actually automated if we could reuse the fixrec-syntax below.

lemma *COPY-rec*:
COPY = *left*?*x* $\rightarrow$ *right*!*x* $\rightarrow$ *COPY*
by(*simp add*: *COPY-def*,*rule trans*, *rule fix-eq*, *simp*)

lemma *SEND-rec*:
SEND = *left*?*x* $\rightarrow$ *mid*!*x* $\rightarrow$ (*ack* $\rightarrow$ *SEND*)
by(*simp add*: *SEND-def*,*rule trans*, *rule fix-eq*, *simp*)

lemma *REC-rec*:
REC = *mid*?*x* $\rightarrow$ *right*!*x* $\rightarrow$ (*ack* $\rightarrow$ *REC*)
by(*simp add*: *REC-def*,*rule trans*, *rule fix-eq*, *simp*)

9.2.3 A Refinement Proof

Third part: No comes the proof by fixpoint induction. Not too bad in automation considering what is inferred, but wouldn't scale for large examples.

lemma *impl-refines-spec* : $COPY \sqsubseteq_{FD} SYSTEM$

apply(*simp add*: *SYSTEM-def COPY-def*)
apply(*rule fix-ind*, *simp-all*)
apply (*intro le-FD-adm*, *simp-all add*: *cont-fun monofunI*)
apply (*subst SEND-rec*, *subst REC-rec*)
by (*simp add*: *Sync-rules Hiding-rules*)

lemma *spec-refines-impl* :
assumes *fin*: *finite* (*SYN*:: *'a channel set*)
shows *SYSTEM* $\sqsubseteq_{FD}$ (*COPY* :: *'a channel process*)
apply(*simp add*: *SYSTEM-def SEND-def*)
apply(*rule fix-ind*, *simp-all*)
apply (*intro le-FD-adm*)
apply (*simp add*: *fin*)
apply (*simp add*: *cont2mono*)
apply (*simp add*: *Hiding-set-BOT Sync-BOT Sync-commute*)
apply (*subst COPY-rec*, *subst REC-rec*)
by (*simp add*: *Sync-rules Hiding-rules*)

Note that this was actually proven for the Process ordering, not the refinement ordering. But the former implies the latter. And due to anti-symmetry, equality follows for the case of finite alphabets ...

lemma *spec-equal-impl* :
assumes *fin*: *finite* (*SYN*::(*'a channel*)*set*)
shows *SYSTEM* = (*COPY*::*'a channel process*)
by (*simp add*: *FD-antisym fin impl-refines-spec spec-refines-impl*)

9.2.4 Deadlock Freeness Proof

HOL-CSP can be used to prove deadlock-freeness of processes with infinite alphabet. In the case of the *COPY* - process, this can be formulated as the following refinement problem:

lemma (*DF* (*range left* ∪ *range right*)) $\sqsubseteq_{FD}$ *COPY*
apply(*simp add*:*DF-def*,*rule fix-ind2*)
proof −
show *adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *COPY*) **by**(*rule le-FD-adm*, *simp-all add*: *monofunI*)
next
show ⊥ $\sqsubseteq_{FD}$ *COPY* **by** *fastforce*
next
have *1*: (⊓*xa*∈ *range left* ∪ *range right* → ⊥) $\sqsubseteq_{FD}$ (⊓*xa*∈ *range left* → ⊥)
by (*rule mono-Mndetprefix-FD-set*, *simp*, *blast*)
have *2*: (⊓*xa*∈ *range left* → ⊥) $\sqsubseteq_{FD}$ (*left*?*x* → ⊥)
unfolding *read-def*
by (*meson Mprefix-refines-Mndetprefix-FD BOT-leFD mono-Mndetprefix-FD trans-FD*)
show (Λ *x*. ⊓*xa*∈*range left* ∪ *range right* → *x*)·⊥ $\sqsubseteq_{FD}$ *COPY*
by *simp* (*metis* (*mono-tags*, *lifting*) *1 2 COPY-rec mono-read-FD BOT-leFD trans-FD*)

next
fix P::$'a$ *channel process*
assume $*$: $P \sqsubseteq_{FD}$ *COPY* **and** $**$: (Λ x. $\sqcap xa \in$*range left* $\cup$ *range right* $\rightarrow$ x)$\cdot P$ $\sqsubseteq_{FD}$ *COPY*
have *1*:($\sqcap xa \in$ *range left* $\cup$ *range right* $\rightarrow$ P) $\sqsubseteq_{FD}$ ($\sqcap xa \in$ *range right* $\rightarrow$ P)
by (*rule mono-Mndetprefix-FD-set*, *simp*, *blast*)
have *2*:($\sqcap xa \in$ *range right* $\rightarrow$ P) $\sqsubseteq_{FD}$ (*right*!x $\rightarrow$ P) **for** x
apply (*unfold write-def*, *rule trans-FD*[*OF mono-Mndetprefix-FD-set*[*of* ‹{*right* x}› ‹*range right*›]])
by *simp-all*
from *1 2* **have** *ab*:($\sqcap xa \in$ *range left* $\cup$ *range right* $\rightarrow$ P) $\sqsubseteq_{FD}$ (*right*!x $\rightarrow$ P) **for** x
using *trans-FD* **by** *blast*
hence *3*:(*left*?x $\rightarrow$ ($\sqcap xa \in$ *range left* $\cup$ *range right* $\rightarrow$ P)) $\sqsubseteq_{FD}$ (*left*?x $\rightarrow$(*right*!x $\rightarrow$ P)) **by** *simp*
have *4*:$\bigwedge X$. ($\sqcap xa \in$ *range left* $\cup$ *range right* $\rightarrow X$) $\sqsubseteq_{FD}$ ($\sqcap xa \in$ *range left* $\rightarrow X$)
by (*rule mono-Mndetprefix-FD-set*, *simp*, *blast*)
have *5*:$\bigwedge X$. ($\sqcap xa \in$ *range left* $\rightarrow X$) $\sqsubseteq_{FD}$ (*left*?x $\rightarrow X$)
by (*unfold read-def*, *subst K-record-comp*, *fact Mprefix-refines-Mndetprefix-FD*)
from *3 4*[*of* ($\sqcap xa \in$ *range left* $\cup$ *range right* $\rightarrow$ P)]
5 [*of* ($\sqcap xa \in$ *range left* $\cup$ *range right* $\rightarrow$ P)]
have *6*:($\sqcap xa \in$ *range left* $\cup$ *range right* $\rightarrow$
($\sqcap xa \in$ *range left* $\cup$ *range right* $\rightarrow$ P)) $\sqsubseteq_{FD}$ (*left*?x $\rightarrow$ (*right*!x $\rightarrow$ P))
using *trans-FD* **by** *blast*
from $*$ $**$ **have** *7*:*left*?x $\rightarrow$ *right*!x $\rightarrow$ $P \sqsubseteq_{FD}$ *left*?x $\rightarrow$ *right*!x $\rightarrow$ *COPY* **by** *simp*

show (Λ x. $\sqcap xa \in$*range left* $\cup$ *range right* $\rightarrow$ x)$\cdot$
((Λ x. $\sqcap xa \in$*range left* $\cup$ *range right* $\rightarrow$ x)$\cdot P$) $\sqsubseteq_{FD}$ *COPY*
by *simp* (*metis* (*mono-tags*, *lifting*) *6 7 COPY-rec trans-FD*)
qed

9.3 An Alternative Approach: Using the fixrec-Package

9.3.1 Channels and Synchronisation Sets

As before.

9.3.2 Process Definitions via fixrec-Package

fixrec
COPY$'$ :: ($'a$ *channel*) *process*
and
SEND$'$ :: ($'a$ *channel*) *process*
and
REC$'$ :: ($'a$ *channel*) *process*

where
  COPY′-rec[*simp del*]: *COPY′* = *left*?*x* → *right*!*x* → *COPY′*
| *SEND′-rec*[*simp del*]: *SEND′* = *left*?*x* → *mid*!*x* → (*ack* → *SEND′*)
| *REC′-rec*[*simp del*] : *REC′* = *mid*?*x* → *right*!*x* → (*ack* → *REC′*)

thm *COPY′-rec*
definition *SYSTEM′* :: (′*a channel*) *process*
where ‹*SYSTEM′* ≡ ((*SEND′* ⟦ *SYN* ⟧ *REC′*) \ *SYN*)›

9.3.3 Another Refinement Proof on fixrec-infrastructure

Third part: No comes the proof by fixpoint induction. Not too bad in automation considering what is inferred, but wouldn't scale for large examples.

thm *COPY′-SEND′-REC′.induct*
lemma *impl-refines-spec′* : (*COPY′*::′*a channel process*) $\sqsubseteq_{FD}$ *SYSTEM′*
 apply (*unfold SYSTEM′-def*)
 apply (*rule-tac P*=‹λ *a b c*. *a* $\sqsubseteq_{FD}$ ((*SEND′* ⟦*SYN*⟧ *REC′*) \ *SYN*)› **in** *COPY′-SEND′-REC′.induct*)
 apply (*subst case-prod-beta′*)+
 apply (*intro le-FD-adm*, *simp-all add*: *monofunI*)
 apply (*subst SEND′-rec*, *subst REC′-rec*)
 by (*simp add*: *Sync-rules Hiding-rules*)

lemma *spec-refines-impl′* :
assumes *fin*: *finite* (*SYN*::(′*a channel*)*set*)
shows *SYSTEM′* $\sqsubseteq_{FD}$ (*COPY′*::′*a channel process*)
proof(*unfold SYSTEM′-def*, *rule-tac P*=‹λ *a b c*. ((*b* ⟦*SYN*⟧ *REC′*) \ *SYN*) $\sqsubseteq_{FD}$ *COPY′*›
 in *COPY′-SEND′-REC′.induct*, *goal-cases*)
 case *1*
 have *aa*:‹*adm* (λ(*a*::′*a channel process*). ((*a* ⟦*SYN*⟧ *REC′*) \ *SYN*) $\sqsubseteq_{FD}$ *COPY′*)›
  apply (*intro le-FD-adm*)
  by (*simp-all add*: *fin cont2mono*)
 thus *?case* **using** *adm-subst*[*of* λ(*a*,*b*,*c*). *b*, *simplified*, *OF aa*] **by** (*simp add*: *split-def*)
next
 case *2*
 then show *?case* **by** (*simp add*: *Hiding-set-BOT Sync-BOT Sync-commute*)
next
 case (*3 a aa b*)
 then show *?case*
  apply (*subst COPY′-rec*, *subst REC′-rec*)
  by (*simp add*: *Sync-rules Hiding-rules*)
qed

lemma *spec-equal-impl′* :
assumes *fin*: *finite* (*SYN*::(′*a channel*)*set*)
shows *SYSTEM′* = (*COPY′*::′*a channel process*)
 apply (*rule FD-antisym*)
 apply (*rule spec-refines-impl′*[*OF fin*])

apply (*rule impl-refines-spec′*)
done

end

Chapter 10

Advanced Induction Schemata

theory *Induction-ext*

imports *HOLCF*

begin

10.1 k-fixpoint-induction

lemma *nat-k-induct* [*case-names base step*]:
  fixes k::*nat*
  assumes $\forall\, i{<}k.\ P\ i$ **and** $\forall\, n_0.\ (\forall\, i{<}k.\ P\ (n_0{+}i)) \longrightarrow P\ (n_0{+}k)$
  shows P (n::*nat*)
proof (*induct rule*: *nat-less-induct*)
  case (*1 n*)
  then show *?case*
    apply(*cases* $n < k$)
     using *assms*(*1*) **apply** *blast*
    using *assms*(*2*)[*rule-format*, *of* $n{-}k$] **by** *auto*
qed

thm *fix-ind fix-ind2*

lemma *fix-ind-k* [*case-names admissibility base-k-steps step*]:
  fixes k::*nat*
  assumes *adm*: *adm P*
  and *base-k-steps*: $\forall\, i{<}k.\ P\ (\mathit{iterate}\ i{\cdot}f{\cdot}\bot)$
  and *step*: $\bigwedge x.\ (\forall\, i{<}k.\ \ P\ (\mathit{iterate}\ i{\cdot}f{\cdot}x)) \Longrightarrow P\ (\mathit{iterate}\ k{\cdot}f{\cdot}x)$
  shows P (*fix*·*f*)
  unfolding *fix-def2* **apply** (*rule admD* [*OF adm chain-iterate*])
  apply(*rule nat-k-induct*[*of k*], *simp add*: *base-k-steps*)
  using *step* **by** (*subst* (*1 2*) *add.commute*, *unfold iterate-iterate*[*symmetric*]) *blast*

```
lemma nat-k-skip-induct [case-names lower-bound base-k step]:
  fixes k::nat
  assumes k ≥ 1 and ∀ i<k. P i and ∀ n0. P (n0) ⟶ P (n0+k)
  shows P (n::nat)
proof(induct rule:nat-less-induct)
  case (1 n)
  then show ?case
    apply(cases n < k)
     using assms(2) apply blast
    using assms(3)[rule-format, of n−k] assms(1) by auto
qed

lemma fix-ind-k-skip [case-names lower-bound admissibility base-k-steps step]:
  fixes k::nat
  assumes k-1: k ≥ 1
  and adm: adm P
  and base-k-steps: ∀ i<k. P (iterate i·f·⊥)
  and step: ⋀x. P x ⟹ P (iterate k·f·x)
  shows P (fix·f)
  unfolding fix-def2 apply (rule admD [OF adm chain-iterate])
  apply(rule nat-k-skip-induct[of k])
  using k-1 base-k-steps apply auto
  using step by (subst add.commute, unfold iterate-iterate[symmetric]) blast

thm parallel-fix-ind
```

10.2 Parallel fixpoint-induction

```
lemma parallel-fix-ind-inc[case-names admissibility base-fst base-snd step]:
  assumes adm: adm (λx. P (fst x) (snd x))
  and base-fst: ⋀y. P ⊥ y and base-snd: ⋀x. P x ⊥
  and step: ⋀x y. P x y ⟹ P (G·x) y ⟹ P x (H·y) ⟹ P (G·x) (H·y)
  shows P (fix·G) (fix·H)
proof −
  from adm have adm′: adm (case-prod P)
    unfolding split-def .
  have P (iterate i·G·⊥) (iterate j·H·⊥) for i j
  proof(induct i+j arbitrary:i j rule:nat-less-induct)
    case 1
    { fix i′ j′
      assume i:i = Suc i′ and j:j = Suc j′
      have P (iterate i′·G·⊥) (iterate j′·H·⊥)
       and P (iterate i′·G·⊥) (iterate j·H·⊥)
       and P (iterate i·G·⊥) (iterate j′·H·⊥)
        using 1.hyps add-strict-mono i j apply blast
        using 1.hyps i apply auto[1]
        using 1.hyps j by auto
```

```
      hence ?case by (simp add: i j step)
    }
    thus ?case
      apply(cases i, simp add:base-fst)
      apply(cases j, simp add:base-snd)
      by assumption
  qed
  then have ⋀i. case-prod P (iterate i·G·⊥, iterate i·H·⊥)
    by simp
  then have case-prod P (⨆i. (iterate i·G·⊥, iterate i·H·⊥))
    by − (rule admD [OF adm'], simp, assumption)
  then have P (⨆i. iterate i·G·⊥) (⨆i. iterate i·H·⊥)
    by (simp add: lub-Pair)
  then show P (fix·G) (fix·H)
    by (simp add: fix-def2)
qed

end
```

Bibliography

[1] P. Broadfoot and B. Roscoe. Tutorial on fdr and its applications. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, pages 322–322, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[2] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[3] A. J. Camilleri. A higher order logic mechanization of the csp failure-divergence semantics. In G. Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 123–150, London, 1991. Springer London.

[4] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, July 1993.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[6] Y. Isobe and M. Roggenbach. Csp-prover: a proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010.

[7] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. Uniform workbench — universelle entwicklungsumgebung für formale methoden. Technical report, Technischer Bericht 8/95, Univ. Bremen, 1995. http://www.informatik.uni-bremen.de/~uniform.

[8] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

[9] P. Noce. Noninterference security in communicating sequential processes. *Archive of Formal Proofs*, May 2014. http://isa-afp.org/entries/Noninterference_CSP.html, Formal proof development.

[10] P. Noce. Conservation of csp noninterference security under concurrent composition. *Archive of Formal Proofs*, June 2016. http://isa-afp.org/entries/Noninterference_Concurrent_Composition.html, Formal proof development.

[11] P. Noce. Conservation of csp noninterference security under sequential composition. *Archive of Formal Proofs*, Apr. 2016. http://isa-afp.org/entries/Noninterference_Sequential_Composition.html, Formal proof development.

[12] A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[13] H. Tej and B. Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME)*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337, Heidelberg, 1997. Springer-Verlag.