HOL-CSP Version 2.0

Benoît Ballenghien Safouan Taha Burkhart Wolff
Lina Ye

March 19, 2025

Contents

1 Context **9**
- 1.1 Preface . . . 9
- 1.2 Introduction . . . 10
- 1.3 An Outline of Failure-Divergence Semantics . . . 11
  - 1.3.1 Non-Determinism . . . 12
  - 1.3.2 Infinite Chatter . . . 13
  - 1.3.3 The Global Architecture of HOL-CSP 2.0 . . . 13

2 The Notion of Processes **17**
- 2.1 Pre-Requisite: Basic Traces and tick-Freeness . . . 17
- 2.2 Basic Types, Traces, Failures and Divergences . . . 22
- 2.3 The Process Type Invariant . . . 22
- 2.4 The Abstraction to the process-Type . . . 24
- 2.5 Some Consequences of the Process Characterization . . . 27
- 2.6 Process Approximation is a Partial Ordering, a Cpo, and a Pcpo . . . 28
- 2.7 Process Refinement is a Partial Ordering . . . 34
- 2.8 Process Refinement is Admissible . . . 37
- 2.9 The Conditional Statement is Continuous . . . 37
- 2.10 Tools for proving continuity . . . 37

3 Constant Processes **39**
- 3.1 The Undefined Process: $\bot$ . . . 39
- 3.2 The SKIP Process . . . 40
- 3.3 The STOP Process . . . 41

4 The Binary Choice Operators **43**
- 4.1 Deterministic Choice . . . 43
  - 4.1.1 The Det Operator Definition . . . 43
  - 4.1.2 The Projections . . . 44
  - 4.1.3 Basic Laws . . . 44
  - 4.1.4 The Continuity-Rule . . . 45
- 4.2 Non Deterministic Choice . . . 46

4.2.1 The Ndet Operator Definition . . . 46
4.2.2 The Projections . . . 47
4.2.3 Basic Laws . . . 47
4.2.4 The Continuity Rule . . . 47
4.3 Sliding Choice . . . 48
4.3.1 Definition . . . 48
4.3.2 Projections . . . 48
4.3.3 Properties . . . 49
4.3.4 Continuity . . . 49

5 The Prefix Choice Operators 51
5.1 Multiple Deterministic Prefix Choice . . . 51
5.1.1 The Definition and some Consequences . . . 51
5.1.2 Projections in Prefix . . . 52
5.1.3 Basic Properties . . . 52
5.1.4 Proof of Continuity Rule . . . 53
5.1.5 High-level Syntax for Read and Write0 . . . 55
5.1.6 CSP_M-Style Syntax for Communication Primitives . . . 55
5.2 Multiple non Deterministic Prefix Choice . . . 56
5.3 Multiple non deterministic prefix operator . . . 56
5.3.1 General case Continuity . . . 59
5.3.2 High-level Syntax for Write . . . 60

6 The Global non Deterministic Choice 63
6.1 General non Deterministic Choice Definition . . . 63
6.2 The projections . . . 64
6.3 Factorization of ($\sqcap$) in front of *GlobalNdet* . . . 65
6.4 First properties . . . 65
6.5 Behaviour of *GlobalNdet* with ($\sqcap$) . . . 66
6.6 Commutativity . . . 66
6.7 Behaviour with injectivity . . . 66
6.8 Cartesian product results . . . 66
6.9 Link with *Mndetprefix* . . . 66
6.10 Continuity . . . 67

7 The Sequential Composition 69
7.1 Definition . . . 69
7.2 The Projections . . . 71
7.3 Continuity Rule . . . 72

8 The Synchronization Product 79
8.1 Basic Concepts . . . 79
8.2 Consequences . . . 80
8.3 The Sync Operator Definition . . . 90

8.4 The Projections . . . 94
8.5 Syntax for Interleave and Parallel Operator . . . 95
8.6 Continuity Rule . . . 96

9 The Renaming Operator 103
9.1 Some preliminaries . . . 103
9.2 The Renaming Operator Definition . . . 104
9.3 The Projections . . . 107
9.4 Continuity Rule . . . 107
9.4.1 Monotonicity of *Renaming*. . . . 107
9.4.2 Some useful results about *finitary*, and preliminaries lemmas for continuity. . . . 108
9.4.3 Finally, continuity ! . . . 112
9.5 Some nice properties . . . 113

10 The Hiding Operator 115
10.1 Preliminaries : primitives and lemmas . . . 115
10.2 The Hiding Operator Definition . . . 117
10.3 Projections . . . 121
10.4 Continuity Rule . . . 122

11 The CSP Refinements 141
11.1 Definitions and first Properties . . . 141
11.1.1 Definitions . . . 141
11.1.2 Idempotency . . . 142
11.1.3 Some obvious refinements . . . 142
11.1.4 Antisymmetry . . . 142
11.1.5 Transitivity . . . 142
11.1.6 Relations between refinements . . . 143
11.1.7 More obvious refinements . . . 143
11.1.8 Admissibility . . . 143
11.2 Monotonies . . . 144
11.2.1 Straight Monotony . . . 144
11.2.2 Monotony Properties . . . 155

12 Algebraic Rules of CSP 159
12.1 The Non-Deterministic Distributivity . . . 159
12.1.1 Global Distributivity . . . 159
12.1.2 Binary Distributivity . . . 160
12.2 The Basic Laws . . . 163
12.2.1 The Laws of $\perp$ . . . 163
12.2.2 The Laws of *STOP* . . . 165
12.2.3 The Laws of *SKIP* . . . 169
12.3 The Step-Laws . . . 185

12.3.1 Deterministic Choice . . . 185
12.3.2 Non-Deterministic Choice . . . 185
12.3.3 Sliding Choice . . . 186
12.3.4 Sequential Composition . . . 187
12.3.5 Hiding . . . 187
12.3.6 Synchronization . . . 197
12.4 Extension of the Step-Laws . . . 205
12.4.1 Derived step-laws for *Sync* . . . 205
12.4.2 Non deterministic step-laws . . . 210
12.5 Read and Write Laws . . . 217
12.5.1 Projections . . . 217
12.5.2 Equality with Constant Process . . . 219
12.5.3 Extensions of Step-Laws . . . 220
12.5.4 *Sync*, *SKIP* and *STOP* . . . 252
12.6 Powerful Laws of CSP . . . 255
12.6.1 Laws for Mndetprefix and Sync . . . 255
12.6.2 Hiding Operator Laws . . . 256
12.6.3 Sync Operator Laws . . . 265
12.6.4 Renaming Operator Laws . . . 278

13 Events and Ticks of a Process 287
13.1 Definitions . . . 287
13.1.1 Events of a Process . . . 287
13.1.2 Ticks of a Process . . . 288
13.2 Laws . . . 289
13.2.1 Preliminaries . . . 289
13.2.2 Events of a Process . . . 292
13.2.3 Strict Events of a Process . . . 295
13.3 Ticks of a Process . . . 299
13.3.1 Strict Events of a Process . . . 303

14 CSP Assertions 307
14.1 Reference Processes . . . 307
14.2 Assertions . . . 308
14.3 Properties . . . 308
14.4 Events and Ticks of Reference Processes . . . 310
14.5 Relations between refinements on reference processes . . . 314
14.6 *deadlock-free* and *deadlock-free*$_{SKIPS}$ with *SKIP* and *STOP* 321

15 Advanced Induction Schemata 325
15.1 k-fixpoint-induction . . . 325
15.2 Parallel fixpoint-induction . . . 326

16 The Main Entry Point 329

17 Conclusion **331**
17.1 Related Work 331
17.2 Lessons learned 331
17.3 A Summary on New Results 332

18 Annex: Refinement Example with Buffer over infinite Alphabet **335**
18.1 Defining the Copy-Buffer Example 335
18.2 The Standard Proof 335
18.2.1 Channels and Synchronization Sets 335
18.2.2 Definitions by Recursors 336
18.2.3 Various Samples of Refinement Proofs 336
18.2.4 Deadlock Freeness Proof 338
18.3 An Alternative Approach: Using the fixrec-Package 339
18.3.1 Channels and Synchronisation Sets 339
18.3.2 Process Definitions via fixrec-Package 339
18.3.3 Another Refinement Proof on fixrec-infrastructure . . 340

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 Brookes "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.

Actually, from the Isabelle-2025 version on, the theory has been extended in two ways:

- new operators known from Roscoe's books had been formally integrated (generalized non deterministic choice, sliding choice, etc.)
- the classical constant tick (✓) of the CSP theory has been replaced by a parameterized version carrying a kind of return value. It turns out that this is a very natural extension of the original setting. Generalizations of the sequential composition and synchronization product that fully enjoy this feature will be added in future versions.

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 paradigmatic 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 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

$$(a \rightarrow STOP) \,\Box\, b \rightarrow STOP \tag{1}$$

and

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

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

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

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. For this reason, the refinement notations $P \sqsubseteq_{FD} P'$, $P \sqsubseteq_{F} P'$, $P \sqsubseteq_{D} P'$, $P \sqsubseteq_{T} P'$, $P \sqsubseteq_{DT} P'$ have been introduced to be notationally closer to Roscoe's book.

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 preservation of `is_process` for this operator). In this new version, the proof is required immediately after the definition of the operator since we use **lift-definition** instead of definition
- The proof of the projections $\mathcal{T}$ and $\mathcal{F}$ and $\mathcal{D}$ for this operator
- The proof of continuity of the operator (which is always possible except for Hiding if applied to infinite hide-sets).

Finally, the theory CSP not only contains the "Laws" of CSP, i.e. the derived algebraic equalities, but also monotonicities rules, allowing to reason abstractly over CSP processes. The overall dependency graph is shown in Figure 1.1.

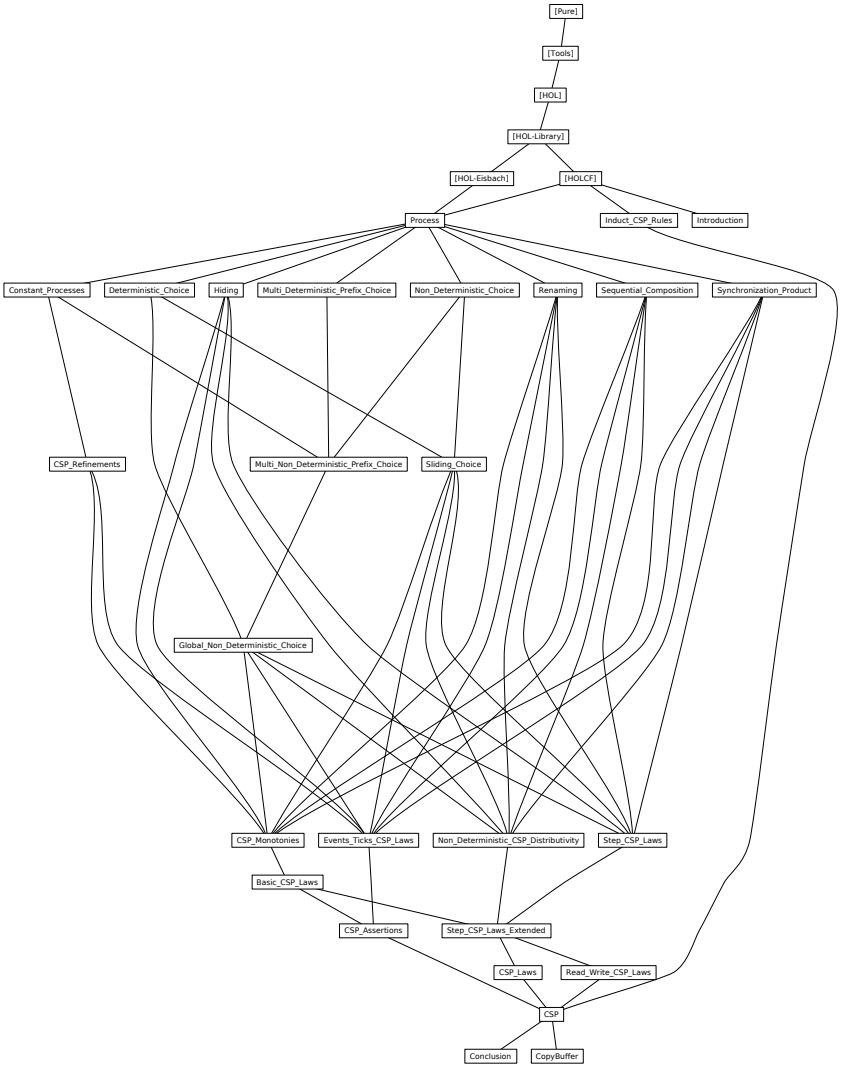

Figure 1.1: The HOL-CSP 2.0 Theory Graph

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.

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.)

From the Isabelle-2025 version on, the classical constant tick (✓) of the CSP theory has been replaced by a parameterized version carrying a kind of return value.

datatype $('a, 'r)\ event_{ptick} =$
  is-ev : *ev* (*of-ev* : $'a$)
| *is-tick* : *tick* (*of-tick* : $'r$) (‹✓$'$(-$'$)›)

This type $('a, 'r)\ event_{ptick}$ is of course isomorphic to the sum type $'a + 'r$.

"ptick" stands for parameterized tick, and we introduce the type synonym for the classical process event type.

type-synonym $'a\ event = ‹('a, unit)\ event_{ptick}›$

abbreviation *tick-unit* :: ‹'*a event*› (‹✓›) **where** ‹✓ ≡ ✓(())›

definition *sum-of-event*$_{ptick}$ :: ‹('*a*, '*r*) *event*$_{ptick}$ ⇒ '*a* + '*r*›
where ‹*sum-of-event*$_{ptick}$ *e* ≡ *case e of ev a* ⇒ *Inl a* | ✓(*r*) ⇒ *Inr r*›

definition *event*$_{ptick}$*-of-sum* :: ‹'*a* + '*r* ⇒ ('*a*, '*r*) *event*$_{ptick}$›
where ‹*event*$_{ptick}$*-of-sum s* ≡ *case s of Inl a* ⇒ *ev a* | *Inr r* ⇒ ✓(*r*)›

lemma *type-definition-event*$_{ptick}$: ‹*type-definition sum-of-event*$_{ptick}$ *event*$_{ptick}$*-of-sum UNIV*›
proof *unfold-locales*
show ‹*sum-of-event*$_{ptick}$ *s* ∈ *UNIV*› **for** *s* :: ‹('*a*, '*r*) *event*$_{ptick}$› **by** *simp*
next
show ‹*event*$_{ptick}$*-of-sum* (*sum-of-event*$_{ptick}$ *e*) = *e*› **for** *e* :: ‹('*a*, '*r*) *event*$_{ptick}$›
by (*cases e*) (*simp-all add*: *event*$_{ptick}$*-of-sum-def sum-of-event*$_{ptick}$*-def*)
next
show ‹*sum-of-event*$_{ptick}$ (*event*$_{ptick}$*-of-sum s*) = *s*› **for** *s* :: ‹'*a* + '*r*›
by (*cases s*) (*simp-all add*: *event*$_{ptick}$*-of-sum-def sum-of-event*$_{ptick}$*-def*)
qed

setup-lifting *type-definition-event*$_{ptick}$

lemma *range-tick-Un-range-ev-is-UNIV* [*simp*] : ‹*range tick* ∪ *range ev* = *UNIV*›
by (*metis UNIV-eq-I UnCI event*$_{ptick}$*.exhaust rangeI*)

The generalization is done in a very straightforward way: the old version is recovered by considering ('*a*, *unit*) *event*$_{ptick}$.

lemma *not-is-ev* [*simp*] : ‹¬ *is-ev* *e* ⟷ *is-tick e*›
and *not-is-tick* [*simp*] : ‹¬ *is-tick e* ⟷ *is-ev* *e›
by (*use event*$_{ptick}$*.exhaust-disc* **in** *blast*)+

type-synonym ('*a*, '*r*) *trace*$_{ptick}$ = ‹('*a*, '*r*) *event*$_{ptick}$ *list*›

We recover the classical version with *unit*.

type-synonym '*a trace* = ‹('*a*, *unit*) *trace*$_{ptick}$›

We chose as standard ordering on traces the prefix ordering.

Some facts on the prefix ordering.

lemma *nil-le* [*simp*]: ‹[] ≤ *s*›
and *nil-le2* [*simp*]: ‹*s* ≤ [] ⟷ *s* = []›
and *nil-less* [*simp*]: ‹¬ *t* < []›
and *nil-less2* [*simp*]: ‹[] < *t* @ [*a*]›
and *less-self* [*simp*]: ‹*t* < *t* @ [*a*]›
and *le-cons* [*simp*]: ‹*a* # *s* ≤ *a* # *t* ⟷ *s* ≤ *t*›
and *le-append* [*simp*]: ‹*b* @ *s* ≤ *b* @ *t* ⟷ *s* ≤ *t*›
and *less-cons* [*simp*]: ‹*a* # *s* < *a* # *t* ⟷ *s* < *t*›
and *less-append*[*simp*]: ‹*b* @ *s* < *b* @ *t* ⟷ *s* < *t*›

and *le-length-mono*: ‹$s \leq t \Longrightarrow length\ s \leq length\ t$›
and *less-length-mono*: ‹$s < t \Longrightarrow length\ s < length\ t$›
and *le-tail*: ‹$s \neq [] \Longrightarrow s \leq t \Longrightarrow tl\ s \leq tl\ t$›
and *less-tail*: ‹$s \neq [] \Longrightarrow s < t \Longrightarrow tl\ s < tl\ t$›
apply (*simp-all add*: *less-eq-list-def less-list-def prefix-length-le*)
apply (*metis prefix-length-less prefix-order.dual-order.not-eq-order-implies-strict*)
apply (*metis prefix-def tl-append2*)
by (*metis prefix-def prefix-order.eq-iff self-append-conv tl-append2*)

lemma *le-same-imp-eq-or-less*: ‹$(s :: 'a\ list) \leq u \Longrightarrow t \leq u \Longrightarrow t = s \vee s < t \vee t < s$›
by (*metis less-eq-list-def linorder-le-cases nless-le prefix-length-prefix*)

lemma *append-eq-first-pref-spec*: ‹$s\ @\ t = r\ @\ [x] \Longrightarrow t \neq [] \Longrightarrow s \leq r$›
by (*metis butlast-append butlast-snoc less-eq-list-def prefix-def*)

lemma *prefixes-fin*: ‹*finite* $\{t.\ t \leq s\} \wedge$ *card* $\{t.\ t \leq s\} =$ *Suc* (*length* s)›
proof (*induct s*)
show ‹*finite* $\{t.\ t \leq []\} \wedge$ *card* $\{t.\ t \leq []\} =$ *Suc* (*length* $[]$)› **by** *simp*
next
case (*Cons x s*)
have $*$: ‹$\{t.\ t \leq x\ \#\ s\} = \{[]\} \cup (\lambda t.\ x\ \#\ t)$ ‘ $\{t.\ t \leq s\}$›
by (*simp add*: *image-def less-eq-list-def set-eq-iff*)
(*meson Sublist.prefix-Cons*)
show ‹*finite* $\{t.\ t \leq x\ \#\ s\} \wedge$ *card* $\{t.\ t \leq x\ \#\ s\} =$ *Suc* (*length* $(x\ \#\ s)$)›
proof (*intro conjI*)
show ‹*finite* $\{t.\ t \leq x\ \#\ s\}$› **by** (*simp add*: $*$ *Cons.hyps*)
next
have ‹*finite* $((\lambda t.\ x\ \#\ t)$ ‘ $\{t.\ t \leq s\})$› **by** (*simp add*: *Cons.hyps*)
show ‹*card* $\{t.\ t \leq x\ \#\ s\} =$ *Suc* (*length* $(x\ \#\ s)$)›
by (*subst card-Un-disjoint*[*of* ‹$\{[]\}$› ‹$(\lambda t.\ x\ \#\ t)$ ‘ $\{t.\ t \leq s\}$›, *folded* $*$])
(*auto simp add*: *card-image Cons.hyps*)
qed
qed

lemma *sublists-fin*: ‹*finite* $\{t.\ \exists\ t1\ t2.\ s = t1\ @\ t\ @\ t2\}$›
proof (*induct s*)
show ‹*finite* $\{t.\ \exists\ t1\ t2.\ [] = t1\ @\ t\ @\ t2\}$› **by** *simp*
next
case (*Cons x s*)
have ‹$\{t.\ t \leq x\ \#\ s\} = \{t.\ \exists\ t2.\ x\ \#\ s = t\ @\ t2\}$›
by (*simp add*: *less-eq-list-def prefix-def*)
with *prefixes-fin*[*of* ‹$x\ \#\ s$›] **have** ‹*finite* $\{t.\ \exists\ t2.\ x\ \#\ s = t\ @\ t2\}$› **by** *simp*
have ‹$\{t.\ \exists\ t1\ t2.\ x\ \#\ s = t1\ @\ t\ @\ t2\} \subseteq$

```
      {t. ∃ t1 t2. s = t1 @ t @ t2} ∪ {t. ∃ t2. x # s = t @ t2}›
   by (simp add: subset-iff) (meson Cons-eq-append-conv)
 show ‹finite {t. ∃ t1 t2. x # s = t1 @ t @ t2}›
   by (rule finite-subset[OF ‹?this›], rule finite-UnI)
     (simp-all add: Cons.hyps ‹finite {t. ∃ t2. x # s = t @ t2}›)
qed


lemma suffixes-fin: ‹finite {t. ∃ t1. s = t1 @ t}›
 by (rule finite-subset[of - ‹{t. ∃ t1 t2. s = t1 @ t @ t2}›];
     simp add: subset-iff sublists-fin) 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 :: ‹('a, 'r) trace_ptick ⇒ bool› (‹tF›)
 where ‹tF s ≡ range tick ∩ set s = {}›

definition front-tickFree :: ‹('a, 'r) trace_ptick ⇒ bool› (‹ftF›)
 where ‹ftF s ≡ s = [] ∨ tickFree (tl (rev s))›

lemma tickFree-Nil         [simp] : ‹tF []›
 and tickFree-Cons-iff     [simp] : ‹tF (a # t) ⟷ is-ev a ∧ tF t›
 and tickFree-append-iff   [simp] : ‹tF (s @ t) ⟷ tF s    ∧ tF t›
 and tickFree-rev-iff      [simp] : ‹tF (rev t) ⟷ tF t›
 and non-tickFree-tick     [simp] : ‹¬ tF [✓(r)]›
 by (cases a; auto simp add: tickFree-def)+

lemma tickFree-iff-is-map-ev : ‹tF t ⟷ (∃ u. t = map ev u)›
 by (induct t) (simp-all add: Cons-eq-map-conv is-ev-def)

lemma front-tickFree-Nil   [simp] : ‹ftF []›
 and front-tickFree-single[simp] : ‹ftF [a]›
 by (simp-all add: front-tickFree-def)


lemma tickFree-tl : ‹tF s ⟹ tF (tl s)›
 by (cases s) simp-all

lemma non-tickFree-imp-not-Nil: ‹¬ tF s ⟹ s ≠ []›
 using tickFree-Nil by blast

lemma tickFree-butlast: ‹tF s ⟷ tF (butlast s) ∧ (s ≠ [] ⟶ is-ev (last s))›
 by (induct s) simp-all

lemma front-tickFree-iff-tickFree-butlast: ‹ftF s ⟷ tF (butlast s)›
 by (induct s) (auto simp add: front-tickFree-def)
```

lemma *front-tickFree-Cons-iff*: ‹*ftF* (*a* # *s*) ⟷ *s* = [] ∨ *is-ev a* ∧ *ftF s*›
by (*simp add*: *front-tickFree-iff-tickFree-butlast*)

lemma *front-tickFree-append-iff*:
‹*ftF* (*s* @ *t*) ⟷ (*if t* = [] *then ftF s else tF s* ∧ *ftF t*)›
by (*simp add*: *butlast-append front-tickFree-iff-tickFree-butlast*)

lemma *tickFree-imp-front-tickFree* [*simp*] : ‹*tF s* ⟹ *ftF s*›
by (*simp add*: *front-tickFree-def tickFree-tl*)

lemma *front-tickFree-charn*: ‹*ftF s* ⟷ *s* = [] ∨ (∃ *a t*. *s* = *t* @ [*a*] ∧ *tF t*)›
by (*cases s rule*: *rev-cases*) (*simp-all add*: *front-tickFree-def*)

lemma *nonTickFree-n-frontTickFree*: ‹¬ *tF s* ⟹ *ftF s* ⟹ ∃ *t r*. *s* = *t* @ [✓(*r*)]›
by (*metis event*$_{ptick}$.*disc*(*1*) *event*$_{ptick}$.*exhaust front-tickFree-append-iff list.distinct*(*1*)
rev-exhaust tickFree-Cons-iff tickFree-Nil tickFree-append-iff)

lemma *front-tickFree-dw-closed* : ‹*ftF* (*s* @ *t*) ⟹ *ftF s*›
by (*metis front-tickFree-append-iff tickFree-imp-front-tickFree*)

lemma *front-tickFree-append*: ‹*tF s* ⟹ *ftF t* ⟹ *ftF* (*s* @ *t*)›
by (*simp add*: *front-tickFree-append-iff*)

lemma *tickFree-imp-front-tickFree-snoc*: ‹*tF s* ⟹ *ftF* (*s* @ [*a*])›
by (*simp add*: *front-tickFree-append*)

lemma *front-tickFree-nonempty-append-imp*: ‹*ftF* (*t* @ *r*) ⟹ *r* ≠ [] ⟹ *tF t* ∧
ftF r›
by (*simp add*: *front-tickFree-append-iff*)

lemma *tickFree-map-ev* [*simp*] : ‹*tF* (*map ev t*)›
by (*induct t*) *simp-all*

lemma *tickFree-map-tick-iff* [*simp*] : ‹*tF* (*map tick t*) ⟷ *t* = []›
by (*induct t*) *simp-all*

lemma *front-tickFree-map-tick-iff* [*simp*] : ‹*ftF* (*map tick t*) ⟷ *t* = [] ∨ (∃ *r*. *t*
= [*r*])›
by (*simp add*: *front-tickFree-iff-tickFree-butlast map-butlast*[*symmetric*])
(*metis append-Nil append-butlast-last-id butlast.simps*(*1*, *2*))

— *map ev* (*map f t*) if automatically simplified into *map* (*ev* ∘ *f*) *t* by the simplified, so we need to add the following versions.

lemma *tickFree-map-ev-comp* [*simp*] : ‹*tF* (*map* (*ev* ∘ *f*) *t*)›
by (*metis list.map-comp tickFree-map-ev*)

lemma *tickFree-map-tick-comp-iff* [*simp*] : ‹*tF* (*map* (*tick* ∘ *f*) *t*) ⟷ *t* = []›

by (*fold map-map*, *unfold tickFree-map-tick-iff*) *simp*

lemma *front-tickFree-map-tick-comp-iff* [*simp*] : ‹*ftF* (*map* (*tick* ∘ *f*) *t*) ⟷ *t* = [] ∨ (∃ *r*. *t* = [*r*])›
by (*fold map-map*, *unfold front-tickFree-map-tick-iff*)
(*simp add*: *map-eq-Cons-conv*)

2.2 Basic Types, Traces, Failures and Divergences

type-synonym ('*a*, '*r*) *refusal*$_{ptick}$ = ‹('*a*, '*r*) *event*$_{ptick}$ *set*›
type-synonym '*a refusal* = ‹('*a*, *unit*) *refusal*$_{ptick}$›
type-synonym ('*a*, '*r*) *failure*$_{ptick}$ = ‹('*a*, '*r*) *trace*$_{ptick}$ × ('*a*, '*r*) *refusal*$_{ptick}$›
type-synonym '*a failure* = ‹('*a*, *unit*) *failure*$_{ptick}$›
type-synonym ('*a*, '*r*) *divergence*$_{ptick}$ = ‹('*a*, '*r*) *trace*$_{ptick}$›
type-synonym '*a divergence* = ‹('*a*, *unit*) *divergence*$_{ptick}$›
type-synonym ('*a*, '*r*) *process*$_0$ = ‹('*a*, '*r*) *failure*$_{ptick}$ *set* × ('*a*, '*r*) *divergence*$_{ptick}$ *set*›

definition *FAILURES* :: ‹('*a*, '*r*) *process*$_0$ ⇒ ('*a*, '*r*) *failure*$_{ptick}$ *set*›
where ‹*FAILURES P* ≡ *fst P*›

definition *TRACES* :: ‹('*a*, '*r*) *process*$_0$ ⇒ ('*a*, '*r*) *trace*$_{ptick}$ *set*›
where ‹*TRACES P* ≡ {*tr*. ∃ *ref*. (*tr*, *ref*) ∈ *FAILURES P*}›

definition *DIVERGENCES* :: ‹('*a*, '*r*) *process*$_0$ ⇒ ('*a*, '*r*) *divergence*$_{ptick}$ *set*›
where ‹*DIVERGENCES P* ≡ *snd P*›

definition *REFUSALS* :: ‹('*a*, '*r*) *process*$_0$ ⇒ ('*a*, '*r*) *refusal*$_{ptick}$ *set*›
where ‹*REFUSALS P* ≡ {*ref*. ([], *ref*) ∈ *FAILURES P*}›

2.3 The Process Type Invariant

definition *is-process* :: ‹('*a*, '*r*) *process*$_0$ ⇒ *bool*› **where**
‹*is-process P* ≡
([], {}) ∈ *FAILURES P* ∧
(∀ *s X*. (*s*, *X*) ∈ *FAILURES P* ⟶ *ftF s*) ∧
(∀ *s t*. (*s* @ *t*, {}) ∈ *FAILURES P* ⟶ (*s*, {}) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*, *Y*) ∈ *FAILURES P* ∧ *X* ⊆ *Y* ⟶ (*s*, *X*) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*, *X*) ∈ *FAILURES P* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *FAILURES P*)
⟶ (*s*, *X* ∪ *Y*) ∈ *FAILURES P*) ∧
(∀ *s r X*. (*s* @ [✓(*r*)], {}) ∈ *FAILURES P* ⟶ (*s*, *X* − {✓(*r*)}) ∈ *FAILURES P*) ∧
(∀ *s t*. *s* ∈ *DIVERGENCES P* ∧ *tF s* ∧ *ftF t* ⟶ *s* @ *t* ∈ *DIVERGENCES P*) ∧
(∀ *s X*. *s* ∈ *DIVERGENCES P* ⟶ (*s*, *X*) ∈ *FAILURES P*) ∧
(∀ *s r*. *s* @ [✓(*r*)] ∈ *DIVERGENCES P* ⟶ *s* ∈ *DIVERGENCES P*)›

lemma *is-process-spec*:
‹*is-process P* ⟷
([], {}) ∈ *FAILURES P* ∧
(∀ *s X*. (*s*, *X*) ∈ *FAILURES P* ⟶ *ftF s*) ∧
(∀ *s t*. (*s* @ *t*, {}) ∉ *FAILURES P* ∨ (*s*, {}) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*, *Y*) ∉ *FAILURES P* ∨ ¬ *X* ⊆ *Y* ∨ (*s*, *X*) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*, *X*) ∈ *FAILURES P* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *FAILURES P*)
⟶ (*s*, *X* ∪ *Y*) ∈ *FAILURES P*) ∧
(∀ *s r X*. (*s* @ [✓(*r*)], {}) ∈ *FAILURES P* ⟶ (*s*, *X* − {✓(*r*)}) ∈ *FAILURES P*) ∧
(∀ *s t*. *s* ∉ *DIVERGENCES P* ∨ ¬ *tF s* ∨ ¬ *ftF t* ∨ *s* @ *t* ∈ *DIVERGENCES P*) ∧
(∀ *s X*. *s* ∉ *DIVERGENCES P* ∨ (*s*, *X*) ∈ *FAILURES P*) ∧
(∀ *s r*. *s* @ [✓(*r*)] ∉ *DIVERGENCES P* ∨ *s* ∈ *DIVERGENCES P*)›
by (*simp only*: *is-process-def HOL.nnf-simps*(*1*)
HOL.nnf-simps(*3*) [*symmetric*] *HOL.imp-conjL*[*symmetric*])

lemma *Process-eqI* :
‹*FAILURES P* = *FAILURES Q* ⟹ *DIVERGENCES P* = *DIVERGENCES Q* ⟹ *P* = *Q*›
by (*metis DIVERGENCES-def FAILURES-def prod-eq-iff*)

lemma *process-eq-spec*:
‹*P* = *Q* ⟷ *FAILURES P* = *FAILURES Q* ∧ *DIVERGENCES P* = *DIVERGENCES Q*›
by (*meson Process-eqI*)

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* ⟹ *TRACES P* = *TRACES Q*›
by (*auto simp*: *FAILURES-def DIVERGENCES-def TRACES-def*)

lemma *is-process1* : ‹([], {}) ∈ *FAILURES P*›
and *is-process2* : ‹(*s*, *X*) ∈ *FAILURES P* ⟹ *ftF s*›
and *is-process3* : ‹(*s* @ *t*, {}) ∈ *FAILURES P* ⟹ (*s*, {}) ∈ *FAILURES P*›
and *is-process4* : ‹⟦*is-process P*; (*s*, *Y*) ∈ *FAILURES P*; *X* ⊆ *Y*⟧ ⟹ (*s*, *X*) ∈ *FAILURES P*›
and *is-process5* : ‹⟦*is-process P*; (*s*, *X*) ∈ *FAILURES P*; ∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *FAILURES P*⟧
⟹ (*s*, *X* ∪ *Y*) ∈ *FAILURES P*›
and *is-process6* : ‹(*s* @ [✓(*r*)], {}) ∈ *FAILURES P* ⟹ (*s*, *X* − {✓(*r*)}) ∈ *FAILURES P*›

and *is-process7* : ‹⟦*s* ∈ *DIVERGENCES P*; *tF s*; *ftF t*⟧ ⟹ *s* @ *t* ∈ *DIVERGENCES P›*
and *is-process8* : ‹*s* ∈ *DIVERGENCES P* ⟹ (*s*, *X*) ∈ *FAILURES P›*
and *is-process9* : ‹*s* @ [✓(*r*)] ∈ *DIVERGENCES P* ⟹ *s* ∈ *DIVERGENCES P›*
if ‹*is-process P›*
using ‹*is-process P›* **unfolding** *is-process-def* **by** *metis+*

lemma *trace-with-Tick-imp-tickFree-front* :
‹*is-process P* ⟹ *t* @ [✓(*r*)] ∈ *TRACES P* ⟹ *tF t›*
by (*simp add*: *TRACES-def*) (*meson front-tickFree-append-iff is-process2 neq-Nil-conv*)

2.4 The Abstraction to the process-Type

typedef ('*a*, '*r*) $process_{ptick}$ = ‹{*p* :: ('*a*, '*r*) $process_0$. *is-process p*}›
morphisms $process_0$*-of-process process-of-process*$_0$
proof −
have ‹({(*s*, *X*). *s* = []}, {}) ∈ {*p*. *is-process p*}›
by (*simp add*: *DIVERGENCES-def FAILURES-def is-process-def*)
thus *?thesis* **by** *auto*
qed

Again, the old version without parameterized termination can be recovered by considering ('*a*, *unit*) $process_{ptick}$.

type-synonym '*a process* = ‹('*a*, *unit*) $process_{ptick}$›

setup-lifting *type-definition-process*$_{ptick}$

This is where we differ from previous versions: we lift definitions using Isabelle's machinery instead of doing it by hand.

lift-definition *Failures* :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $failure_{ptick}$ *set›* (‹$\mathcal{F}$›) **is** *FAILURES* **.**

lift-definition *Traces* :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $trace_{ptick}$ *set›* (‹$\mathcal{T}$›) **is** *TRACES* **.**

lift-definition *Divergences* :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $divergence_{ptick}$ *set›* (‹$\mathcal{D}$›) **is** *DIVERGENCES* **.**

lift-definition *Refusals* :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $refusal_{ptick}$ *set›* (‹$\mathcal{R}$›) **is** *REFUSALS* **.**

lemma *Refusals-def-bis* : ‹$\mathcal{R}$ *P* = {*X*. ([], *X*) ∈ $\mathcal{F}$ *P*}›
by (*simp add*: *Failures.rep-eq REFUSALS-def Refusals.rep-eq*)

lemma *Refusals-iff* : ‹*X* ∈ $\mathcal{R}$ *P* ⟷ ([], *X*) ∈ $\mathcal{F}$ *P*›

by (*simp add*: *Failures-def Refusals-def-bis*)

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

lemma *T-F-spec* : ‹$(t, \{\}) \in \mathcal{F}\ P \longleftrightarrow t \in \mathcal{T}\ P$›
by *transfer* (*auto simp add*: *TRACES-def intro*: *is-process4*)

lemma *Process-spec*: ‹*process-of-process*$_0$ $(\mathcal{F}\ P, \mathcal{D}\ P) = P$›
by (*simp add*: *Divergences.rep-eq Failures.rep-eq*
process$_0$*-of-process-inverse process-surj-pair*)

lemma *Process-eq-spec*: ‹$P = Q \longleftrightarrow \mathcal{F}\ P = \mathcal{F}\ Q \wedge \mathcal{D}\ P = \mathcal{D}\ Q$›
by (*metis Process-spec*)

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

lemma *is-processT*:
‹$([], \{\}) \in \mathcal{F}\ P \wedge$
$(\forall s\ X.\ (s, X) \in \mathcal{F}\ P \longrightarrow ftF\ s) \wedge$
$(\forall s\ t.\ (s\ @\ t, \{\}) \in \mathcal{F}\ P \longrightarrow (s, \{\}) \in \mathcal{F}\ P) \wedge$
$(\forall s\ X\ Y.\ (s, Y) \in \mathcal{F}\ P \wedge X \subseteq Y \longrightarrow (s, X) \in \mathcal{F}\ P) \wedge$
$(\forall s\ X\ Y.\ (s, X) \in \mathcal{F}\ P \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \notin \mathcal{F}\ P) \longrightarrow (s, X \cup Y) \in \mathcal{F}\ P) \wedge$
$(\forall s\ r\ X.\ (s\ @\ [\checkmark(r)], \{\}) \in \mathcal{F}\ P \longrightarrow (s, X - \{\checkmark(r)\}) \in \mathcal{F}\ P) \wedge$
$(\forall s\ t.\ s \in \mathcal{D}\ P \wedge tF\ s \wedge ftF\ t \longrightarrow s\ @\ t \in \mathcal{D}\ P) \wedge$
$(\forall s\ r\ X.\ s \in \mathcal{D}\ P \longrightarrow (s, X) \in \mathcal{F}\ P) \wedge (\forall s.\ s\ @\ [\checkmark(r)] \in \mathcal{D}\ P \longrightarrow s \in \mathcal{D}\ P)$›
by *transfer* (*unfold is-process-def*, *fast*)

When the second type is set to *unit*, we recover the classical definition as defined in the book by Roscoe.

lemma *is-processT-unit*:
‹$([], \{\}) \in \mathcal{F}\ P \wedge$
$(\forall s\ X.\ (s, X) \in \mathcal{F}\ P \longrightarrow ftF\ s) \wedge$
$(\forall s\ t.\ (s\ @\ t, \{\}) \in \mathcal{F}\ P \longrightarrow (s, \{\}) \in \mathcal{F}\ P) \wedge$
$(\forall s\ X\ Y.\ (s, Y) \in \mathcal{F}\ P \wedge X \subseteq Y \longrightarrow (s, X) \in \mathcal{F}\ P) \wedge$
$(\forall s\ X\ Y.\ (s, X) \in \mathcal{F}\ P \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \notin \mathcal{F}\ P) \longrightarrow (s, X \cup Y) \in \mathcal{F}\ P) \wedge$
$(\forall s\ X.\ (s\ @\ [\checkmark], \{\}) \in \mathcal{F}\ P \longrightarrow (s, X - \{\checkmark\}) \in \mathcal{F}\ P) \wedge$
$(\forall s\ t.\ s \in \mathcal{D}\ P \wedge tF\ s \wedge ftF\ t \longrightarrow s\ @\ t \in \mathcal{D}\ P) \wedge$
$(\forall s\ X.\ s \in \mathcal{D}\ P \longrightarrow (s, X) \in \mathcal{F}\ P) \wedge (\forall s.\ s\ @\ [\checkmark] \in \mathcal{D}\ P \longrightarrow s \in \mathcal{D}\ P)$›
by *transfer* (*unfold is-process-def*, *fast*)

lemma *process-charn*:
‹([], {}) ∈ $\mathcal{F}$ P ∧
(∀ s X. (s, X) ∈ $\mathcal{F}$ P ⟶ *ftF* s) ∧
(∀ s t. (s @ t, {}) ∉ $\mathcal{F}$ P ∨ (s, {}) ∈ $\mathcal{F}$ P) ∧
(∀ s X Y. (s, Y) ∉ $\mathcal{F}$ P ∨ ¬ X ⊆ Y ∨ (s, X) ∈ $\mathcal{F}$ P) ∧
(∀ s X Y. (s, X) ∈ $\mathcal{F}$ P ∧ (∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ $\mathcal{F}$ P) ⟶ (s, X ∪ Y) ∈ $\mathcal{F}$ P) ∧
(∀ s r X. (s @ [✓(r)], {}) ∈ $\mathcal{F}$ P ⟶ (s, X − {✓(r)}) ∈ $\mathcal{F}$ P) ∧
(∀ s t. s ∉ $\mathcal{D}$ P ∨ ¬ *tF* s ∨ ¬ *ftF* t ∨ s @ t ∈ $\mathcal{D}$ P) ∧
(∀ s r X. s ∉ $\mathcal{D}$ P ∨ (s, X) ∈ $\mathcal{F}$ P) ∧ (∀ s. s @ [✓(r)] ∉ $\mathcal{D}$ P ∨ s ∈ $\mathcal{D}$ P)›
by (*meson is-processT*)

split of `is_processT`:

lemma *is-processT1* : ‹([], {}) ∈ $\mathcal{F}$ P›
and *is-processT1-TR* : ‹[] ∈ $\mathcal{T}$ P›
and *is-processT2* : ‹(s, X) ∈ $\mathcal{F}$ P ⟹ *ftF* s›
and *is-processT2-TR* : ‹s ∈ $\mathcal{T}$ P ⟹ *ftF* s›
and *is-processT3* : ‹(s @ t, {}) ∈ $\mathcal{F}$ P ⟹ (s, {}) ∈ $\mathcal{F}$ P›
and *is-processT3-pref* : ‹(t, {}) ∈ $\mathcal{F}$ P ⟹ s ≤ t ⟹ (s, {}) ∈ $\mathcal{F}$ P›
and *is-processT3-TR* : ‹t ∈ $\mathcal{T}$ P ⟹ s ≤ t ⟹ s ∈ $\mathcal{T}$ P›
and *is-processT3-TR-pref* : ‹(t, {}) ∈ $\mathcal{F}$ P ⟹ s ≤ t ⟹ (s, {}) ∈ $\mathcal{F}$ P›
and *is-processT4* : ‹(s, Y) ∈ $\mathcal{F}$ P ⟹ X ⊆ Y ⟹ (s, X) ∈ $\mathcal{F}$ P›
and *is-processT5* : ‹(s, X) ∈ $\mathcal{F}$ P ⟹ ∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ $\mathcal{F}$ P ⟹ (s, X ∪ Y) ∈ $\mathcal{F}$ P›
and *is-processT6* : ‹(s @ [✓(r)], {}) ∈ $\mathcal{F}$ P ⟹ (s, X − {✓(r)}) ∈ $\mathcal{F}$ P›
and *is-processT6-TR* : ‹s @ [✓(r)] ∈ $\mathcal{T}$ P ⟹ (s, X − {✓(r)}) ∈ $\mathcal{F}$ P›
and *is-processT7* : ‹s ∈ $\mathcal{D}$ P ⟹ *tF* s ⟹ *ftF* t ⟹ s @ t ∈ $\mathcal{D}$ P›
and *is-processT8* : ‹s ∈ $\mathcal{D}$ P ⟹ (s, X) ∈ $\mathcal{F}$ P›
and *is-processT9* : ‹s @ [✓(r)] ∈ $\mathcal{D}$ P ⟹ s ∈ $\mathcal{D}$ P›
by (*fold T-F-spec*)
(*use is-processT* **in** ‹*metis* [[*metis-verbose*=*false*]] *prefixE*›)+

lemma *is-processT6-notin* : ‹(s @ [✓(r)], {}) ∈ $\mathcal{F}$ P ⟹ ✓(r) ∉ X ⟹ (s, X) ∈ $\mathcal{F}$ P›
and *is-processT6-TR-notin* : ‹s @ [✓(r)] ∈ $\mathcal{T}$ P ⟹ ✓(r) ∉ X ⟹ (s, X) ∈ $\mathcal{F}$ P›
by (*metis Diff-insert-absorb is-processT6*)
(*metis Diff-insert-absorb is-processT6-TR*)

lemma *is-processT3-TR-append* : ‹t @ u ∈ $\mathcal{T}$ P ⟹ t ∈ $\mathcal{T}$ P›
using *is-processT3-TR* **by** *fastforce*

lemma *nonempty-divE* :
‹$\mathcal{D}$ P ≠ {} ⟹ (⋀t. *tF* t ⟹ t ∈ $\mathcal{D}$ P ⟹ *thesis*) ⟹ *thesis*›
by (*metis ex-in-conv front-tickFree-nonempty-append-imp is-processT2 is-processT8 is-processT9 neq-Nil-conv nonTickFree-n-frontTickFree*)

lemma *div-butlast-when-non-tickFree-iff* :

‹*ftF s* ⟹ (*if tF s then s else butlast s*) ∈ 𝒟 *P* ⟷ *s* ∈ 𝒟 *P*›
by (*cases s rule*: *rev-cases*; *simp add*: *front-tickFree-iff-tickFree-butlast*)
(*metis front-tickFree-Cons-iff is-processT7 is-processT9 is-tick-def*)

2.5 Some Consequences of the Process Characterization

lemma *F-T*: ‹(*s*, *X*) ∈ ℱ *P* ⟹ *s* ∈ 𝒯 *P*›
by (*simp add*: *T-def-spec split-def*, *metis*)

lemma *T-F*: ‹*s* ∈ 𝒯 *P* ⟹ (*s*, {}) ∈ ℱ *P*›
using *is-processT4* **by** (*auto simp add*: *T-def-spec*)

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

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

lemma *is-processT5-S7*: ‹*t* ∈ 𝒯 *P* ⟹ (*t*, *A*) ∉ ℱ *P* ⟹ ∃ *x*. *x* ∈ *A* ∧ *t* @ [*x*] ∈ 𝒯 *P*›
by (*metis T-F-spec is-processT5 sup-bot-left*)

lemma *is-processT5-S7'*:
‹(*t*, *X*) ∈ ℱ *P* ⟹ (*t*, *X* ∪ *A*) ∉ ℱ *P* ⟹ ∃ *x*. *x* ∈ *A* ∧ *x* ∉ *X* ∧ *t* @ [*x*] ∈ 𝒯 *P*›
by (*erule contrapos-np*, *subst Un-Diff-cancel*[*symmetric*])
(*rule is-processT5*, *auto simp*: *T-F-spec*)

lemma *trace-tick-continuation-or-all-tick-failuresE*:
‹⟦(*s*, {}) ∈ ℱ *P*; ⋀*r*. *s* @ [✓(*r*)] ∈ 𝒯 *P* ⟹ *thesis*; (*s*, *range tick*) ∈ ℱ *P* ⟹ *thesis*⟧ ⟹ *thesis*›
by (*metis F-T f-inv-into-f is-processT5-S7*)

lemmas *Nil-elem-T* [*simp*] = *is-processT1-TR*

lemmas *F-imp-front-tickFree* = *is-processT2*
and *D-imp-front-tickFree* = *is-processT8*[*THEN is-processT2*]
and *T-imp-front-tickFree* = *T-F*[*THEN is-processT2*]

lemma *D-front-tickFree-subset* : ‹𝒟 *P* ⊆ *Collect ftF*›
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 simp add*: *is-processT8*)

lemma *D-F* : ‹$\{(s,\ x).\ s \in \mathcal{D}\ P\} \subseteq \mathcal{F}\ P$›
using *F-D-part* **by** *blast*

lemma *append-T-imp-tickFree*: ‹$t\ @\ s \in \mathcal{T}\ P \Longrightarrow s \neq [] \Longrightarrow tF\ t$›
by (*meson front-tickFree-append-iff is-processT2-TR*)

lemma *tick-T-F*: ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow (t\ @\ [\checkmark(r)],\ X) \in \mathcal{F}\ P$›
by (*meson append-T-imp-tickFree is-processT5-S7 list.discI non-tickFree-tick tick-Free-append-iff*)

lemma *is-processT9-tick*: ‹$[\checkmark(r)] \in \mathcal{D}\ P \Longrightarrow ftF\ s \Longrightarrow s \in \mathcal{D}\ P$›
by (*metis append-Nil is-processT7 is-processT9 tickFree-Nil*)

lemma *T-nonTickFree-imp-decomp*: ‹$t \in \mathcal{T}\ P \Longrightarrow \neg\ tF\ t \Longrightarrow \exists\, s\ r.\ t = s\ @\ [\checkmark(r)]$›
by (*simp add*: *is-processT2-TR 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 $\mathcal{R}_a$ (abbreviating *refusals after*). The former provides just a set of minimal elements from a given set of elements of typeclass *ord* ...

definition *min-elems* :: ‹$('s{::}ord)\ set \Rightarrow 's\ set$›
where ‹$min\text{-}elems\ X \equiv \{s \in X.\ \forall\, t \in X.\ \neg\ (t < s)\}$›

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

lemma *min-elems-le-self*[*simp*] : ‹$(min\text{-}elems\ A) \subseteq A$›
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 ftF*) = {[]}›
by (*simp add*: *min-elems-def less-eq-list-def set-eq-iff*)
(*metis front-tickFree-charn nil-less nil-less2*)

lemma *min-elems5* : ‹(*s* :: *'a list*) ∈ *A* ⟹ ∃ *t*≤*s*. *t* ∈ *min-elems A*›
proof −
have ∗ : ‹*x* ∈ *A* ⟹ *length x* ≤ *n* ⟹ ∃ *s*≤*x*. *s* ∈ *min-elems A*› **for** *x* :: ‹*'a list*›
and *A n*
proof (*induct n arbitrary*: *x rule*: *nat-induct*)
show ‹*x* ∈ *A* ⟹ *length x* ≤ *0* ⟹ ∃ *s*≤*x*. *s* ∈ *min-elems A*› **for** *x* **by** (*simp add*: *Nil-min-elems*)
next
fix *n x*
assume ‹*x* ∈ *A*› ‹*length x* ≤ *Suc n*›
assume *hyp* : ‹*x* ∈ *A* ⟹ *length x* ≤ *n* ⟹ ∃ *s*≤*x*. *s* ∈ *min-elems A*› **for** *x*
show ‹∃ *s*≤*x*. *s* ∈ *min-elems A*›
proof (*cases* ‹∃ *y* ∈ *A*. *y* < *x*›)
show ‹∃ *y*∈*A*. *y* < *x* ⟹ ∃ *s*≤*x*. *s* ∈ *min-elems A*›
by (*elim bexE*, *frule hyp*, *drule less-length-mono*, *use* ‹*length x* ≤ *Suc n*› **in** *simp*)
(*meson dual-order.strict-trans2 less-list-def*)
next
show ‹¬ (∃ *y*∈*A*. *y* < *x*) ⟹ ∃ *s*≤*x*. *s* ∈ *min-elems A*›
using ‹*x* ∈ *A*› **unfolding** *min-elems-def* **by** *auto*
qed
qed
thus ‹*s* ∈ *A* ⟹ ∃ *t*≤*s*. *t* ∈ *min-elems A*› **by** *auto*
qed

lemma *min-elems4*: ‹*A* ≠ {} ⟹ ∃ *s*. (*s* :: (*'a*, *'r*) *trace*$_{ptick}$) ∈ *min-elems A*›
by (*auto dest*: *min-elems5*)

lemma *min-elems-charn*: ‹*t* ∈ *A* ⟹ ∃ *t' r*. *t* = (*t'* @ *r*) ∧ *t'* ∈ *min-elems A*›
by (*meson prefixE min-elems5*)

lemma *min-elems-no*: ‹(*s*::*'a list*) ∈ *min-elems A* ⟹ *t* ∈ *A* ⟹ *t* ≤ *s* ⟹ *s* = *t*›
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq min-elems-def order-neq-le-trans*)

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

definition *Refusals-after* :: ‹[(*'a*, *'r*) *process*$_{ptick}$, (*'a*, *'r*) *trace*$_{ptick}$] ⇒ (*'a*, *'r*) *refusal*$_{ptick}$ *set*› (‹$\mathcal{R}_a$›)
where ‹$\mathcal{R}_a$ *P tr* ≡ {*ref*. (*tr*, *ref*) ∈ $\mathcal{F}$ *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$_{ptick}$ :: (*type*, *type*) *below*
begin

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

definition *le-approx-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-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*: ‹$P \sqsubseteq Q \Longrightarrow s \notin \mathcal{D}\ P \Longrightarrow ((s, X) \in \mathcal{F}\ Q) = ((s, X) \in \mathcal{F}\ P)$›
by (*auto simp*: *Refusals-after-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*: ‹$P \sqsubseteq Q \Longrightarrow s \notin \mathcal{D}\ P \Longrightarrow s \in \mathcal{T}\ Q \longleftrightarrow s \in \mathcal{T}\ P$›
by (*auto simp*: *le-approx2 T-F-spec*[*symmetric*])

lemma *le-approx-lemma-F* : ‹$P \sqsubseteq Q \Longrightarrow \mathcal{F}\ Q \subseteq \mathcal{F}\ P$›
by (*meson le-approx2 process-charn subrelI*)

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* : ‹$P \sqsubseteq Q \Longrightarrow s \notin \mathcal{D}\ P \Longrightarrow (s, X) \in \mathcal{F}\ P \longleftrightarrow (s, X) \in \mathcal{F}\ Q$›
by (*auto simp*: *le-approx-def Refusals-after-def*)

instance *process*$_{ptick}$ :: (*type*, *type*) *po*
proof *intro-classes*

```
  show ‹P ⊑ P› for P :: ‹('a, 'r) process_ptick›
    by (metis D-T elem-min-elems le-approx-def subsetI)
next
  show ‹P ⊑ Q ⟹ Q ⊑ P ⟹ P = Q› for P Q :: ‹('a, 'r) process_ptick›
    by (simp add: Process-eq-spec le-approx1 le-approx-lemma-F subset-antisym)
next
  fix P Q R :: ‹('a, 'r) process_ptick›
  assume ‹P ⊑ Q› and ‹Q ⊑ R›
  show ‹P ⊑ R›
  proof (unfold le-approx-def, intro conjI allI impI)
    show ‹𝒟 R ⊆ 𝒟 P› by (meson ‹P ⊑ Q› ‹Q ⊑ R› dual-order.trans le-approx1)
  next
    show ‹s ∉ 𝒟 P ⟹ ℛ_a P s = ℛ_a R s› for s
      by (metis ‹P ⊑ Q› ‹Q ⊑ R› le-approx-def subsetD)
  next
    from ‹P ⊑ Q›[THEN le-approx1]  ‹P ⊑ Q›[THEN le-approx3]
      ‹Q ⊑ R›[THEN le-approx2T] ‹Q ⊑ R›[THEN le-approx3]
    show ‹min-elems (𝒟 P) ⊆ 𝒯 R›
      by (simp add: min-elems-def subset-iff) blast
  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:

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

- `po_class.is_lub_range_shift`:
  chain ?S $\Longrightarrow$ *range* (λi. *?S* (i + *?j*)) <<| *?x* = *range ?S* <<| *?x*
- `po_class.is_lub_rangeD1`: *range ?S* <<| *?x* $\Longrightarrow$ *?S ?i* $\sqsubseteq$ *?x*
- `po_class.lub_eqI`: *?M* <<| *?l* $\Longrightarrow$ *lub ?M* = *?l*
- `po_class.is_lub_unique`:⟦*?S* <<| *?x*; *?S* <<| *?y*⟧ $\Longrightarrow$ *?x* = *?y*

lemma *min-elems3*: ‹*s* @ [*c*] ∈ $\mathcal{D}$ *P* $\Longrightarrow$ *s* @ [*c*] ∉ *min-elems* ($\mathcal{D}$ *P*) $\Longrightarrow$ *s* ∈ $\mathcal{D}$ *P*›
by (*simp add*: *min-elems-def less-eq-list-def less-list-def*)
(*metis D-imp-front-tickFree append.right-neutral front-tickFree-append-iff front-tickFree-dw-closed is-processT7 prefix-def*)

lemma *min-elems1*: ‹*s* ∉ $\mathcal{D}$ *P* $\Longrightarrow$ *s* @ [*c*] ∈ $\mathcal{D}$ *P* $\Longrightarrow$ *s* @ [*c*] ∈ *min-elems* ($\mathcal{D}$ *P*)›
using *min-elems3* **by** *blast*

lemma *min-elems2*: ‹*s* ∉ $\mathcal{D}$ *P* $\Longrightarrow$ *s* @ [*c*] ∈ $\mathcal{D}$ *P* $\Longrightarrow$ *P* $\sqsubseteq$ *S* $\Longrightarrow$ *Q* $\sqsubseteq$ *S* $\Longrightarrow$ (*s* @ [*c*], {}) ∈ $\mathcal{F}$ *Q*›
by (*meson T-F in-mono le-approx3 le-approx-lemma-F min-elems3*)

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

lemma *ND-F-dir2*: ‹*s* ∉ $\mathcal{D}$ *P* $\Longrightarrow$ (*s*, {}) ∈ $\mathcal{F}$ *P* $\Longrightarrow$ *P* $\sqsubseteq$ *S* $\Longrightarrow$ *Q* $\sqsubseteq$ *S* $\Longrightarrow$ (*s*, {}) ∈ $\mathcal{F}$ *Q*›
by (*meson is-processT8 le-approx2*)

lemma *ND-F-dir2′*: ‹*s* ∉ $\mathcal{D}$ *P* $\Longrightarrow$ *s* ∈ $\mathcal{T}$ *P* $\Longrightarrow$ *P* $\sqsubseteq$ *S* $\Longrightarrow$ *Q* $\sqsubseteq$ *S* $\Longrightarrow$ *s* ∈ $\mathcal{T}$ *Q*›
by (*meson D-T le-approx2T*)

lemma *chain-lemma*: ‹*chain S* $\Longrightarrow$ *S i* $\sqsubseteq$ *S k* ∨ *S k* $\sqsubseteq$ *S i*›
by (*metis chain-mono-less not-le-imp-less po-class.chain-mono*)

context fixes *S* :: ‹*nat* ⇒ ('*a*, '*r*) *process*$_{ptick}$›
assumes ‹*chain S*›
begin

lift-definition *lim-proc* :: ‹('*a*, '*r*) *process*$_{ptick}$›
is ‹($\bigcap$ ($\mathcal{F}$ ‘ *range S*), $\bigcap$ ($\mathcal{D}$ ‘ *range S*))›
proof (*unfold is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv*, *intro conjI allI impI*)
show ‹([], {}) ∈ $\bigcap$ ($\mathcal{F}$ ‘ *range S*)› **by** (*simp add*: *is-processT*)
next
show ‹(*s*, *X*) ∈ $\bigcap$ ($\mathcal{F}$ ‘ *range S*) $\Longrightarrow$ *ftF s*› **for** *s X*

by (*meson INT-iff UNIV-I image-eqI is-processT2*)
next
show ‹(*s* @ *t*, {}) ∈ ⋂ (*F* ‘ *range S*) ⟹
(*s*, {}) ∈ ⋂ (*F* ‘ *range S*)› **for** *s t* **by** (*auto intro*: *is-processT3*)
next
show ‹(*s*, *Y*) ∈ ⋂ (*F* ‘ *range S*) ∧ *X* ⊆ *Y* ⟹ (*s*, *X*) ∈ ⋂ (*F* ‘ *range S*)› **for**
s X Y
by (*metis* (*full-types*) *INT-iff is-processT4*)
next
show ‹(*s*, *X* ∪ *Y*) ∈ ⋂ (*F* ‘ *range S*)›
if *assm* : ‹(*s*, *X*) ∈ ⋂ (*F* ‘ *range S*) ∧
(∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ ⋂ (*F* ‘ *range S*))› **for** *s X Y*
proof (*rule ccontr*)
assume ‹(*s*, *X* ∪ *Y*) ∉ ⋂ (*F* ‘ *range S*)›
then obtain *i* **where** ‹(*s*, *X* ∪ *Y*) ∉ *F* (*S i*)› **by** *blast*
moreover have ‹(*s*, *X*) ∈ *F* (*S j*)› **for** *j* **using** *assm* **by** *blast*
ultimately obtain *c* **where** ‹*c* ∈ *Y*› **and** ∗ : ‹(*s* @ [*c*], {}) ∈ *F* (*S i*)›
using *is-processT5* **by** *blast*
from ‹(*s*, *X* ∪ *Y*) ∉ *F* (*S i*)› *is-processT8* **have** ‹*s* ∉ *D* (*S i*)› **by** *blast*
from *assm* ‹*c* ∈ *Y*› **obtain** *j* **where** ∗∗ : ‹(*s* @ [*c*], {}) ∉ *F* (*S j*)› **by** *blast*

from *chain-lemma*[*OF* ‹*chain S*›, *of i j*] ∗ ∗∗ **show** *False*
by (*elim disjE*; *use* ‹*s* ∉ *D* (*S i*)› *is-processT8 min-elems6 proc-ord2a* **in** *blast*)
qed
next
show ‹(*s* @ [✓(*r*)], {}) ∈ ⋂ (*F* ‘ *range S*) ⟹
(*s*, *X* − {✓(*r*)}) ∈ ⋂ (*F* ‘ *range S*)› **for** *s r X* **by** (*simp add*: *is-processT6*)
next
show ‹*s* ∈ ⋂ (*D* ‘ *range S*) ∧ *tF s* ∧ *ftF t* ⟹
s @ *t* ∈ ⋂ (*D* ‘ *range S*)› **for** *s t* **by** (*simp add*: *is-processT7*)
next
show ‹*s* ∈ ⋂ (*D* ‘ *range S*) ⟹ (*s*, *X*) ∈ ⋂ (*F* ‘ *range S*)› **for** *s X*
by (*simp add*: *is-processT8*)
next
show ‹*s* @ [✓(*r*)] ∈ ⋂ (*D* ‘ *range S*) ⟹ *s* ∈ ⋂ (*D* ‘ *range S*)› **for** *s r*
by (*auto intro*: *is-processT9*)
qed

lemma *F-LUB*: ‹*F lim-proc* = ⋂ (*F* ‘ *range S*)›
by (*metis Failures.rep-eq lim-proc.rep-eq process-surj-pair prod.sel*(*1*))

lemma *D-LUB*: ‹*D lim-proc* = ⋂ (*D* ‘ *range S*)›
by (*metis Divergences.rep-eq lim-proc.rep-eq process-surj-pair prod.inject*)

lemma *T-LUB*: ‹*T lim-proc* = ⋂ (*T* ‘ *range S*)›
by (*insert F-LUB*, *auto simp add*: *T-def-spec*) (*meson F-T T-F*)

lemmas *LUB-projs* = *F-LUB D-LUB T-LUB*

lemma *Refusals-LUB*: ‹$\mathcal{R}$ *lim-proc* = $\bigcap$ ($\mathcal{R}$ ‘ *range S*)›
by (*auto simp add*: *Refusals-def-bis F-LUB*)

lemma *Refusals-after-LUB*: ‹$\mathcal{R}_a$ *lim-proc s* = ($\bigcap i.$ ($\mathcal{R}_a$ (*S i*) *s*))›
by (*auto simp add*: *Refusals-after-def F-LUB*)

lemma *F-LUB-2*: ‹(*s*, *X*) ∈ $\mathcal{F}$ *lim-proc* ⟷ ($\forall i.$ (*s*, *X*) ∈ $\mathcal{F}$ (*S i*))›
and *D-LUB-2*: ‹*t* ∈ $\mathcal{D}$ *lim-proc* ⟷ ($\forall i.$ *t* ∈ $\mathcal{D}$ (*S i*))›
and *T-LUB-2*: ‹*t* ∈ $\mathcal{T}$ *lim-proc* ⟷ ($\forall i.$ *t* ∈ $\mathcal{T}$ (*S i*))›
and *Refusals-LUB-2*: ‹*X* ∈ $\mathcal{R}$ *lim-proc* ⟷ ($\forall i.$ *X* ∈ $\mathcal{R}$ (*S i*))›
and *Refusals-after-LUB-2*: ‹*X* ∈ $\mathcal{R}_a$ *lim-proc s* ⟷ ($\forall i.$ *X* ∈ $\mathcal{R}_a$ (*S i*) *s*)›
by (*simp-all add*: *F-LUB D-LUB T-LUB Refusals-LUB Refusals-after-LUB*)

end

By exiting the context, terms like $\mathcal{F}$ *lim-proc* will become $\mathcal{F}$ (*lim-proc S*) and the assumption *chain S* will be added.

2.7 Process Refinement is a Partial Ordering

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

instantiation $process_{ptick}$ :: (*type*, *type*) *ord*
begin

definition $less\text{-}eq\text{-}process_{ptick}$ :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $process_{ptick}$ ⇒ *bool*›
where ‹$less\text{-}eq\text{-}process_{ptick}$ *P Q* ≡ $\mathcal{D}$ *Q* ⊆ $\mathcal{D}$ *P* ∧ $\mathcal{F}$ *Q* ⊆ $\mathcal{F}$ *P*›

definition $less\text{-}process_{ptick}$ :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $process_{ptick}$ ⇒ *bool*›
where ‹$less\text{-}process_{ptick}$ *P Q* ≡ *P* ≤ *Q* ∧ *P* ≠ *Q*›

instance ..

end

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

lemma *le-ref1* : ‹*P* ≤ *Q* ⟹ $\mathcal{D}$ *Q* ⊆ $\mathcal{D}$ *P*›
and *le-ref2* : ‹*P* ≤ *Q* ⟹ $\mathcal{F}$ *Q* ⊆ $\mathcal{F}$ *P*›
and *le-ref2T* : ‹*P* ≤ *Q* ⟹ $\mathcal{T}$ *Q* ⊆ $\mathcal{T}$ *P*›
and *le-approx-imp-le-ref*: ‹(*P*::('*a*, '*r*) $process_{ptick}$) ⊑ *Q* ⟹ *P* ≤ *Q*›
by (*simp-all add*: $less\text{-}eq\text{-}process_{ptick}$*-def le-approx1 le-approx-lemma-F*)
(*use T-F-spec* **in** *blast*)

lemma *F-subset-imp-T-subset* : ‹$\mathcal{F}\ P \subseteq \mathcal{F}\ Q \Longrightarrow \mathcal{T}\ P \subseteq \mathcal{T}\ Q$›
using *T-F-spec* **by** *blast*

lemma *D-extended-is-D* :
‹$\{t\ @\ u\ |t\ u.\ t \in \mathcal{D}\ P \wedge tF\ t \wedge ftF\ u\} = \mathcal{D}\ P$›
by (*auto simp add*: *is-processT7*)
(*metis D-imp-front-tickFree append.right-neutral butlast-snoc front-tickFree-append-iff front-tickFree-charn is-processT9 nonTickFree-n-frontTickFree tickFree-Nil*)

lemma *Process-eq-optimizedI* :
‹$[\![\bigwedge t.\ t \in \mathcal{D}\ P \Longrightarrow t \in \mathcal{D}\ Q;\ \bigwedge t.\ t \in \mathcal{D}\ Q \Longrightarrow t \in \mathcal{D}\ P;$
$\bigwedge t\ X.\ (t, X) \in \mathcal{F}\ P \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow t \notin \mathcal{D}\ Q \Longrightarrow (t, X) \in \mathcal{F}\ Q;$
$\bigwedge t\ X.\ (t, X) \in \mathcal{F}\ Q \Longrightarrow t \notin \mathcal{D}\ Q \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow (t, X) \in \mathcal{F}\ P]\!] \Longrightarrow P = Q$›
by (*simp add*: *Process-eq-spec-optimized*, *safe*, *auto intro*: *is-processT8*)

instance $process_{ptick}$ :: (*type*, *type*) *order*
by *intro-classes* (*auto simp*: *less-eq-process*$_{ptick}$*-def less-process*$_{ptick}$*-def Process-eq-spec*)

lemma *lim-proc-is-ub*: ‹*chain S* $\Longrightarrow$ *range S* <| *lim-proc S*›
by (*simp add*: *is-ub-def le-approx-def F-LUB D-LUB T-LUB Refusals-after-def*)
(*intro allI conjI*, *blast*, *use chain-lemma is-processT8 le-approx2* **in** *blast*,
use D-T chain-lemma le-approx2T le-approx-def **in** *blast*)

lemma *chain-min-elem-div-is-min-for-sequel*:
‹*chain S* $\Longrightarrow$ *s* $\in$ *min-elems* ($\mathcal{D}$ (*S i*)) $\Longrightarrow$ *i* $\leq$ *j* $\Longrightarrow$ *s* $\in$ $\mathcal{D}$ (*S j*) $\Longrightarrow$ *s* $\in$ *min-elems* ($\mathcal{D}$ (*S j*))›
by (*metis elem-min-elems insert-absorb insert-subset le-approx1 min-elems5 min-elems-no po-class.chain-mono*)

lemma *limproc-is-lub*: ‹*range S* <<| *lim-proc S*› **if** ‹*chain S*›
proof (*unfold is-lub-def*, *intro conjI allI impI*)
show ‹*range S* <| *lim-proc S*› **by** (*simp add*: *lim-proc-is-ub* ‹*chain S*›)
next
show ‹*lim-proc S* $\sqsubseteq$ *P*› **if** ‹*range S* <| *P*› **for** *P*
proof (*unfold le-approx-def*, *intro conjI allI impI subsetI*)
show ‹*s* $\in$ $\mathcal{D}$ *P* $\Longrightarrow$ *s* $\in$ $\mathcal{D}$ (*lim-proc S*)› **for** *s*
by (*meson D-LUB-2* ‹*chain S*› ‹*range S* <| *P*› *is-ub-def le-approx1 rangeI subsetD*)
next

```
    show ‹s ∉ 𝒟 (lim-proc S) ⟹ ℛa (lim-proc S) s = ℛa P s› for s
      by (metis ‹chain S› ‹range S <| P› D-LUB-2 le-approx-def lim-proc-is-ub
ub-rangeD)
  next
    fix s
    assume ‹s ∈ min-elems (𝒟 (lim-proc S))›
    from elem-min-elems[OF this] have ‹∀ i. s ∈ 𝒟 (S i)›
      by (simp add: ‹chain S› D-LUB)
    have ‹∃ i. ∀ j≥i. s ∈ min-elems (𝒟 (S j))›
    proof (rule ccontr)
      assume ‹∄ i. ∀ j≥i. s ∈ min-elems (𝒟 (S j))›
      hence ‹∀ i. ∃ j≥i. s ∉ min-elems (𝒟 (S j))› by simp
      with ‹∀ i. s ∈ 𝒟 (S i)› chain-min-elem-div-is-min-for-sequel ‹chain S›
      have ‹∀ j. s ∉ min-elems (𝒟 (S j))› by blast
      from ‹s ∈ min-elems (𝒟 (lim-proc S))› ‹∀ i. s ∈ 𝒟 (S i)› show False
        by (cases s rule: rev-cases; simp add: min-elems-def D-LUB ‹chain S›)
          (use Nil-min-elems ‹∀ j. s ∉ min-elems (𝒟 (S j))› in blast,
            metis (no-types, lifting) INT-iff ‹∀ j. s ∉ min-elems (𝒟 (S j))› less-self
min-elems3)
    qed
     thus ‹s ∈ 𝒯 P› by (meson le-approx3 order.refl subset-eq ‹range S <| P›
ub-rangeD)
  qed
qed


lemma limproc-is-thelub: ‹chain S ⟹ (⨆ i. S i) = lim-proc S›
  by (frule limproc-is-lub, frule po-class.lub-eqI, simp)


instance process_ptick :: (type, type) cpo
  by intro-classes (use limproc-is-lub in blast)



instance process_ptick :: (type, type) pcpo
proof
  define bot0 :: ‹('a, 'r) process0› where ‹bot0 ≡ ({(s, X). ftF s}, {d. ftF d})›
  define bot :: ‹('a, 'r) process_ptick› where ‹bot ≡ process-of-process0 bot0›

  have ‹is-process bot0›
    unfolding is-process-def bot0-def
    by (simp add: FAILURES-def DIVERGENCES-def)
      (meson front-tickFree-append-iff front-tickFree-dw-closed)
  have F-bot : ‹ℱ bot = {(s, X). ftF s}›
    by (metis CollectI FAILURES-def Failures.rep-eq ‹is-process bot0›
        bot0-def bot-def fst-eqD process-of-process0-inverse)
  have D-bot : ‹𝒟 bot = {d. ftF d}›
    by (metis CollectI DIVERGENCES-def Divergences.rep-eq ‹is-process bot0›
```

bot$_0$-def bot-def process-of-process$_0$-inverse prod.sel(2))

show ‹$\exists\, x :: ('a, 'r)\ process_{ptick}.\ \forall\ y.\ x \sqsubseteq y$›
proof (*intro exI allI*)
show ‹$bot \sqsubseteq y$› **for** y
proof (*unfold le-approx-def*, *intro conjI allI impI subsetI*)
show ‹$s \in \mathcal{D}\ y \Longrightarrow s \in \mathcal{D}\ bot$› **for** s
by (*simp add*: *D-bot D-imp-front-tickFree*)
next
from *F-imp-front-tickFree* **show** ‹$s \notin \mathcal{D}\ bot \Longrightarrow \mathcal{R}_a\ bot\ s = \mathcal{R}_a\ y\ s$› **for** s
by (*auto simp add*: *D-bot Refusals-after-def F-bot*)
next
show ‹$s \in min\text{-}elems\ (\mathcal{D}\ bot) \Longrightarrow s \in \mathcal{T}\ y$› **for** s
by (*simp add*: *D-bot min-elems-Collect-ftF-is-Nil*)
qed
qed
qed

2.8 Process Refinement is Admissible

lemma *le-FD-adm* : ‹*cont* ($u :: ('b{::}cpo) \Rightarrow ('a, 'r)\ process_{ptick}$) $\Longrightarrow$ *monofun v* $\Longrightarrow$ *adm* ($\lambda x.\ u\ x \leq v\ x$)›
apply (*unfold less-eq-process$_{ptick}$-def adm-def*)
apply (*simp add*: *cont2contlubE D-LUB F-LUB ch2ch-cont limproc-is-thelub monofun-def*)
by (*meson INF-greatest dual-order.trans is-ub-thelub le-approx1 le-approx-lemma-F*)

lemmas *le-FD-adm-cont*[*simp*] = *le-FD-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*]:
‹$[\![\bigwedge x.\ P\ x \Longrightarrow cont\ (f\ x);\ \bigwedge x.\ \neg\ P\ x \Longrightarrow cont\ (g\ x)]\!] \Longrightarrow$
cont ($\lambda y.$ *if P x then f x y else g x y*)›
for $f :: ‹'c \Rightarrow 'b :: cpo \Rightarrow ('a, 'r)\ process_{ptick}›$
by (*auto simp*: *cont-def*)

2.10 Tools for proving continuity

lemma *cont-process-rec*: ‹$P = (\mu\ X.\ f\ X) \Longrightarrow cont\ f \Longrightarrow P = f\ P$›
by (*simp add*: *def-cont-fix-eq*)

lemma *Inter-nonempty-finite-chained-sets*: ‹$(\bigcap i.\ S\ i) \neq \{\}$›

if ‹$\bigwedge i.\ j \leq i \Longrightarrow S\ i \neq \{\}$› ‹*finite* $(S\ j)$› ‹$\bigwedge i.\ S\ (Suc\ i) \subseteq S\ i$› **for** $S :: ‹nat \Rightarrow 'a\ set›$
proof −
have $* :$ ‹$\forall i.\ S\ i \neq \{\} \Longrightarrow$ *finite* $(S\ 0) \Longrightarrow \forall i.\ S\ (Suc\ i) \subseteq S\ i \Longrightarrow (\bigcap i.\ S\ i) \neq \{\}$›
for $S :: ‹nat \Rightarrow 'a\ set›$
proof (*induct* ‹*card* $(S\ 0)$› *arbitrary*: *S rule*: *nat-less-induct*)
case *1*
show *?case*
proof (*cases* ‹$\forall i.\ S\ i = S\ 0$›)
case *True*
thus *?thesis* **by** (*metis 1.prems*(*1*) *INT-iff ex-in-conv*)
next
case *False*
have *f1*: ‹$i \leq j \Longrightarrow S\ j \subseteq S\ i$› **for** *i j* **by** (*simp add*: *1.prems*(*3*) *lift-Suc-antimono-le*)
with *False* **obtain** *j m* **where** *f2*: ‹$m <$ *card* $(S\ 0)$› **and** *f3*: ‹$m =$ *card* $(S\ j)$›
by (*metis 1.prems*(*2*) *psubsetI psubset-card-mono zero-le*)
define *T* **where** ‹$T\ i \equiv S\ (i + j)$› **for** *i*
have *f4*: ‹$m =$ *card* $(T\ 0)$› **unfolding** *T-def* **by** (*simp add*: *f3*)
from *f1* **have** *f5*: ‹$(\bigcap i.\ S\ i) = (\bigcap i.\ T\ i)$› **unfolding** *T-def* **by** (*auto intro*: *le-add1*)
show *?thesis*
apply (*subst f5*)
apply (*rule 1.hyps*[*rule-format*, *OF f2*, *of T*, *OF f4*], *unfold T-def*)
by (*simp-all add*: *1.prems*(*1*, *3*) *lift-Suc-antimono-le*)
(*metis 1.prems*(*2*) *add-0 f1 finite-subset le-add1*)
qed
qed
define S' **where** ‹$S'\ i \equiv S\ (j + i)$› **for** *i*
have ‹$\forall i.\ S'\ i \neq \{\}$› **by** (*simp add*: *S'-def* ‹$\bigwedge i.\ j \leq i \Longrightarrow S\ i \neq \{\}$›)
moreover have ‹*finite* $(S'\ 0)$› **by** (*simp add*: ‹$S' \equiv \lambda i.\ S\ (j + i)$› ‹*finite* $(S\ j)$›)
moreover have ‹$\forall i.\ S'\ (Suc\ i) \subseteq S'\ i$› **by** (*simp add*: *S'-def* ‹$\bigwedge i.\ S\ (Suc\ i) \subseteq S\ i$›)
ultimately have ‹$(\bigcap i.\ S'\ i) \neq \{\}$› **by** (*fact* $*$)
also from *lift-Suc-antimono-le*[**where** $f = S$, *OF* ‹$\bigwedge i.\ S\ (Suc\ i) \subseteq S\ i$›]
have ‹$(\bigcap i.\ S'\ i) = (\bigcap i.\ S\ i)$›
by (*simp add*: *INF-greatest INF-lower INF-mono' S'-def equalityI*)
finally show ‹$(\bigcap i.\ S\ i) \neq \{\}$› **.**
qed

method *prove-finite-subset-of-prefixes* **for** $t :: ‹('a, 'r)\ trace_{ptick}›$ =
— Useful for establishing the second hypothesis
solves ‹(*rule finite-UnI*; *prove-finite-subset-of-prefixes t*) |
(*rule finite-subset*[*of* - ‹$\{u.\ u \leq t\}$›],
use prefixI in blast, *simp add*: *prefixes-fin*)›

Chapter 3

Constant Processes

3.1 The Undefined Process: $\bot$

This is the key result: $\bot$ — which we know to exist from the process instantiation — can be explicitly written with its projections.

lemma *F-BOT* : ‹$\mathcal{F}\ \bot = \{(s :: ('a, 'r)\ trace_{ptick},\ X).\ ftF\ s\}$›
  and *D-BOT* : ‹$\mathcal{D}\ \bot = \{d :: ('a, 'r)\ trace_{ptick}.\ ftF\ d\}$›
  and *T-BOT* : ‹$\mathcal{T}\ \bot = \{s :: ('a, 'r)\ trace_{ptick}.\ ftF\ s\}$›
proof −
  define bot_0 :: ‹$('a, 'r)\ process_0$› **where** ‹$bot_0 \equiv (\{(s, X).\ ftF\ s\}, \{d.\ ftF\ d\})$›
  define *bot* :: ‹$('a, 'r)\ process_{ptick}$› **where** ‹$bot \equiv process\text{-}of\text{-}process_0\ bot_0$›

  have ‹*is-process* bot_0›
    unfolding *is-process-def* bot_0*-def*
    by (*simp add*: *FAILURES-def DIVERGENCES-def*)
      (*meson front-tickFree-append-iff front-tickFree-dw-closed*)
  have *F-bot* : ‹$\mathcal{F}\ bot = \{(s, X).\ ftF\ s\}$›
    by (*metis CollectI FAILURES-def Failures.rep-eq* ‹*is-process* bot_0›
      bot_0*-def bot-def fst-eqD process-of-process*$_0$*-inverse*)
  have *D-bot* : ‹$\mathcal{D}\ bot = \{d.\ ftF\ d\}$›
    by (*metis CollectI DIVERGENCES-def Divergences.rep-eq* ‹*is-process* bot_0›
      bot_0*-def bot-def process-of-process*$_0$*-inverse prod.sel*(*2*))
  hence *T-bot* : ‹$\mathcal{T}\ bot = \{s.\ ftF\ s\}$›
   by (*metis D-T T-imp-front-tickFree mem-Collect-eq subsetI subset-Un-eq sup.orderE*)
  have ‹$bot = \bot$›
    by (*simp add*: *eq-bottom-iff le-approx-def Refusals-after-def F-bot D-bot*
      min-elems-Collect-ftF-is-Nil)
      (*metis D-front-tickFree-subset process-charn*)
  with *F-bot D-bot T-bot*
  show ‹$\mathcal{F}\ \bot = \{(s :: ('a, 'r)\ trace_{ptick},\ X).\ ftF\ s\}$›
    ‹$\mathcal{D}\ \bot = \{d :: ('a, 'r)\ trace_{ptick}.\ ftF\ d\}$›
    ‹$\mathcal{T}\ \bot = \{s :: ('a, 'r)\ trace_{ptick}.\ ftF\ s\}$› **by** *simp-all*
qed

lemmas *BOT-projs* = *F-BOT D-BOT T-BOT*

lemma *BOT-iff-Nil-D* : ‹$P = \bot \longleftrightarrow [] \in \mathcal{D}\ P$›
proof (*rule iffI*)
 show ‹$P = \bot \Longrightarrow [] \in \mathcal{D}\ P$› **by** (*simp add*: *D-BOT*)
next
 show ‹$P = \bot$› **if** ‹$[] \in \mathcal{D}\ P$›
 proof (*subst Process-eq-spec-optimized*, *safe*)
  show ‹$s \in \mathcal{D}\ P \Longrightarrow s \in \mathcal{D}\ \bot$› **for** *s* **by** (*simp add*: *D-BOT D-imp-front-tickFree*)
 next
  show ‹$s \in \mathcal{D}\ \bot \Longrightarrow s \in \mathcal{D}\ P$› **for** *s*
   by (*metis* ‹$[] \in \mathcal{D}\ P$› *append-Nil process-charn tickFree-Nil*)
 next
  show ‹$(s, X) \in \mathcal{F}\ P \Longrightarrow (s, X) \in \mathcal{F}\ \bot$› **for** *s X*
   using *le-approx-lemma-F* **by** *blast*
 next
  show ‹$\mathcal{D}\ P = \mathcal{D}\ \bot \Longrightarrow (s, X) \in \mathcal{F}\ \bot \Longrightarrow (s, X) \in \mathcal{F}\ P$› **for** *s X*
   using *D-BOT F-T T-BOT is-processT8* **by** *blast*
 qed
qed

lemma *BOT-iff-tick-D* : ‹$P = \bot \longleftrightarrow (\exists\, r.\ [\checkmark(r)] \in \mathcal{D}\ P)$›
 by (*metis BOT-iff-Nil-D D-BOT front-tickFree-single is-processT9-tick mem-Collect-eq*)

3.2 The SKIP Process

In this new parameterized version, the SKIP process is of course also parameterized.

lift-definition *SKIP* :: ‹$'r \Rightarrow ('a, 'r)\ process_{ptick}$›
 is ‹$\lambda r.\ (\{([], X) |\ X.\ \checkmark(r) \notin X\} \cup \{(s, X).\ s = [\checkmark(r)]\}, \{\})$›
 unfolding *is-process-def FAILURES-def DIVERGENCES-def*
 by (*auto simp add*: *append-eq-Cons-conv*)

abbreviation *SKIP-unit* :: ‹*'a process*› (‹*Skip*›) **where** ‹*Skip* ≡ *SKIP* ()›

lemma *F-SKIP* :
 ‹$\mathcal{F}\ (SKIP\ r) = \{([], X) |\ X.\ \checkmark(r) \notin X\} \cup \{(s, X).\ s = [\checkmark(r)]\}$›
 by (*simp add*: *FAILURES-def Failures.rep-eq SKIP.rep-eq*)

lemma *D-SKIP* : ‹$\mathcal{D}\ (SKIP\ r) = \{\}$›
 by (*simp add*: *DIVERGENCES-def Divergences.rep-eq SKIP.rep-eq*)

lemma *T-SKIP* : ‹$\mathcal{T}\ (SKIP\ r) = \{[], [\checkmark(r)]\}$›
 by (*auto simp add*: *Traces.rep-eq TRACES-def Failures.rep-eq*[*symmetric*] *F-SKIP*)

lemmas *SKIP-projs* = *F-SKIP D-SKIP T-SKIP*

lemma *inj-SKIP* : ‹*inj SKIP*›
by (*rule injI*, *simp add*: *Process-eq-spec F-SKIP*) *blast*

3.3 The STOP Process

lift-definition *STOP* :: ‹('*a*, '*r*) *process*$_{ptick}$›
is ‹({(*s*, *X*). *s* = []}, {})›
unfolding *is-process-def FAILURES-def DIVERGENCES-def* **by** *simp*

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

lemma *D-STOP* : ‹$\mathcal{D}$ *STOP* = {}›
by (*simp add*: *DIVERGENCES-def Divergences.rep-eq STOP.rep-eq*)

lemma *T-STOP* : ‹$\mathcal{T}$ *STOP* = {[]}›
by (*simp add*: *Traces.rep-eq TRACES-def Failures.rep-eq*[*symmetric*] *F-STOP*)

lemmas *STOP-projs* = *F-STOP D-STOP T-STOP*

lemma *STOP-iff-T* : ‹*P* = *STOP* ⟷ $\mathcal{T}$ *P* = {[]}›
proof (*rule iffI*)
show ‹*P* = *STOP* ⟹ $\mathcal{T}$ *P* = {[]}› **by** (*simp add*: *T-STOP*)
next
assume ‹$\mathcal{T}$ *P* = {[]}›
show ‹*P* = *STOP*›
proof (*subst Process-eq-spec*, *safe*)
show ‹*s* ∈ $\mathcal{D}$ *P* ⟹ *s* ∈ $\mathcal{D}$ *STOP*› **for** *s*
by (*metis D-T* ‹$\mathcal{T}$ *P* = {[]}› *front-tickFree-single not-Cons-self*
process-charn self-append-conv2 singletonD tickFree-Nil)
next
show ‹*s* ∈ $\mathcal{D}$ *STOP* ⟹ *s* ∈ $\mathcal{D}$ *P*› **for** *s* **by** (*simp add*: *D-STOP*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *P* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *STOP*› **for** *s X*
using *F-STOP F-T* ‹$\mathcal{T}$ *P* = {[]}› **by** *blast*
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *STOP* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *P*› **for** *s X*
by (*metis F-T T-STOP* ‹$\mathcal{T}$ *P* = {[]}› *is-processT5-S7 singletonD snoc-eq-iff-butlast*)
qed
qed

Chapter 4

The Binary Choice Operators

4.1 Deterministic Choice

4.1.1 The Det Operator Definition

lift-definition *Det* :: ‹[('*a*, '*r*)*process*$_{ptick}$, ('*a*, '*r*)*process*$_{ptick}$] ⇒ ('*a*, '*r*)*process*$_{ptick}$›
(**infixl** ‹[+]› *82*)
is ‹λ*P Q*. ({(*s*, *X*). *s* = [] ∧ (*s*, *X*) ∈ ℱ *P* ∩ ℱ *Q*} ∪
{(*s*, *X*). *s* ≠ [] ∧ (*s*, *X*) ∈ ℱ *P* ∪ ℱ *Q*} ∪
{(*s*, *X*). *s* = [] ∧ *s* ∈ 𝒟 *P* ∪ 𝒟 *Q*} ∪
{(*s*, *X*). ∃ *r*. *s* = [] ∧ ✓(*r*) ∉ *X* ∧ [✓(*r*)] ∈ 𝒯 *P* ∪ 𝒯 *Q*},
𝒟 *P* ∪ 𝒟 *Q*)›
proof −
show ‹*?thesis P Q*› (**is** ‹*is-process* (*?f*, 𝒟 *P* ∪ 𝒟 *Q*)›) **for** *P Q* :: ‹('*a*, '*r*)*process*$_{ptick}$›
proof (*unfold is-process-def DIVERGENCES-def FAILURES-def fst-conv snd-conv*, *intro conjI allI impI*)
show ‹([], {}) ∈ *?f*›
by (*simp add*: *is-processT1*)
next
show ‹(*s*, *X*) ∈ *?f* ⟹ *front-tickFree s*› **for** *s X*
by (*auto simp*: *is-processT2*)
next
show ‹(*s* @ *t*, {}) ∈ *?f* ⟹ (*s*, {}) ∈ *?f*› **for** *s t*
by (*auto simp*: *is-processT1 dest*!: *is-processT3*[*rule-format*])
next
show ‹(*s*, *Y*) ∈ *?f* ∧ *X* ⊆ *Y* ⟹ (*s*, *X*) ∈ *?f*› **for** *s X Y*
by (*cases s*; *use is-processT4* **in** *blast*)
next
show ‹(*s*, *X*) ∈ *?f* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *?f*) ⟹ (*s*, *X* ∪ *Y*) ∈ *?f*›
for *s X Y*
by (*cases s*) (*auto simp*: *is-processT5 T-F*)
next
show ‹(*s* @ [✓(*r*)], {}) ∈ *?f* ⟹ (*s*, *X* − {✓(*r*)}) ∈ *?f*› **for** *s r X*

apply (*cases s*; *simp add*: *T-F-spec*)
by *blast* (*metis T-F-spec append-Cons is-processT6*)
next
show ‹$s \in \mathcal{D}\ P \cup \mathcal{D}\ Q \wedge$ *tickFree* $s \wedge$ *front-tickFree* $t \Longrightarrow s\ @\ t \in \mathcal{D}\ P \cup \mathcal{D}\ Q$› **for** $s\ t$
by (*auto simp*: *is-processT7*)
next
show ‹$s \in \mathcal{D}\ P \cup \mathcal{D}\ Q \Longrightarrow (s, X) \in ?f$› **for** $s\ X$
by (*auto simp*: *is-processT8*)
next
show ‹$s\ @\ [\checkmark(r)] \in \mathcal{D}\ P \cup \mathcal{D}\ Q \Longrightarrow s \in \mathcal{D}\ P \cup \mathcal{D}\ Q$› **for** $s\ r$
by (*meson UnCI UnE is-processT9*)
qed
qed

notation *Det* (**infixl** ‹$\Box$› *82*)

4.1.2 The Projections

lemma *F-Det* :
‹$\mathcal{F}\ (P \Box Q) = \{(s, X).\ s = [] \wedge (s, X) \in \mathcal{F}\ P \cap \mathcal{F}\ Q\} \cup$
$\{(s, X).\ s \neq [] \wedge (s, X) \in \mathcal{F}\ P \cup \mathcal{F}\ Q\} \cup$
$\{(s, X).\ s = [] \wedge s \in \mathcal{D}\ P \cup \mathcal{D}\ Q\} \cup$
$\{(s, X).\ \exists r.\ s = [] \wedge \checkmark(r) \notin X \wedge [\checkmark(r)] \in \mathcal{T}\ P \cup \mathcal{T}\ Q\}$›
by (*subst Failures.rep-eq*, *simp add*: *Det.rep-eq FAILURES-def*)

lemma *D-Det* : ‹$\mathcal{D}\ (P \Box Q) = \mathcal{D}\ P \cup \mathcal{D}\ Q$›
by (*subst Divergences.rep-eq*, *simp add*: *Det.rep-eq DIVERGENCES-def*)

lemma *T-Det* : ‹$\mathcal{T}\ (P \Box Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$›
by (*unfold Traces.rep-eq TRACES-def F-Det Failures.rep-eq*[*symmetric*])
(*simp add*: *T-F F-T set-eq-iff*, *metis F-T T-F is-processT1*)

lemmas *Det-projs* = *F-Det D-Det T-Det*

4.1.3 Basic Laws

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

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

lemma *Det-id* [*simp*] : ‹$P \Box P = P$›
by (*auto simp add*: *Process-eq-spec F-Det D-Det*
intro: *is-processT8 is-processT6-TR-notin*)

4.1.4 The Continuity-Rule

lemma *mono-Det* : ‹$(P :: ('a, 'r)process_{ptick}) \sqsubseteq P' \Longrightarrow Q \sqsubseteq Q' \Longrightarrow P \,\Box\, Q \sqsubseteq P' \,\Box\, Q'$›
proof −
have ‹$(P :: ('a, 'r)process_{ptick}) \,\Box\, S \sqsubseteq Q \,\Box\, S$› **if** ‹$P \sqsubseteq Q$› **for** *P S Q*
proof (*unfold le-approx-def*, *intro conjI impI allI subsetI*)
show ‹$s \in \mathcal{D}\ (Q \,\Box\, S) \Longrightarrow s \in \mathcal{D}\ (P \,\Box\, S)$› **for** *s*
apply (*simp add*: *D-Det*)
using *le-approx-imp-le-ref le-ref1 that* **by** *blast*
next
show ‹$s \notin \mathcal{D}\ (P \,\Box\, S) \Longrightarrow \mathcal{R}_a\ (P \,\Box\, S)\ s = \mathcal{R}_a\ (Q \,\Box\, S)\ s$› **for** *s*
apply (*cases s*; *simp add*: *D-Det Refusals-after-def F-Det set-eq-iff*)
apply (*meson front-tickFree-Nil in-mono is-processT9-tick le-approx1 le-approx2T proc-ord2a that*)
using *le-approx2 that* **by** *auto*[*1*]
next
show ‹$s \in min\text{-}elems\ (\mathcal{D}\ (P \,\Box\, S)) \Longrightarrow s \in \mathcal{T}\ (Q \,\Box\, S)$› **for** *s*
by (*simp add*: *min-elems-def D-Det T-Det*)
(*metis D-T UnCI elem-min-elems in-mono le-approx3 min-elems5 order-neq-le-trans that*)
qed

thus ‹$P \sqsubseteq P' \Longrightarrow Q \sqsubseteq Q' \Longrightarrow P \,\Box\, Q \sqsubseteq P' \,\Box\, Q'$›
by (*metis Det-commute below-trans*)
qed

lemma *chain-Det* : ‹$chain\ Y \Longrightarrow chain\ (\lambda i.\ Y\ i \,\Box\, S)$›
by (*simp add*: *po-class.chain-def mono-Det*)

lemma *cont-Det-prem* : ‹$((\bigsqcup\ i.\ Y\ i) \,\Box\, S) = (\bigsqcup\ i.\ (Y\ i \,\Box\, S))$› **if** ‹$chain\ Y$›
proof (*subst Process-eq-spec*, *safe*)
show ‹$(s, X) \in \mathcal{F}\ (Lub\ Y \,\Box\, S) \Longrightarrow (s, X) \in \mathcal{F}\ (\bigsqcup i.\ Y\ i \,\Box\, S)$› **for** *s X*
by (*auto simp add*: *limproc-is-thelub F-Det D-LUB F-LUB T-LUB chain-Det* ‹$chain\ Y$›)
next
show ‹$(s, X) \in \mathcal{F}\ (\bigsqcup i.\ Y\ i \,\Box\, S) \Longrightarrow (s, X) \in \mathcal{F}\ (Lub\ Y \,\Box\, S)$› **for** *s X*
using *le-approx2T*[*OF is-ub-thelub*[*OF* ‹$chain\ Y$›]]
apply (*cases s*; *simp add*: *limproc-is-thelub F-Det D-LUB F-LUB T-LUB chain-Det* ‹$chain\ Y$›)
by (*metis append-Nil is-processT8 is-processT9*)
next
show ‹$s \in \mathcal{D}\ (Lub\ Y \,\Box\, S) \Longrightarrow s \in \mathcal{D}\ (\bigsqcup i.\ Y\ i \,\Box\, S)$›
and ‹$s \in \mathcal{D}\ (\bigsqcup i.\ Y\ i \,\Box\, S) \Longrightarrow s \in \mathcal{D}\ (Lub\ Y \,\Box\, S)$› **for** *s*
by (*simp-all add*: *limproc-is-thelub D-Det D-LUB chain-Det* ‹$chain\ Y$›)
qed

lemma *Det-cont*[*simp*]: ‹$cont\ (\lambda x.\ f\ x \,\Box\, g\ x)$› **if** ‹$cont\ f$› **and** ‹$cont\ g$›

proof (*rule cont-apply*[**where** $f = \langle \lambda x\ g.\ f\ x\ \Box\ g \rangle$])
show ‹*cont g*› **by** (*fact* ‹*cont g*›)
next
show ‹*cont* ((□) (*f x*))› **for** *x*
by (*rule contI2*, *simp add*: *monofun-def mono-Det*)
(*subst* (*1 2*) *Det-commute*, *simp add*: *cont-Det-prem*)
next
show ‹*cont* (λ*x*. *f x* □ *y*)› **for** *y*
by (*rule contI2*, *simp add*: *monofun-def mono-Det cont2monofunE* ‹*cont f*›)
(*simp add*: *ch2ch-cont cont2contlubE cont-Det-prem* ‹*cont f*›)
qed

4.2 Non Deterministic Choice

4.2.1 The Ndet Operator Definition

lift-definition *Ndet* :: ‹[('*a*, '*r*)*process*$_{ptick}$, ('*a*, '*r*)*process*$_{ptick}$] ⇒ ('*a*, '*r*)*process*$_{ptick}$›
(**infixl** ‹|−|› *83*)
is ‹λ*P Q*. (*F P* ∪ *F Q* , *D P* ∪ *D Q*)›
proof −
show ‹*is-process* (*F P* ∪ *F Q* , *D P* ∪ *D Q*)› **for** *P Q* :: ‹('*a*, '*r*)*process*$_{ptick}$›
proof (*unfold is-process-def DIVERGENCES-def FAILURES-def fst-conv snd-conv*, *intro conjI allI impI*)
show ‹([], {}) ∈ *F P* ∪ *F Q*›
by (*simp add*: *is-processT1*)
next
show ‹(*s*, *X*) ∈ *F P* ∪ *F Q* ⟹ *front-tickFree s*› **for** *s X*
by (*auto intro*: *is-processT2*)
next
show ‹(*s* @ *t*, {}) ∈ *F P* ∪ *F Q* ⟹ (*s*, {}) ∈ *F P* ∪ *F Q*› **for** *s t*
by (*auto intro*: *is-processT3*)
next
show ‹(*s*, *Y*) ∈ *F P* ∪ *F Q* ∧ *X* ⊆ *Y* ⟹ (*s*, *X*) ∈ *F P* ∪ *F Q*› **for** *s X Y*
by (*auto intro*: *is-processT4*)
next
show ‹(*s*, *X*) ∈ *F P* ∪ *F Q* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *F P* ∪ *F Q*)
⟹ (*s*, *X* ∪ *Y*) ∈ *F P* ∪ *F Q*› **for** *s X Y* **by** (*auto intro*: *is-processT5*)
next
show ‹(*s* @ [✓(*r*)], {}) ∈ *F P* ∪ *F Q* ⟹ (*s*, *X* − {✓(*r*)}) ∈ *F P* ∪ *F Q*› **for** *s r X*
by (*auto intro*: *is-processT6*)
next
show ‹*s* ∈ *D P* ∪ *D Q* ∧ *tickFree s* ∧ *front-tickFree t* ⟹ *s* @ *t* ∈ *D P* ∪ *D Q*› **for** *s t*
by (*auto intro*: *is-processT7*)
next
show ‹*s* ∈ *D P* ∪ *D Q* ⟹ (*s*, *X*) ∈ *F P* ∪ *F Q*› **for** *s X*

by (*auto intro*: *is-processT8*)
next
show ‹*s* @ [✓(*r*)] ∈ $\mathcal{D}$ *P* ∪ $\mathcal{D}$ *Q* ⟹ *s* ∈ $\mathcal{D}$ *P* ∪ $\mathcal{D}$ *Q*› **for** *s r*
by (*auto intro*: *is-processT9*)
qed
qed

notation *Ndet* (**infixl** ‹⊓› *83*)

4.2.2 The Projections

lemma *F-Ndet* : ‹$\mathcal{F}$ (*P* ⊓ *Q*) = $\mathcal{F}$ *P* ∪ $\mathcal{F}$ *Q*›
by (*simp add*: *FAILURES-def Failures.rep-eq Ndet.rep-eq*)

lemma *D-Ndet* : ‹$\mathcal{D}$ (*P* ⊓ *Q*) = $\mathcal{D}$ *P* ∪ $\mathcal{D}$ *Q*›
by (*simp add*: *DIVERGENCES-def Divergences.rep-eq Ndet.rep-eq*)

lemma *T-Ndet* : ‹$\mathcal{T}$ (*P* ⊓ *Q*) = $\mathcal{T}$ *P* ∪ $\mathcal{T}$ *Q*›
by (*subst Traces.rep-eq*)
(*use T-F-spec* **in** ‹*auto simp*: *TRACES-def Failures.rep-eq*[*symmetric*] *F-Ndet F-T*›)

lemmas *Ndet-projs* = *F-Ndet D-Ndet T-Ndet*

4.2.3 Basic Laws

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

lemma *Ndet-commute*: ‹*P* ⊓ *Q* = *Q* ⊓ *P*›
by (*auto simp*: *Process-eq-spec F-Ndet D-Ndet*)

lemma *Ndet-id* [*simp*] : ‹*P* ⊓ *P* = *P*› **by** (*simp add*: *Process-eq-spec F-Ndet D-Ndet*)

4.2.4 The Continuity Rule

lemma *mono-Ndet* : ‹*P* ⊓ *Q* ⊑ *P'* ⊓ *Q'*› **if** ‹*P* ⊑ *P'*› **and** ‹*Q* ⊑ *Q'*›
proof (*unfold le-approx-def*, *intro conjI allI impI*)
show ‹$\mathcal{D}$ (*P'* ⊓ *Q'*) ⊆ $\mathcal{D}$ (*P* ⊓ *Q*)› **by** (*metis D-Ndet Un-mono le-approx1 that*)
next
show ‹*s* ∉ $\mathcal{D}$ (*P* ⊓ *Q*) ⟹ $\mathcal{R}_a$ (*P* ⊓ *Q*) *s* = $\mathcal{R}_a$ (*P'* ⊓ *Q'*) *s*› **for** *s*
using *that*[*THEN le-approx2*] **by** (*simp add*: *D-Ndet Refusals-after-def F-Ndet*)
next
show ‹*min-elems* ($\mathcal{D}$ (*P* ⊓ *Q*)) ⊆ $\mathcal{T}$ (*P'* ⊓ *Q'*)›
using *that*[*THEN le-approx3*] **by** (*auto simp add*: *min-elems-def D-Ndet T-Ndet*)
qed

lemma *cont-Ndet-prem* : ‹$(\bigsqcup\ i.\ Y\ i) \sqcap S = (\bigsqcup\ i.\ Y\ i \sqcap S)$› **if** ‹*chain* Y›
proof −
have ‹*chain* $(\lambda i.\ Y\ i \sqcap S)$›
by (*rule chainI*) (*simp add*: ‹*chain* Y› *chainE mono-Ndet*)
with ‹*chain* Y› **show** *?thesis*
by (*simp add*: *limproc-is-thelub Process-eq-spec D-Ndet F-Ndet F-LUB D-LUB*)
qed

lemma *Ndet-cont*[*simp*]: ‹*cont* $(\lambda x.\ f\ x \sqcap g\ x)$› **if** ‹*cont* f› **and** ‹*cont* g›
proof (*rule cont-apply*[**where** $f = ‹\lambda x\ g.\ f\ x \sqcap g›$])
show ‹*cont* g› **by** (*fact* ‹*cont* g›)
next
show ‹*cont* $((\sqcap)\ (f\ x))$› **for** x
by (*rule contI2*, *simp add*: *monofun-def mono-Ndet*)
(*subst* (*1 2*) *Ndet-commute*, *simp add*: *cont-Ndet-prem*)
next
show ‹*cont* $(\lambda x.\ f\ x \sqcap y)$› **for** y
by (*rule contI2*, *simp add*: *monofun-def mono-Ndet cont2monofunE* ‹*cont* f›)
(*simp add*: *ch2ch-cont cont2contlubE cont-Ndet-prem* ‹*cont* f›)
qed

4.3 Sliding Choice

4.3.1 Definition

definition *Sliding* :: ‹$('a, 'r)process_{ptick} \Rightarrow ('a, 'r)process_{ptick} \Rightarrow ('a, 'r)process_{ptick}$›
(**infixl** ‹$\rhd$› *80*)
where ‹$P \rhd Q \equiv (P \Box Q) \sqcap Q$›

4.3.2 Projections

lemma *F-Sliding*:
‹$\mathcal{F}\ (P \rhd Q) = \mathcal{F}\ Q \cup \{(s, X).\ s \neq [] \wedge (s, X) \in \mathcal{F}\ P \vee$
$s = [] \wedge (\exists r.\ s \in \mathcal{D}\ P \vee tick\ r \notin X \wedge [tick\ r] \in \mathcal{T}\ P)\}$›
by (*auto simp add*: *Sliding-def F-Ndet F-Det intro*: *is-processT8 is-processT6-TR-notin*)

lemma *D-Sliding*: ‹$\mathcal{D}\ (P \rhd Q) = \mathcal{D}\ P \cup \mathcal{D}\ Q$›
by (*simp add*: *Sliding-def D-Ndet D-Det*)

lemma *T-Sliding*: ‹$\mathcal{T}\ (P \rhd Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$›
by (*simp add*: *Sliding-def T-Ndet T-Det*)

lemmas *Sliding-projs* = *F-Sliding D-Sliding T-Sliding*

4.3.3 Properties

lemma *Sliding-id*: ‹$P \triangleright P = P$›
by (*simp add*: *Sliding-def*)

4.3.4 Continuity

From the definition, monotony and continuity is obvious.

lemma *mono-Sliding* : ‹$P \sqsubseteq P' \Longrightarrow Q \sqsubseteq Q' \Longrightarrow P \triangleright Q \sqsubseteq P' \triangleright Q'$›
unfolding *Sliding-def* **by** (*simp add*: *mono-Det mono-Ndet*)

lemma *Sliding-cont*[*simp*] : ‹$cont\ f \Longrightarrow cont\ g \Longrightarrow cont\ (\lambda x.\ f\ x \triangleright g\ x)$›
by (*simp add*: *Sliding-def*)

Chapter 5

The Prefix Choice Operators

5.1 Multiple Deterministic Prefix Choice

5.1.1 The Definition and some Consequences

lift-definition *Mprefix* :: ‹['a *set*, 'a ⇒ ('a, 'r)*process*$_{ptick}$] ⇒ ('a, 'r)*process*$_{ptick}$›
is ‹λ*A P*. ({(*tr*, *ref*). *tr* = [] ∧ *ref* ∩ *ev* ‘ *A* = {}} ∪
{(*tr*, *ref*). *tr* ≠ [] ∧ *hd tr* ∈ *ev* ‘ *A* ∧ (∃ *a*. *ev a* = *hd tr* ∧ (*tl tr*, *ref*) ∈ $\mathcal{F}$ (*P a*))},
{*d*. *d* ≠ [] ∧ *hd d* ∈ *ev* ‘ *A* ∧ (∃ *a*. *ev a* = *hd d* ∧ *tl d* ∈ $\mathcal{D}$ (*P a*))})›
proof −
show ‹*?thesis A P*› (**is** ‹*is-process*(*?f*, *?d*)›) **for** *P* **and** *A*
proof (*unfold is-process-def DIVERGENCES-def FAILURES-def fst-conv snd-conv*, *intro conjI allI impI*)
show ‹([], {}) ∈ *?f*›
by (*simp add*: *is-processT1*)
next
show ‹(*s*, *X*) ∈ *?f* ⟹ *front-tickFree s*› **for** *s X*
apply (*cases s*; *simp add*: *image-iff*)
by (*meson F-imp-front-tickFree event*$_{ptick}$.*disc*(*1*) *front-tickFree-Cons-iff*)
next
show ‹(*s* @ *t*, {}) ∈ *?f* ⟹ (*s*, {}) ∈ *?f*› **for** *s t*
by (*auto intro*: *is-processT3*)
next
show ‹(*s*, *Y*) ∈ *?f* ∧ *X* ⊆ *Y* ⟹ (*s*, *X*) ∈ *?f*› **for** *s X Y*
by (*auto intro*: *is-processT4*)
next
show ‹(*s*, *X*) ∈ *?f* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *?f*) ⟹ (*s*, *X* ∪ *Y*) ∈ *?f*›
for *s X Y*
apply (*cases s*; *simp add*: *disjoint-iff image-iff*)
using *is-processT1* **apply** *blast*
by (*metis is-processT5*)
next
show ‹(*s* @ [✓(*r*)], {}) ∈ *?f* ⟹ (*s*, *X* − {✓(*r*)}) ∈ *?f*› **for** *s r X*
by (*cases s*) (*auto intro*: *is-processT6*)

next
show ‹$s \in ?d \wedge$ *tickFree* $s \wedge$ *front-tickFree* $t \Longrightarrow s$ @ $t \in ?d$› **for** s t
by (*cases s*) (*auto intro*: *is-processT7*)
next
show ‹$s \in ?d \Longrightarrow (s, X) \in ?f$› **for** s X
by (*auto intro*: *is-processT8*)
next
show ‹s @ $[\checkmark(r)] \in ?d \Longrightarrow s \in ?d$› **for** s r
by (*cases s*) (*auto intro*: *is-processT9*)
qed
qed

syntax -*Mprefix* :: ‹[*pttrn*, *'a set*, (*'a*, *'r*)*process*$_{ptick}$] ⇒ (*'a*, *'r*)*process*$_{ptick}$›
(‹(*3*□((-)/∈(-))/ → (-))› [*78*,*78*,*77*] *77*)
syntax-consts -*Mprefix* ⇌ *Mprefix*
translations □*a*∈*A* → *P* ⇌ *CONST Mprefix A* (λ*a*. *P*)

Syntax Check:

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

5.1.2 Projections in Prefix

lemma *F-Mprefix* :
$\mathcal{F}$ (□$a \in A \to P\ a$) = {([], *X*)| *X*. *X* ∩ *ev* ‘ *A* = {}} ∪
{(*ev a* # *s*, *X*)| *a s X*. *a* ∈ *A* ∧ (*s*, *X*) ∈ $\mathcal{F}$ (*P a*)}
by (*subst Failures.rep-eq*, *auto simp add*: *Mprefix.rep-eq FAILURES-def*)
(*metis list.collapse*)

lemma *D-Mprefix*: ‹$\mathcal{D}$ (□$a \in A \to P\ a$) = {*ev a* # *s*| *a s*. *a* ∈ *A* ∧ *s* ∈ $\mathcal{D}$ (*P a*)}›
by (*subst Divergences.rep-eq*, *auto simp add*: *Mprefix.rep-eq DIVERGENCES-def*)
(*metis list.collapse*)

lemma *T-Mprefix*: ‹$\mathcal{T}$ (□$a \in A \to P\ a$) = *insert* [] {*ev a* # *s*| *a s*. *a* ∈ *A* ∧ *s* ∈ $\mathcal{T}$ (*P a*)}›
by (*subst Traces.rep-eq*, *auto simp add*: *TRACES-def Failures.rep-eq*[*symmetric*] *F-Mprefix*)
(*use F-T T-F* **in** *blast*)+

lemmas *Mprefix-projs* = *F-Mprefix D-Mprefix T-Mprefix*

lemma *mono-Mprefix-eq*: ‹(⋀*a*. *a* ∈ *A* ⟹ *P a* = *Q a*) ⟹ *Mprefix A P* = *Mprefix A Q*›
by (*subst Process-eq-spec*) (*auto simp add*: *F-Mprefix D-Mprefix*)

5.1.3 Basic Properties

lemma *tick-notin-T-Mprefix* [*simp*]: ‹$[\checkmark(r)] \notin \mathcal{T}$ (□$x \in A \to P\ x$)›

by (*simp add*: *T-Mprefix*)

lemma *Nil-notin-D-Mprefix* [*simp*]: ‹[] ∉ $\mathcal{D}$ (□*x* ∈ *A* → *P x*)›
by (*simp add*: *D-Mprefix*)

5.1.4 Proof of Continuity Rule

Backpatch Isabelle 2009-1

definition
contlub :: ('*a*::*cpo* ⇒ '*b*::*cpo*) ⇒ *bool*
where
contlub f = (∀ *Y*. *chain Y* ⟶ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*)))

lemma *contlubE*:
⟦*contlub f*; *chain Y*⟧ ⟹ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*))
by (*simp add*: *contlub-def*)

lemma *monocontlub2cont*: ⟦*monofun f*; *contlub f*⟧ ⟹ *cont f*
apply (*rule contI*)
apply (*rule thelubE*)
apply (*erule* (*1*) *ch2ch-monofun*)
apply (*erule* (*1*) *contlubE* [*symmetric*])
done

lemma *contlubI*:
(⋀*Y*. *chain Y* ⟹ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*))) ⟹ *contlub f*
by (*simp add*: *contlub-def*)

lemma *cont2contlub*: *cont f* ⟹ *contlub f*
apply (*rule contlubI*)
apply (*rule po-class.lub-eqI* [*symmetric*])
apply (*erule* (*1*) *contE*)
done

Core of Proof

lemma *mono-Mprefix* : ‹□*a* ∈ *A* → *P a* ⊑ □*a* ∈ *A* → *Q a*› (**is** ‹*?P* ⊑ *?Q*›)
if ‹⋀*a*. *a* ∈ *A* ⟹ *P a* ⊑ *Q a*›
proof (*unfold le-approx-def*, *intro conjI impI allI subsetI*)
from *that*[*THEN le-approx1*] **show** ‹*s* ∈ $\mathcal{D}$ *?Q* ⟹ *s* ∈ $\mathcal{D}$ *?P*› **for** *s*
by (*auto simp add*: *D-Mprefix*)
next
from *that*[*THEN le-approx2*] **show** ‹*s* ∉ $\mathcal{D}$ *?P* ⟹ $\mathcal{R}_a$ *?P s* = $\mathcal{R}_a$ *?Q s*› **for** *s*
by (*auto simp add*: *Refusals-after-def D-Mprefix F-Mprefix*)
next
from *that*[*THEN le-approx3*] **show** ‹*s* ∈ *min-elems* ($\mathcal{D}$ *?P*) ⟹ *s* ∈ $\mathcal{T}$ *?Q*› **for** *s*

by (*simp add*: *min-elems-def D-Mprefix T-Mprefix subset-iff*) (*metis less-cons*)
qed

lemma *chain-Mprefix* : ‹*chain Y* ⟹ *chain* (λ*i*. □*a*∈*A* → *Y i a*)›
by (*simp add*: *fun-belowD mono-Mprefix chain-def*)

lemma *cont-Mprefix-prem* : ‹□*a* ∈ *A* → (⨆ *i*. *Y i a*) = (⨆ *i*. □*a* ∈ *A* → *Y i a*)›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain Y*›
proof (*subst Process-eq-spec*, *safe*)
show ‹(*s*, *X*) ∈ ℱ *?lhs* ⟹ (*s*, *X*) ∈ ℱ *?rhs*› **for** *s X*
by (*auto simp add*: *F-Mprefix limproc-is-thelub* ‹*chain Y*› *F-LUB ch2ch-fun chain-Mprefix*)
next
show ‹(*s*, *X*) ∈ ℱ *?rhs* ⟹ (*s*, *X*) ∈ ℱ *?lhs*› **for** *s X*
by (*simp add*: *F-Mprefix limproc-is-thelub* ‹*chain Y*› *F-LUB ch2ch-fun chain-Mprefix*) *blast*
next
show ‹*s* ∈ 𝒟 *?lhs* ⟹ *s* ∈ 𝒟 *?rhs*› **for** *s*
by (*auto simp add*: *D-Mprefix limproc-is-thelub* ‹*chain Y*› *D-LUB ch2ch-fun chain-Mprefix*)
next
show ‹*s* ∈ 𝒟 *?rhs* ⟹ *s* ∈ 𝒟 *?lhs*› **for** *s*
by (*simp add*: *D-Mprefix limproc-is-thelub* ‹*chain Y*› *D-LUB ch2ch-fun chain-Mprefix*) *blast*
qed

lemma *Mprefix-cont* [*simp*] : ‹*cont* (λ*b*. □*a* ∈ *A* → *f a b*)› **if** ∗ : ‹⋀*a*. *a* ∈ *A* ⟹ *cont* (*f a*)›
proof −
define *g* **where** ‹*g a b* ≡ *if a* ∈ *A then f a b else undefined*› **for** *a b*
have ‹(λ*b*. □*a* ∈ *A* → *f a b*) = (λ*b*. □*a* ∈ *A* → *g a b*)›
by (*intro ext mono-Mprefix-eq*) (*simp add*: *g-def*)
moreover have ‹*cont* (λ*b*. □*a* ∈ *A* → *g a b*)›
proof (*rule cont-compose*[**where** *c* = ‹*Mprefix A*›])
show ‹*cont* (*Mprefix A*)›
proof (*rule contI2*)
show ‹*monofun* (*Mprefix A*)› **by** (*simp add*: *fun-belowD mono-Mprefix mono-funI*)
next
show ‹*chain Y* ⟹ *Mprefix A* (⨆ *i*. *Y i*) ⊑ (⨆ *i*. *Mprefix A* (*Y i*))›
for *Y* :: ‹*nat* ⇒ *'a* ⇒ (*'a*, *'r*)*process*$_{ptick}$›
by (*simp add*: *cont-Mprefix-prem lub-fun*)
qed
next
show ‹*cont* (λ*b a*. *g a b*)›
by (*rule cont2cont-lambda*, *rule contI2*)

(*simp-all add*: *cont2monofunE cont2contlubE g-def monofunI* ∗)
qed
ultimately show ‹*cont* (λ*b*. □*a*∈*A* → *f a b*)› **by** *simp*
qed

5.1.5 High-level Syntax for Read and Write0

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* ⇒ '*b*, '*a set*, '*a* ⇒ ('*b*, '*r*)*process*$_{ptick}$] ⇒ ('*b*, '*r*)*process*$_{ptick}$›
where ‹*read c A P* ≡ *Mprefix* (*c* ‘ *A*) (*P* ∘ (*inv-into A c*))›

definition *write0* :: ‹['*a*, ('*a*, '*r*)*process*$_{ptick}$] ⇒ ('*a*, '*r*)*process*$_{ptick}$› (**infixr** ‹→› *77*)
where ‹*write0 a P* ≡ *Mprefix* {*a*} (λ*x*. *P*)›

5.1.6 CSP$_M$-Style Syntax for Communication Primitives

syntax
-read :: ‹['*a* ⇒ '*b*, *pttrn*, ('*b*, '*r*)*process*$_{ptick}$] ⇒ ('*b*, '*r*)*process*$_{ptick}$›
(‹(*3*((-)**?**(-)) /→ -)› [*78*,*78*,*77*] *77*)
-readX :: ‹['*a* ⇒ '*b*, *pttrn*, *bool*, ('*b*, '*r*)*process*$_{ptick}$] ⇒ ('*b*, '*r*)*process*$_{ptick}$›
(‹(*3*((-)**?**(-))|- /→ -)› [*78*,*78*,*78*,*77*] *77*)
-readS :: ‹['*a* ⇒ '*b*, *pttrn*, '*a set*, ('*b*, '*r*)*process*$_{ptick}$] ⇒ ('*b*, '*r*)*process*$_{ptick}$›
(‹(*3*((-)**?**(-))∈- /→ -)› [*78*,*78*,*78*,*77*] *77*)

translations
-read c p P ⇌ *CONST read c CONST UNIV* (λ*p*. *P*)
-readX c p b P => *CONST read c* {*p*. *b*} (λ*p*. *P*)

-readS c p A P => *CONST read c A* (λ*p*. *P*)

syntax-consts *-read* ⇌ *read*
and *-readX* ⇌ *read*
and *-readS* ⇌ *read*

Syntax Check:

term ‹*a* → *P*›

term ‹*c***?***x* → *d***?***y* → *P a y*›
term ‹*c***?***x*∈*X* → *P x*›
term ‹*c***?***x*|(*x*<*0*) → *P x*›

term ‹*c***?***x* → *d***?***y*∈*B* → *e* → *u***?***t*|(*t* ≥ *1*) → *P a y*›

term ‹$(c \circ d)?a \rightarrow P\ a$›

lemma *mono-write0* : ‹$P \sqsubseteq Q \Longrightarrow a \rightarrow P \sqsubseteq a \rightarrow Q$›
unfolding *write0-def* **by** (*simp add*: *mono-Mprefix*)

lemma *mono-read* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq Q\ a) \Longrightarrow c?a{\in}A \rightarrow P\ a \sqsubseteq c?a{\in}A \rightarrow Q\ a$›
unfolding *read-def* **by** (*simp add*: *inv-into-into mono-Mprefix*)

lemma *read-cont*[*simp*]:
‹$(\bigwedge a.\ a \in A \Longrightarrow cont\ (\lambda b.\ P\ b\ a)) \Longrightarrow cont\ (\lambda y.\ read\ c\ A\ (P\ y))$›
unfolding *read-def o-def* **by** (*rule Mprefix-cont*) (*simp add*: *inv-into-into*)

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*]: ‹$(\bigwedge a.\ cont\ (f\ a)) \Longrightarrow cont\ (\lambda y.\ c?x \rightarrow f\ x\ y)$› **by** *simp*

lemma *write0-cont*[*simp*]: ‹$cont\ (P{::}('b{::}cpo \Rightarrow ('a,\ 'r)process_{ptick})) \Longrightarrow cont(\lambda x.\ a \rightarrow P\ x)$›
by (*simp add*: *write0-def*)

lemma *Mprefix-singl* : ‹$\Box x \in \{a\} \rightarrow P\ x = a \rightarrow P\ a$›
by (*auto simp add*: *Process-eq-spec write0-def Mprefix-projs*)

5.2 Multiple non Deterministic Prefix Choice

5.3 Multiple non deterministic prefix operator

lift-definition *Mndetprefix* :: ‹$['a\ set,\ 'a \Rightarrow ('a,\ 'r)process_{ptick}] \Rightarrow ('a,\ 'r)process_{ptick}$›
is ‹$\lambda A\ P.\quad if\ A = \{\}\ then\ process_0\text{-}of\text{-}process\ STOP$
$\quad else\ (\bigsqcup a{\in}A.\ \mathcal{F}\ (a \rightarrow P\ a),\ \bigsqcup a{\in}A.\ \mathcal{D}\ (a \rightarrow P\ a))$›
proof (*split if-split*, *intro conjI impI*)
show ‹*is-process* ($process_0$*-of-process STOP*)›
by (*metis STOP.rep-eq STOP.rsp eq-onp-def*)
next
show ‹*is-process* $(\bigsqcup a{\in}A.\ \mathcal{F}\ (a \rightarrow P\ a),\ \bigsqcup a{\in}A.\ \mathcal{D}\ (a \rightarrow P\ a))$› (**is** ‹*is-process* (*?f*, *?d*)›)
if ‹$A \neq \{\}$› **for** A **and** P :: ‹$'a \Rightarrow ('a,\ 'r)process_{ptick}$›
proof (*unfold is-process-def DIVERGENCES-def FAILURES-def fst-conv snd-conv*,

intro conjI allI impI)
show ‹([], {}) ∈ *?f*› **by** (*simp add*: *ex-in-conv is-processT1* ‹*A* ≠ {}›)
next
show ‹(*s*, *X*) ∈ *?f* ⟹ *front-tickFree s*› **for** *s X*
by (*blast intro*: *is-processT2*)
next
show ‹(*s* @ *t*, {}) ∈ *?f* ⟹ (*s*, {}) ∈ *?f*› **for** *s t*
by (*blast intro*: *is-processT3*)
next
show ‹(*s*, *Y*) ∈ *?f* ∧ *X* ⊆ *Y* ⟹ (*s*, *X*) ∈ *?f*› **for** *s X Y*
by (*blast intro*: *is-processT4*)
next
show ‹(*s*, *X*) ∈ *?f* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *?f*) ⟹
(*s*, *X* ∪ *Y*) ∈ *?f*› **for** *s X Y* **by** (*blast intro*!: *is-processT5*)
next
show ‹(*s* @ [✓(*r*)], {}) ∈ *?f* ⟹ (*s*, *X* − {✓(*r*)}) ∈ *?f*› **for** *s r X*
by (*blast intro*: *is-processT6*)
next
show ‹*s* ∈ *?d* ∧ *tickFree s* ∧ *front-tickFree t* ⟹ *s* @ *t* ∈ *?d*› **for** *s t*
by (*blast intro*: *is-processT7*)
next
show ‹*s* ∈ *?d* ⟹ (*s*, *X*) ∈ *?f*› **for** *s X*
by (*blast intro*!: *is-processT8*)
next
show ‹*s* @ [✓(*r*)] ∈ *?d* ⟹ *s* ∈ *?d*› **for** *s r*
by (*blast intro*!: *is-processT9*)
qed
qed

syntax -*Mndetprefix* :: *pttrn* ⇒ *'a set* ⇒ (*'a*, *'r*)*process*$_{ptick}$ ⇒ (*'a*, *'r*)*process*$_{ptick}$

(‹(*3*⊓((-)/∈(-))/ → (-))› [*78*,*78*,*77*] *77*)
syntax-consts -*Mndetprefix* ⇌ *Mndetprefix*

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

lemma *F-Mndetprefix*:
‹*F* (⊓*a* ∈ *A* → *P a*) = (*if A* = {} *then* {(*s*, *X*). *s* = []} *else* ⋃*x*∈*A*. *F* (*x* → *P x*))›
by (*simp add*: *Failures.rep-eq FAILURES-def STOP.rep-eq Mndetprefix.rep-eq*)

lemma *D-Mndetprefix* : ‹*D* (⊓*a* ∈ *A* → *P a*) = (*if A* = {} *then* {} *else* ⋃*x*∈*A*. *D* (*x* → *P x*))›
by (*simp add*: *Divergences.rep-eq DIVERGENCES-def STOP.rep-eq Mndetpre-*

fix.rep-eq)

lemma *T-Mndetprefix*: ‹$\mathcal{T}$ ($\sqcap a \in A \to P\ a$) = (*if* $A = \{\}$ *then* $\{[]\}$ *else* $\bigcup x{\in}A.$ $\mathcal{T}$ ($x \to P\ x$))›
by (*auto simp add*: *Traces.rep-eq TRACES-def Failures.rep-eq*[*symmetric*] *F-Mndetprefix*)

lemma *mono-Mndetprefix-eq*: ‹($\bigwedge a.\ a \in A \Longrightarrow P\ a = Q\ a$) $\Longrightarrow$ $\sqcap a \in A \to P\ a$ = $\sqcap a \in A \to Q\ a$›
by (*cases* ‹$A = \{\}$›, *simp*; *subst Process-eq-spec*, *auto simp add*: *F-Mndetprefix D-Mndetprefix*)

lemma *F-Mndetprefix'* :
‹$\mathcal{F}$ ($\sqcap a \in A \to P\ a$) =
(*if* $A = \{\}$ *then* $\{(s, X).\ s = []\}$
else $\{([], X) |\ X.\ \exists a \in A.\ ev\ a \notin X\} \cup \{(ev\ a \#\ s, X)\ |a\ s\ X.\ a \in A \wedge (s, X) \in \mathcal{F}\ (P\ a)\})$›
by (*simp add*: *F-Mndetprefix write0-def F-Mprefix*) *blast*

lemma *D-Mndetprefix'* : ‹$\mathcal{D}$ ($\sqcap a \in A \to P\ a$) = $\{ev\ a \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{D}\ (P\ a)\}$›
by (*auto simp add*: *D-Mndetprefix write0-def D-Mprefix*)

lemma *T-Mndetprefix'* : ‹$\mathcal{T}$ ($\sqcap a \in A \to P\ a$) = *insert* [] $\{ev\ a \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{T}\ (P\ a)\}$›
by (*auto simp add*: *T-Mndetprefix write0-def T-Mprefix*)

lemmas *Mndetprefix-projs* = *F-Mndetprefix' D-Mndetprefix' T-Mndetprefix'*

Thus we know now, that Mndetprefix yields processes. Direct consequences are the following distributivities:

lemma *Mndetprefix-singl* [*simp*] : ‹$\sqcap\ a \in \{a\} \to P\ a = a \to P\ a$›
by (*auto simp add*: *Process-eq-spec F-Mndetprefix D-Mndetprefix*)

lemma *Mndetprefix-Un-distrib* :
‹$A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow \sqcap x \in (A \cup B) \to P\ x = (\sqcap a \in A \to P\ a) \sqcap (\sqcap b \in B \to P\ b)$›
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-singl*)

This also implies that *Mndetprefix* is continuous when *finite A*, but this is not really useful since we have the general case.

5.3.1 General case Continuity

lemma *mono-Mndetprefix* : ‹$\sqcap a \in A \to P\ a \sqsubseteq \sqcap a \in A \to Q\ a$›
(**is** ‹*?P* $\sqsubseteq$ *?Q*›) **if** ‹$\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq Q\ a$›
proof (*unfold le-approx-def*, *intro conjI impI allI subsetI*)
from *that*[*THEN le-approx1*] **show** ‹$s \in \mathcal{D}$ *?Q* $\Longrightarrow s \in \mathcal{D}$ *?P*› **for** *s*
by (*auto simp add*: *D-Mndetprefix'*)
next
from *that*[*THEN le-approx2*] **show** ‹$s \notin \mathcal{D}$ *?P* $\Longrightarrow \mathcal{R}_a$ *?P s* $= \mathcal{R}_a$ *?Q s*› **for** *s*
by (*auto simp add*: *Refusals-after-def D-Mndetprefix' F-Mndetprefix'*)
next
from *that*[*THEN le-approx3*] **show** ‹*s* $\in$ *min-elems* ($\mathcal{D}$ *?P*) $\Longrightarrow s \in \mathcal{T}$ *?Q*› **for** *s*
by (*simp add*: *min-elems-def D-Mndetprefix' T-Mndetprefix' subset-iff*) (*metis less-cons*)
qed

lemma *chain-Mndetprefix* : ‹*chain Y* $\Longrightarrow$ *chain* ($\lambda i.\ \sqcap a \in A \to Y\ i\ a$)›
by (*simp add*: *fun-belowD mono-Mndetprefix chain-def*)

lemma *cont-Mndetprefix-prem* : ‹$\sqcap a \in A \to (\bigsqcup i.\ Y\ i\ a) = (\bigsqcup i.\ \sqcap a \in A \to Y\ i\ a)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain Y*›
proof (*subst Process-eq-spec*, *safe*)
show ‹$(s, X) \in \mathcal{F}$ *?lhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*› **for** *s X*
by (*auto simp add*: *F-Mndetprefix' limproc-is-thelub* ‹*chain Y*› *F-LUB ch2ch-fun chain-Mndetprefix split*: *if-split-asm*)
next
show ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*› **for** *s X*
by (*simp add*: *F-Mndetprefix' limproc-is-thelub* ‹*chain Y*› *F-LUB ch2ch-fun chain-Mndetprefix split*: *if-split-asm*) *blast*
next
show ‹*s* $\in \mathcal{D}$ *?lhs* $\Longrightarrow s \in \mathcal{D}$ *?rhs*› **for** *s*
by (*auto simp add*: *D-Mndetprefix' limproc-is-thelub* ‹*chain Y*› *D-LUB ch2ch-fun chain-Mndetprefix*)
next
show ‹*s* $\in \mathcal{D}$ *?rhs* $\Longrightarrow s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*simp add*: *D-Mndetprefix' limproc-is-thelub* ‹*chain Y*› *D-LUB ch2ch-fun chain-Mndetprefix*) *blast*
qed

lemma *Mndetprefix-cont* [*simp*] : ‹*cont* ($\lambda b.\ \sqcap a \in A \to f\ a\ b$)› **if** $*$: ‹$\bigwedge a.\ a \in A \Longrightarrow$ *cont* ($f\ a$)›
proof −

define *g* **where** ‹*g a b* ≡ *if a* ∈ *A then f a b else undefined*› **for** *a b*
have ‹(λ*b*. ⊓*a* ∈ *A* → *f a b*) = (λ*b*. ⊓*a* ∈ *A* → *g a b*)›
by (*intro ext mono-Mndetprefix-eq*) (*simp add*: *g-def*)
moreover have ‹*cont* (λ*b*. ⊓*a* ∈ *A* → *g a b*)›
proof (*rule cont-compose*[**where** *c* = ‹*Mndetprefix A*›])
show ‹*cont* (*Mndetprefix A*)›
proof (*rule contI2*)
show ‹*monofun* (*Mndetprefix A*)› **by** (*simp add*: *fun-belowD mono-Mndetprefix monofunI*)
next
show ‹*chain Y* ⟹ *Mndetprefix A* (⨆ *i*. *Y i*) ⊑ (⨆ *i*. *Mndetprefix A* (*Y i*))›
for *Y* :: ‹*nat* ⇒ 'a ⇒ ('a, 'r)*process*$_{ptick}$›
by (*simp add*: *cont-Mndetprefix-prem lub-fun*)
qed
next
show ‹*cont* (λ*b a*. *g a b*)›
by (*rule cont2cont-lambda*, *rule contI2*)
(*simp-all add*: *cont2monofunE cont2contlubE g-def monofunI* ∗)
qed
ultimately show ‹*cont* (λ*b*. ⊓*a*∈*A* → *f a b*)› **by** *simp*
qed

5.3.2 High-level Syntax for Write

A version with a non deterministic choice is also introduced.

definition *ndet-write* :: ‹['a ⇒ 'b, 'a *set*, 'a ⇒ ('b, 'r)*process*$_{ptick}$] ⇒ ('b, 'r)*process*$_{ptick}$›
where ‹*ndet-write c A P* ≡ *Mndetprefix* (*c* ‘ *A*) (*P o* (*inv-into A c*))›

definition *write* :: ‹['a ⇒ 'b, 'a, ('b, 'r)*process*$_{ptick}$] ⇒ ('b, 'r)*process*$_{ptick}$›
where ‹*write c a P* ≡ *Mprefix* {*c a*} (λ*x*. *P*)›

syntax
-ndet-write :: ‹['a ⇒ 'b, *pttrn*, ('b, 'r)*process*$_{ptick}$] ⇒ ('b, 'r)*process*$_{ptick}$›
(‹(*3*((-)**!!**(-)) /→ -)› [*78*,*78*,*77*] *77*)
-ndet-writeX :: ‹['a ⇒ 'b, *pttrn*, *bool*,('b, 'r)*process*$_{ptick}$] ⇒ ('b, 'r)*process*$_{ptick}$›
(‹(*3*((-)**!!**(-))|- /→ -)› [*78*,*78*,*78*,*77*] *77*)
-ndet-writeS :: ‹['a ⇒ 'b, *pttrn*, 'b *set*,('b, 'r)*process*$_{ptick}$] ⇒ ('b, 'r)*process*$_{ptick}$›
(‹(*3*((-)**!!**(-))∈- /→ -)› [*78*,*78*,*78*,*77*] *77*)
-write :: ‹['a ⇒ 'b, 'a, ('a, 'r)*process*$_{ptick}$] ⇒ ('a, 'r)*process*$_{ptick}$› (‹(*3*(-)**!**(-) /→ -)› [*78*,*78*,*77*] *77*)

syntax-consts *-ndet-write* ⇌ *ndet-write*
and *-ndet-writeX* ⇌ *ndet-write*
and *-ndet-writeS* ⇌ *ndet-write*
and *-write* ⇌ *ndet-write*

translations

-ndet-write c p P $\rightleftharpoons$ *CONST ndet-write c CONST UNIV* ($\lambda p.$ *P*)
-ndet-writeX c p b P => *CONST ndet-write c* {*p. b*} ($\lambda p.$ *P*)
-ndet-writeS c p A P => *CONST ndet-write c A* ($\lambda p.$ *P*)
-write c a P $\rightleftharpoons$ *CONST write c a P*

Syntax checks.

term ‹*c*!!*x* → *P x*›
term ‹*c*!!*x*∈*A* → *P x*›
term ‹*c*!!*x*|(*0*<*x*) → *P x*›

term ‹(*c* ∘ *c'*)!!*a*∈*A* → *d*?*b*∈*B* → *event* → *e*!*a'* → *P a b*›

term ‹*c*!*x* → *P*›

lemma *mono-ndet-write*: ‹(⋀*a. a* ∈ *A* ⟹ *P a* ⊑ *Q a*) ⟹ (*c*!!*a*∈*A* → *P a*) ⊑ (*c*!!*a*∈*A* → *Q a*)›
unfolding *ndet-write-def* **by** (*simp add*: *inv-into-into mono-Mndetprefix*)

lemma *ndet-write-cont*[*simp*]:
‹(⋀*a. a* ∈ *A* ⟹ *cont* (λ*b. P b a*)) ⟹ *cont* (λ*y. c*!!*a*∈*A* → *P y a*)›
unfolding *ndet-write-def o-def* **by** (*rule Mndetprefix-cont*) (*simp add*: *inv-into-into*)

lemma *mono-write*: ‹*P* ⊑ *Q* ⟹ *c*!*a* → *P* ⊑ *c*!*a* → *Q*›
unfolding *write-def* **by** (*simp add*: *mono-Mprefix*)

lemma *write-cont*[*simp*]: ‹*cont P* ⟹ *cont* (λ*x. c*!*a* → *P x*)›
unfolding *write-def* **by** *simp*

lemma *read-singl*[*simp*] : ‹*c*?*a*∈{*x*} → *P a* = *c*!*x* → *P x*›
by (*simp add*: *read-def Mprefix-singl write-def*)

lemma *ndet-write-singl*[*simp*] : ‹*c*!!*a*∈{*x*} → *P a* = *c*!*x* → *P x*›
by (*simp add*: *ndet-write-def Mprefix-singl write-def*)

lemma *write-is-write0* : ‹*c*!*x* → *P* = *c x* → *P*›
by (*simp add*: *write0-def write-def*)

Chapter 6

The Global non Deterministic Choice

6.1 General non Deterministic Choice Definition

This is an experimental definition of a generalized non deterministic choice $a \sqcap b$ for an arbitrary set. The present version is "totalised" for the case of $A = \{\}$ by *STOP*, which is not the neutral element of the $(\sqcap)$ operator (because there is no neutral element for $(\sqcap)$).

lemma ‹$\nexists P.\ \forall Q.\ (P :: ('a, 'r)\ process_{ptick}) \sqcap Q = Q$›
proof (*rule ccontr*, *simp*, *elim exE*)
  fix *P*
  assume $*$: ‹$\forall Q.\ (P :: ('a, 'r)\ process_{ptick}) \sqcap Q = Q$›
  hence ‹$P = STOP$›
   by (*erule-tac x* = *STOP* **in** *allE*) (*auto simp add*: *STOP-iff-T T-Ndet T-STOP*)
  with $*$ **show** *False*
    apply (*erule-tac x* = ‹*SKIP undefined*› **in** *allE*)
    apply (*simp add*: *Process-eq-spec F-STOP F-Ndet F-SKIP set-eq-iff*)
    by (*metis UNIV-I not-Cons-self*)
qed

lift-definition *GlobalNdet* :: ‹['b *set*, 'b ⇒ ('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$›
  is ‹$\lambda A\ P.$ *if* $A = \{\}$
          then $(\{(s, X).\ s = []\}, \{\})$
          else $(\bigcup a{\in}A.\ \mathcal{F}\ (P\ a), \bigcup a{\in}A.\ \mathcal{D}\ (P\ a))$›
proof (*split if-split*, *intro conjI impI*)
  show ‹*is-process* $(\{(s, X).\ s = []\}, \{\})$› **by** (*metis STOP.rsp eq-onp-def*)
next
  show ‹*is-process* $(\bigcup a{\in}A.\ \mathcal{F}\ (P\ a), \bigcup a{\in}A.\ \mathcal{D}\ (P\ a))$› (**is** ‹*is-process* (*?f*, *?d*)›)
    if ‹$A \neq \{\}$› **for** $P :: 'b \Rightarrow ('a, 'r)\ process_{ptick}$› **and** *A*
  proof (*unfold is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv*,

intro conjI impI allI)
from ‹$A \neq \{\}$› *is-processT1* **show** ‹$([], \{\}) \in ?f$› **by** *blast*
next
from *is-processT2* **show** ‹$(s, X) \in ?f \Longrightarrow ftF\ s$› **for** $s\ X$ **by** *blast*
next
from *is-processT3* **show** ‹$(s\ @\ t, \{\}) \in ?f \Longrightarrow (s, \{\}) \in ?f$› **for** $s\ t$ **by** *blast*
next
from *is-processT4* **show** ‹$(s, Y) \in ?f \wedge X \subseteq Y \Longrightarrow (s, X) \in ?f$› **for** $s\ X\ Y$
by *blast*
next
from *is-processT5*
show ‹$(s, X) \in ?f \wedge (\forall\, c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \notin ?f)$
$\Longrightarrow (s, X \cup Y) \in ?f$› **for** $s\ X\ Y$ **by** *simp blast*
next
from *is-processT6*
show ‹$(s\ @\ [✓(r)], \{\}) \in ?f \Longrightarrow (s, X - \{✓(r)\}) \in ?f$› **for** $s\ r\ X$ **by** *fast*
next
from *is-processT7* **show** ‹$s \in ?d \wedge tF\ s \wedge ftF\ t \Longrightarrow s\ @\ t \in ?d$› **for** $s\ t$ **by**
blast
next
from *is-processT8* **show** ‹$s \in ?d \Longrightarrow (s, X) \in ?f$› **for** $s\ X$ **by** *blast*
next
from *is-processT9* **show** ‹$s\ @\ [✓(r)] \in ?d \Longrightarrow s \in ?d$› **for** $s\ r$ **by** *fast*
qed
qed

syntax -*GlobalNdet* :: ‹[*pttrn*,'*b set*,('*a*, '*r*) $process_{ptick}$] ⇒ ('*a*, '*r*) $process_{ptick}$›
(‹(*3*⊓((-)/∈(-))./ (-))› [*78*,*78*,*77*] *77*)
syntax-consts -*GlobalNdet* == *GlobalNdet*
translations ⊓ *a* ∈ *A*. *P* ⇌ *CONST GlobalNdet A* (λ*a*. *P*)

Note that the global non deterministic choice $\sqcap a{\in}A.\ P\ a$ is different from the multiple non deterministic prefix choice which guarantees continuity even when A is *infinite* due to the fact that it communicates its choice via an internal prefix operator.

6.2 The projections

lemma *F-GlobalNdet*:
‹$\mathcal{F}\ (\sqcap\ a \in A.\ P\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \{(s, X).\ s = []\}\ \mathit{else}\ (\bigcup\ a{\in}A.\ \mathcal{F}\ (P\ a)))$›
by (*simp add*: *Failures.rep-eq FAILURES-def GlobalNdet.rep-eq*)

lemma *D-GlobalNdet*: ‹$\mathcal{D}\ (\sqcap\ a \in A.\ P\ a) = (\bigcup\ a{\in}A.\ \mathcal{D}\ (P\ a))$›
by (*simp add*: *Divergences.rep-eq DIVERGENCES-def GlobalNdet.rep-eq*)

lemma *T-GlobalNdet*:
‹$\mathcal{T}\ (\sqcap\ a \in A.\ P\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \{[]\}\ \mathit{else}\ (\bigcup\ a{\in}A.\ \mathcal{T}\ (P\ a)))$›

by (*auto simp add*: *Traces.rep-eq TRACES-def Failures.rep-eq*[*symmetric*] *F-GlobalNdet intro*: *F-T T-F*)

lemma *T-GlobalNdet'*: ‹$\mathcal{T}$ ($\sqcap$ $a \in A$. $P\ a$) = *insert* [] ($\bigcup$ $a \in A$. $\mathcal{T}$ ($P\ a$))›
by (*simp add*: *T-GlobalNdet*)
(*metis T-GlobalNdet insert-absorb is-processT1-TR*)

lemmas *GlobalNdet-projs* = *F-GlobalNdet D-GlobalNdet T-GlobalNdet'*

lemma *mono-GlobalNdet-eq*:
‹($\bigwedge a$. $a \in A \Longrightarrow P\ a = Q\ a$) $\Longrightarrow$ *GlobalNdet A P* = *GlobalNdet A Q*›
by (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)

lemma *mono-GlobalNdet-eq2*:
‹($\bigwedge a$. $a \in A \Longrightarrow P\ (f\ a) = Q\ a$) $\Longrightarrow$ *GlobalNdet* (f ‘ A) P = *GlobalNdet A Q*›
by (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)

6.3 Factorization of (⊓) in front of *GlobalNdet*

lemma *GlobalNdet-factorization-union*:
‹[$\![A \neq \{\}; B \neq \{\}]\!$] $\Longrightarrow$
($\sqcap$ $p \in A$. $P\ p$) $\sqcap$ ($\sqcap$ $p \in B$. $P\ p$) = ($\sqcap$ $p \in (A \cup B)$. $P\ p$)›
by (*subst Process-eq-spec*) (*simp add*: *F-GlobalNdet D-GlobalNdet F-Ndet D-Ndet*)

lemma *GlobalNdet-Union* :
‹($\sqcap a \in (\bigcup i \in I.\ A\ i)$. $P\ a$) = $\sqcap i \in \{i \in I.\ A\ i \neq \{\}\}$. $\sqcap a \in A\ i$. $P\ a$›
by (*auto simp add*: *Process-eq-spec GlobalNdet-projs*)

6.4 First properties

lemma *GlobalNdet-id* [*simp*] : ‹$A \neq \{\} \Longrightarrow$ ($\sqcap$ $p \in A$. P) = P›
by (*subst Process-eq-spec*) (*simp add*: *F-GlobalNdet D-GlobalNdet*)

lemma *GlobalNdet-unit*[*simp*] : ‹($\sqcap$ $x \in \{a\}$. $P\ x$) = $P\ a$›
by(*auto simp* : *Process-eq-spec F-GlobalNdet D-GlobalNdet*)

lemma *GlobalNdet-distrib-unit*:
‹$A - \{a\} \neq \{\} \Longrightarrow$ ($\sqcap$ $x \in$ *insert a A*. $P\ x$) = $P\ a \sqcap$ ($\sqcap$ $x \in (A - \{a\})$. $P\ x$)›
by (*metis GlobalNdet-factorization-union GlobalNdet-unit*
empty-not-insert insert-Diff-single insert-is-Un)

lemma *GlobalNdet-distrib-unit-bis* :
‹($\sqcap$ $x \in$ *insert a A*. $P\ x$) = (*if* $A - \{a\} = \{\}$ *then* $P\ a$ *else* $P\ a \sqcap$ ($\sqcap$ $x \in (A - \{a\})$. $P\ x$))›
by (*metis GlobalNdet-distrib-unit GlobalNdet-unit insert-Diff-single*)

6.5 Behaviour of *GlobalNdet* with ($\sqcap$)

lemma *GlobalNdet-Ndet*:
‹$(\sqcap\ a \in A.\ P\ a) \sqcap (\sqcap\ a \in A.\ Q\ a) = \sqcap\ a \in A.\ P\ a \sqcap Q\ a$›
by (*auto simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet F-Ndet D-Ndet*)

6.6 Commutativity

lemma *GlobalNdet-sets-commute*:
‹$(\sqcap\ a \in A.\ \sqcap\ b \in B.\ P\ a\ b) = \sqcap\ b \in B.\ \sqcap\ a \in A.\ P\ a\ b$›
by (*auto simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet*)

6.7 Behaviour with injectivity

lemma *inj-on-mapping-over-GlobalNdet*:
‹*inj-on f A* $\Longrightarrow (\sqcap\ x \in A.\ P\ x) = \sqcap\ x \in f\ `\ A.\ P\ (inv\text{-}into\ A\ f\ x)$›
by (*simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet*)

6.8 Cartesian product results

lemma *GlobalNdet-cartprod-σs-set-σs-set*:
‹$(\sqcap\ (s,\ t) \in A \times B.\ P\ (s\ @\ t)) = \sqcap\ u \in \{s\ @\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
apply (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)
by *safe auto*

lemma *GlobalNdet-cartprod-s-set-σs-set*:
‹$(\sqcap\ (s,\ t) \in A \times B.\ P\ (s\ \#\ t)) = \sqcap\ u \in \{s\ \#\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
apply (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)
by *safe auto*

lemma *GlobalNdet-cartprod-s-set-s-set*:
‹$(\sqcap\ (s,\ t) \in A \times B.\ P\ [s,\ t]) = \sqcap\ u \in \{[s,\ t]\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
apply (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)
by *safe auto*

lemma *GlobalNdet-cartprod*: ‹$(\sqcap(s,\ t) \in A \times B.\ P\ s\ t) = \sqcap s \in A.\ \sqcap t \in B.\ P\ s\ t$›
apply (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)
by *safe auto*

6.9 Link with *Mndetprefix*

This is a trick to make proof of *Mndetprefix* using *GlobalNdet* as it has an easier denotational definition.

lemma *Mndetprefix-GlobalNdet*: ‹$\sqcap\ a \in A \rightarrow P\ a = \sqcap\ a \in A.\ a \rightarrow P\ a$›
by (*cases* ‹$A = \{\}$›; *subst Process-eq-spec-optimized*)
(*simp-all add*: *F-Mndetprefix D-Mndetprefix F-GlobalNdet D-GlobalNdet*)

lemma *write0-GlobalNdet*:
‹$x \rightarrow \sqcap\ a \in A.\ P\ a = (\mathit{if}\ A = \{\}\ \mathit{then}\ x \rightarrow \mathit{STOP}\ \mathit{else}\ \sqcap\ a \in A.\ x \rightarrow P\ a)$›
by (*auto simp add*: *Process-eq-spec write0-def STOP-projs*
F-GlobalNdet D-GlobalNdet F-Mprefix D-Mprefix)

lemma *ndet-write-is-GlobalNdet-write0* :
‹$c!!a{\in}A \rightarrow P\ a = \sqcap b{\in}c\ `\ A.\ b \rightarrow P\ (\mathit{inv\text{-}into}\ A\ c\ b)$›
by (*simp add*: *ndet-write-def Mndetprefix-GlobalNdet*)

lemma *ndet-write-is-GlobalNdet-write* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow c!!a{\in}A \rightarrow P\ a = \sqcap a{\in}A.\ c!a \rightarrow P\ a$›
by (*auto simp add*: *ndet-write-is-GlobalNdet-write0 write-def write0-def*
intro: *mono-GlobalNdet-eq2*)

lemma *GlobalNdet-Mprefix-distr*:
‹$A \neq \{\} \Longrightarrow (\sqcap\ a \in A.\ \Box\ b \in B \rightarrow P\ a\ b) = \Box\ b \in B \rightarrow (\sqcap\ a \in A.\ P\ a\ b)$›
by (*auto simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet F-Mprefix D-Mprefix*)

lemma *GlobalNdet-Det-distrib*:
‹$(\sqcap\ a \in A.\ P\ a\ \Box\ Q\ a) = (\sqcap\ a \in A.\ P\ a)\ \Box\ (\sqcap\ a \in A.\ Q\ a)$›
if ‹$\exists\ Q'\ b.\ \forall\ a.\ Q\ a = (b \rightarrow Q'\ a)$›
proof −
from *that* **obtain** $b\ Q'$ **where** ‹$\forall\ a.\ (Q\ a = (b \rightarrow Q'\ a))$› **by** *blast*
thus *?thesis*
by (*auto simp add*: *Process-eq-spec F-Det D-Det write0-def*
F-GlobalNdet D-GlobalNdet T-GlobalNdet F-STOP D-STOP
F-Mprefix D-Mprefix T-Mprefix)
qed

6.10 Continuity

lemma *mono-GlobalNdet* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq \sqcap a \in A.\ Q\ a$›

by (*simp add*: *le-approx-def D-GlobalNdet Refusals-after-def F-GlobalNdet min-elems-def T-GlobalNdet subset-iff*) *blast*

lemma *chain-GlobalNdet* : ‹*chain Y* ⟹ *chain* (λ*i*. ⊓*a* ∈ *A*. *Y i a*)›
by (*simp add*: *ch2ch-monofun fun-belowD mono-GlobalNdet monofunI*)

lemma *GlobalNdet-cont* [*simp*] : ‹⟦*finite A*; ⋀*a*. *a* ∈ *A* ⟹ *cont* (*P a*)⟧ ⟹ *cont* (λ*y*. ⊓ *z*∈*A*. *P z y*)›
proof (*induct A rule*: *finite-induct*)
case *empty*
thus *?case* **by** (*simp add*: *GlobalNdet.abs-eq*)
next
case (*insert a A*)
thus *?case*
by (*cases* ‹*A* = {}›, *simp*)
(*subst GlobalNdet-distrib-unit*[*of A a*], *auto*)
qed

Chapter 7

The Sequential Composition

7.1 Definition

lift-definition *Seq* :: ‹[('a, 'r) $process_{ptick}$, ('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$› (**infixl** ‹;› 74)
is ‹λP Q. ({(t, X) |t X. (t, X ∪ range tick) ∈ ℱ P ∧ tF t} ∪
{(t @ u, X) |t u r X. t @ [✓(r)] ∈ 𝒯 P ∧ (u, X) ∈ ℱ Q} ∪
{(t, X). t ∈ 𝒟 P},
𝒟 P ∪ {t @ u |t u r. t @ [✓(r)] ∈ 𝒯 P ∧ u ∈ 𝒟 Q})›
proof −
show ‹?thesis P Q› (**is** ‹is-process(?f, ?d)›) **for** P **and** Q
proof (*unfold is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv, intro conjI allI impI*)
show ‹([], {}) ∈ ?f›
by *simp* (*metis* (*no-types*, *opaque-lifting*) *T-F-spec append-Nil f-inv-into-f is-processT1 is-processT5-S7*)
next
show ‹(s, X) ∈ ?f ⟹ *front-tickFree* s› **for** s X
by (*auto simp*: *is-processT2 append-T-imp-tickFree front-tickFree-append D-imp-front-tickFree*)
next
show ‹(s @ t, {}) ∈ ?f ⟹ (s, {}) ∈ ?f› **for** s t
proof (*induct t arbitrary*: s)
show ‹(s @ [], {}) ∈ ?f ⟹ (s, {}) ∈ ?f› **for** s **by** *simp*
next
fix s e t **assume** *prem* : ‹(s @ e # t, {}) ∈ ?f›
assume *hyp* : ‹(s @ t, {}) ∈ ?f ⟹ (s, {}) ∈ ?f› **for** s
from *prem* **have** ‹((s @ [e]) @ t, {}) ∈ ?f› **by** *simp*
with *hyp* **have** ‹(s @ [e], {}) ∈ ?f› **by** *presburger*
then consider (*F-P*) ‹(s @ [e], *range tick*) ∈ ℱ P› ‹tF s›
| (*F-Q*) t u r **where** ‹s @ [e] = t @ u› ‹t @ [✓(r)] ∈ 𝒯 P› ‹(u, {}) ∈ ℱ Q›
by (*auto intro*: *is-processT8*)
(*meson F-T append-T-imp-tickFree is-processT8 not-Cons-self2*)
thus ‹(s, {}) ∈ ?f›
proof *cases*

```
      case F-P
     from F-P(1) is-processT3 is-processT4-empty have ‹(s, {}) ∈ ℱ P› by blast
      with F-P(2) show ‹(s, {}) ∈ ?f›
        by (elim trace-tick-continuation-or-all-tick-failuresE)
          (simp-all, metis append.right-neutral is-processT1)
    next
      case F-Q
      show ‹(s, {}) ∈ ?f›
      proof (cases u rule: rev-cases)
        assume ‹u = []›
        with F-Q(1) have ‹s @ [e] = t› by simp
        with F-Q(2) T-F is-processT3 have ‹(s, {}) ∈ ℱ P› by blast
        with F-Q(2) ‹s @ [e] = t› show ‹(s, {}) ∈ ?f›
          by (elim trace-tick-continuation-or-all-tick-failuresE, simp-all)
            (metis append.right-neutral is-processT1,
              metis append-T-imp-tickFree not-Cons-self2 tickFree-append-iff)
      next
        from F-Q show ‹u = u' @ [e'] ⟹ (s, {}) ∈ ?f› for u' e'
          by simp (metis is-processT3)
      qed
    qed
  qed
next
  show ‹(s, Y) ∈ ?f ∧ X ⊆ Y ⟹ (s, X) ∈ ?f› for s X Y
    by simp (metis Un-mono image-mono is-processT4 top.extremum-unique)
next
  fix s X Y assume ∗ : ‹(s, X) ∈ ?f ∧ (∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ ?f)›
  from ∗ consider ‹s ∈ ?d›
    | (F-P) ‹(s, X ∪ range tick) ∈ ℱ P› ‹tickFree s›
    | (F-Q) t u r where ‹s = t @ u› ‹t @ [✓(r)] ∈ 𝒯 P› ‹(u, X) ∈ ℱ Q› by fast
  thus ‹(s, X ∪ Y) ∈ ?f›
  proof cases
    from is-processT8 show ‹s ∈ ?d ⟹ (s, X ∪ Y) ∈ ?f› by simp blast
  next
    case F-P
    let ?Y-evs = ‹Y − range tick›
    have ‹(s, X ∪ range tick ∪ ?Y-evs) ∈ ℱ P›
    proof (intro is-processT5[OF F-P(1)] allI impI)
      fix c assume ‹c ∈ ?Y-evs›
      then obtain r where ‹c = ev r› ‹ev r ∈ Y›
        by (metis Diff-iff event_ptick.exhaust rangeI)
      from ∗[THEN conjunct2, rule-format, OF ‹ev r ∈ Y›]
      show ‹(s @ [c], {}) ∉ ℱ P›
        by (simp add: ‹c = ev r› ‹tickFree s› image-iff)
       (metis append-Nil2 is-processT1 trace-tick-continuation-or-all-tick-failuresE)
    qed
    also have ‹X ∪ range tick ∪ ?Y-evs = X ∪ Y ∪ range tick› by fast
    finally have ‹(s, X ∪ Y ∪ range tick) ∈ ℱ P› .
    with F-P(2) show ‹(s, X ∪ Y) ∈ ?f› by simp
```

```
    next
      case F-Q
      from ∗ have ‹∀ c. c ∈ Y ⟶ (u @ [c], {}) ∉ ℱ Q›
        by (simp add: F-Q(1)) (metis F-Q(2))
      with F-Q(3) have ‹(u, X ∪ Y) ∈ ℱ Q› by (simp add: is-processT5)
      with F-Q(1, 2) show ‹(s, X ∪ Y) ∈ ?f› by auto
    qed
  next
    show ‹s ∈ ?d ∧ tF s ∧ ftF t ⟹ s @ t ∈ ?d› for s t
      by (simp, elim conjE disjE exE)
        (metis is-processT7, meson append-assoc is-processT7 tickFree-append-iff)
  next
    from is-processT8 show ‹s ∈ ?d ⟹ (s, X) ∈ ?f› for s X by simp blast
  next
    show ‹s @ [✓(r)] ∈ ?d ⟹ s ∈ ?d› for s r
      by (simp, elim conjE disjE exE)
        (meson is-processT9,
       metis (no-types) D-T T-nonTickFree-imp-decomp append.assoc append-T-imp-tickFree
       butlast-snoc is-processT9 non-tickFree-tick not-Cons-self2 tickFree-append-iff)
    fix s r X assume ‹(s @ [✓(r)], {}) ∈ ?f›
    then consider ‹s @ [✓(r)] ∈ ?d›
      | (F-Q) t u r' where ‹s @ [✓(r)] = t @ u› ‹t @ [✓(r')] ∈ 𝒯 P› ‹(u, X) ∈ ℱ
Q›
      by (auto simp add: is-processT8)
        (metis F-T append-T-imp-tickFree is-processT5-S7
          non-tickFree-tick not-Cons-self2 tickFree-append-iff)
    thus ‹(s, X − {✓(r)}) ∈ ?f›
    proof cases
      assume ‹s @ [✓(r)] ∈ ?d›
      hence ‹s ∈ ?d› by (fact ‹s @ [✓(r)] ∈ ?d ⟹ s ∈ ?d›)
      with is-processT8 show ‹(s, X − {✓(r)}) ∈ ?f› by simp blast
    next
      case F-Q
      from F-Q(1, 2) obtain u' where ‹u = u' @ [✓(r)]›
        by (cases u rule: rev-cases, simp-all)
            (meson append-T-imp-tickFree non-tickFree-tick not-Cons-self2 tick-
Free-append-iff)
     with F-Q(3) have ‹(u', X − {✓(r)}) ∈ ℱ Q› by (simp add: F-T is-processT6-TR)
      with F-Q(1, 2) ‹u = u' @ [✓(r)]› show ‹(s, X − {✓(r)}) ∈ ?f› by auto
    qed
  qed
qed
```

7.2 The Projections

```
lemma F-Seq : ‹ℱ (P ; Q) = {(t, X) |t X. (t, X ∪ range tick) ∈ ℱ P ∧ tF t} ∪
                          {(t @ u, X) |t u r X. t @ [✓(r)] ∈ 𝒯 P ∧ (u, X) ∈ ℱ Q} ∪
                          {(t, X). t ∈ 𝒟 P}›
  by (simp add: Failures.rep-eq FAILURES-def Seq.rep-eq)
```

lemma *D-Seq* : ‹$\mathcal{D}\ (P\ ;;\ Q) = \mathcal{D}\ P \cup \{t\ @\ u\ |t\ u\ r.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \wedge u \in \mathcal{D}\ Q\}$›
by (*simp add*: *Divergences.rep-eq DIVERGENCES-def Seq.rep-eq*)

lemma *T-Seq-bis* :
‹$\mathcal{T}\ (P\ ;;\ Q) = \{t.\ (t, \mathit{range\ tick}) \in \mathcal{F}\ P \wedge tF\ t\} \cup$
$\{t\ @\ u\ |t\ u\ r.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q\} \cup$
$\mathcal{D}\ P$›
by (*auto simp add*: *Traces.rep-eq TRACES-def F-Seq simp flip*: *Failures.rep-eq*)
(*meson is-processT4 sup-ge2*, *meson is-processT5-S7'*, *blast*)

lemma *T-Seq* :
‹$\mathcal{T}\ (P\ ;;\ Q) = \{t \in \mathcal{T}\ P.\ tF\ t\} \cup \{t\ @\ u\ |t\ u\ r.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q\} \cup \mathcal{D}\ P$›
— Often easier to use
by (*auto simp add*: *T-Seq-bis F-T*)
(*metis T-F-spec append.right-neutral is-processT1-TR*
trace-tick-continuation-or-all-tick-failuresE)

lemmas *Seq-projs* = *F-Seq D-Seq T-Seq*

7.3 Continuity Rule

lemma *mono-Seq* : ‹$P \sqsubseteq Q \Longrightarrow R \sqsubseteq S \Longrightarrow P\ ;;\ R \sqsubseteq Q\ ;;\ S$› **for** P :: ‹$('a, 'r)\ process_{ptick}$›
proof −
have ‹$P\ ;;\ S \sqsubseteq Q\ ;;\ S$› **if** ‹$P \sqsubseteq Q$› **for** $P\ Q$ :: ‹$('a, 'r)\ process_{ptick}$› **and** S
proof (*unfold le-approx-def*, *intro conjI allI impI subset-antisym*)
show ‹$\mathcal{D}\ (Q\ ;;\ S) \subseteq \mathcal{D}\ (P\ ;;\ S)$›
apply (*rule subsetI*, *simp add*: *D-Seq*, *elim disjE exE conjE*)
by (*meson in-mono le-approx1* ‹$P \sqsubseteq Q$›) (*metis D-T le-approx2T* ‹$P \sqsubseteq Q$›)
next
show ‹$s \notin \mathcal{D}\ (P\ ;;\ S) \Longrightarrow \mathcal{R}_a\ (P\ ;;\ S)\ s \subseteq \mathcal{R}_a\ (Q\ ;;\ S)\ s$› **for** s
apply (*auto simp add*: *D-Seq Refusals-after-def F-Seq set-eq-iff append-T-imp-tickFree*)[*1*]
by (*meson le-approx2* ‹$P \sqsubseteq Q$›)
(*metis F-imp-front-tickFree append-T-imp-tickFree is-processT7 is-processT9 le-approx2T not-Cons-self2* ‹$P \sqsubseteq Q$›)+
next
show ‹$s \notin \mathcal{D}\ (P\ ;;\ S) \Longrightarrow \mathcal{R}_a\ (Q\ ;;\ S)\ s \subseteq \mathcal{R}_a\ (P\ ;;\ S)\ s$› **for** s
apply (*auto simp add*: *D-Seq Refusals-after-def F-Seq set-eq-iff append-T-imp-tickFree*)
apply (*meson le-approx2* ‹$P \sqsubseteq Q$›)
by ((*metis D-T le-approx2T* ‹$P \sqsubseteq Q$›)+)[*2*]
((*meson in-mono le-approx1* ‹$P \sqsubseteq Q$›)+)[*2*]
next
show ‹*min-elems* $(\mathcal{D}\ (P\ ;;\ S)) \subseteq \mathcal{T}\ (Q\ ;;\ S)$›

proof (*rule subset-trans*; *rule subsetI*)
fix t **assume** ‹$t \in$ *min-elems* ($\mathcal{D}$ (P ; S))›
hence ‹$t \in \mathcal{D}$ (P ; S)› **by** (*meson elem-min-elems*)
then consider ‹$t \in \mathcal{D}$ P› | u v r **where** ‹$t = u$ @ v› ‹u @ [✓(r)] $\in \mathcal{T}$ P› ‹$u \notin \mathcal{D}$ P› ‹$v \in \mathcal{D}$ S›
by (*simp add*: *D-Seq*)
(*metis D-imp-front-tickFree append-T-imp-tickFree is-processT7 not-Cons-self2*)
thus ‹$t \in$ *min-elems* ($\mathcal{D}$ P) $\cup$ {u @ v |u v r. u @ [✓(r)] $\in \mathcal{T}$ $P \wedge u \notin \mathcal{D}$ $P \wedge v \in$ *min-elems* ($\mathcal{D}$ S)}›
proof *cases*
assume ‹$t \in \mathcal{D}$ P›
with ‹$t \in$ *min-elems* ($\mathcal{D}$ (P ; S))› **have** ‹$t \in$ *min-elems* ($\mathcal{D}$ P)›
by (*auto simp add*: *D-Seq min-elems-def*)
thus ‹$t \in$ *min-elems* ($\mathcal{D}$ P) $\cup$ {u @ v |u v r. u @ [✓(r)] $\in \mathcal{T}$ $P \wedge u \notin \mathcal{D}$ $P \wedge v \in$ *min-elems* ($\mathcal{D}$ S)}› **..**
next
fix u v r **assume** ‹$t = u$ @ v› ‹u @ [✓(r)] $\in \mathcal{T}$ P› ‹$u \notin \mathcal{D}$ P› ‹$v \in \mathcal{D}$ S›
with ‹$t \in$ *min-elems* ($\mathcal{D}$ (P ; S))›
have ‹$t \in$ {u @ v |u v r. u @ [✓(r)] $\in \mathcal{T}$ $P \wedge u \notin \mathcal{D}$ $P \wedge v \in$ *min-elems* ($\mathcal{D}$ S)}›
by (*simp add*: *D-Seq min-elems-def*)
(*metis* (*mono-tags*, *lifting*) *Un-iff less-append mem-Collect-eq*)
thus ‹$t \in$ *min-elems* ($\mathcal{D}$ P) $\cup$ {u @ v |u v r. u @ [✓(r)] $\in \mathcal{T}$ $P \wedge u \notin \mathcal{D}$ $P \wedge v \in$ *min-elems* ($\mathcal{D}$ S)}› **..**
qed
next
show ‹$t \in$ *min-elems* ($\mathcal{D}$ P) $\cup$ {u @ v |u v r. u @ [✓(r)] $\in \mathcal{T}$ $P \wedge u \notin \mathcal{D}$ $P \wedge v \in$ *min-elems* ($\mathcal{D}$ S)}
$\Longrightarrow t \in \mathcal{T}$ (Q ; S)› **for** t
by (*simp add*: *T-Seq*, *elim disjE exE conjE*)
(*metis* (*no-types*, *lifting*) *Prefix-Order.prefixI T-nonTickFree-imp-decomp elem-min-elems in-mono is-processT9 le-approx3 min-elems-no not-Cons-self2 self-append-conv* ‹$P \sqsubseteq Q$›,
metis D-T elem-min-elems is-processT9 le-approx2T ‹$P \sqsubseteq Q$›)
qed
qed

moreover have ‹S ; $P \sqsubseteq S$; Q› **if** ‹$P \sqsubseteq Q$› **for** P Q **and** S :: ‹('a, 'r) $process_{ptick}$›
proof (*unfold le-approx-def*, *intro conjI allI impI subset-antisym*)
show ‹$\mathcal{D}$ (S ; Q) $\subseteq \mathcal{D}$ (S ; P)›
by (*rule subsetI*, *simp add*: *D-Seq*) (*metis in-mono le-approx1 that*)
next
show ‹$s \notin \mathcal{D}$ (S ; P) $\Longrightarrow \mathcal{R}_a$ (S ; P) $s \subseteq \mathcal{R}_a$ (S ; Q) s› **for** s
apply (*auto simp add*: *D-Seq Refusals-after-def F-Seq append-T-imp-tickFree*)
by (*metis proc-ord2a that*)+
next
show ‹$s \notin \mathcal{D}$ (S ; P) $\Longrightarrow \mathcal{R}_a$ (S ; Q) $s \subseteq \mathcal{R}_a$ (S ; P) s› **for** s
apply (*auto simp add*: *D-Seq Refusals-after-def F-Seq append-T-imp-tickFree*)

```
    by (metis proc-ord2a that)+
  next
    show ‹min-elems (𝒟 (S ; P)) ⊆ 𝒯 (S ; Q)›
    proof (rule subset-trans; rule subsetI)
      fix t assume ‹t ∈ min-elems (𝒟 (S ; P))›
      hence ‹t ∈ 𝒟 (S ; P)› by (meson elem-min-elems)
      then consider ‹t ∈ 𝒟 S› | u v r where ‹t = u @ v› ‹u @ [✓(r)] ∈ 𝒯 S› ‹u
∉ 𝒟 S› ‹v ∈ 𝒟 P›
        by (simp add: D-Seq)
       (metis D-imp-front-tickFree append-T-imp-tickFree is-processT7 not-Cons-self2)
      thus ‹t ∈ min-elems (𝒟 S) ∪ {u @ v |u v r. u @ [✓(r)] ∈ 𝒯 S ∧ u ∉ 𝒟 S ∧
v ∈ min-elems (𝒟 P)}›
      proof cases
        assume ‹t ∈ 𝒟 S›
        with ‹t ∈ min-elems (𝒟 (S ; P))› have ‹t ∈ min-elems (𝒟 S)›
          by (auto simp add: D-Seq min-elems-def)
        thus ‹t ∈ min-elems (𝒟 S) ∪ {u @ v |u v r. u @ [✓(r)] ∈ 𝒯 S ∧ u ∉ 𝒟 S
∧ v ∈ min-elems (𝒟 P)}› ..
      next
        fix u v r assume ‹t = u @ v› ‹u @ [✓(r)] ∈ 𝒯 S› ‹u ∉ 𝒟 S› ‹v ∈ 𝒟 P›
        with ‹t ∈ min-elems (𝒟 (S ; P))›
        have ‹t ∈ {u @ v |u v r. u @ [✓(r)] ∈ 𝒯 S ∧ u ∉ 𝒟 S ∧ v ∈ min-elems (𝒟
P)}›
          by (simp add: D-Seq min-elems-def)
            (metis (mono-tags, lifting) Un-iff less-append mem-Collect-eq)
        thus ‹t ∈ min-elems (𝒟 S) ∪ {u @ v |u v r. u @ [✓(r)] ∈ 𝒯 S ∧ u ∉ 𝒟 S
∧ v ∈ min-elems (𝒟 P)}› ..
      qed
    next
      show ‹t ∈ min-elems (𝒟 S) ∪ {u @ v |u v r. u @ [✓(r)] ∈ 𝒯 S ∧ u ∉ 𝒟 S
∧ v ∈ min-elems (𝒟 P)}
          ⟹ t ∈ 𝒯 (S ; Q)› for t
        by (simp add:  T-Seq, elim disjE exE conjE)
          (use elem-min-elems in blast,
            metis insert-absorb insert-subset le-approx3 ‹P ⊑ Q›)
    qed
  qed

  ultimately show ‹P ⊑ Q ⟹ R ⊑ S ⟹ P ; R ⊑ Q ; S› by (meson below-trans)
qed


context begin

private lemma chain-Seq-left: ‹chain Y ⟹ chain (λi. Y i ; S)›
  by (simp add: mono-Seq po-class.chain-def)

private lemma chain-Seq-right: ‹chain Y ⟹ chain (λi. S ; Y i)›
  by (simp add: mono-Seq po-class.chain-def)
```

private lemma *cont-left-prem-Seq* :
‹$(\bigsqcup i.\ Y\ i)$ **;** $S = (\bigsqcup i.\ Y\ i$ **;** $S)$› (**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain* Y›
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$t \in \mathcal{D}$ *?lhs* $\Longrightarrow$ $t \in \mathcal{D}$ *?rhs*› **for** t
by (*auto simp add*: *D-Seq limproc-is-thelub* ‹*chain* Y› *chain-Seq-left LUB-projs*)
next
fix t **assume** ‹$t \in \mathcal{D}$ *?rhs*›
define T **where** ‹$T\ i \equiv \{u \in \mathcal{D}\ (Y\ i).\ t = u\} \cup$
$\{u.\ \exists r\ v.\ t = u\ @\ v \wedge u\ @\ [\checkmark(r)] \in \mathcal{T}\ (Y\ i) \wedge v \in \mathcal{D}\ S\}$› **for** i
from ‹$t \in \mathcal{D}$ *?rhs*› **have** ‹$T\ i \neq \{\}$› **for** i
by (*auto simp add*: *T-def D-Seq limproc-is-thelub* ‹*chain* Y› *chain-Seq-left D-LUB*)
moreover have ‹*finite* $(T\ 0)$› **unfolding** *T-def* **by** (*prove-finite-subset-of-prefixes* t)
moreover have ‹$T\ (Suc\ i) \subseteq T\ i$› **for** i
unfolding *T-def* **apply** (*intro allI Un-mono subsetI*, *simp-all*)
by (*meson* ‹*chain* Y› *in-mono le-approx1 po-class.chainE*)
(*metis* ‹*chain* Y› *le-approx-lemma-T po-class.chain-def subset-iff*)
ultimately have ‹$(\bigcap i.\ T\ i) \neq \{\}$› **by** (*rule Inter-nonempty-finite-chained-sets*)
then obtain u **where** ‹$\forall i.\ u \in T\ i$› **by** *fast*
show ‹$t \in \mathcal{D}$ *?lhs*›
proof (*cases* ‹$\forall i.\ t \in \mathcal{D}\ (Y\ i)$›)
show ‹$\forall i.\ t \in \mathcal{D}\ (Y\ i) \Longrightarrow t \in \mathcal{D}$ *?lhs*›
by (*simp add*: *D-Seq limproc-is-thelub D-LUB* ‹*chain* Y›)
next
assume ‹$\neg\ (\forall i.\ t \in \mathcal{D}\ (Y\ i))$›
with ‹$\forall i.\ u \in T\ i$› **obtain** $r\ v\ j$
where ‹$t = u\ @\ v$› ‹$u\ @\ [\checkmark(r)] \in \mathcal{T}\ (Y\ j)$› ‹$u \notin \mathcal{D}\ (Y\ j)$› ‹$v \in \mathcal{D}\ S$›
by (*simp add*: *T-def*)
(*metis D-imp-front-tickFree append-T-imp-tickFree is-processT7 not-Cons-self2*)
from ‹$u \notin \mathcal{D}\ (Y\ j)$› ‹$u\ @\ [\checkmark(r)] \in \mathcal{T}\ (Y\ j)$› **have** ‹$\forall i{\geq}j.\ u\ @\ [\checkmark(r)] \in \mathcal{T}\ (Y\ i)$›
by (*meson ND-F-dir2'* ‹*chain* Y› ‹$u \notin \mathcal{D}\ (Y\ j)$› *is-processT9 is-ub-thelub*)
hence ‹$u\ @\ [\checkmark(r)] \in \mathcal{T}\ (\bigsqcup i.\ Y\ i)$›
by (*simp add*: *limproc-is-thelub T-LUB* ‹*chain* Y›)
(*meson D-T* ‹*chain* Y› *le-approx2T nle-le po-class.chain-mono*)
with ‹$v \in \mathcal{D}\ S$› **show** ‹$t \in \mathcal{D}$ *?lhs*› **by** (*auto simp add*: ‹$t = u\ @\ v$› *D-Seq* ‹*chain* Y›)
qed
next
show ‹$(t,\ X) \in \mathcal{F}$ *?lhs* $\Longrightarrow$ $(t,\ X) \in \mathcal{F}$ *?rhs*› **for** $t\ X$
by (*auto simp add*: *F-Seq limproc-is-thelub* ‹*chain* Y› *chain-Seq-left LUB-projs*)
next
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
fix $t\ X$ **assume** ‹$(t,\ X) \in \mathcal{F}$ *?rhs*›
then consider ‹$\forall i.\ t \in \mathcal{D}\ (Y\ i)$›
| j **where** ‹$t \notin \mathcal{D}\ (Y\ j)$› ‹$(t,\ X \cup range\ tick) \in \mathcal{F}\ (Y\ j)$› ‹$tF\ t$›

| *j u r v* **where** ‹$t \notin \mathcal{D}\ (Y\ j)$› ‹$t = u\ @\ v$› ‹$u\ @\ [\checkmark(r)] \in \mathcal{T}\ (Y\ j)$› ‹$(v,\ X) \in \mathcal{F}\ S$›
by (*simp add*: *limproc-is-thelub* ‹*chain Y*› *chain-Seq-left F-LUB F-Seq*) *blast*
thus ‹$(t,\ X) \in \mathcal{F}$ *?lhs*›
proof *cases*
show ‹$\forall i.\ t \in \mathcal{D}\ (Y\ i) \Longrightarrow (t,\ X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *D-LUB-2 D-Seq is-processT8 limproc-is-thelub* ‹*chain Y*›)
next
fix *j* **assume** ‹$t \notin \mathcal{D}\ (Y\ j)$› ‹$(t,\ X \cup$ *range tick*$) \in \mathcal{F}\ (Y\ j)$› ‹*tF t*›
hence ‹$(t,\ X \cup$ *range tick*$) \in \mathcal{F}\ (\bigsqcup i.\ Y\ i)$›
by (*simp add*: *limproc-is-thelub* ‹*chain Y*› *F-LUB*)
(*metis is-processT8 nle-le po-class.chain-mono proc-ord2a* ‹*chain Y*›)
with ‹*tF t*› **show** ‹$(t,\ X) \in \mathcal{F}$ *?lhs*› **by** (*simp add*: *F-Seq*)
next
fix *j u r v* **assume** ‹$t \notin \mathcal{D}\ (Y\ j)$› ‹$t = u\ @\ v$› ‹$u\ @\ [\checkmark(r)] \in \mathcal{T}\ (Y\ j)$› ‹$(v,\ X) \in \mathcal{F}\ S$›
hence ‹$u\ @\ [\checkmark(r)] \in \mathcal{T}\ (\bigsqcup i.\ Y\ i)$›
by (*simp add*: *limproc-is-thelub* ‹*chain Y*› *T-LUB*)
(*meson F-imp-front-tickFree ND-F-dir2' T-F-spec append-T-imp-tickFree is-processT7 is-ub-thelub min-elems2 not-Cons-self2* ‹*chain Y*›)
with ‹$(v,\ X) \in \mathcal{F}\ S$› **show** ‹$(t,\ X) \in \mathcal{F}$ *?lhs*› **by** (*auto simp add*: *F-Seq* ‹$t = u\ @\ v$›)
qed
qed

private lemma *cont-right-prem-Seq* :
‹S **;** $(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ S$ **;** $Y\ i)$› (**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain Y*›
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$t \in \mathcal{D}$ *?lhs* $\Longrightarrow t \in \mathcal{D}$ *?rhs*› **for** *t*
by (*auto simp add*: *D-Seq limproc-is-thelub* ‹*chain Y*› *chain-Seq-right D-LUB*)
next
show ‹$t \in \mathcal{D}$ *?lhs*› **if** ‹$t \in \mathcal{D}$ *?rhs*› **for** *t*
proof (*cases* ‹$t \in \mathcal{D}\ S$›)
show ‹$t \in \mathcal{D}\ S \Longrightarrow t \in \mathcal{D}\ (S$ **;** $(\bigsqcup i.\ Y\ i))$› **by** (*simp add*: *D-Seq*)
next
assume ‹$t \notin \mathcal{D}\ S$›
define *T* **where** ‹$T\ i \equiv \{u.\ \exists v\ r.\ t = u\ @\ v \wedge u\ @\ [\checkmark(r)] \in \mathcal{T}\ S \wedge v \in \mathcal{D}\ (Y\ i)\}$› **for** *i*
from ‹$t \notin \mathcal{D}\ S$› ‹$t \in \mathcal{D}$ *?rhs*› **have** ‹$T\ i \neq \{\}$› **for** *i*
by (*simp add*: *T-def limproc-is-thelub chain-Seq-right* ‹*chain Y*› *D-LUB D-Seq*)
moreover have ‹*finite* $(T\ 0)$›
unfolding *T-def* **by** (*prove-finite-subset-of-prefixes t*)
moreover have ‹$T\ (Suc\ i) \subseteq T\ i$› **for** *i*
unfolding *T-def* **apply** (*intro allI Un-mono subsetI*; *simp*)
by (*metis le-approx1 po-class.chainE subset-iff* ‹*chain Y*›)
ultimately have ‹$(\bigcap i.\ T\ i) \neq \{\}$› **by** (*rule Inter-nonempty-finite-chained-sets*)
then obtain *u* **where** ‹$\forall i.\ u \in T\ i$› **by** *fast*
then obtain *v r* **where** ‹$t = u\ @\ v$› ‹$u\ @\ [\checkmark(r)] \in \mathcal{T}\ S$› ‹$\forall i.\ v \in \mathcal{D}\ (Y\ i)$›

```
    by (simp add: T-def) blast
  from ‹∀ i. v ∈ 𝒟 (Y i)› have ‹v ∈ 𝒟 (⨆ i. Y i)›
    by (simp add: D-LUB ‹chain Y› limproc-is-thelub)
  with ‹u @ [✓(r)] ∈ 𝒯 S› show ‹t ∈ 𝒟 ?lhs› by (auto simp add: ‹t = u @ v›
D-Seq)
 qed
next
 show ‹(t, X) ∈ ℱ ?lhs ⟹ (t, X) ∈ ℱ ?rhs› for t X
  by (auto simp add: F-Seq limproc-is-thelub ‹chain Y› chain-Seq-right F-LUB)
next
 assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
 show ‹(t, X) ∈ ℱ ?lhs› if ‹(t, X) ∈ ℱ ?rhs› for t X
 proof (cases ‹t ∈ 𝒟 ?rhs›)
  show ‹t ∈ 𝒟 ?rhs ⟹ (t, X) ∈ ℱ ?lhs›
    by (simp add: is-processT8 same-div)
 next
  assume ‹t ∉ 𝒟 ?rhs›
  then obtain j where ‹t ∉ 𝒟 (S ; Y j)›
    by (auto simp add: limproc-is-thelub chain-Seq-right ‹chain Y› D-LUB)
  moreover from ‹(t, X) ∈ ℱ ?rhs› have ‹(t, X) ∈ ℱ (S ; Y j)›
    by (simp add: limproc-is-thelub chain-Seq-right ‹chain Y› F-LUB)
  ultimately show ‹(t, X) ∈ ℱ ?lhs›
    by (fact le-approx2[OF mono-Seq[OF below-refl is-ub-thelub[OF ‹chain Y›]],
THEN iffD2])
 qed
qed


lemma Seq-cont [simp] : ‹cont (λx. f x ; g x)› if ‹cont f› and ‹cont g›
proof (rule cont-apply[where f = ‹λx y. f x ; y›])
 show ‹cont g› by (fact ‹cont g›)
next
 show ‹cont ((;) (f x))› for x
 proof (rule contI2)
  show ‹monofun ((;) (f x))› by (simp add: mono-Seq monofunI)
 next
  show ‹chain Y ⟹ f x ; (⨆ i. Y i) ⊑ (⨆ i. f x ; Y i)› for Y
    by (simp add: cont-right-prem-Seq)
 qed
next
 show ‹cont (λx. f x ; y)› for y
 proof (rule contI2)
  show ‹monofun (λx. f x ; y)› by (simp add: cont2monofunE mono-Seq monofunI
‹cont f›)
 next
  show ‹chain Y ⟹ f (⨆ i. Y i) ; y ⊑ (⨆ i. f (Y i) ; y)› for Y
    by (simp add: ch2ch-cont cont2contlubE cont-left-prem-Seq ‹cont f›)
 qed
qed
```

end

Chapter 8

The Synchronization Product

8.1 Basic Concepts

fun *setinterleaving* :: ‹('*a*, '*r*) $trace_{ptick}$ × ('*a*, '*r*) $refusal_{ptick}$ × ('*a*, '*r*) $trace_{ptick}$ ⇒ ('*a*, '*r*) $trace_{ptick}$ *set*›
where

si-empty1: ‹*setinterleaving* ([], *X*, []) = {[]}›
| *si-empty2*: ‹*setinterleaving* ([], *X*, *y* # *t*) =
(*if* *y* ∈ *X*
then {}
else {*z*. ∃ *u*. *z* = *y* # *u* ∧ *u* ∈ *setinterleaving* ([], *X*, *t*)})›
| *si-empty3*: ‹*setinterleaving* ((*x* # *s*), *X*, []) =
(*if* *x* ∈ *X*
then {}
else {*z*. ∃ *u*. *z* = *x* # *u* ∧ *u* ∈ *setinterleaving* (*s*, *X*, [])})›

| *si-neq* : ‹*setinterleaving* (*x* # *s*, *X*, *y* # *t*) =
(*if* *x* ∈ *X*
then if *y* ∈ *X*
then if *x* = *y*
then {*z*. ∃ *u*. *z* = *x* # *u* ∧ *u* ∈ *setinterleaving* (*s*, *X*, *t*)}
else {}
else {*z*. ∃ *u*. *z* = *y* # *u* ∧ *u* ∈ *setinterleaving* (*x* # *s*, *X*, *t*)}
else if *y* ∉ *X*
then {*z*. ∃ *u*. *z* = *x* # *u* ∧ *u* ∈ *setinterleaving* (*s*, *X*, *y* # *t*)} ∪
{*z*. ∃ *u*. *z* = *y* # *u* ∧ *u* ∈ *setinterleaving* (*x* # *s*, *X*, *t*)}
else {*z*. ∃ *u*. *z* = *x* # *u* ∧ *u* ∈ *setinterleaving* (*s*, *X*, *y* # *t*)})›

fun *setinterleavingList* :: ‹('*a*, '*r*) $trace_{ptick}$ × ('*a*, '*r*) $event_{ptick}$ *set* × ('*a*, '*r*) $trace_{ptick}$ ⇒ ('*a*, '*r*) $trace_{ptick}$ *list*›
where

si-empty1l: ‹*setinterleavingList* ([], *X*, []) = [[]]›
| *si-empty2l*: ‹*setinterleavingList* ([], *X*, (*y* # *t*)) =
(*if* *y* ∈ *X*
then []
else [*y* # *z*. *z* ← *setinterleavingList* ([], *X*, *t*)])›
| *si-empty3l*: ‹*setinterleavingList* (*x* # *s*, *X*, []) =
(*if* *x* ∈ *X*
then []
else [*x* # *z*. *z* ← *setinterleavingList* (*s*, *X*, [])])›

| *si-neql* : ‹*setinterleavingList* ((*x* # *s*), *X*, (*y* # *t*)) =
(*if* *x* ∈ *X*
then if *y* ∈ *X*
then if *x* = *y*
then [*x* # *z*. *z* ← *setinterleavingList* (*s*, *X*, *t*)]
else []
else [*y* # *z*. *z* ← *setinterleavingList* (*x* # *s*, *X*, *t*)]
else if *y* ∉ *X*
then [*x* # *z*. *z* ← *setinterleavingList* (*s*, *X*, *y* # *t*)] @
[*y* # *z*. *z* ← *setinterleavingList* (*x* # *s*, *X*, *t*)]
else [*x* # *z*. *z* ← *setinterleavingList* (*s*, *X*, *y* # *t*)])›

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

lemma *setinterleaving-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* ∈ *setinterleaving*(*s*, *X*, *t*))

8.2 Consequences

lemma *emptyLeftProperty*: ‹*s setinterleaves* (([], *t* :: (′*a*, ′*r*) *trace*$_{ptick}$), *A*) ⟹ *s* = *t*›
by (*induct* ‹([] :: (′*a*, ′*r*) *trace*$_{ptick}$, *A*, *t*)› *arbitrary*: *s* *t* *rule*: *setinterleaving.induct*)
(*auto* *split*: *if-split-asm*)

lemma *emptyRightProperty*: ‹*s setinterleaves* ((*t*, []), *A*) ⟹ *s* = *t*›
by (*simp* *add*: *setinterleaving-sym* *emptyLeftProperty*)

lemma *emptyLeftSelf*: ‹∀ *t1*. *t1* ∈ *set* *t* ⟶ *t1* ∉ *A* ⟹ *t* *setinterleaves* (([], *t*), *A*)›
by (*induct* *t*) *auto*

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

lemma *emptyLeftNonSync*: ‹*s setinterleaves* (([], *t*), *A*) ⟹ ∀ *a* ∈ *set t*. *a* ∉ *A*›
proof (*induct t arbitrary*: *s*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons a t*)
thus *?case* **by** (*cases s*; *simp split*: *if-split-asm*)
qed

lemma *ftf-Sync1*: ‹⟦*a* ∉ *set u*; *a* ∉ *set t*; *s setinterleaves* ((*t*, *u*), *A*)⟧ ⟹ *a* ∉ *set s*›
by (*induct* ‹(*t*, *A*, *u*)› *arbitrary*: *s t u rule*: *setinterleaving.induct*)
(*auto intro*: *emptyLeftProperty emptyRightProperty split*: *if-split-asm*)

lemma *addNonSync*:
‹*sa setinterleaves* ((*t*, *u*), *A*) ⟹ *y1* ∉ *A* ⟹
(*sa* @ [*y1*]) *setinterleaves* ((*t* @ [*y1*], *u*), *A*) ∧ (*sa* @ [*y1*]) *setinterleaves* ((*t*, *u* @ [*y1*]), *A*)›
by (*induct* ‹(*t*, *A*, *u*)› *arbitrary*: *sa t u rule*: *setinterleaving.induct*)
(*auto split*: *if-split-asm*)

lemma *addSync*:‹*sa setinterleaves* ((*t*, *u*), *A*) ⟹ *y1* ∈ *A* ⟹
(*sa* @ [*y1*]) *setinterleaves* ((*t* @ [*y1*], *u* @ [*y1*]), *A*)›
by (*induct* ‹(*t*, *A*, *u*)› *arbitrary*: *sa t u rule*: *setinterleaving.induct*)
(*auto split*: *if-split-asm*)

lemma *doubleReverse*: ‹*s1 setinterleaves* ((*t*, *u*), *A*) ⟹ *rev s1 setinterleaves* ((*rev t*, *rev u*), *A*)›
by (*induct* ‹(*t*, *A*, *u*)› *arbitrary*: *s1 t u rule*: *setinterleaving.induct*; *simp split*: *if-split-asm*)
(*metis* [[*metis-verbose* = *false*]] *addNonSync addSync rev.simps*(*2*))+

lemma *SyncTlEmpty*: ‹*a setinterleaves* (([], *u*), *A*) ⟹ *tl a setinterleaves* (([], *tl u*), *A*)›
by (*cases u*; *cases a*) (*simp-all split*: *if-splits*)

lemma *SyncHd-Tl*:
‹*a setinterleaves* ((*t*, *u*), *A*) ⟹ *hd t* ∈ *A* ⟹ *hd u* ∉ *A*

$\Longrightarrow$ *hd a = hd u ∧ tl a setinterleaves ((t, tl u), A)*›
by (*cases u*; *cases t*) (*auto split*: *if-split-asm*)

lemma *SyncHdAddEmpty*:
(*tl a*) *setinterleaves ((*[], *u), A)* $\Longrightarrow$ *hd a* ∉ *A* $\Longrightarrow$ *a* ≠ []
$\Longrightarrow$ *a setinterleaves ((*[], *hd a # u), A)*
by (*cases a*) *simp-all*

lemma *SyncHdAdd*:
(*tl a*) *setinterleaves ((t, u), A)* $\Longrightarrow$ *hd a* ∉ *A* $\Longrightarrow$ *hd t* ∈ *A* $\Longrightarrow$ *a* ≠ []
$\Longrightarrow$ *a setinterleaves ((t, hd a # u), A)*
by (*cases a*, *simp*, *cases t*, *auto*)

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

lemma *SyncSameHdTl*:
a setinterleaves ((t, u), A) $\Longrightarrow$ *hd t* ∈ *A* $\Longrightarrow$ *hd u* ∈ *A*
$\Longrightarrow$ *hd t = hd u ∧ hd a = hd t ∧ (tl a) setinterleaves ((tl t, tl u), A)*
by (*cases u*) (*cases t*, *auto split*:*if-splits*)+

lemma *SyncSingleHeadAdd*:
a setinterleaves ((t, u), A) $\Longrightarrow$ *h* ∉ *A*
$\Longrightarrow$ *(h # a) setinterleaves ((h # t, u), A)*
by (*cases u*, *auto split*:*if-splits*)

lemma *TickLeftSync*:
‹⟦*tick r* ∈ *A*; *front-tickFree t*; *s setinterleaves (([tick r], t* :: (′*a*, ′*r*) *trace*$_{ptick}$*), A)*⟧ $\Longrightarrow$ *s = t ∧ last t = tick r*›
apply (*induct* ‹*([tick r]* :: (′*a*, ′*r*) *trace*$_{ptick}$, *A*, *t)*› *arbitrary*: *s t rule*: *setinterleaving.induct*)
apply (*simp-all add*: *front-tickFree-Cons-iff split*: *if-split-asm*)
by *force* (*metis distinct.simps(1) distinct-length-2-or-more emptyRightProperty front-tickFree-Nil*)

lemma *EmptyLeftSync*: ‹*s setinterleaves ((*[], *t), A)* $\Longrightarrow$ *s = t ∧ set t ∩ A = {}*›
by (*meson Int-emptyI emptyLeftNonSync emptyLeftProperty*)

lemma *EmptyRightSync*: ‹*s setinterleaves ((t,* []*), A)* $\Longrightarrow$ *s = t ∧ set t ∩ A = {}*›
by (*simp add*: *EmptyLeftSync setinterleaving-sym*)

lemma *ftf-Sync21*: ‹$a \in set\ u \wedge a \notin set\ t \vee a \in set\ t \wedge a \notin set\ u \Longrightarrow a \in A \Longrightarrow$
setinterleaving $(u, A, t) = \{\}$›
by (*induct* ‹(t, A, u)› *arbitrary*: *t u rule*: *setinterleaving.induct*)
(*auto split*: *if-split-asm*)

lemma *ftf-Sync32*:
assumes ‹$t = t1 \ @\ [tick\ r]$› **and** ‹$u = u1 \ @\ [tick\ r]$›
and ‹*s setinterleaves* $((t, u)$, *range tick* $\cup$ *ev* ‘ $A)$›
shows ‹$\exists\, s1.$ *s1 setinterleaves* $((t1, u1)$, *range tick* $\cup$ *ev* ‘ $A) \wedge s = s1 \ @\ [tick\ r]$›
proof −
from *assms*(*1*) **have** *b* : ‹*rev t* = *tick r* # *rev t1*› **by** *auto*
from *assms*(*2*) **have** *a* : ‹*rev u* = *tick r* # *rev u1*› **by** *auto*
from *assms* **have** *ab*: ‹(*rev s*) *setinterleaves* ((*tick r* # *rev t1*, *tick r* # *rev u1*), *range tick* ∪ *ev* ‘ *A*)›
using *doubleReverse* **by** *fastforce*
from *ab* **have** *c*: ‹*tl* (*rev s*) *setinterleaves* ((*rev t1*, *rev u1*), *range tick* ∪ *ev* ‘ *A*)›
by *auto*
hence *d*: ‹(*rev* (*tl* (*rev s*))) *setinterleaves* ((*t1*, *u1*), *range tick* ∪ *ev* ‘ *A*)›
using *doubleReverse* **by** *fastforce*
from *ab append-Cons-eq-iff c* **have** ‹*rev s* = *tick r* # *tl* (*rev s*)› **by** *auto*
with *d* **show** *?thesis* **by** *blast*
qed

lemma *SyncWithTick-imp-NTF*:
assumes ‹(*s* @ [*tick r*]) *setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*)›
and ‹*front-tickFree t*› **and** ‹*front-tickFree u*›
shows ‹$\exists$ *t1 u1*. *t* = *t1* @ [*tick r*] ∧ *u* = *u1* @ [*tick r*] ∧ *s setinterleaves* ((*t1*, *u1*), *range tick* ∪ *ev* ‘ *A*)›
proof−
from *assms*(*1*) **have** *a*: ‹(*tick r* # *rev s*) *setinterleaves* ((*rev t*, *rev u*), *range tick* ∪ *ev* ‘ *A*)›
using *doubleReverse* **by** *fastforce*
from *assms*(*1*)[*THEN doubleReverse*] **obtain** *tt uu* **where** *b*: ‹*t* = *tt* @ [*tick r*]› ‹*u* = *uu* @ [*tick r*]›
apply (*cases t rule*: *rev-cases*; *cases u rule*: *rev-cases*; *simp add*: *image-iff*)
by ((*split if-split-asm*, *safe*)+, *auto*)+
from *ftf-Sync32*[*OF this assms*(*1*)] **have** ‹*s setinterleaves* ((*tt*, *uu*), *range tick* ∪ *ev* ‘ *A*)› **by** *blast*
with *b* **show** *?thesis* **by** *blast*
qed

lemma *suffix-tick-le-ftf-imp-eq*: ‹*front-tickFree t* $\Longrightarrow$ $s \ @\ [tick\ r] \leq t \Longrightarrow s \ @\ [tick\ r] = t$›
by (*metis prefixE append-Nil2 front-tickFree-nonempty-append-imp non-tickFree-tick*

tickFree-append-iff)

lemma *synPrefix*: ‹(*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟹ ∃ *t1 u1*. *t1* ≤ *ta* ∧ *u1* ≤ *u* ∧ *s setinterleaves* ((*t1*, *u1*), *A*)›
proof (*induct* ‹(*ta*, *A*, *u*)› *arbitrary*: *s t ta u rule*: *setinterleaving.induct*)
case *1*
thus *?case* **by** *simp*
next
{ case (*2 y u*)
thus *?case*
by (*cases s*; *simp split*: *if-split-asm*)
(*metis si-empty1 insertI1 nil-le*,
metis (*no-types*) *2.prems emptyLeftNonSync emptyLeftProperty emptyLeftSelf less-eq-list-def prefix-def set-ConsD*)
} note ∗ = *this*

case (*3 x ta*)
thus *?case* **by** (*metis* (*full-types*) ∗ *setinterleaving-sym*)
next
case (*4 x ta y u*)
show *?case*
proof (*cases s*)
show ‹*s* = [] ⟹ *?case*› **by** (*metis si-empty1 insertI1 nil-le*)
next
fix *a s′*
assume ‹*s* = *a* # *s′*›
consider ‹*x* ∈ *A*› ‹*y* ∈ *A*› | ‹*x* ∈ *A*› ‹*y* ∉ *A*› | ‹*x* ∉ *A*› ‹*y* ∈ *A*› | ‹*x* ∉ *A*› ‹*y* ∉ *A*› **by** *blast*
thus *?case*
proof *cases*
assume ‹*x* ∈ *A*› ‹*y* ∈ *A*›
with *4.prems* **have** ‹*x* = *y*› ‹*a* = *y*› ‹(*s′* @ *t*) *setinterleaves* ((*ta*, *u*), *A*)›
by (*simp-all add*: ‹*s* = *a* # *s′*› *split*: *if-split-asm*)
from *4.hyps*(*1*)[*OF* ‹*x* ∈ *A*› ‹*y* ∈ *A*› ‹*x* = *y*› *this*(*3*)] **obtain** *t1 u1*
where ‹*t1* ≤ *ta*› ‹*u1* ≤ *u* ∧ *s′ setinterleaves* ((*t1*, *u1*), *A*)› **by** *blast*
hence ‹*x* # *t1* ≤ *x* # *ta* ∧ *y* # *u1* ≤ *y* # *u* ∧ *s setinterleaves* ((*x* # *t1*, *y* # *u1*), *A*)›
by (*simp add*: ‹*s* = *a* # *s′*› ‹*x* ∈ *A*› ‹*y* ∈ *A*› ‹*x* = *y*› ‹*a* = *y*›)
thus *?case* **by** *blast*
next
{ fix *x y ta u s a s′*
assume ‹*x* ∈ *A*› ‹*y* ∉ *A*› ‹*s* = *a* # *s′*›
and *prems*: ‹(*s* @ *t*) *setinterleaves* ((*x* # *ta*, *y* # *u*), *A*)›
assume *hyps*: ‹*x* ∈ *A* ⟹ *y* ∉ *A* ⟹ (*s* @ *t*) *setinterleaves* ((*x* # *ta*, *u*), *A*) ⟹
∃ *t1 u1*. *t1* ≤ *x* # *ta* ∧ *u1* ≤ *u* ∧ *s setinterleaves* ((*t1*, *u1*), *A*)›

for *s t*
from ‹$x \in A$› ‹$y \notin A$› ‹$s = a \# s'$› *prems*
have ‹$a = y$› ‹$(s' @ t)$ *setinterleaves* $((x \# ta, u), A)$›
by (*simp-all add*: ‹$s = a \# s'$› *split*: *if-split-asm*)
from *hyps*[*OF* ‹$x \in A$› ‹$y \notin A$› *this*(*2*)] **obtain** *t1 u1*
where ‹$t1 \leq x \# ta$› ‹$u1 \leq u$› ‹s' *setinterleaves* $((t1, u1), A)$› **by** *blast*
hence ‹$t1 \leq x \# ta \wedge y \# u1 \leq y \# u \wedge s$ *setinterleaves* $((t1, y \# u1),$
$A)$›
by (*cases t1*; *simp add*: ‹$s = a \# s'$› ‹$y \notin A$› ‹$a = y$›)
hence ‹$\exists t1\ u1.\ t1 \leq x \# ta \wedge u1 \leq y \# u \wedge s$ *setinterleaves* $((t1, u1),$
$A)$› **by** *blast*
} **note** $* = this$

show ‹*?case*› **if** ‹$x \in A$› **and** ‹$y \notin A$›
using $*$[*OF* ‹$x \in A$› ‹$y \notin A$› ‹$s = a \# s'$› *4.prems 4.hyps*(*2*)] **by** *simp*

show ‹*?case*› **if** ‹$x \notin A$› **and** ‹$y \in A$›
by (*metis* $*$[*OF* ‹$y \in A$› ‹$x \notin A$› ‹$s = a \# s'$›, *of u ta*]
4.hyps(*5*) *4.prems setinterleaving-sym*)
next
assume ‹$x \notin A$› **and** ‹$y \notin A$›
with *4.prems* **have** ‹$a = x \wedge (s' @ t)$ *setinterleaves* $((ta, y \# u), A) \vee$
$a = y \wedge (s' @ t)$ *setinterleaves* $((x \# ta, u), A)$› (**is** ‹*?c* $\vee$ -›)
by (*simp add*: ‹$s = a \# s'$› *split*: *if-split-asm*)
thus *?case*
proof (*elim disjE conjE*)
assume ‹$a = x$› ‹$(s' @ t)$ *setinterleaves* $((ta, y \# u), A)$›
from *4.hyps*(*3*)[*OF* ‹$x \notin A$› ‹$y \notin A$› *this*(*2*)] **obtain** *t1 u1*
where ‹$t1 \leq ta$› ‹$u1 \leq y \# u$› ‹s' *setinterleaves* $((t1, u1), A)$› **by** *blast*
hence ‹$x \# t1 \leq x \# ta \wedge u1 \leq y \# u \wedge s$ *setinterleaves* $((x \# t1, u1),$
$A)$›
by (*cases u1*; *simp add*: ‹$s = a \# s'$› ‹$a = x$› ‹$x \notin A$›)
thus *?case* **by** *blast*
next
assume ‹$a = y$› ‹$(s' @ t)$ *setinterleaves* $((x \# ta, u), A)$›
from *4.hyps*(*4*)[*OF* ‹$x \notin A$› ‹$y \notin A$› *this*(*2*)] **obtain** *t1 u1*
where ‹$t1 \leq x \# ta$› ‹$u1 \leq u$› ‹s' *setinterleaves* $((t1, u1), A)$› **by** *blast*
hence ‹$t1 \leq x \# ta \wedge y \# u1 \leq y \# u \wedge s$ *setinterleaves* $((t1, y \# u1),$
$A)$›
by (*cases t1*; *simp add*: ‹$s = a \# s'$› ‹$a = y$› ‹$y \notin A$›)
thus *?case* **by** *blast*
qed
qed
qed
qed

lemma *interleave-less-left* :
‹s *setinterleaves* $((t, u), A) \Longrightarrow t1 < t \Longrightarrow$

```
  ∃ u1 s1. u1 ≤ u ∧ s1 < s ∧ s1 setinterleaves ((t1, u1), A)›
proof (induct ‹(t, A, u)› arbitrary: s t u t1 rule: setinterleaving.induct)
 case 1
 thus ?case by simp
next
 case (2 y u)
 hence False by simp
 thus ?case by simp
next
 case (3 x t)
 thus ?case by (metis (no-types, lifting) emptyRightProperty less-eq-list-def pre-
fix-def
       nil-le2 order-less-imp-le synPrefix)
next
 case (4 x t y u)
 show ?case
 proof (cases ‹t1 = []›)
  show ‹t1 = [] ⟹ ?case›
   by (metis 4.prems(1) si-empty1 empty-setinterleaving insertI1
       list.distinct(1) nil-le order-le-imp-less-or-eq)
 next
  assume ‹t1 ≠ []›
  with ‹t1 < x # t› obtain t1' where ‹t1 = x # t1'› ‹t1' < t›
   by (metis hd-append2 less-eq-list-def prefix-def less-list-def less-tail list.exhaust-sel
list.sel(1, 3))
  consider ‹x ∈ A› ‹y ∈ A› | ‹x ∈ A› ‹y ∉ A› | ‹x ∉ A› ‹y ∈ A› | ‹x ∉ A› ‹y ∉
A› by blast
  thus ?case
  proof cases
   assume ‹x ∈ A› ‹y ∈ A›
   with 4.prems(1) obtain s' where ‹x = y› ‹s = y # s'› ‹s' setinterleaves ((t,
u), A)›
    by (simp split: if-split-asm) blast
   from 4.hyps(1)[OF ‹x ∈ A› ‹y ∈ A› ‹x = y› this(3) ‹t1' < t›]
   obtain u1 s1 where ‹u1 ≤ u› ‹s1 < s'› ‹s1 setinterleaves ((t1', u1), A)› by
blast
    hence ‹y # u1 ≤ y # u ∧ y # s1 < s ∧ (y # s1) setinterleaves ((t1, y #
u1), A)›
     by (simp add: ‹t1 = x # t1'› ‹x = y› ‹y ∈ A› ‹s = y # s'›)
    thus ?case by blast
  next
   assume ‹x ∈ A› ‹y ∉ A›
   with 4.prems(1) obtain s' where ‹s = y # s'› ‹s' setinterleaves ((x # t,
u), A)›
    by (simp split: if-split-asm) blast
   from 4.hyps(2)[OF ‹x ∈ A› ‹y ∉ A› this(2) less-cons[THEN iffD2, OF ‹t1'
< t›]]
   obtain u1 s1 where ‹u1 ≤ u› ‹s1 < s'› ‹s1 setinterleaves ((x # t1', u1),
A)› by blast
```

hence ‹$y \# u1 \leq y \# u \wedge y \# s1 < s \wedge (y \# s1)$ *setinterleaves* $((t1, y \# u1), A)$›
by (*simp add*: ‹$t1 = x \# t1'$› ‹$y \notin A$› ‹$s = y \# s'$›)
thus *?case* **by** *blast*
next
assume ‹$x \notin A$› ‹$y \in A$›
with *4.prems*(*1*) **obtain** s' **where** ‹$s = x \# s'$› ‹s' *setinterleaves* $((t, y \# u), A)$›
by (*simp split*: *if-split-asm*) *blast*
from *4.hyps*(*5*)[*OF* ‹$x \notin A$› - *this*(*2*) ‹$t1' < t$›] ‹$y \in A$›
obtain *u1 s1* **where** ‹$u1 \leq y \# u$› ‹$s1 < s'$› ‹*s1 setinterleaves* $((t1', u1), A)$› **by** *blast*
hence ‹$u1 \leq y \# u \wedge x \# s1 < s \wedge (x \# s1)$ *setinterleaves* $((t1, u1), A)$›
by (*cases u1*; *simp add*: ‹$t1 = x \# t1'$› ‹$x \notin A$› ‹$s = x \# s'$›)
thus *?case* **by** *blast*
next
assume ‹$x \notin A$› ‹$y \notin A$›
with *4.prems*(*1*) **obtain** s'
where ‹$s = x \# s' \wedge s'$ *setinterleaves* $((t, y \# u), A) \vee$
$s = y \# s' \wedge s'$ *setinterleaves* $((x \# t, u), A)$›
by (*simp split*: *if-split-asm*) *blast*
thus *?case*
proof (*elim disjE conjE*)
assume ‹$s = x \# s'$› ‹s' *setinterleaves* $((t, y \# u), A)$›
from *4.hyps*(*3*)[*OF* ‹$x \notin A$› ‹$y \notin A$› *this*(*2*) ‹$t1' < t$›]
obtain *u1 s1* **where** ‹$u1 \leq y \# u$› ‹$s1 < s'$› ‹*s1 setinterleaves* $((t1', u1), A)$› **by** *blast*
hence ‹$u1 \leq y \# u \wedge x \# s1 < s \wedge (x \# s1)$ *setinterleaves* $((t1, u1), A)$›
by (*cases u1*; *simp add*: ‹$t1 = x \# t1'$› ‹$x \notin A$› ‹$s = x \# s'$›)
thus *?case* **by** *blast*
next
assume ‹$s = y \# s'$› ‹s' *setinterleaves* $((x \# t, u), A)$›
from *4.hyps*(*4*)[*OF* ‹$x \notin A$› ‹$y \notin A$› *this*(*2*) *less-cons*[*THEN iffD2*, *OF* ‹$t1' < t$›]]
obtain *u1 s1* **where** ‹$u1 \leq u$› ‹$s1 < s'$› ‹*s1 setinterleaves* $((x \# t1', u1), A)$› **by** *blast*
hence ‹$y \# u1 \leq y \# u \wedge y \# s1 < s \wedge (y \# s1)$ *setinterleaves* $((t1, y \# u1), A)$›
by (*simp add*: ‹$t1 = x \# t1'$› ‹$y \notin A$› ‹$s = y \# s'$›)
thus *?case* **by** *blast*
qed
qed
qed
qed

lemma *interleave-less-right* :
‹s *setinterleaves* $((t, u), A) \Longrightarrow u1 < u \Longrightarrow$
$\exists t1\ s1.\ t1 \leq t \wedge s1 < s \wedge s1$ *setinterleaves* $((t1, u1), A)$›
by (*metis* (*no-types*) *interleave-less-left setinterleaving-sym*)

lemma *interleave-le-left* :
‹*s setinterleaves* ((*t*, *u*), *A*) ⟹ *t1* ≤ *t* ⟹
∃ *u1 s1*. *u1* ≤ *u* ∧ *s1* ≤ *s* ∧ *s1 setinterleaves* ((*t1*, *u1*), *A*)›
by (*metis interleave-less-left order-le-less*)

lemma *interleave-le-right* :
‹*s setinterleaves* ((*t*, *u*), *A*) ⟹ *u1* ≤ *u* ⟹
∃ *t1 s1*. *t1* ≤ *t* ∧ *s1* ≤ *s* ∧ *s1 setinterleaves* ((*t1*, *u1*), *A*)›
by (*metis* (*no-types*) *interleave-le-left setinterleaving-sym*)

lemma *SyncWithTick-imp-NTF1*:
assumes ‹(*s* @ [*tick r*]) *setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*)›
and ‹*t* ∈ ℱ *P*› **and** ‹*u* ∈ ℱ *Q*›
shows ‹∃ *t u X-P X-Q*. (*t*, *X-P*) ∈ ℱ *P* ∧ (*u*, *X-Q*) ∈ ℱ *Q* ∧
s setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
X − {*tick r*} = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *A*) ∪ *X-P* ∩ *X-Q*›
proof −
from *SyncWithTick-imp-NTF*[*OF assms*(*1*)] *assms*(*2*, *3*) *is-processT2-TR*
obtain *t1 u1*
where ∗ : ‹*t* = *t1* @ [*tick r*]› ‹*u* = *u1* @ [*tick r*]›
‹*s setinterleaves* ((*t1*, *u1*), *range tick* ∪ *ev* ‘ *A*)› **by** *blast*
from ∗(*1*, *2*) *assms*(*2*, *3*) **have** *E*: ‹(*t1*, *X* − {*tick r*}) ∈ ℱ *P*› ‹(*u1*, *X* − {*tick r*}) ∈ ℱ *Q*›
by (*simp-all add*: *T-F process-charn*)
with ∗(*3*) **show** *?thesis* **by** *blast*
qed

lemma *interleave-size*:
‹*s setinterleaves* ((*t*, *u*), *C*) ⟹ *length s* = *length t* + *length u* − *length* (*filter* (λ*x*. *x* ∈ *C*) *t*)›
apply (*induct* ‹(*t*, *C*, *u*)› *arbitrary*: *t u s rule*: *setinterleaving.induct*)
apply (*simp*; *fail*)
apply (*metis add-diff-cancel-left′ emptyLeftProperty filter.simps*(*1*))
apply (*metis add-diff-cancel-right′ emptyLeftNonSync*
emptyRightProperty filter-False setinterleaving-sym)
by (*simp split*: *if-split-asm*;
metis (*no-types*) [[*metis-verbose* = *false*]] *Suc-diff-le add-Suc-right*
length-Cons length-filter-le trans-le-add1)

lemma *interleave-eq-size*:
‹*s setinterleaves* ((*t*, *u*), *C*) ⟹ *s′ setinterleaves* ((*t*, *u*), *C*) ⟹ *length s* = *length s′*›

by (*simp add*: *interleave-size*)

lemma *ftf-Sync*:
assumes ‹*front-tickFree t*› **and** ‹*front-tickFree u*›
and ‹*s setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*)›
shows ‹*front-tickFree s*›
proof (*cases* ‹*tickFree s*›)
show ‹*tickFree s* ⟹ *front-tickFree s*› **by** (*fact tickFree-imp-front-tickFree*)
next
assume ‹¬ *tickFree s*›
hence ‹∃ *s1 s2 r*. *tickFree s1* ∧ *s* = (*s1* @ [*tick r*]) @ *s2*›
proof (*induct s rule*: *rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc a s*)
from ‹¬ *tickFree* (*s* @ [*a*])› **consider** *r* **where** ‹*tickFree s*› ‹*a* = *tick r*› | ‹¬ *tickFree s*›
by (*auto simp add*: *tickFree-def*)
thus *?case*
proof *cases*
show ‹*tickFree s* ⟹ *a* = *tick r* ⟹
∃ *s1 s2 r*. *tickFree s1* ∧ *s* @ [*a*] = (*s1* @ [*tick r*]) @ *s2*› **for** *r* **by** *blast*
next
assume ‹¬ *tickFree s*›
with *snoc.hyps* **obtain** *s1 s2 r* **where** ‹*tickFree s1*› ‹*s* = (*s1* @ [*tick r*]) @ *s2*› **by** *blast*
hence ‹*tickFree s1* ∧ *s* @ [*a*] = (*s1* @ [*tick r*]) @ (*s2* @ [*a*])› **by** *simp*
thus *?case* **by** *blast*
qed
qed
then obtain *s1 s2 r* **where** ‹*tickFree s1*› ‹*s* = (*s1* @ [*tick r*]) @ *s2*› **by** *blast*
from *synPrefix*[*of* ‹*s1* @ [*tick r*]› *s2*, *folded* ‹*s* = (*s1* @ [*tick r*]) @ *s2*›, *OF assms*(*3*)]
obtain *t1 u1*
where ∗ : ‹*t1* ≤ *t*› ‹*u1* ≤ *u*›
‹(*s1* @ [*tick r*]) *setinterleaves* ((*t1*, *u1*), *range tick* ∪ *ev* ‘ *A*)› **by** *blast*
from ∗(*1*, *2*) *assms*(*1*, *2*) **have** ‹*front-tickFree t1*› ‹*front-tickFree u1*›
by (*metis prefixE front-tickFree-dw-closed*)+
from *SyncWithTick-imp-NTF*[*OF* ∗(*3*) *this*]
obtain *t2 u2* **where** ‹*t1* = *t2* @ [*tick r*]› ‹*u1* = *u2* @ [*tick r*]›
‹*s1 setinterleaves* ((*t2*, *u2*), *range tick* ∪ *ev* ‘ *A*)› **by** *blast*
from ‹*t1* ≤ *t*› ‹*t1* = *t2* @ [*tick r*]› *assms*(*1*) *suffix-tick-le-ftf-imp-eq* **have** ‹*t* = *t1*› **by** *metis*
from ‹*u1* ≤ *u*› ‹*u1* = *u2* @ [*tick r*]› *assms*(*2*) *suffix-tick-le-ftf-imp-eq* **have** ‹*u* = *u1*› **by** *metis*
have ‹(*s1* @ [*tick r*]) *setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*)›
by (*simp add*: ∗(*3*) ‹*t* = *t1*› ‹*u* = *u1*›)
with *assms*(*3*) *interleave-eq-size* **have** ‹*length s* = *length* (*s1* @ [*tick r*])› **by** *blast*

with ‹$s = (s1 \mathbin{@} [tick\ r]) \mathbin{@} s2$› **have** ‹$s = s1 \mathbin{@} [tick\ r]$› **by** *simp*
thus ‹*front-tickFree s*› **by** (*simp add*: ‹*tickFree s1*› *front-tickFree-append*)
qed

8.3 The Sync Operator Definition

lift-definition *Sync* :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a set*, ('*a*, '*r*) *process*$_{ptick}$] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
(‹(*3*(- ⟦-⟧/ -))› [*70*, *0*, *71*] *70*)
is ‹$\lambda P\ A\ Q.\ (\{(s, R).\ \exists t\ u\ X\ Y.\ (t, X) \in \mathcal{F}\ P \wedge (u, Y) \in \mathcal{F}\ Q \wedge$
$s\ setinterleaves\ ((t, u), range\ tick \cup ev\ `\ A) \wedge$
$R = (X \cup Y) \cap (range\ tick \cup ev\ `\ A) \cup X \cap Y\} \cup$
$\{(s, R).\ \exists t\ u\ r\ v.\ ftF\ v \wedge (tF\ r \vee v = []) \wedge s = r \mathbin{@} v \wedge$
$r\ setinterleaves\ ((t, u), range\ tick \cup ev\ `\ A) \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\},$
$\{s.\ \exists t\ u\ r\ v.\ ftF\ v \wedge (tF\ r \vee v = []) \wedge s = r \mathbin{@} v \wedge$
$r\ setinterleaves\ ((t, u), range\ tick \cup ev\ `\ A) \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\})$›
proof −
show ‹*?thesis P A Q*› (**is** ‹*is-process*(*?f*, *?d P Q*)›) **for** *P Q* :: ‹('*a*, '*r*) *process*$_{ptick}$› **and** *A*
proof (*unfold is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv*, *intro conjI impI allI*)
show ‹([], {}) ∈ *?f*›
by *simp* (*metis Int-commute Int-empty-right si-empty1*
Un-empty insert-iff is-processT1)
next
show ‹(*s*, *X*) ∈ *?f* ⟹ *ftF s*› **for** *s X*
apply (*simp*, *elim disjE*)
using *F-imp-front-tickFree ftf-Sync* **apply** *blast*
by (*metis D-imp-front-tickFree T-imp-front-tickFree append.right-neutral front-tickFree-append ftf-Sync*)

next
fix *s t* **assume** ‹(*s* @ *t*, {}) ∈ *?f*›
then consider (*fail*) *t-P t-Q X-P X-Q* **where**
‹(*t-P*, *X-P*) ∈ $\mathcal{F}$ *P*› ‹(*t-Q*, *X-Q*) ∈ $\mathcal{F}$ *Q*›
‹(*s* @ *t*) *setinterleaves* ((*t-P*, *t-Q*), *range tick* ∪ *ev* ‘ *A*)›
‹(*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *A*) ∪ *X-P* ∩ *X-Q* = {}›
| (*div*) *t′ u r v* **where** ‹*ftF v*› ‹*tF r* ∨ *v* = []› ‹*s* @ *t* = *r* @ *v*›
‹*r setinterleaves* ((*t′*, *u*), *range tick* ∪ *ev* ‘ *A*)›
‹*t′* ∈ $\mathcal{D}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q* ∨ *t′* ∈ $\mathcal{D}$ *Q* ∧ *u* ∈ $\mathcal{T}$ *P*›

by (*smt* (*verit*, *ccfv-SIG*) *Un-iff case-prod-conv mem-Collect-eq*)
thus ‹(*s*, {}) ∈ *?f*›
proof *cases*
case *fail*
from *fail*(*3*)[*THEN synPrefix*] **show** ‹(*s*, {}) ∈ *?f*›
apply (*elim exE conjE*)

```
      apply (drule fail(1, 2)[THEN F-T, THEN is-processT3-TR])+
      by simp (metis Int-empty-left T-F Un-absorb)
    next
     case div
     show ‹(s, {}) ∈ ?f›
     proof (cases ‹length r ≤ length s›)
      assume ‹length r ≤ length s›
      with div have ‹front-tickFree (take (length s − length r) v) ∧
                    (tickFree r ∨ take (length s − length r) v = []) ∧
                    s = r @ take (length s − length r) v›
       by simp (metis append-eq-conv-conj append-take-drop-id
          front-tickFree-dw-closed take-all-iff take-append)
      show ‹(s, {}) ∈ ?f› by simp (use ‹?this› div in blast)
     next
      assume ‹¬ length r ≤ length s›
      with div obtain r' where ‹r = s @ r'›
       by (metis append-eq-append-conv-if append-take-drop-id)
      from div ‹r = s @ r'› synPrefix obtain t1 u1
       where ** : ‹t1 ≤ t'› ‹u1 ≤ u›
        ‹s setinterleaves ((t1, u1), range tick ∪ ev ‘ A)› by metis
      from div **(1, 2) D-T is-processT3-TR
      have ‹t1 ∈ 𝒯 P ∧ u1 ∈ 𝒯 Q ∨ t1 ∈ 𝒯 Q ∧ u1 ∈ 𝒯 P› by blast
      thus ‹(s, {}) ∈ ?f›
     by simp (metis **(3) Int-Un-eq(1) T-F setinterleaving-sym sup-bot.right-neutral)
     qed
    qed
  next
    fix s X Y
    assume ‹(s, Y) ∈ ?f ∧ X ⊆ Y›
    then consider ‹s ∈ ?d P Q›
     | (fail) s-P s-Q X-P X-Q
     where ‹(s-P, X-P) ∈ ℱ P› ‹(s-Q, X-Q) ∈ ℱ Q›
      ‹s setinterleaves ((s-P, s-Q), range tick ∪ ev ‘ A)›
      ‹Y = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ A) ∪ X-P ∩ X-Q›
     by simp blast
    thus ‹(s, X) ∈ ?f›
    proof cases
     show ‹s ∈ ?d P Q ⟹ (s, X) ∈ ?f› by blast
    next
     case fail
    have ‹(s-P, X-P ∩ X) ∈ ℱ P› using fail(1) by (meson inf-le1 process-charn)
     moreover have ‹(s-Q, X-Q ∩ X) ∈ ℱ Q› using fail(2) by (meson inf-le1
process-charn)
    moreover have ‹X = (X-P ∩ X ∪ X-Q ∩ X) ∩ (range tick ∪ ev ‘ A) ∪ X-P
∩ X ∩ (X-Q ∩ X)›
       by (subst ‹(s, Y) ∈ ?f ∧ X ⊆ Y›[THEN conjunct2, THEN Int-absorb1,
symmetric])
        (use fail(4) in blast)
     ultimately show ‹(s, X) ∈ ?f› using fail(3) by simp blast
```

```
  qed
 next
  fix s X Y
  assume ‹(s, X) ∈ ?f ∧ (∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ ?f)›
  then consider ‹s ∈ ?d P Q› | ‹(s, X) ∈ ?f ∧ s ∉ ?d P Q› by linarith
  thus ‹(s, X ∪ Y) ∈ ?f›
  proof cases
   show ‹s ∈ ?d P Q ⟹ (s, X ∪ Y) ∈ ?f› by blast
  next
   assume ‹(s, X) ∈ ?f ∧ s ∉ ?d P Q›
   then obtain s-P X-P s-Q X-Q
    where assms : ‹(s-P, X-P) ∈ ℱ P› ‹(s-Q, X-Q) ∈ ℱ Q›
      ‹s setinterleaves ((s-P, s-Q), range tick ∪ ev ‘ A)›
      ‹X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ A) ∪ X-P ∩ X-Q›
    by simp blast
   have assms4 : ‹c ∈ Y ⟹ ((s @ [c]) setinterleaves ((t1, u1), range tick ∪ ev
‘ A) ⟹
                ((t1, {}) ∈ ℱ P ⟶ (u1, {}) ∉ ℱ Q) ∧
                ((t1, {}) ∈ ℱ Q ⟶ (u1, {}) ∉ ℱ P))› for c t1 u1
    using ‹(s, X) ∈ ?f ∧ s ∉ ?d P Q›
      ‹(s, X) ∈ ?f ∧ (∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ ?f)›
   by simp (metis (no-types, lifting) Int-empty-left setinterleaving-sym sup-bot.right-neutral)
   show ‹(s, X ∪ Y) ∈ ?f›
   proof−
    define Y1 Y2 where Y1-def: ‹Y1 ≡ Y ∩ (range tick ∪ ev ‘ A)›
     and Y2-def: ‹Y2 ≡ Y − (range tick ∪ ev ‘ A)›
    hence ‹Y1 ∪ Y2 = Y› by blast
    have $ : ‹∀ z∈Y1. (s-P @ [z], {}) ∉ ℱ P ∨ (s-Q @ [z], {}) ∉ ℱ Q›
    proof (rule ccontr)
     assume ‹¬ (∀ z∈Y1. (s-P @ [z], {}) ∉ ℱ P ∨ (s-Q @ [z], {}) ∉ ℱ Q)›
     then obtain z1 where ∗ : ‹z1 ∈ Y1› ‹(s-P @ [z1], {}) ∈ ℱ P› ‹(s-Q @
[z1], {}) ∈ ℱ Q› by blast
     have ‹(s @ [z1]) setinterleaves ((s-P @ [z1], s-Q @ [z1]), range tick ∪ ev
‘ A)›
      by (metis ∗(1) IntD2 Y1-def addSync assms(3))
     with ∗ assms4 show False by (metis IntD1 Y1-def)
    qed
    define Z1 Z2 where Z1-def: ‹Z1 ≡ Y1 ∩ {z. (s-P @ [z], {}) ∉ ℱ P}›
     and Z2-def: ‹Z2 ≡ Y1 − {z. (s-P @ [z], {}) ∉ ℱ P}›
    hence ‹Y1 = Z1 ∪ Z2› by blast
    have $$ : ‹∀ y∈Y2. (s-P @ [y], {}) ∉ ℱ P ∧ (s-Q @ [y], {}) ∉ ℱ Q›
    proof (rule ccontr)
     assume ‹¬ (∀ y∈Y2. (s-P @ [y], {}) ∉ ℱ P ∧ (s-Q @ [y], {}) ∉ ℱ Q)›
     then obtain y1 where ∗ : ‹y1 ∈ Y2› ‹((s-P @ [y1], {}) ∈ ℱ P ∨ (s-Q
@ [y1], {}) ∈ ℱ Q)› by blast
     hence ‹(s @ [y1]) setinterleaves ((s-P @ [y1], s-Q), range tick ∪ ev ‘ A) ∨
          (s @ [y1]) setinterleaves ((s-P, s-Q @ [y1]), range tick ∪ ev ‘ A)›
      by (simp add: Y2-def addNonSync assms(3))
     with ∗ assms show False
```

```
        by (metis DiffD1 DiffD2 Y2-def addNonSync assms(1, 2, 3) assms4
is-processT4-empty)
      qed

      define X-P' where ‹X-P' = X-P ∪ Z1 ∪ Y2›
       hence f1: ‹(s-P, X-P') ∈ ℱ P› by (simp add: $$ Z1-def assms(1) pro-
cess-charn)
      define X-Q' where ‹X-Q' = X-Q ∪ Z2 ∪ Y2›
      hence f2: ‹(s-Q, X-Q') ∈ ℱ Q› using $ $$ by (simp add: Z2-def assms(2)
is-processT5)
      have ‹(X-P ∪ X-Q ∪ Y1 ∪ Y2) ∩ range tick ∪ ev ‘ A =
          (X-P ∪ X-Q) ∩ range tick ∪ ev ‘ A ∪ Y1›
        unfolding Y1-def Y2-def by auto
      hence ‹(X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ A) ∪ X-P ∩ X-Q ∪ Y =
            (X-P' ∪ X-Q') ∩ (range tick ∪ ev ‘ A) ∪ X-P' ∩ X-Q'›
        using X-P'-def X-Q'-def ‹Y1 = Z1 ∪ Z2› ‹Y1 ∪ Y2 = Y› by blast
      thus ‹(s, X ∪ Y) ∈ ?f› by simp (use f1 f2 assms(3, 4) in blast)
    qed
  qed
 next
  show processT9: ‹s @ [✓(res)] ∈ ?d P Q ⟹ s ∈ ?d P Q› for s res
  proof −
   { fix s t u r v and res :: 'r and P Q
    assume assms : ‹front-tickFree v› ‹tickFree r ∨ v = []› ‹s @ [✓(res)] = r @
v›
      ‹r setinterleaves ((t, u), range tick ∪ ev ‘ A)› ‹t ∈ 𝒟 P› ‹u ∈ 𝒯 Q›
    from assms(2) have ‹s ∈ ?d P Q›
    proof (elim disjE)
      assume ‹tickFree r›
      with assms(1, 3) have ‹s @ [✓(res)] = r @ (butlast v @ [✓(res)])›
        by (cases v rule: rev-cases) auto
      with ‹tickFree r› assms(1, 4, 5, 6) show ‹s ∈ ?d P Q›
       by simp (use front-tickFree-iff-tickFree-butlast tickFree-imp-front-tickFree
in blast)
    next
      assume ‹v = []›
      with assms(3) obtain r' where ‹r = r' @ [✓(res)]› ‹s = r'› by auto
     with D-imp-front-tickFree SyncWithTick-imp-NTF assms(4, 5, 6) is-processT2-TR
      obtain t1 u1 where ‹t = t1 @ [✓(res)]› ‹u = u1 @ [✓(res)]›
        ‹r' setinterleaves ((t1, u1), range tick ∪ ev ‘ A)› by metis
      with assms(5, 6) show ‹s ∈ ?d P Q›
        apply (simp add: ‹s = r'›)
       by (metis prefixI ‹v = []› append.right-neutral assms(1) is-processT3-TR
is-processT9)
    qed
   }
   thus ‹s @ [✓(res)] ∈ ?d P Q ⟹ s ∈ ?d P Q› for s by simp blast
  qed
```

fix *s X res*
assume ‹(*s* @ [✓(*res*)], {}) ∈ *?f*›
hence ‹*s* @ [✓(*res*)] ∈ *?d P Q* ∨
(∃ *s-P s-Q X-P X-Q*. (*s-P*, *X-P*) ∈ ℱ *P* ∧ (*s-Q*, *X-Q*) ∈ ℱ *Q* ∧
(*s* @ [✓(*res*)]) *setinterleaves* ((*s-P*, *s-Q*), *range tick* ∪ *ev* ‘ *A*) ∧
{} = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *A*) ∪ *X-P* ∩ *X-Q*)›
by *auto*
thus ‹(*s*, *X* − {✓(*res*)}) ∈ *?f*›
proof (*elim disjE exE conjE*)
from *processT9* **show** ‹*s* @ [✓(*res*)] ∈ *?d P Q* ⟹ (*s*, *X* − {✓(*res*)}) ∈ *?f*›
by *blast*
next
fix *s-P s-Q X-P X-Q res*
assume ∗ : ‹(*s-P*, *X-P*) ∈ ℱ *P*› ‹(*s-Q*, *X-Q*) ∈ ℱ *Q*›
‹(*s* @ [✓(*res*)]) *setinterleaves* ((*s-P*, *s-Q*), *range tick* ∪ *ev* ‘ *A*)›
‹{} = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *A*) ∪ *X-P* ∩ *X-Q*›
from *SyncWithTick-imp-NTF1*[*OF* ∗(*3*) ∗(*1*, *2*)[*THEN F-T*]]
show ‹(*s*, *X* − {✓(*res*)}) ∈ *?f*› **by** *simp*
qed
next
show ‹*s* ∈ *?d P Q* ∧ *tickFree s* ∧ *front-tickFree t* ⟹ *s* @ *t* ∈ *?d P Q*› **for** *s t*
using *front-tickFree-append* **by** *fastforce*
next
show ‹*s* ∈ *?d P Q* ⟹ (*s*, *X*) ∈ *?f*› **for** *s X* **by** *blast*
qed
qed

8.4 The Projections

lemma *F-Sync* :
‹ℱ (*P* ⟦ *A* ⟧ *Q*) =
{(*s*, *R*). ∃ *t u X Y*. (*t*, *X*) ∈ ℱ *P* ∧ (*u*, *Y*) ∈ ℱ *Q* ∧
s setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
R = (*X* ∪ *Y*) ∩ (*range tick* ∪ *ev* ‘ *A*) ∪ *X* ∩ *Y*} ∪
{(*s*, *R*). ∃ *t u r v*. *front-tickFree v* ∧ (*tickFree r* ∨ *v* = []) ∧ *s* = *r* @ *v* ∧
r setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
(*t* ∈ 𝒟 *P* ∧ *u* ∈ 𝒯 *Q* ∨ *t* ∈ 𝒟 *Q* ∧ *u* ∈ 𝒯 *P*)}›
by (*simp add*: *Failures.rep-eq Sync.rep-eq FAILURES-def*)

lemma *D-Sync* :
‹𝒟 (*P* ⟦ *A* ⟧ *Q*) =
{*s*. ∃ *t u r v*. *front-tickFree v* ∧ (*tickFree r* ∨ *v* = []) ∧ *s* = *r* @ *v* ∧
r setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
(*t* ∈ 𝒟 *P* ∧ *u* ∈ 𝒯 *Q* ∨ *t* ∈ 𝒟 *Q* ∧ *u* ∈ 𝒯 *P*)}›
by (*simp add*: *Divergences.rep-eq Sync.rep-eq DIVERGENCES-def*)

lemma *T-Sync* :
‹𝒯 (*P* ⟦ *A* ⟧ *Q*) =

$\{s.\ \exists t\ u.\ t \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q \wedge s\ setinterleaves\ ((t,\ u),\ range\ tick \cup ev\ `\ A)\} \cup$
$\{s.\ \exists t\ u\ r\ v.\ front\text{-}tickFree\ v \wedge (tickFree\ r \vee v = []) \wedge$
$s = r\ @\ v \wedge r\ setinterleaves\ ((t,\ u),\ range\ tick \cup ev\ `\ A) \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\}\rangle$
by (*simp add*: *Traces.rep-eq TRACES-def Failures.rep-eq*[*symmetric*] *F-Sync*) *blast*

lemmas *Sync-projs* = *F-Sync D-Sync T-Sync*

8.5 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 $P\ ||\ Q \equiv (P\ [\![\ UNIV\ ]\!]\ Q)$

lemma *setinterleaving-UNIV-iff* : $\langle s\ setinterleaves\ ((t,\ u),\ UNIV) \longleftrightarrow s = t \wedge s = u\rangle$
for $s :: \langle('a,\ 'r)\ trace_{ptick}\rangle$
by (*induct* $\langle(u,\ UNIV :: ('a,\ 'r)\ event_{ptick}\ set,\ t)\rangle$
arbitrary: *s t u rule*: *setinterleaving.induct*) *auto*

lemma *F-Par* :
$\langle\mathcal{F}\ (P\ ||\ Q) = \{(s,\ R).\ \exists X\ Y.\ (s,\ X) \in \mathcal{F}\ P \wedge (s,\ Y) \in \mathcal{F}\ Q \wedge R = X \cup Y \cup X \cap Y\} \cup$
$\{(s,\ R).\ \exists t\ v.\ ftF\ v \wedge (tF\ t \vee v = []) \wedge s = t\ @\ v \wedge$
$(t \in \mathcal{D}\ P \wedge t \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge t \in \mathcal{T}\ P)\}\rangle$
by (*auto simp add*: *F-Sync setinterleaving-UNIV-iff*)

lemma *D-Par* :
$\langle\mathcal{D}\ (P\ ||\ Q) = \{s.\ \exists t\ v.\ ftF\ v \wedge (tF\ t \vee v = []) \wedge s = t\ @\ v \wedge$
$(t \in \mathcal{D}\ P \wedge t \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge t \in \mathcal{T}\ P)\}\rangle$
by (*simp add*: *D-Sync setinterleaving-UNIV-iff*)

lemma *T-Par* :
$\langle\mathcal{T}\ (P\ ||\ Q) = \mathcal{T}\ P \cap \mathcal{T}\ Q \cup \{t\ @\ v|\ t\ v.\ ftF\ v \wedge (tF\ t \vee v = []) \wedge t \in \mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P\}\rangle$
by (*simp add*: *set-eq-iff T-Sync setinterleaving-UNIV-iff*) *blast*

lemmas *Par-projs* = *F-Par D-Par T-Par*

lemma *superset-T-Inter* : $\langle\{t.\ tF\ t \wedge t \in \mathcal{T}\ P \cup \mathcal{T}\ Q\} \subseteq \mathcal{T}\ (P\ |||\ Q)\rangle$
proof (*rule subsetI*, *clarify*)
fix *t* **assume** $\langle tF\ t\rangle$ $\langle t \in \mathcal{T}\ P \cup \mathcal{T}\ Q\rangle$
from $\langle tF\ t\rangle$ **have** $\langle t\ setinterleaves\ ((t,\ []),\ range\ tick)\rangle$
by (*metis disjoint-iff emptyLeftSelf setinterleaving-sym tickFree-def*)

with ‹$t \in \mathcal{T}\ P \cup \mathcal{T}\ Q$› **show** ‹$t \in \mathcal{T}\ (P \mid\mid\mid Q)$›
by (*simp add*: *T-Sync*) (*use Nil-elem-T setinterleaving-sym* **in** *blast*)
qed

lemma *superset-D-Inter* : ‹$\mathcal{D}\ P \cup \mathcal{D}\ Q \subseteq \mathcal{D}\ (P \mid\mid\mid Q)$›
proof (*rule subsetI*)
fix t **assume** ‹$t \in \mathcal{D}\ P \cup \mathcal{D}\ Q$›
define t' **where** ‹$t' \equiv$ *if tF t then t else butlast t*›
from ‹$t \in \mathcal{D}\ P \cup \mathcal{D}\ Q$› *D-imp-front-tickFree front-tickFree-iff-tickFree-butlast*
have ‹*tF t'*› **unfolding** *t'-def* **by** *auto*
with ‹$t \in \mathcal{D}\ P \cup \mathcal{D}\ Q$› **have** ‹$t' \in \mathcal{D}\ P \cup \mathcal{D}\ Q$›
by (*metis t'-def D-imp-front-tickFree Un-iff butlast-snoc*
is-processT9 nonTickFree-n-frontTickFree)
from ‹*tF t'*› **have** ‹t' *setinterleaves* ((t', []), *range tick*)›
by (*metis disjoint-iff emptyLeftSelf setinterleaving-sym tickFree-def*)
with ‹$t' \in \mathcal{D}\ P \cup \mathcal{D}\ Q$› **have** ‹$t' \in \mathcal{D}\ (P \mid\mid\mid Q)$›
by (*simp add*: *D-Sync*) (*use front-tickFree-Nil Nil-elem-T* **in** *blast*)
thus ‹$t \in \mathcal{D}\ (P \mid\mid\mid Q)$›
by (*metis* ‹*tF t'*› *butlast-snoc front-tickFree-iff-tickFree-butlast*
front-tickFree-single is-processT7 nonTickFree-n-frontTickFree t'-def)
qed

8.6 Continuity Rule

lemma *Sync-commute*: ‹$P \llbracket S \rrbracket\ Q = Q \llbracket S \rrbracket\ P$›
— This will simplify the following proofs.
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}\ (P \llbracket S \rrbracket\ Q) \Longrightarrow s \in \mathcal{D}\ (Q \llbracket S \rrbracket\ P)$› **for** s **by** (*simp add*: *D-Sync*)
blast
next
show ‹$s \in \mathcal{D}\ (Q \llbracket S \rrbracket\ P) \Longrightarrow s \in \mathcal{D}\ (P \llbracket S \rrbracket\ Q)$› **for** s **by** (*simp add*: *D-Sync*)
blast
next
show ‹$(s, X) \in \mathcal{F}\ (P \llbracket S \rrbracket\ Q) \Longrightarrow (s, X) \in \mathcal{F}\ (Q \llbracket S \rrbracket\ P)$› **for** $s\ X$
by (*simp add*: *F-Sync*, *elim disjE exE*)
(*metis* (*no-types*) *Int-commute Un-commute setinterleaving-sym*, *blast*)
next
show ‹$(s, X) \in \mathcal{F}\ (Q \llbracket S \rrbracket\ P) \Longrightarrow (s, X) \in \mathcal{F}\ (P \llbracket S \rrbracket\ Q)$› **for** $s\ X$
by (*simp add*: *F-Sync*, *elim disjE exE*)
(*metis* (*no-types*) *Int-commute Un-commute setinterleaving-sym*, *blast*)
qed

lemma *mono-Sync* : ‹$P \sqsubseteq P' \Longrightarrow Q \sqsubseteq Q' \Longrightarrow P \llbracket A \rrbracket\ Q \sqsubseteq P' \llbracket A \rrbracket\ Q'$›
for P :: ‹$('a, 'r)$ $process_{ptick}$›
proof −
have ‹$P \llbracket A \rrbracket\ S \sqsubseteq Q \llbracket A \rrbracket\ S$› **if** ‹$P \sqsubseteq Q$› **for** $P\ S\ Q$ :: ‹$('a, 'r)$ $process_{ptick}$›
proof (*unfold le-approx-def*, *safe*)

```
    from le-approx1[OF ‹P ⊑ Q›] le-approx-lemma-T[OF ‹P ⊑ Q›]
    show ‹s ∈ 𝒟 (Q ⟦A⟧ S) ⟹ s ∈ 𝒟 (P ⟦A⟧ S)› for s by (simp add: D-Sync)
blast
  next
    show ‹s ∉ 𝒟 (P ⟦A⟧ S) ⟹ X ∈ ℛa (P ⟦A⟧ S) s ⟹ X ∈ ℛa (Q ⟦A⟧ S) s›
for s X
      by (simp add: D-Sync Refusals-after-def F-Sync, elim disjE)
        (metis F-T front-tickFree-Nil proc-ord2a[OF that] self-append-conv, blast)
  next
    show ‹s ∉ 𝒟 (P ⟦A⟧ S) ⟹ X ∈ ℛa (Q ⟦A⟧ S) s ⟹ X ∈ ℛa (P ⟦A⟧ S) s›
for s X
      apply (simp add: D-Sync Refusals-after-def F-Sync, elim disjE)
      using le-approx-lemma-F that apply blast

      by (smt (verit, ccfv-SIG) in-mono le-approx1 le-approx-lemma-T that)
  next
    fix s
    assume ‹s ∈ min-elems (𝒟 (P ⟦A⟧ S))›
    hence ‹s ∈ 𝒟 (P ⟦A⟧ S)› by (simp add: elem-min-elems)
    then obtain t u r
      where ∗ : ‹r ≤ s› ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 S ∨ t ∈ 𝒟 S ∧ u ∈ 𝒯 P›
        ‹r setinterleaves ((t, u), range tick ∪ ev ‘ A)›
      by (simp add: D-Sync less-eq-list-def prefix-def) blast

    have ‹∃ t u r. r ≤ s ∧ r setinterleaves ((t, u), range tick ∪ ev ‘ A) ∧
              (t ∈ min-elems (𝒟 P) ∧ u ∈ 𝒯 S ∨ t ∈ min-elems (𝒟 S) ∧ u ∈ 𝒯
P)›
    proof (rule ccontr)
      assume assm : ‹∄ t u r1. r1 ≤ s ∧ r1 setinterleaves ((t, u), range tick ∪ ev ‘
A) ∧
                          (t ∈ min-elems (𝒟 P) ∧ u ∈ 𝒯 S ∨ t ∈ min-elems (𝒟 S)
∧ u ∈ 𝒯 P)›

      obtain tm1 tm2 where $: ‹t ∈ 𝒟 P ⟹ tm1 ≤ t ∧ tm1 ∈ min-elems (𝒟
P)›
        ‹t ∈ 𝒟 S ⟹ tm2 ≤ t ∧ tm2 ∈ min-elems (𝒟 S)›
        by (metis min-elems5)
      hence $$ : ‹t ∈ 𝒟 P ⟹ u ∈ 𝒯 S ⟹ tm1 < t› ‹t ∈ 𝒟 S ⟹ u ∈ 𝒯 P ⟹
tm2 < t›
        by (metis ∗(1, 3) assm order-neq-le-trans)+
      then obtain um1 rm1 um2 rm2 where
        ‹t ∈ 𝒟 P ⟹ u ∈ 𝒯 S ⟹ um1 ≤ u ∧ rm1 setinterleaves ((tm1, um1),
range tick ∪ ev ‘ A)›
        ‹t ∈ 𝒟 S ⟹ u ∈ 𝒯 P ⟹ um2 ≤ u ∧ rm2 setinterleaves ((tm2, um2),
range tick ∪ ev ‘ A)›
        by (metis ∗(3) interleave-less-left)
      moreover have ‹t ∈ 𝒟 P ⟹ u ∈ 𝒯 S ⟹ rm1 < s ∧ rm1 ∈ 𝒟 (P ⟦A⟧ S)›
        ‹t ∈ 𝒟 S ⟹ u ∈ 𝒯 P ⟹ rm2 < s ∧ rm2 ∈ 𝒟 (P ⟦A⟧ S)›
        by (meson assm $(1, 2) $$(1, 2) ∗(1) ∗(3)
```

interleave-le-left is-processT3-TR order-trans)+
ultimately show *False* **by** (*metis* ∗(*2*) ‹*s* ∈ *min-elems* (*D* (*P* ⟦*A*⟧ *S*))› *less-list-def min-elems-no*)
qed

— We override.
then obtain *t u r*
where ∗ : ‹*r* ≤ *s*› ‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*)›
‹*t* ∈ *min-elems* (*D P*) ∧ *u* ∈ *T S* ∨ *t* ∈ *min-elems* (*D S*) ∧ *u* ∈ *T P*› **by** *blast*

from ∗(*2*, *3*) **have** ‹*r* ∈ *D* (*P* ⟦*A*⟧ *S*)›
by (*simp add*: *D-Sync*) (*use elem-min-elems front-tickFree-Nil* **in** *blast*)
with ∗(*1*) ‹*s* ∈ *min-elems* (*D* (*P* ⟦*A*⟧ *S*))› *min-elems-no* **have** ‹*s* = *r*› **by** *blast*

have ‹*t* ∈ *D S* ⟹ *u* ∈ *T P* ⟹ *u* ∈ *T P* − (*D P* − *min-elems* (*D P*))›
proof (*rule ccontr*)
assume *assms* : ‹*t* ∈ *D S*› ‹*u* ∈ *T P*› ‹*u* ∉ *T P* − (*D P* − *min-elems* (*D P*))›
from *assms*(*2*, *3*) **have** ‹*u* ∈ *D P*› ‹*u* ∉ *min-elems* (*D P*)› **by** *simp-all*
then obtain *u1* **where** ‹*u1* < *u*› ‹*u1* ∈ *min-elems* (*D P*)›
by (*metis min-elems5 order-neq-le-trans*)

obtain *t1 r1* **where** ‹*t1* ≤ *t*› ‹*r1 setinterleaves* ((*t1*, *u1*), *range tick* ∪ *ev* ‘ *A*)› ‹*r1* < *r*›
by (*metis* (*no-types*, *lifting*) ∗(*2*) ‹*u1* < *u*› *interleave-less-left setinterleaving-sym*)
from ‹*t1* ≤ *t*› ‹*u1* ∈ *min-elems* (*D P*)› *assms*(*1*) **have** ‹*u1* ∈ *D P* ∧ *t1* ∈ *T S*›
using *D-T elem-min-elems is-processT3-TR* **by** *blast*
moreover have ‹*r1* ∈ *D* (*P* ⟦*A*⟧ *S*)›
by (*simp add*: *D-Sync*)
(*use* ‹*r1 setinterleaves* ((*t1*, *u1*), *range tick* ∪ *ev* ‘ *A*)›
‹*u1* ∈ *D P* ∧ *t1* ∈ *T S*› *front-tickFree-Nil setinterleaving-sym* **in** *blast*)
ultimately show *False*
by (*metis* ‹*r1* < *r*› ‹*s* = *r*› ‹*s* ∈ *min-elems* (*D* (*P* ⟦*A*⟧ *S*))› *less-list-def min-elems-no*)
qed

hence ‹∃ *t u*. *s setinterleaves* ((*t*, *u*), *range 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*)))›
using ∗(*2*, *3*) ‹*s* = *r*› *elem-min-elems* **by** *blast*
thus ‹*s* ∈ *T* (*Q* ⟦*A*⟧ *S*)›
by (*simp add*: *T-Sync*)
(*metis* (*no-types*) *append.right-neutral elem-min-elems front-tickFree-Nil le-approx2T*[*OF* ‹*P* ⊑ *Q*›] ‹*P* ⊑ *Q*›[*unfolded le-approx-def*] *in-mono*)
qed
thus ‹*P* ⊑ *P*′ ⟹ *Q* ⊑ *Q*′ ⟹ *P* ⟦*A*⟧ *Q* ⊑ *P*′ ⟦*A*⟧ *Q*′›

by (*metis Sync-commute below-trans*)
qed

lemma *chain-Sync-left* : ‹*chain Y* ⟹ *chain* (λ*i*. *Y i* ⟦*A*⟧ *S*)›
and *chain-Sync-right* : ‹*chain Y* ⟹ *chain* (λ*i*. *S* ⟦*A*⟧ *Y i*)›
by (*simp-all add*: *chain-def mono-Sync*)

lemma *finite-interleaves*: ‹*finite* {(*t*, *u*). (*s* :: ('*a*, '*r*) *trace*$_{ptick}$) *setinterleaves* ((*t*, *u*), *A*)}›
proof (*induction* ‹*length s*› *arbitrary*: *s rule*: *nat-less-induct*)
fix *s* :: ‹('*a*, '*r*) *trace*$_{ptick}$›
assume *hyp* : ‹∀ *m* < *length s*. ∀ *s*. *m* = *length s* ⟶ *finite* {(*t*, *u*). *s setinterleaves* ((*t*, *u*), *A*)}›
show ‹*finite* {(*t*, *u*). *s setinterleaves* ((*t*, *u*), *A*)}›
proof (*cases s*)
have ‹[] *setinterleaves* ((*t*, *u*), *A*) ⟹ *t* = [] ∧ *u* = []› **for** *t u*
using *EmptyLeftSync empty-setinterleaving* **by** *blast*
hence ‹{(*t*, *u*). [] *setinterleaves* ((*t*, *u*), *A*)} = {([], [])}› **by** *auto*
thus ‹*s* = [] ⟹ *finite* {(*t*, *u*). *s setinterleaves* ((*t*, *u*), *A*)}› **by** *simp*
next
fix *x s'*
assume ‹*s* = *x* # *s'*›
define *S* **where** ‹*S* ≡ {(*t*, *u*). *s' setinterleaves* ((*t*, *u*), *A*)}›
with *hyp* ‹*s* = *x* # *s'*› **have** ‹*finite S*› **by** *auto*
have ‹*s setinterleaves* ((*t*, *u*), *A*) ⟹
t ≠ [] ∧ *hd t* = *x* ∧ *s' setinterleaves* ((*tl t*, *u*), *A*) ∨
u ≠ [] ∧ *hd u* = *x* ∧ *s' setinterleaves* ((*t*, *tl u*), *A*) ∨
t ≠ [] ∧ *hd t* = *x* ∧ *u* ≠ [] ∧ *hd u* = *x* ∧
s' setinterleaves ((*tl t*, *tl u*), *A*)› **for** *t u*
by (*cases t*; *cases u*; *simp add*: ‹*s* = *x* # *s'*› *split*: *if-split-asm*) *blast*
hence ∗ : ‹{(*t*, *u*). *s setinterleaves* ((*t*, *u*), *A*)} ⊆
{(*t*, *u*). *t* ≠ [] ∧ *hd t* = *x* ∧ *s' setinterleaves* ((*tl t*, *u*), *A*)} ∪
{(*t*, *u*). *u* ≠ [] ∧ *hd u* = *x* ∧ *s' setinterleaves* ((*t*, *tl u*), *A*)} ∪
{(*t*, *u*). *t* ≠ [] ∧ *hd t* = *x* ∧ *u* ≠ [] ∧ *hd u* = *x* ∧
s' setinterleaves ((*tl t*, *tl u*), *A*)}›
(**is** ‹{(*t*, *u*). *s setinterleaves* ((*t*, *u*), *A*)} ⊆ *?S1* ∪ *?S2* ∪ *?S3*›) **by** *blast*

have ‹*?S1* = {(*x* # *t*, *u*)| *t u*. *s' setinterleaves* ((*t*, *u*), *A*)}› **by** *force*
also have ‹... = (λ(*t*, *u*). (*x* # *t*, *u*)) ‘ *S*› **unfolding** *S-def* **by** *auto*
finally have ‹*finite ?S1*› **by** (*simp add*: ‹*finite S*›)

have ‹*?S2* = {(*t*, *x* # *u*)| *t u*. *s' setinterleaves* ((*t*, *u*), *A*)}› **by** *force*
also have ‹... = (λ(*t*, *u*). (*t*, *x* # *u*)) ‘ *S*› **unfolding** *S-def* **by** *auto*
finally have ‹*finite ?S2*› **by** (*simp add*: ‹*finite S*›)

```
    have ‹?S3 = {(x # t, x # u)| t u. s' setinterleaves ((t, u), A)}›
      by (simp add: set-eq-iff) (metis list.collapse list.distinct(1) list.sel(1, 3))
    also have ‹... = (λ(t, u). (x # t, x # u)) ‘ S› unfolding S-def by auto
    finally have ‹finite ?S3› by (simp add: ‹finite S›)

    from ∗ ‹finite ?S1› ‹finite ?S2› ‹finite ?S3› finite-subset
    show ‹finite {(t, u). s setinterleaves ((t, u), A)}› by blast
  qed
qed


lemma finite-interleaves-Sync:
  ‹finite {(t, u, r). r setinterleaves ((t, u), range tick ∪ ev ‘ A) ∧
                    (∃ v. s = r @ v ∧ front-tickFree v ∧ (tickFree r ∨ v = []))}›
  (is ‹finite ?A›)
proof −
  have ‹?A ⊆ (⋃r∈{r. r ≤ s}. {(t, u, r) |t u. r setinterleaves ((t, u), range tick
∪ ev ‘ A)})›
    (is ‹?A ⊆ (⋃ r ∈ {r. r ≤ s}. ?P r)›)
    unfolding less-eq-list-def prefix-def by blast

  have ‹?P r ⊆ (λ(t, u). (t, u, r)) ‘
              {(t, u). r setinterleaves ((t, u), range tick ∪ ev ‘ A)}› for r by auto
  hence ‹finite (?P r)› for r by (rule finite-subset) (simp add: finite-interleaves)

  have ‹{r. ∃ v. s = r @ v} = {r. ∃ v. r @ v = s}› by blast
  hence ‹finite {r. r ≤ s}›
    by (fold prefix-def less-eq-list-def)
      (use prefixes-fin[of s, simplified Let-def] in presburger)

  show ?thesis
    by (rule finite-subset[OF ‹?A ⊆ (⋃ r ∈ {r. r ≤ s}. ?P r)›])
      (rule finite-UN-I[OF ‹finite {r. r ≤ s}› ‹⋀r. finite (?P r)›])
qed


lemma Cont-Sync-prem:
  ‹(⨆ i. Y i) ⟦A⟧ Q = (⨆ i. Y i ⟦A⟧ Q)› if chain: ‹chain Y›
proof (subst Process-eq-spec-optimized, safe)
  show ‹s ∈ 𝒟 ((⨆ i. Y i) ⟦A⟧ Q) ⟹ s ∈ 𝒟 (⨆i. Y i ⟦A⟧ Q)› for s
    by (simp add: limproc-is-thelub chain chain-Sync-left D-Sync D-LUB T-LUB)
blast
next
  show ‹(s, X) ∈ ℱ ((⨆ i. Y i) ⟦A⟧ Q) ⟹ (s, X) ∈ ℱ (⨆i. Y i ⟦A⟧ Q)› for s X
    by (simp add: limproc-is-thelub chain chain-Sync-left F-Sync D-LUB T-LUB
F-LUB) blast
next
```

fix s
assume ‹$s \in \mathcal{D}\ (\bigsqcup i.\ Y\ i\ [\![A]\!]\ Q)$›
define S
where ‹$S\ i \equiv \{(t,\ u,\ r).\ \exists\, v.\ \mathit{front\text{-}tickFree}\ v \wedge (\mathit{tickFree}\ r \vee v = []) \wedge s = r\ @\ v \wedge$
$r\ \mathit{setinterleaves}\ ((t,\ u),\ \mathit{range}\ \mathit{tick} \cup \mathit{ev}\ `\ A) \wedge$
$(t \in \mathcal{D}\ (Y\ i) \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ (Y\ i))\}$›
for i
have ‹$(\bigcap i.\ S\ i) \neq \{\}$›
apply (*rule Inter-nonempty-finite-chained-sets*, *unfold S-def*)
apply (*use* ‹$s \in \mathcal{D}\ (\bigsqcup i.\ Y\ i\ [\![A]\!]\ Q)$› **in**
‹*simp add*: *limproc-is-thelub chain chain-Sync-left D-Sync T-LUB D-LUB*, *blast*›)
apply (*rule finite-subset*[*OF - finite-interleaves-Sync*]; *auto*)
using *D-T le-approx1*[*OF po-class.chainE*[*OF chain*]]
le-approx2T[*OF po-class.chainE*[*OF chain*]] **by** *blast*
then obtain $t\ u\ r$ **where** ‹$(t,\ u,\ r) \in (\bigcap i.\ S\ i)$› **by** *auto*
hence ‹$\mathit{front\text{-}tickFree}\ (\mathit{drop}\ (\mathit{length}\ r)\ s) \wedge (\mathit{tickFree}\ r \vee \mathit{drop}\ (\mathit{length}\ r)\ s = [])$
$\wedge$
$s = r\ @\ \mathit{drop}\ (\mathit{length}\ r)\ s \wedge r\ \mathit{setinterleaves}\ ((t,\ u),\ \mathit{range}\ \mathit{tick} \cup \mathit{ev}\ `\ A)$
$\wedge$
$((\forall i.\ t \in \mathcal{D}\ (Y\ i)) \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge (\forall i.\ u \in \mathcal{T}\ (Y\ i)))$›
by (*auto simp add*: *S-def*) (*meson chain-lemma le-approx1 le-approx-lemma-T subsetD chain*)

show ‹$s \in \mathcal{D}\ ((\bigsqcup\ i.\ Y\ i)\ [\![A]\!]\ Q)$›
by (*simp add*: *limproc-is-thelub chain chain-Sync-left D-Sync T-LUB D-LUB*)
(*use* ‹*?this*› **in** *blast*)
next
assume *same-div* : ‹$\mathcal{D}\ ((\bigsqcup\ i.\ Y\ i)\ [\![A]\!]\ Q) = \mathcal{D}\ (\bigsqcup\ i.\ Y\ i\ [\![A]\!]\ Q)$›
fix $s\ X$ **assume** ‹$(s,\ X) \in \mathcal{F}\ (\bigsqcup i.\ Y\ i\ [\![A]\!]\ Q)$›
show ‹$(s,\ X) \in \mathcal{F}\ ((\bigsqcup\ i.\ Y\ i)\ [\![A]\!]\ Q)$›
proof (*cases* ‹$s \in \mathcal{D}\ (\bigsqcup\ i.\ Y\ i\ [\![A]\!]\ Q)$›)
show ‹$s \in \mathcal{D}\ (\bigsqcup i.\ Y\ i\ [\![A]\!]\ Q) \Longrightarrow (s,\ X) \in \mathcal{F}\ ((\bigsqcup i.\ Y\ i)\ [\![A]\!]\ Q)$›
by (*simp add*: *is-processT8 same-div*)
next
assume ‹$s \notin \mathcal{D}\ (\bigsqcup i.\ Y\ i\ [\![A]\!]\ Q)$›
then obtain j **where** ‹$s \notin \mathcal{D}\ (Y\ j\ [\![A]\!]\ Q)$›
by (*auto simp add*: *limproc-is-thelub chain-Sync-left* ‹*chain Y*› *D-LUB*)
moreover from ‹$(s,\ X) \in \mathcal{F}\ (\bigsqcup i.\ Y\ i\ [\![A]\!]\ Q)$› **have** ‹$(s,\ X) \in \mathcal{F}\ (Y\ j\ [\![A]\!]\ Q)$›
by (*simp add*: *limproc-is-thelub chain-Sync-left* ‹*chain Y*› *F-LUB*)
ultimately show ‹$(s,\ X) \in \mathcal{F}\ ((\bigsqcup\ i.\ Y\ i)\ [\![A]\!]\ Q)$›
by (*fact le-approx2*[*OF mono-Sync*[*OF is-ub-thelub*[*OF* ‹*chain Y*›] *below-refl*], *THEN iffD2*])
qed
qed

lemma *Sync-cont*[*simp*]: ‹*cont* (λ*x*. *f x* ⟦*A*⟧ *g x*)› **if** ‹*cont f*› ‹*cont g*›
proof (*rule cont-apply*[**where** *f* = ‹λ*x y*. *y* ⟦*A*⟧ *g x*›])
from ‹*cont f*› **show** ‹*cont f*› **.**
next
show ‹*cont* (λ*y*. *y* ⟦*A*⟧ *g x*)› **for** *x*
proof (*rule contI2*)
show ‹*monofun* (λ*y*. *y* ⟦*A*⟧ *g x*)› **by** (*simp add*: *monofunI mono-Sync*)
next
show ‹*chain Y* ⟹ (⨆ *i*. *Y i*) ⟦*A*⟧ *g x* ⊑ (⨆ *i*. *Y i* ⟦*A*⟧ *g x*)› **for** *Y*
by (*simp add*: *Cont-Sync-prem*)
qed
next
show ‹*cont* (λ*x*. *y* ⟦*A*⟧ *g x*)› **for** *y*
proof (*rule cont-compose*[*of* ‹λ*x*. *y* ⟦*A*⟧ *x*›])
show ‹*cont* (*Sync y A*)›
proof (*rule contI2*)
show ‹*monofun* (*Sync y A*)› **by** (*simp add*: *monofunI mono-Sync*)
next
show ‹*chain Y* ⟹ *y* ⟦*A*⟧ (⨆ *i*. *Y i*) ⊑ (⨆ *i*. *y* ⟦*A*⟧ *Y i*)› **for** *Y*
by (*subst* (*1 2*) *Sync-commute*) (*simp add*: *Cont-Sync-prem*)
qed
next
from ‹*cont g*› **show** ‹*cont g*› **.**
qed
qed

Chapter 9

The Renaming Operator

9.1 Some preliminaries

term ‹f $-`$ B›

definition *finitary* :: ‹($'a \Rightarrow 'b$) $\Rightarrow$ *bool*›
  where ‹*finitary* $f \equiv \forall x.$ *finite*(f $-`$ $\{x\}$)›

We start with some simple results.

lemma ‹f $-`$ $\{\} = \{\}$› **by** *simp*

lemma ‹$X \subseteq Y \Longrightarrow f$ $-`$ $X \subseteq f$ $-`$ Y› **by** (*rule vimage-mono*)

lemma ‹f $-`(X \cup Y) = f$ $-`$ $X \cup f$ $-`$ Y› **by** (*rule vimage-Un*)

lemma *map-event*$_{ptick}$*-eq-ev-iff* : ‹*map-event*$_{ptick}$ f g $e = ev$ $b \longleftrightarrow (\exists a.\ e = ev$ $a \ \wedge\ b = f$ $a)$›
  and *map-event*$_{ptick}$*-eq-tick-iff* : ‹*map-event*$_{ptick}$ f g $e = \checkmark(s) \longleftrightarrow (\exists r.\ e = \checkmark(r)$ $\wedge\ s = g$ $r)$›
  and *ev-eq-map-event*$_{ptick}$*-iff* : ‹ev $b =$ *map-event*$_{ptick}$ f g $e \longleftrightarrow (\exists a.\ e = ev$ a $\wedge\ b = f$ $a)$›
  and *tick-eq-map-event*$_{ptick}$*-iff* : ‹$\checkmark(s) =$ *map-event*$_{ptick}$ f g $e \longleftrightarrow (\exists r.\ e = \checkmark(r)$ $\wedge\ s = g$ $r)$›
  by (*cases* e; *auto*)+

lemma *inj-left-map-event*$_{ptick}$: ‹*inj* ($\lambda f.$ *map-event*$_{ptick}$ f g)›
proof (*intro injI ext*)
  show ‹*map-event*$_{ptick}$ *f1* $g =$ *map-event*$_{ptick}$ *f2* $g \Longrightarrow$ *f1* $a =$ *f2* a› **for** *f1 f2* :: ‹$'a \Rightarrow 'b$› **and** a :: $'a$
    by (*drule fun-cong*[**where** $x =$ ‹ev a›]) *simp*
qed

lemma *inj-right-map-event*$_{ptick}$: ‹*inj* (*map-event*$_{ptick}$ f)›

proof (*intro injI ext*)
show ‹*map-event*$_{ptick}$ *f g1* = *map-event*$_{ptick}$ *f g2* ⟹ *g1 r* = *g2 r*› **for** *g1 g2* :: ‹*'r* ⇒ *'s*› **and** *r* :: *'r*
by (*drule fun-cong*[**where** *x* = ‹✓(*r*)›]) *simp*
qed

lemma *map-event*$_{ptick}$*-tickFree* : ‹*tickFree* (*map* (*map-event*$_{ptick}$ *f g*) *s*) ⟷ *tickFree s*›
by (*induct s*) (*simp-all add*: *event*$_{ptick}$*.case-eq-if*)

lemma *map-event*$_{ptick}$*-front-tickFree* : ‹*front-tickFree* (*map* (*map-event*$_{ptick}$ *f g*) *s*) ⟷ *front-tickFree s*›
by (*simp add*: *front-tickFree-iff-tickFree-butlast map-butlast*[*symmetric*] *map-event*$_{ptick}$*-tickFree*)

lemma *map-map-event*$_{ptick}$*-eq-tick-iff* :
‹*map* (*map-event*$_{ptick}$ *f g*) *t* = [✓(*s*)] ⟷ (∃ *r*. *s* = *g r* ∧ *t* = [✓(*r*)])›
and *tick-eq-map-map-event*$_{ptick}$*-iff* :
‹[✓(*s*)] = *map* (*map-event*$_{ptick}$ *f g*) *t* ⟷ (∃ *r*. *s* = *g r* ∧ *t* = [✓(*r*)])›
by (*auto simp add*: *map-event*$_{ptick}$*-eq-tick-iff tick-eq-map-event*$_{ptick}$*-iff*)

9.2 The Renaming Operator Definition

Our new renaming operator does not only deal with events but also with termination.

lift-definition *Renaming* :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a* ⇒ '*b*, '*r* ⇒ '*s*] ⇒ ('*b*, '*s*) *process*$_{ptick}$›
is ‹λ*P f g*. ({(*map* (*map-event*$_{ptick}$ *f g*) *s1*, *X*)| *s1 X*. (*s1*, (*map-event*$_{ptick}$ *f g*) −‘ *X*) ∈ ℱ *P*} ∪
{((*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*, *X*)| *s1 s2 X*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ 𝒟 *P*},
{(*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*| *s1 s2*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ 𝒟 *P*})›
proof −
show ‹*?thesis P f g*› (**is** ‹*is-process*(*?f*, *?d*)›) **for** *P f g*
proof (*unfold is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv*, *intro conjI allI impI*)
show ‹([], {}) ∈ *?f*›
by (*simp add*: *process-charn*)
next
show ‹(*s*, *X*) ∈ *?f* ⟹ *front-tickFree s*› **for** *s X*
by (*auto simp add*: *F-imp-front-tickFree map-event*$_{ptick}$*-front-tickFree map-event*$_{ptick}$*-tickFree front-tickFree-append*)
next
show ‹(*s* @ *t*, {}) ∈ *?f* ⟹ (*s*, {}) ∈ *?f*› **for** *s t*
proof (*induct t rule*: *rev-induct*)
case *Nil* **thus** ‹(*s*, {}) ∈ *?f*› **by** *simp*

next
case (*snoc e t*)
from *snoc.prems is-processT8*
consider (*fail*) *s1* **where** ‹*s* @ *t* @ [*e*] = *map* (*map-event*$_{ptick}$ *f g*) *s1*› ‹(*s1*, {}) ∈ $\mathcal{F}$ *P*›
| (*div*) *s1 s2* **where** ‹*s* @ *t* @ [*e*] = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2*› ‹*s2* ≠ []›
‹*tickFree s1*› ‹*front-tickFree s2*› ‹*s1* ∈ $\mathcal{D}$ *P*› **by** *auto blast*
thus ‹(*s*, {}) ∈ *?f*›
proof *cases*
case *fail*
from *fail*(*1*) **obtain** *s1′ e′* **where** ‹*s1* = *s1′* @ [*e′*]› ‹*s* @ *t* = *map* (*map-event*$_{ptick}$ *f g*) *s1′*›
by (*metis Nil-is-append-conv butlast-append list.map-disc-iff map-butlast snoc-eq-iff-butlast*)
from *this*(*1*) *fail*(*2*) *is-processT3* **have** ‹(*s1′*, {}) ∈ $\mathcal{F}$ *P*› **by** *blast*
with ‹*s* @ *t* = *map* (*map-event*$_{ptick}$ *f g*) *s1′*› **have** ‹(*s* @ *t*, {}) ∈ *?f*› **by** *auto*
with *snoc.hyps* **show** ‹(*s*, {}) ∈ *?f*› **by** *presburger*
next
case *div*
from *div*(*2*) **obtain** *s2′ e′* **where** ‹*s2* = *s2′* @ [*e′*]› **by** (*meson rev-exhaust*)
with *div*(*1*, *3*, *4*, *5*) ‹*s2* = *s2′* @ [*e′*]› *front-tickFree-dw-closed*
have ‹(*s* @ *t*, {}) ∈ *?f*› **by** *simp blast*
with *snoc.hyps* **show** ‹(*s*, {}) ∈ *?f*› **by** *presburger*
qed
qed
next
show ‹(*s*, *Y*) ∈ *?f* ∧ *X* ⊆ *Y* ⟹ (*s*, *X*) ∈ *?f*› **for** *s X Y*
by (*induct s rule*: *rev-induct*, *simp-all*)
(*meson is-processT4 vimage-mono*)+
next
show ‹(*s*, *X*) ∈ *?f* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *?f*) ⟹ (*s*, *X* ∪ *Y*) ∈ *?f*› **for** *s X Y*
by *auto* (*metis* (*no-types*, *lifting*) *is-processT5 list.simps*(*8*, *9*) *map-append vimageE*)
next
fix *s r X* **assume** ‹(*s* @ [✓(*r*)], {}) ∈ *?f*›
then consider (*fail*) *s1* **where** ‹*s* @ [✓(*r*)] = *map* (*map-event*$_{ptick}$ *f g*) *s1*› ‹(*s1*, {}) ∈ $\mathcal{F}$ *P*›
| (*div*) *s1 s2* **where** ‹*s* @ [✓(*r*)] = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2*› ‹*s2* ≠ []›
‹*tickFree s1*› ‹*front-tickFree s2*› ‹*s1* ∈ $\mathcal{D}$ *P*›
using *is-processT8* **by** *auto blast*
thus ‹(*s*, *X* − {✓(*r*)}) ∈ *?f*›
proof *cases*
case *fail*
from *fail*(*1*) **obtain** *s1′ r′* **where** ∗ : ‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1′*› ‹*s1* = *s1′* @ [✓(*r′*)]› ‹*r* = *g r′*›

by (*simp add*: *append-eq-map-conv*) (*metis* (*mono-tags*, *opaque-lifting*) *map-map-event*$_{ptick}$*-eq-tick-iff*)
from *this*(*2*) *fail*(*2*) **have** ‹(*s1'*, *map-event*$_{ptick}$ *f g* −‘ *X* − {✓(*r'*)}) ∈ ℱ *P*›

by (*simp add*: *is-processT6*)
hence ‹(*s1'*, *map-event*$_{ptick}$ *f g* −‘ (*X* − {✓(*r*)})) ∈ ℱ *P*›
by (*rule is-processT4*) (*auto simp add*: ‹*r* = *g r'*›)
thus ‹(*s*, *X* − {✓(*r*)}) ∈ *?f*› **by** (*unfold* ∗(*1*)) *blast*
next
case *div*
from *div*(*2*, *4*) *front-tickFree-charn tickFree-imp-front-tickFree*
obtain *s2'* *e'* **where** ‹*s2* = *s2'* @ [*e'*]› ‹*front-tickFree s2'*› **by** *blast*
from ‹*s2* = *s2'* @ [*e'*]› *div*(*1*) **have** ‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2'*›
by *simp*
with *div*(*3*, *5*) ‹*front-tickFree s2'*› **show** ‹(*s*, *X* − {✓(*r*)}) ∈ *?f*› **by** *blast*
qed
next
show ‹*s* ∈ *?d* ∧ *tickFree s* ∧ *front-tickFree t* ⟹ *s* @ *t* ∈ *?d*› **for** *s t*
using *front-tickFree-append* **by** *safe auto*
next
show ‹*s* ∈ *?d* ⟹ (*s*, *X*) ∈ *?f*› **for** *s X* **by** *blast*

next
fix *s r* **assume** ‹*s* @ [✓(*r*)] ∈ *?d*›
then obtain *s1 s2* **where** ∗ : ‹*s* @ [✓(*r*)] = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2*›
‹*tickFree s1*› ‹*front-tickFree s2*› ‹*s1* ∈ 𝒟 *P*› **by** *blast*
from ∗(*1*, *2*, *3*) **obtain** *s2'* *e* **where** ‹*s2* = *s2'* @ [*e*]› ‹*front-tickFree s2'*›
by (*metis append-self-conv front-tickFree-charn front-tickFree-dw-closed*
non-tickFree-tick map-event$_{ptick}$*-tickFree tickFree-append-iff*)
from *this*(*1*) ∗(*1*) **have** ‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2'*› **by** *simp*
with ∗(*2*, *4*) ‹*front-tickFree s2'*› **show** ‹*s* ∈ *?d*› **by** *blast*
qed
qed

Some syntactic sugar.

syntax
-Renaming :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ *updbinds* ⇒ *updbinds* ⇒ ('*a*, '*r*) *process*$_{ptick}$› (‹- ⟦-⟧ ⟦-⟧› [*100*, *100*, *100*])
-Renaming-left :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ *updbinds* ⇒ ('*a*, '*r*) *process*$_{ptick}$› (‹- ⟦-⟧ [][]› [*100*, *100*])
-Renaming-right :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ *updbinds* ⇒ ('*a*, '*r*) *process*$_{ptick}$› (‹- [][] ⟦-⟧› [*100*, *100*])
syntax-consts *-Renaming* ⇌ *Renaming*
and *-Renaming-left* ⇌ *Renaming*
and *-Renaming-right* ⇌ *Renaming*
translations
-Renaming P f-updates g-updates ⇌ *CONST Renaming P* (*-Update* (*CONST id*) *f-updates*) (*-Update* (*CONST id*) *g-updates*)

-Renaming-left P f-updates ⇌ *CONST Renaming P* (*-Update* (*CONST id*) *f-updates*) (*CONST id*)
-Renaming-right P g-updates ⇌ *CONST Renaming P* (*CONST id*) (*-Update* (*CONST id*) *g-updates*)

Now we can write $P\ [\![a := b]\!]\ [\![c := d]\!]$. If we only want to rename events, or results, we use respectively $P\ [\![a := b]\!]\ [\![]\!]$ and $P\ [\![]\!]\ [\![c := d]\!]$. Like in *HOL.Fun*, we can write this kind of expression with as many updates we want: $P\ [\![a := b,\ c := d,\ e := f,\ g := h]\!]\ [\![r1 := r2,\ r3 := r4]\!]$. By construction we also inherit all the results about *fun-upd*, for example $?z \neq ?x \Longrightarrow (?f(?x := ?y))\ ?z = ?f\ ?z$

$?f(?x := ?y,\ ?x := ?z) = ?f(?x := ?z)$

$?a \neq ?c \Longrightarrow ?m(?a := ?b,\ ?c := ?d) = ?m(?c := ?d,\ ?a := ?b)$.

lemma ‹$P\ [\![x := y,\ x := z]\!]\ [\![]\!] = P[\![x := z]\!]\ [\![]\!]$› **by** *simp*

lemma ‹$a \neq c \Longrightarrow P\ [\![a := b,\ c := d]\!]\ [\![r1 := r2]\!] = P\ [\![c := d,\ a := b]\!]\ [\![r1 := r2]\!]$› **by** (*simp add*: *fun-upd-twist*)

9.3 The Projections

lemma *F-Renaming*: ‹$\mathcal{F}$ (*Renaming P f g*) =
{(*map* (*map-event*$_{ptick}$ *f g*) *s1*, *X*)| *s1 X*. (*s1*, (*map-event*$_{ptick}$ *f g*) −‘ *X*) ∈ $\mathcal{F}$ *P*} ∪
{((*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*, *X*)| *s1 s2 X*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ $\mathcal{D}$ *P*}›
by (*simp add*: *Failures.rep-eq FAILURES-def Renaming.rep-eq*)

lemma *D-Renaming*: ‹$\mathcal{D}$ (*Renaming P f g*) =
{(*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*| *s1 s2*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ $\mathcal{D}$ *P*}›
by (*simp add*: *Divergences.rep-eq DIVERGENCES-def Renaming.rep-eq*)

lemma *T-Renaming*: ‹$\mathcal{T}$ (*Renaming P f g*) =
{ *map* (*map-event*$_{ptick}$ *f g*) *s1*| *s1*. *s1* ∈ $\mathcal{T}$ *P*} ∪
{(*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*| *s1 s2*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ $\mathcal{D}$ *P*}›
by (*subst set-eq-iff*, *fold T-F-spec*, *simp add*: *F-Renaming*)

lemmas *Renaming-projs* = *F-Renaming D-Renaming T-Renaming*

9.4 Continuity Rule

9.4.1 Monotonicity of *Renaming*.

lemma *mono-Renaming* : ‹*Renaming P f g* ⊑ *Renaming Q f g*› **if** ‹*P* ⊑ *Q*›
proof (*subst le-approx-def*, *intro conjI subset-antisym allI impI subsetI*)
show ‹*s* ∈ $\mathcal{D}$ (*Renaming Q f g*) ⟹ *s* ∈ $\mathcal{D}$ (*Renaming P f g*)› **for** *s*

unfolding *D-Renaming* **using** *le-approx1* ‹$P \sqsubseteq Q$› **by** *blast*
next
show ‹$s \notin \mathcal{D}\ (Renaming\ P\ f\ g) \Longrightarrow X \in \mathcal{R}_a\ (Renaming\ P\ f\ g)\ s \Longrightarrow X \in \mathcal{R}_a\ (Renaming\ Q\ f\ g)\ s$› **for** *s X*
by (*simp add*: *D-Renaming Refusals-after-def F-Renaming*)
(*metis* (*no-types*, *opaque-lifting*) *F-imp-front-tickFree append.right-neutral front-tickFree-Nil*
front-tickFree-append-iff front-tickFree-single is-processT9 map-append tick-Free-Nil
map-map-event$_{ptick}$-eq-tick-iff nonTickFree-n-frontTickFree non-tickFree-tick proc-ord2a ‹$P \sqsubseteq Q$›)
next
show ‹$s \notin \mathcal{D}\ (Renaming\ P\ f\ g) \Longrightarrow X \in \mathcal{R}_a\ (Renaming\ Q\ f\ g)\ s \Longrightarrow X \in \mathcal{R}_a\ (Renaming\ P\ f\ g)\ s$› **for** *s X*
by (*simp add*: *D-Renaming Refusals-after-def F-Renaming*)
(*metis* (*no-types*, *lifting*) *is-processT8 le-approx1 proc-ord2a subset-eq* ‹$P \sqsubseteq Q$›)
next
show ‹$s \in min\text{-}elems\ (\mathcal{D}\ (Renaming\ P\ f\ g)) \Longrightarrow s \in \mathcal{T}\ (Renaming\ Q\ f\ g)$› **for** *s*
apply (*rule set-mp*[*of* ‹$\{map\ (map\text{-}event_{ptick}\ f\ g)\ s1\ |s1.\ s1 \in min\text{-}elems\ (\mathcal{D}\ P)\}$›])
apply (*simp add*: *T-Renaming*, *use le-approx3* ‹$P \sqsubseteq Q$› **in** *blast*)
apply (*drule set-rev-mp*[*of* - - ‹$min\text{-}elems\ \{map\ (map\text{-}event_{ptick}\ f\ g)\ s1\ |s1.\ tickFree\ s1 \wedge s1 \in \mathcal{D}\ P\}$›])
apply (*simp-all add*: *D-Renaming min-elems-def subset-iff*)
by (*metis prefixI antisym-conv2 append.right-neutral front-tickFree-Nil*)
(*metis* (*no-types*, *lifting*) *antisym-conv2 append-butlast-last-id*
front-tickFree-iff-tickFree-butlast less-self list.map-disc-iff map-butlast
min-elems1 min-elems-no nil-less order-less-imp-le tickFree-imp-front-tickFree)
qed

lemma ‹$\{ev\ c\ |c.\ f\ c = a\} = ev`\{c\ .\ f\ c = a\}$› **by** (*rule setcompr-eq-image*)

lemma *vimage-map-event$_{ptick}$-subset-vimage*: ‹$map\text{-}event_{ptick}\ f\ g\ -`\ (ev\ `\ A) \subseteq range\ tick \cup (ev\ `\ (f\ -`\ A))$›
and *vimage-subset-vimage-map-event$_{ptick}$*: ‹$(ev\ `\ (f\ -`\ A)) \subseteq (map\text{-}event_{ptick}\ f\ g\ -`\ (ev\ `\ A)) - range\ tick$›
by (*auto simp add*: *image-def vimage-def*)
(*metis event$_{ptick}$.exhaust-sel event$_{ptick}$.sel*(1) *event$_{ptick}$.simps*(9))

9.4.2 Some useful results about *finitary*, and preliminaries lemmas for continuity.

lemma *inj-imp-finitary*[*simp*] : ‹$inj\ f \Longrightarrow finitary\ f$› **by** (*simp add*: *finitary-def finite-vimageI*)

lemma *finitary-comp-iff* : ‹$finitary\ g \Longrightarrow finitary\ (g\ o\ f) \longleftrightarrow finitary\ f$›

proof (*unfold finitary-def*, *intro iffI allI*;
(*subst vimage-comp*[*symmetric*] | *subst* (*asm*) *vimage-comp*[*symmetric*]))
have *f1*: ‹$f -` g -` \{x\} = (\bigcup y \in g -` \{x\}.\ f -` \{y\})$› **for** x **by** *blast*
show ‹*finite* $(f -` g -` \{x\})$› **if** ‹$\forall x.$ *finite* $(f -` \{x\})$› **and** ‹$\forall x.$ *finite* $(g -` \{x\})$› **for** x
apply (*subst f1*, *subst finite-UN*)
by (*rule that*(*2*)[*unfolded finitary-def*, *rule-format*])
(*intro ballI that*(*1*)[*rule-format*])
qed (*metis finite-Un insert-absorb vimage-insert vimage-singleton-eq*)

lemma *finitary-updated-iff*[*simp*] : ‹*finitary* $(f\ (a := b)) \longleftrightarrow$ *finitary* f›
proof (*unfold fun-upd-def finitary-def vimage-def*, *intro iffI allI*)
show ‹*finite* $\{x.\ (\textit{if } x = a \textit{ then } b \textit{ else } f\ x) \in \{y\}\}$› **if** ‹$\forall y.$ *finite* $\{x.\ f\ x \in \{y\}\}$› **for** y
apply (*rule finite-subset*[*of* - ‹$\{x.\ x = a\} \cup \{x.\ f\ x \in \{y\}\}$›], (*auto*)[*1*])
apply (*rule finite-UnI*)
by *simp* (*fact that*[*rule-format*])
next
show ‹*finite* $\{x.\ f\ x \in \{y\}\}$› **if** ‹$\forall y.$ *finite* $\{x.\ (\textit{if } x = a \textit{ then } b \textit{ else } f\ x) \in \{y\}\}$› **for** y
apply (*rule finite-subset*[*of* - ‹$\{x.\ x = a \wedge f\ x \in \{y\}\} \cup \{x.\ x \neq a \wedge f\ x \in \{y\}\}$›], (*auto*)[*1*])
apply (*rule finite-UnI*)
using *that*[*rule-format*, *of y*] **by** (*simp-all add*: *Collect-mono rev-finite-subset*)
qed

By declaring this simp, we automatically obtain this kind of result.

lemma ‹*finitary* $f \longleftrightarrow$ *finitary* $(f\ (a := b,\ c := d,\ e := g,\ h := i))$› **by** *simp*

lemma *Cont-RenH2*:
‹*finitary* $((\textit{map-event}_{ptick}\ f\ g) :: ('a,\ 'r)\ \textit{event}_{ptick} \Rightarrow ('b,\ 's)\ \textit{event}_{ptick}) \longleftrightarrow$ *finitary* $f \wedge$ *finitary* g›
proof (*intro iffI conjI*)
show ‹*finitary* f› **if** ‹*finitary* $(\textit{map-event}_{ptick}\ f\ g)$›
proof (*unfold finitary-def*, *rule allI*)
fix $b :: 'b$
have ‹*finite* $((\textit{map-event}_{ptick}\ f\ g) -` \{\textit{ev}\ b\})$›
by (*fact* ‹*finitary* $(\textit{map-event}_{ptick}\ f\ g)$›[*unfolded finitary-def*, *rule-format*])
also have ‹$\textit{map-event}_{ptick}\ f\ g -` \{\textit{ev}\ b\} = \textit{ev}\ `\ (f -` \{b\})$›
by (*unfold vimage-def image-def*) (*auto simp add*: $\textit{map-event}_{ptick}$-*eq-ev-iff*)
finally show ‹*finite* $(f -` \{b\})$› **by** (*simp add*: *inj-on-def finite-image-iff*)
qed
next
show ‹*finitary* g› **if** ‹*finitary* $(\textit{map-event}_{ptick}\ f\ g)$›
proof (*unfold finitary-def*, *rule allI*)
fix $s :: 's$

```
    have ‹finite ((map-event_ptick f g) -‘ {tick s})›
      by (fact ‹finitary (map-event_ptick f g)›[unfolded finitary-def, rule-format])
    also have ‹map-event_ptick f g -‘ {tick s} = tick ‘ (g -‘ {s})›
      by (unfold vimage-def image-def) (auto simp add: map-event_ptick-eq-tick-iff)
    finally show ‹finite (g -‘ {s})› by (simp add: inj-on-def finite-image-iff)
  qed
next
  show ‹finitary (map-event_ptick f g)› if ‹finitary f ∧ finitary g›
  proof (unfold finitary-def, rule allI)
    fix e :: ‹('b, 's) event_ptick›
    show ‹finite (map-event_ptick f g -‘ {e})›
    proof (cases e)
      fix b assume ‹e = ev b›
      have ‹map-event_ptick f g -‘ {ev b} = ev ‘ (f -‘ {b})›
        by (unfold vimage-def image-def) (auto simp add: map-event_ptick-eq-ev-iff)
      thus ‹finite (map-event_ptick f g -‘ {e})›
        by (simp add: ‹e = ev b›) (meson finitary-def finite-imageI that)
    next
      fix s assume ‹e = tick s›
      have ‹map-event_ptick f g -‘ {tick s} = tick ‘ (g -‘ {s})›
       by (unfold vimage-def image-def) (auto simp add: map-event_ptick-eq-tick-iff)
      thus ‹finite (map-event_ptick f g -‘ {e})›
        by (simp add: ‹e = tick s›) (meson finitary-def finite-imageI that)
    qed
  qed
qed



lemma Cont-RenH3:
  ‹{s1 @ t1 |s1 t1. (∃ b. s1 = [b] & f b = a) ∧ list = map f t1} =
   (⋃ b ∈ f -‘{a}. (λt. b # t) ‘ {t. list = map f t})›
  by auto (metis append-Cons append-Nil)


lemma Cont-RenH4: ‹finitary f ⟷ (∀ s. finite {t. s = map f t})›
proof (unfold finitary-def, intro iffI allI)
  show ‹finite {t. s = map f t}› if ‹∀ x. finite (f -‘ {x})› for s
  proof (induct s)
    show ‹finite {t. [] = map f t}› by simp
  next
    case (Cons a s)
    have ‹{t. a # s = map f t} = (⋃ b ∈ f -‘ {a}. {b # t |t. s = map f t})›
      by (subst Cons-eq-map-conv) blast
    thus ?case by (simp add: finite-UN[OF that[rule-format]] local.Cons)
  qed
next
  have map1: ‹[a] = map f s = (∃ b. s = [b] ∧ f b = a)› for a s by fastforce
  show ‹finite (f -‘ {x}) › if ‹∀ s. finite {t. s = map f t}› for x
```

by (*simp add*: *vimage-def*)
(*fact finite-vimageI*[*OF that*[*rule-format*, *of* ‹[*x*]›, *simplified map1*],
of ‹λ*x*. [*x*]›, *unfolded inj-on-def*, *simplified*])
qed

lemma *Cont-RenH5*: ‹*finitary f* ⟹ *finitary g* ⟹ *finite* (⋃ *t* ∈ {*t*. *t* ≤ *s*}. {*s*. *t*=*map* (*map-event*$_{ptick}$ *f g*) *s*})›
apply (*rule finite-UN-I*)
unfolding *less-eq-list-def prefix-def*
apply (*induct s rule*: *rev-induct*, *simp*)
apply (*subgoal-tac* ‹{*t*. ∃ *r*. *t* @ *r* = *xs* @ [*x*]} = *insert* (*xs* @ [*x*]) {*t*. ∃ *r*. *t* @ *r* = *xs*}›, *auto*)

apply (*smt* (*verit*, *ccfv-SIG*) *Collect-cong finite-insert*)
apply (*metis Sublist.prefix-snoc prefix-def*)
using *Cont-RenH2 Cont-RenH4* **apply** *blast*
done

lemma *Cont-RenH6*: ‹{*t*. ∃ *t2*. *tickFree t* ∧ *front-tickFree t2* ∧ *x* = *map* (*map-event*$_{ptick}$ *f g*) *t* @ *t2*}
⊆ {*y*. ∃ *xa*. *xa* ∈ {*t*. *t* ≤ *x*} ∧ *y* ∈ {*s*. *xa* = *map* (*map-event*$_{ptick}$ *f g*) *s*}}›
by *auto*

lemma *Cont-RenH7*: ‹*finite* {*t*. ∃ *t2*. *tickFree t* ∧ *front-tickFree t2* ∧ *s* = *map* (*map-event*$_{ptick}$ *f g*) *t* @ *t2*}›
if ‹*finitary f*› **and** ‹*finitary g*›
proof −
have *f1*: ‹{*y*. *map* (*map-event*$_{ptick}$ *f g*) *y* ≤ *s*} = (⋃ *z* ∈ {*z*. *z* ≤ *s*}. {*y*. *z* = *map* (*map-event*$_{ptick}$ *f g*) *y*})› **by** *fast*
show *?thesis*
apply (*rule finite-subset*[*OF Cont-RenH6*])
apply (*simp add*: *f1*)
apply (*rule finite-UN-I*)
apply (*induct s rule*: *rev-induct*, *simp*)
apply (*subgoal-tac* ‹{*z*. *z* ≤ *xs* @ [*x*]} = *insert* (*xs* @ [*x*]) {*z*. *z* ≤ *xs*}›, *auto*)
using *Cont-RenH2 Cont-RenH4 that* **by** *blast*
qed

lemma *chain-Renaming*: ‹*chain Y* ⟹ *chain* (λ*i*. *Renaming* (*Y i*) *f g*)›
by (*simp add*: *mono-Renaming chain-def*)

9.4.3 Finally, continuity !

lemma *Cont-Renaming-prem*:
‹($\bigsqcup$ *i. Renaming* (*Y i*) *f g*) = *Renaming* ($\bigsqcup$ *i. Y i*) *f g*›
if *finitary*: ‹*finitary f*› ‹*finitary g*› **and** *chain*: ‹*chain Y*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ $\mathcal{D}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*)›
define *S* **where** ‹*S i* ≡ {*s1*. ∃ *t2*. *tickFree s1* ∧ *front-tickFree t2*
∧ *s* = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *t2* ∧ *s1* ∈ $\mathcal{D}$ (*Y i*)}› **for** *i*
have ‹($\bigcap$ *i. S i*) ≠ {}›
apply (*rule Inter-nonempty-finite-chained-sets*, *unfold S-def*)
apply (*use* ‹*s* ∈ $\mathcal{D}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*)› **in**
‹*simp add*: *limproc-is-thelub chain chain-Renaming D-Renaming D-LUB*, *blast*›)
apply (*rule finite-subset*[*OF - Cont-RenH7*[*OF finitary*, *of s*]], *blast*)
using *chain* **unfolding** *chain-def le-approx-def* **by** *blast*
then obtain *s1* **where** *f5*: ‹*s1* ∈ ($\bigcap$ *i. S i*)› **by** *blast*
thus ‹*s* ∈ $\mathcal{D}$ (*Renaming* (*Lub Y*) *f g*)›
by (*simp add*: *limproc-is-thelub chain chain-Renaming D-Renaming D-LUB S-def*) *blast*
next
show ‹*s* ∈ $\mathcal{D}$ (*Renaming* (*Lub Y*) *f g*) ⟹ *s* ∈ $\mathcal{D}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*)› **for** *s*
by (*auto simp add*: *limproc-is-thelub chain chain-Renaming D-Renaming D-LUB*)
next
next
assume *same-div* : ‹$\mathcal{D}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*) = $\mathcal{D}$ (*Renaming* ($\bigsqcup$ *i. Y i*) *f g*)›
fix *s X* **assume** ‹(*s*, *X*) ∈ $\mathcal{F}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*)›
show ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Renaming* ($\bigsqcup$ *i. Y i*) *f g*)›
proof (*cases* ‹*s* ∈ $\mathcal{D}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*)›)
show ‹*s* ∈ $\mathcal{D}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ (*Renaming* ($\bigsqcup$ *i. Y i*) *f g*)›
by (*simp add*: *is-processT8 same-div*)
next
assume ‹*s* ∉ $\mathcal{D}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*)›
then obtain *j* **where** ‹*s* ∉ $\mathcal{D}$ (*Renaming* (*Y j*) *f g*)›
by (*auto simp add*: *limproc-is-thelub chain-Renaming* ‹*chain Y*› *D-LUB*)
moreover from ‹(*s*, *X*) ∈ $\mathcal{F}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*)› **have** ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Renaming* (*Y j*) *f g*)›
by (*simp add*: *limproc-is-thelub chain-Renaming* ‹*chain Y*› *F-LUB*)
ultimately show ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Renaming* ($\bigsqcup$ *i. Y i*) *f g*)›
by (*fact le-approx2*[*OF mono-Renaming*[*OF is-ub-thelub*[*OF* ‹*chain Y*›]], *THEN iffD2*])
qed
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Renaming* (*Lub Y*) *f g*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ ($\bigsqcup$ *i. Renaming* (*Y i*) *f g*)› **for** *s X*

by (*auto simp add*: *limproc-is-thelub chain chain-Renaming F-Renaming D-LUB F-LUB*)
qed

lemma *Renaming-cont*[*simp*] : ‹*cont P* ⟹ *finitary f* ⟹ *finitary g* ⟹ *cont* (λ*x*. (*Renaming* (*P x*) *f g*))›
by (*rule contI2*)
(*simp add*: *cont2monofunE mono-Renaming monofun-def*,
simp add: *Cont-Renaming-prem ch2ch-cont cont2contlubE*)

lemma *RenamingF-cont* : ‹*cont P* ⟹ *cont* (λ*x*. (*P x*) ⟦*a* := *b*⟧ ⟦*c* := *d*⟧)›

by (*simp only*: *Renaming-cont finitary-updated-iff inj-imp-finitary inj-on-id*)

lemma ‹*cont P* ⟹ *cont* (λ*x*. (*P x*)⟦*a* := *b*, *c* := *d*, *e* := *f*, *g* := *a*⟧ ⟦*r1* := *r2*, *r3*:= *r4*, *r5*:= *r6*⟧)›
by *simp*

9.5 Some nice properties

lemma *map-event*$_{ptick}$-*comp*: ‹*map-event*$_{ptick}$ (*f2* ∘ *f1*) (*g2* ∘ *g1*) = *map-event*$_{ptick}$ *f2 g2* ∘ *map-event*$_{ptick}$ *f1 g1*›
by (*rule ext*) (*simp add*: *event*$_{ptick}$.*map-comp*)

lemma *Renaming-comp* : ‹*Renaming P* (*f2* ∘ *f1*) (*g2* ∘ *g1*) = *Renaming* (*Renaming P f1 g1*) *f2 g2*›
proof (*subst Process-eq-spec*, *intro conjI subset-antisym*)
show ‹*F* (*Renaming P* (*f2* ∘ *f1*) (*g2* ∘ *g1*)) ⊆ *F* (*Renaming* (*Renaming P f1 g1*) *f2 g2*)›
apply (*simp add*: *F-Renaming D-Renaming subset-iff*, *safe*)
by (*metis* (*no-types*, *opaque-lifting*) *map-event*$_{ptick}$-*comp list.map-comp set.compositionality*)
(*metis* (*no-types*, *opaque-lifting*) *map-event*$_{ptick}$-*comp map-event*$_{ptick}$-*tickFree append.right-neutral*
front-tickFree-Nil list.map-comp)
next
show ‹*F* (*Renaming* (*Renaming P f1 g1*) *f2 g2*) ⊆ *F* (*Renaming P* (*f2* ∘ *f1*) (*g2* ∘ *g1*))›
apply (*simp add*: *F-Renaming D-Renaming map-event*$_{ptick}$-*comp*, *safe*)
subgoal by *simp* (*metis* (*no-types*, *lifting*) *vimage-comp*)
subgoal using *map-event*$_{ptick}$-*front-tickFree* **by** *simp blast*
subgoal by *simp* (*metis front-tickFree-append map-event*$_{ptick}$-*tickFree*) .
next
show ‹*D* (*Renaming P* (*f2* ∘ *f1*) (*g2* ∘ *g1*)) ⊆ *D* (*Renaming* (*Renaming P f1 g1*) *f2 g2*)›

by (*simp add*: *D-Renaming subset-iff*, *safe*)
(*metis* (*no-types*, *opaque-lifting*) *map-event*$_{ptick}$*-comp map-event*$_{ptick}$*-tickFree append.right-neutral*
front-tickFree-Nil list.map-comp)
next
show ‹$\mathcal{D}$ (*Renaming* (*Renaming P f1 g1*) *f2 g2*) ⊆ $\mathcal{D}$ (*Renaming P* (*f2* ∘ *f1*) (*g2* ∘ *g1*))›
by (*auto simp add*: *D-Renaming subset-iff*)
(*metis map-event*$_{ptick}$*-comp map-event*$_{ptick}$*-tickFree front-tickFree-append*)
qed

lemma *Renaming-id*: ‹*Renaming P id id* = *P*›
by (*subst Process-eq-spec*, *auto simp add*: *F-Renaming D-Renaming event*$_{ptick}$*.map-id0 is-processT7 is-processT8*)
(*metis append.right-neutral front-tickFree-append-iff*
nonTickFree-n-frontTickFree not-Cons-self2 process-charn)

lemma *Renaming-inv*: ‹*Renaming* (*Renaming P f g*) (*inv f*) (*inv g*) = *P*› **if** ‹*inj f*› **and** ‹*inj g*›
proof −
have ‹ *Renaming* (*Renaming P f g*) (*inv f*) (*inv g*)
= *Renaming P* (*inv f* ∘ *f*) (*inv g* ∘ *g*)› **by** (*simp add*: *Renaming-comp*)
also have ‹... = *Renaming P id id*› **by** (*simp add*: ‹*inj f*› ‹*inj g*›)
also have ‹... = *P*› **by** (*fact Renaming-id*)
finally show ‹*Renaming* (*Renaming P f g*) (*inv f*) (*inv g*) = *P*› **.**
qed

lemma *inv-Renaming*: ‹*Renaming* (*Renaming P* (*inv f*) (*inv g*)) *f g* = *P*›
if ‹*bij f*› **and** ‹*bij g*›
proof −
have ‹ *Renaming* (*Renaming P* (*inv f*) (*inv g*)) *f g*
= *Renaming P* (*f* ∘ *inv f*) (*g* ∘ *inv g*)› **by** (*simp add*: *Renaming-comp*)
also have ‹... = *Renaming P id id*›
by (*metis bij-betw-def surj-iff* ‹*bij f*› ‹*bij g*›)
also have ‹... = *P*› **by** (*fact Renaming-id*)
finally show ‹*Renaming* (*Renaming P* (*inv f*) (*inv g*)) *f g* = *P*› **.**
qed

Chapter 10

The Hiding Operator

10.1 Preliminaries : primitives and lemmas

abbreviation ‹*trace-hide t A* ≡ *filter* (λ*x*. *x* ∉ *A*) *t*›

lemma *Hiding-tickFree* : ‹*tF* (*trace-hide s* (*ev* ‘ *A*)) ⟷ *tF s*›
by (*auto simp add*: *tickFree-def*)

lemma *Hiding-front-tickFree* : ‹*ftF s* ⟹ *ftF* (*trace-hide s* (*ev* ‘ *A*))›
apply (*induct s*; *simp add*: *image-iff front-tickFree-Cons-iff*)
by (*metis* (*no-types*, *lifting*) *filter.simps*(*1*) *front-tickFree-charn*)

lemma *trace-hide-union*: ‹*trace-hide t* (*A* ∪ *B*) = *trace-hide* (*trace-hide t A*) *B*›
by *simp*

lemma *trace-hide-ev-union* [*simp*] :
‹*trace-hide t* (*ev* ‘ (*A* ∪ *B*)) = *trace-hide* (*trace-hide t* (*ev* ‘ *A*)) (*ev* ‘ *B*)›
apply (*fold trace-hide-union*)
apply (*rule arg-cong*[**where** *f* = ‹λ*S*. *trace-hide t S*›])
by *blast*

abbreviation *isInfHiddenRun* :: ‹[*nat* ⇒ (′*a*, ′*r*) *event*$_{ptick}$ *list*, (′*a*, ′*r*) *process*$_{ptick}$, ′*a set*] ⇒ *bool*›
where ‹*isInfHiddenRun f P A* ≡ *strict-mono f* ∧ (∀ *i*. *f i* ∈ *T P*) ∧
(∀ *i*. *trace-hide* (*f i*) (*ev* ‘ *A*) = *trace-hide* (*f 0*) (*ev* ‘ *A*))›

lemma *isInfHiddenRun-1* :
‹*isInfHiddenRun f P A* ⟷ *strict-mono f* ∧ (∀ *i*. *f i* ∈ *T P*) ∧
(∀ *i*. ∃ *t*. *f i* = *f 0* @ *t* ∧ *set t* ⊆ *ev* ‘ *A*)›
proof (*intro iffI conjI*; *elim conjE*, *assumption?*)
assume ∗ : ‹*strict-mono f*› ‹∀ *i*. *trace-hide* (*f i*) (*ev* ‘ *A*) = *trace-hide* (*f 0*) (*ev* ‘ *A*)›
show ‹∀ *i*. ∃ *t*. *f i* = *f 0* @ *t* ∧ *set t* ⊆ *ev* ‘ *A*›

```
  proof (rule allI)
    fix i
    from *(1) obtain t where ‹f i = f 0 @ t›
      by (meson prefixE less-eq-nat.simps(1) strict-mono-less-eq)
    with *(2)[THEN spec, of i] have ‹set t ⊆ ev ‘ A›
      by simp (metis empty-filter-conv subset-code(1))
    with ‹f i = f 0 @ t› show ‹∃t. f i = f 0 @ t ∧ set t ⊆ ev ‘ A› by blast
  qed
next
  assume * : ‹∀i. ∃t. f i = f 0 @ t ∧ set t ⊆ ev ‘ A›
  show ‹∀i. trace-hide (f i) (ev ‘ A) = trace-hide (f 0) (ev ‘ A)›
  proof (rule allI)
    fix i
    from *[rule-format, of i] show ‹trace-hide (f i) (ev ‘ A) = trace-hide (f 0) (ev
‘ A)›
      by (elim exE, simp) (meson filter-False in-mono)
  qed
qed


abbreviation div-hide :: ‹('a, 'r) process_ptick ⇒ 'a set ⇒ ('a, 'r) event_ptick list
set›
  where ‹div-hide P A ≡ {s. ∃t u. ftF u ∧ tF t ∧
                                s = trace-hide t (ev ‘ A) @ u ∧
                                 (t ∈ 𝒟 P ∨ (∃f. isInfHiddenRun f P A ∧ t ∈ range
f))}›

lemma inf-hidden:
  ‹∃f. isInfHiddenRun f P A ∧ s ∈ range f›
  if ‹∀t. trace-hide t (ev ‘ A) = trace-hide s (ev ‘ A) ⟶ (t, ev ‘ A) ∉ ℱ P› and
‹s ∈ 𝒯 P›
proof (rule exI)
  define f where ‹f ≡ rec-nat s (λi t. let e = SOME e. e ∈ ev ‘ A ∧ t @ [e] ∈ 𝒯
P in t @ [e])›
  have ‹strict-mono f› by (simp add: f-def strict-mono-def lift-Suc-mono-less-iff)
   moreover have ‹s ∈ range f› unfolding f-def by (metis (no-types, lifting)
nat.rec(1) range-eqI)
  moreover have ‹f i ∈ 𝒯 P ∧ trace-hide (f i) (ev ‘ A) = trace-hide s (ev ‘ A)›
for i
  proof (induct i)
    show ‹f 0 ∈ 𝒯 P ∧ trace-hide (f 0) (ev ‘ A) = trace-hide s (ev ‘ A)›
      by (simp add: f-def that(2))
  next
    fix i
    assume hyp : ‹f i ∈ 𝒯 P ∧ trace-hide (f i) (ev ‘ A) = trace-hide s (ev ‘ A)›
   from is-processT5-S7[OF hyp[THEN conjunct1] that(1)[rule-format, OF hyp[THEN
conjunct2]]]
    obtain e where $ : ‹e ∈ ev ‘ A› ‹f i @ [e] ∈ 𝒯 P› by blast
    have ‹f (Suc i) = (let e = SOME e. e ∈ ev ‘ A ∧ f i @ [e] ∈ 𝒯 P in  f i @ [e])›
```

by (*simp add*: *f-def*)
thus ‹*f* (*Suc i*) ∈ $\mathcal{T}$ *P* ∧ *trace-hide* (*f* (*Suc i*)) (*ev* ‘ *A*) = *trace-hide s* (*ev* ‘ *A*)›
by (*simp add*: *hyp*[*THEN conjunct2*]) (*metis* (*no-types*, *lifting*) $ *someI-ex*)
qed
ultimately show ‹*isInfHiddenRun f P A* ∧ *s* ∈ *range f*› **by** *simp*
qed

lemma *trace-hide-append* :
‹*s* @ *t* = *trace-hide u* (*ev* ‘ *A*) ⟹
∃ *s*′ *t*′. *u* = *s*′ @ *t*′ ∧ *s* = *trace-hide s*′ (*ev* ‘ *A*) ∧ *t* = *trace-hide t*′ (*ev* ‘ *A*)›
proof (*induct u arbitrary*: *s t*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons e u*)
show *?case*
proof (*cases* ‹*e* ∈ *ev* ‘ *A*›)
assume ‹*e* ∈ *ev* ‘ *A*›
with *Cons.prems* **have** ‹*s* @ *t* = *trace-hide u* (*ev* ‘ *A*)› **by** *simp*
with *Cons.hyps* **obtain** *s*′ *t*′ **where**
‹*u* = *s*′ @ *t*′› ‹*s* = *trace-hide s*′ (*ev* ‘ *A*)› ‹*t* = *trace-hide t*′ (*ev* ‘ *A*)› **by** *blast*
hence ‹*e* # *u* = (*e* # *s*′) @ *t*′ ∧ *s* = *trace-hide* (*e* # *s*′) (*ev* ‘ *A*) ∧ *t* = *trace-hide* *t*′ (*ev* ‘ *A*)›
by (*simp add*: ‹*e* ∈ *ev* ‘ *A*›)
thus *?case* **by** *blast*
next
assume ‹*e* ∉ *ev* ‘ *A*›
with *Cons.prems* **consider** ‹*s* = []› ‹*t* ≠ []› ‹*hd t* = *e*› ‹[] @ *tl t* = *trace-hide* *u* (*ev* ‘ *A*)›
| ‹*s* ≠ []› ‹*hd s* = *e*› ‹*tl s* @ *t* = *trace-hide u* (*ev* ‘ *A*)› **by** (*cases s*) *simp-all*
thus *?case*
proof *cases*
show ‹⟦*s* = []; *t* ≠ []; *hd t* = *e*; [] @ *tl t* = *trace-hide u* (*ev* ‘ *A*)⟧ ⟹ *?case*›
by (*drule Cons.hyps*) (*metis Cons.prems append-Nil filter.simps*(*1*))
next
show ‹⟦*s* ≠ []; *hd s* = *e*; *tl s* @ *t* = *trace-hide u* (*ev* ‘ *A*)⟧ ⟹ *?case*›
by (*drule Cons.hyps*) (*metis Cons.prems Cons-eq-appendI append-same-eq* *filter-append*)
qed
qed
qed

10.2 The Hiding Operator Definition

lift-definition *Hiding* :: ‹[('*a*, '*r*) *process*$_{ptick}$,'*a set*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
(**infixl** ‹\› *69*)
is ‹λ*P A*. ({(*s*, *X*). ∃ *t*. *s* = *trace-hide t* (*ev* ‘ *A*) ∧ (*t*, *X* ∪ *ev* ‘ *A*) ∈ $\mathcal{F}$ *P*} ∪
{(*s*, *X*). *s* ∈ *div-hide P A*},

```
        div-hide P A)›
proof −
  show ‹?thesis P A› (is ‹is-process(?f, div-hide P A)›) for P and A
  proof (unfold is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv,
intro conjI allI impI)
    from inf-hidden[of A ‹[]› P] show ‹([], {}) ∈ ?f›
      by simp (metis filter.simps(1) tickFree-Nil)
  next
    show ‹(s, X) ∈ ?f ⟹ ftF s› for s X
      by simp (metis F-imp-front-tickFree Hiding-front-tickFree Hiding-tickFree
          front-tickFree-append-iff tickFree-imp-front-tickFree)
  next
    show ‹(s @ t, {}) ∈ ?f ⟹ (s, {}) ∈ ?f› for s t
    proof (induct t rule: rev-induct)
      case Nil
      then show ?case by auto
    next
      case (snoc e t)
       from snoc.prems consider u where ‹s @ t @ [e] = trace-hide u (ev ‘ A)›
‹(u, ev ‘ A) ∈ ℱ P›
        | u v where ‹ftF v› ‹tF u› ‹s @ t @ [e] = trace-hide u (ev ‘ A) @ v›
          ‹u ∈ 𝒟 P ∨ (∃f. isInfHiddenRun f P A ∧ u ∈ range f)›
        by simp blast
      thus ?case
      proof cases
        fix u assume ‹s @ t @ [e] = trace-hide u (ev ‘ A)› ‹(u, ev ‘ A) ∈ ℱ P›
       from this(1) obtain u' u'' where ‹u = u' @ u''› ‹s @ t = trace-hide u' (ev
‘ A)›
          by (metis append-assoc trace-hide-append)
        have ‹(s @ t, {}) ∈ ?f›
       proof (cases ‹∃t. trace-hide t (ev ‘ A) = trace-hide u' (ev ‘ A) ∧ (t, ev ‘ A)
∈ ℱ P›)
          show ‹∃t. trace-hide t (ev ‘ A) = trace-hide u' (ev ‘ A) ∧ (t, ev ‘ A) ∈ ℱ
P ⟹
                    (s @ t, {}) ∈ ?f›
            by (simp add: ‹s @ t = trace-hide u' (ev ‘ A)›) metis
        next
          assume ∗ : ‹∄t. trace-hide t (ev ‘ A) = trace-hide u' (ev ‘ A) ∧ (t, ev ‘ A)
∈ ℱ P›
          from this[simplified, THEN inf-hidden] obtain f where ‹isInfHiddenRun
f P A ∧ u' ∈ range f›
              using T-F-spec ‹(u, ev ‘ A) ∈ ℱ P› ‹u = u' @ u''› is-processT3
is-processT4-empty by blast
          hence ‹s @ t ∈ div-hide P A›
            by (simp add: ‹s @ t = trace-hide u' (ev ‘ A)›)
              (metis F-imp-front-tickFree ‹(u, ev ‘ A) ∈ ℱ P› ∗ ‹u = u' @ u''›
                append.right-neutral front-tickFree-Nil front-tickFree-append-iff)
          thus ‹(s @ t, {}) ∈ ?f› by simp
        qed
```

```
      from snoc.hyps[OF this] show ‹(s, {}) ∈ ?f› by blast
    next
      fix u v assume ∗ : ‹ftF v› ‹tF u› ‹s @ t @ [e] = trace-hide u (ev ‘ A) @ v›
        ‹u ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P A ∧ u ∈ range f)›
      show ?case
      proof (cases v rule: rev-cases)
        assume ‹v = []›
        with ∗(3) obtain u' u'' where ‹u = u' @ u''› ‹s @ t = trace-hide u' (ev
‘ A)›
          by simp (metis append-assoc trace-hide-append)
        from ∗(4) D-T have ‹u ∈ 𝒯 P› by blast
        have ‹(s @ t, {}) ∈ ?f›
        proof (cases ‹∃ t. trace-hide t (ev ‘ A) = trace-hide u' (ev ‘ A) ∧ (t, ev ‘
A) ∈ ℱ P›)
          show ‹∃ t. trace-hide t (ev ‘ A) = trace-hide u' (ev ‘ A) ∧ (t, ev ‘ A) ∈
ℱ P ⟹
                  (s @ t, {}) ∈ ?f›
            by (simp add: ‹s @ t = trace-hide u' (ev ‘ A)›) metis
        next
          assume ∗ : ‹∄ t. trace-hide t (ev ‘ A) = trace-hide u' (ev ‘ A) ∧ (t, ev ‘
A) ∈ ℱ P›
          from this[simplified, THEN inf-hidden] obtain f where ‹isInfHiddenRun
f P A ∧ u' ∈ range f›
            by (metis T-F-spec ‹u = u' @ u''› ‹u ∈ 𝒯 P› is-processT3)
          hence ‹s @ t ∈ div-hide P A›
            by (simp add: ‹s @ t = trace-hide u' (ev ‘ A)›)
              (metis (no-types, lifting) ∗ append-Nil2 append-T-imp-tickFree
                front-tickFree-Nil is-processT5-S7 list.distinct(1) rangeE)
          thus ‹(s @ t, {}) ∈ ?f› by simp
        qed
        from snoc.hyps[OF this] show ‹(s, {}) ∈ ?f› by blast
      next
        fix v' e'
        assume ‹v = v' @ [e']›
        with ∗ front-tickFree-dw-closed have ‹s @ t ∈ div-hide P A› by auto
        hence ‹(s @ t, {}) ∈ ?f› by simp
        from snoc.hyps[OF this] show ‹(s, {}) ∈ ?f› by blast
      qed
    qed
  qed

next
  show $ : ‹(s, Y) ∈ ?f ∧ X ⊆ Y ⟹ (s, X) ∈ ?f› for s X Y
    by simp (metis is-processT4 subset-iff-psubset-eq sup.mono)
next
  fix s X Y assume ∗ : ‹(s, X) ∈ ?f ∧ (∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ ?f)›
  from ∗[THEN conjunct1] consider ‹s ∈ div-hide P A›
    | u where ‹s = trace-hide u (ev ‘ A)› ‹(u, X ∪ ev ‘ A) ∈ ℱ P› by simp blast
  thus ‹(s, X ∪ Y) ∈ ?f›
```

```
  proof cases
    show ‹s ∈ div-hide P A ⟹ (s, X ∪ Y) ∈ ?f› by simp
  next
    fix u assume ‹s = trace-hide u (ev ‘ A)› ‹(u, X ∪ ev ‘ A) ∈ ℱ P›
    show ‹(s, X ∪ Y) ∈ ?f›
    proof (cases ‹s ∈ div-hide P A›)
      show ‹s ∈ div-hide P A ⟹ (s, X ∪ Y) ∈ ?f› by simp
    next
      assume ‹s ∉ div-hide P A›
      have ‹(u, X ∪ Y ∪ ev ‘ A) ∈ ℱ P›
      proof (rule ccontr)
        assume ‹(u, X ∪ Y ∪ ev ‘ A) ∉ ℱ P›
        hence ‹(u, X ∪ ev ‘ A ∪ Y) ∉ ℱ P› by (metis sup.assoc sup-commute)
        from is-processT5-S7'[OF ‹(u, X ∪ ev ‘ A) ∈ ℱ P› this] obtain c
          where ‹c ∈ Y› ‹c ∉ X› ‹c ∉ ev ‘ A› ‹u @ [c] ∈ 𝒯 P› by blast
        show False
        proof (cases ‹∃ t. trace-hide t (ev ‘ A) = trace-hide (u @ [c]) (ev ‘ A) ∧
(t, ev ‘ A) ∈ ℱ P›)
          show ‹∃ t. trace-hide t (ev ‘ A) = trace-hide (u @ [c]) (ev ‘ A) ∧ (t, ev ‘
A) ∈ ℱ P ⟹ False›
            using ∗ ‹c ∈ Y› ‹c ∉ ev ‘ A› ‹s = trace-hide u (ev ‘ A)› by force
        next
          assume ‹∄ t. trace-hide t (ev ‘ A) = trace-hide (u @ [c]) (ev ‘ A) ∧ (t,
ev ‘ A) ∈ ℱ P›
          hence ‹∀ t. trace-hide t (ev ‘ A) = trace-hide (u @ [c]) (ev ‘ A)
                  ⟶ (t, ev ‘ A) ∉ ℱ P› by simp
          from inf-hidden[OF this ‹u @ [c] ∈ 𝒯 P›] have ‹s @ [c] ∈ div-hide P A›

            by (smt (verit, ccfv-threshold) Prefix-Order.strict-prefix-simps(1)
                Prefix-Order.strict-prefix-simps(2)
                ‹∄ t. trace-hide t (ev ‘ A) = trace-hide (u @ [c]) (ev ‘ A) ∧ (t, ev ‘
A) ∈ ℱ P›
                  ‹s = trace-hide u (ev ‘ A)› ‹s ∉ div-hide P A› ‹u @ [c] ∈ 𝒯 P›
append-Nil2
                append-T-imp-tickFree filter.simps(1) filter.simps(2) filter-append
                front-tickFree-Nil is-processT5-S7 mem-Collect-eq)
          thus False using ∗ ‹c ∈ Y› by blast
        qed
      qed
      thus ‹(s, X ∪ Y) ∈ ?f› by (auto simp add: ‹s = trace-hide u (ev ‘ A)›)
    qed
  qed
next
  { fix s r assume ‹s @ [✓(r)] ∈ div-hide P A›
    then obtain t u where ∗ : ‹ftF u› ‹tF t›
      ‹s @ [✓(r)] = trace-hide t (ev ‘ A) @ u›
      ‹t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P A ∧ t ∈ range f)› by blast
    from ∗(2, 3) have ‹u ≠ []›
      by (cases u; cases t rule: rev-cases; simp split: if-split-asm)
```

```
        (metis Hiding-tickFree non-tickFree-tick tickFree-append-iff)
      with *(2, 3) have ‹s = trace-hide t (ev ‘ A) @ butlast u›
        by (cases u rule: rev-cases; simp)
    moreover from *(1)front-tickFree-iff-tickFree-butlast tickFree-imp-front-tickFree
      have ‹ftF (butlast u)› by blast
      ultimately show ‹s ∈ div-hide P A› using *(2, 4) by auto
    } note local-is-processT9 = this

    fix s r X assume ‹(s @ [✓(r)], {}) ∈ ?f›
    then consider ‹s @ [✓(r)] ∈ div-hide P A›
      | u where ‹s @ [✓(r)] = trace-hide u (ev ‘ A)› ‹(u, ev ‘ A) ∈ ℱ P› by simp
blast
    thus ‹(s, X − {tick r}) ∈ ?f›
    proof cases
      assume ‹s @ [✓(r)] ∈ div-hide P A›
      with local-is-processT9 have ‹s ∈ div-hide P A› by blast
      thus ‹(s, X − {tick r}) ∈ ?f› by simp
    next
      fix u assume ‹s @ [✓(r)] = trace-hide u (ev ‘ A)› ‹(u, ev ‘ A) ∈ ℱ P›
      from this(1) obtain u' where ‹u = u' @ [✓(r)]› ‹s = trace-hide u' (ev ‘ A)›
        by (cases u rule: rev-cases; simp split: if-split-asm)
           (metis F-imp-front-tickFree Hiding-tickFree ‹(u, ev ‘ A) ∈ ℱ P› tick-
Free-append-iff
          front-tickFree-iff-tickFree-butlast non-tickFree-tick snoc-eq-iff-butlast)
      have ‹X − {✓(r)} ∪ ev ‘ A = X ∪ ev ‘ A − {✓(r)}› by auto
      with is-processT6-TR[OF ‹(u, ev ‘ A) ∈ ℱ P›[THEN F-T, unfolded ‹u = u'
@ [✓(r)]›]]
      have ‹(u', X − {✓(r)} ∪ ev ‘ A) ∈ ℱ P› by presburger
     thus ‹(s, X − {tick r}) ∈ ?f› by (auto simp add: ‹s = trace-hide u' (ev ‘ A)›)
    qed
  next
    fix s t :: ‹('a, 'r) trace_ptick› assume * : ‹s ∈ div-hide P A ∧ tF s ∧ ftF t›
    from *[THEN conjunct1] obtain u v
      where ** : ‹ftF v› ‹tF u› ‹s = trace-hide u (ev ‘ A) @ v›
        ‹u ∈ 𝒟 P ∨ (∃f. isInfHiddenRun f P A ∧ u ∈ range f)› by blast
    from **(3) have ‹s @ t = trace-hide u (ev ‘ A) @ v @ t› by simp
    moreover from * **(3) front-tickFree-append have ‹ftF (v @ t)› by auto
    ultimately show ‹s @ t ∈ div-hide P A› using **(2) **(4) by blast
  next
    show ‹s ∈ div-hide P A ⟹ (s, X) ∈ ?f› for s X by simp
  qed
qed
```

10.3 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 (simp add: Failures.rep-eq Hiding.rep-eq FAILURES-def)
```

lemma *D-Hiding*: ‹$\mathcal{D}\ (P \setminus A) = \textit{div-hide}\ P\ A$›
by (*simp add*: *Divergences.rep-eq Hiding.rep-eq DIVERGENCES-def*)

lemma *T-Hiding*: ‹$\mathcal{T}\ (P \setminus A) = \{\textit{trace-hide}\ t\ (ev\ `\ A)\ |t.\ (t,\ ev\ `\ A) \in \mathcal{F}\ P\} \cup \textit{div-hide}\ P\ A$›
by (*auto simp add*: *Traces.rep-eq TRACES-def Failures.rep-eq*[*symmetric*] *F-Hiding*)
(*metis is-processT4 sup-ge2*, *metis sup-bot-left*)

lemma *Hiding-empty*: ‹$P \setminus \{\} = P$›
by (*auto simp add*: *Process-eq-spec D-Hiding F-Hiding strict-mono-eq intro*: *is-processT7 is-processT8*)
(*metis append.right-neutral front-tickFree-append-iff list.distinct*(*1*) *nonTickFree-n-frontTickFree process-charn*)

lemma *mem-D-imp-mem-D-Hiding*: ‹*trace-hide t* (*ev* ` *A*) ∈ $\mathcal{D}\ (P \setminus A)$› **if** ‹$t \in \mathcal{D}\ P$›
proof (*cases* ‹*tF t*›)
assume ‹*tF t*›
with ‹$t \in \mathcal{D}\ P$› **show** ‹*trace-hide t* (*ev* ` *A*) ∈ $\mathcal{D}\ (P \setminus A)$›
by (*simp add*: *D-Hiding*) (*use front-tickFree-Nil* **in** *blast*)
next
assume ‹¬ *tF t*›
with ‹$t \in \mathcal{D}\ P$› **obtain** $t'\ r$ **where** ‹$t = t' @ [\checkmark(r)]$› ‹*tF t'*› ‹$t' \in \mathcal{D}\ P$›
by (*metis D-imp-front-tickFree butlast-snoc is-processT9 front-tickFree-iff-tickFree-butlast nonTickFree-n-frontTickFree*)
thus ‹*trace-hide t* (*ev* ` *A*) ∈ $\mathcal{D}\ (P \setminus A)$›
by (*simp add*: *D-Hiding*) (*use front-tickFree-single* **in** *blast*)
qed

lemma *mem-T-imp-mem-T-Hiding*: ‹*trace-hide t* (*ev* ` *A*) ∈ $\mathcal{T}\ (P \setminus A)$› **if** ‹$t \in \mathcal{T}\ P$›
proof (*cases* ‹∃ *t'*. *trace-hide t* (*ev* ` *A*) = *trace-hide t'* (*ev* ` *A*) ∧ (*t'*, *ev* ` *A*) ∈ $\mathcal{F}\ P$›)
show ‹∃ *t'*. *trace-hide t* (*ev* ` *A*) = *trace-hide t'* (*ev* ` *A*) ∧ (*t'*, *ev* ` *A*) ∈ $\mathcal{F}\ P$ ⟹ *trace-hide t* (*ev* ` *A*) ∈ $\mathcal{T}\ (P \setminus A)$› **by** (*simp add*: *T-Hiding*)
next
assume ‹∄ *t'*. *trace-hide t* (*ev* ` *A*) = *trace-hide t'* (*ev* ` *A*) ∧ (*t'*, *ev* ` *A*) ∈ $\mathcal{F}\ P$›
with *inf-hidden*[*of A*, *OF* - ‹$t \in \mathcal{T}\ P$›] **obtain** *f*
where ‹*isInfHiddenRun f P A*› ‹$t \in \textit{range}\ f$› ‹*tF t*›
by (*metis T-nonTickFree-imp-decomp* ‹$t \in \mathcal{T}\ P$› *tick-T-F*)
thus ‹*trace-hide t* (*ev* ` *A*) ∈ $\mathcal{T}\ (P \setminus A)$›
by (*simp add*: *T-Hiding*) (*use front-tickFree-Nil* **in** *blast*)
qed

10.4 Continuity Rule

lemma *mono-Hiding* : ‹$(P \setminus A) \sqsubseteq (Q \setminus A)$› **if** ‹$(P{::}('a, 'r)\ process_{ptick}) \sqsubseteq Q$›

```
proof (unfold le-approx-def, intro conjI allI impI subsetI)
  from le-approx1[OF ‹P ⊑ Q›] le-approx-lemma-T[OF ‹P ⊑ Q›]
  show ‹s ∈ 𝒟 (Q \ A) ⟹ s ∈ 𝒟 (P \ A)› for s
    by (simp add: D-Hiding) blast
next
  { fix s t X
    assume assms : ‹s ∉ 𝒟 (P \ A)› ‹s = trace-hide t (ev ‘ A)›
    have ‹t ∉ 𝒟 P›
    proof (cases ‹ftF t›)
      from D-imp-front-tickFree show ‹¬ ftF t ⟹ t ∉ 𝒟 P› by blast
    next
      assume ‹ftF t›
      show ‹t ∉ 𝒟 P›
      proof (cases ‹tF t›)
        from ‹s ∉ 𝒟 (P \ A)› show ‹tF t ⟹ t ∉ 𝒟 P›
          by (simp add: assms(2) D-Hiding)
      next
        assume ‹¬ tF t›
        then obtain t' r where ‹t = t' @ [tick r]›
          by (meson ‹ftF t› nonTickFree-n-frontTickFree process-charn)
        with ‹s ∉ 𝒟 (P \ A)› show ‹t ∉ 𝒟 P›
          by (simp add: assms(2) D-Hiding image-iff)
            (metis front-tickFree-append-iff list.distinct(1) process-charn)
      qed
    qed
  } note ∗ = this

  fix s
  assume ‹s ∉ 𝒟 (P \ A)›
  show ‹ℛa (P \ A) s = ℛa (Q \ A) s›
  proof (intro subset-antisym subsetI)
    fix X
    assume ‹X ∈ ℛa (P \ A) s›
    with le-approx1[OF ‹P ⊑ Q›] le-approx-lemma-T[OF ‹P ⊑ Q›] ‹s ∉ 𝒟 (P \
A)›
    obtain t where $ : ‹s = trace-hide t (ev ‘ A)› ‹(t, X ∪ ev ‘ A) ∈ ℱ P›
      by (simp add: Refusals-after-def F-Hiding D-Hiding) blast
    from ∗[OF ‹s ∉ 𝒟 (P \ A)› ‹s = trace-hide t (ev ‘ A)›] have ‹t ∉ 𝒟 P› .
    from le-approx2[OF ‹P ⊑ Q› this] $
    show ‹X ∈ ℛa (Q \ A) s› by (simp add: Refusals-after-def F-Hiding) blast
  next
    fix X
    assume ‹X ∈ ℛa (Q \ A) s›
    with le-approx1[OF ‹P ⊑ Q›] le-approx-lemma-T[OF ‹P ⊑ Q›] ‹s ∉ 𝒟 (P \
A)›
    obtain t where $ : ‹s = trace-hide t (ev ‘ A)› ‹(t, X ∪ ev ‘ A) ∈ ℱ Q›
      by (simp add: Refusals-after-def F-Hiding D-Hiding) blast
    from ∗[OF ‹s ∉ 𝒟 (P \ A)› ‹s = trace-hide t (ev ‘ A)›] have ‹t ∉ 𝒟 P› .
    from le-approx2[OF ‹P ⊑ Q› this] $
```

```
    show ‹X ∈ ℛa (P \ A) s› by (simp add: Refusals-after-def F-Hiding) blast
  qed
next
  fix s
  assume ‹s ∈ min-elems (𝒟 (P \ A))›

  { fix t
    assume assms : ‹t ∈ 𝒟 P› ‹s = trace-hide t (ev ‘ A)› ‹tF t›
    from assms(1) obtain t' t'' where ‹t = t' @ t''› and ‹t' ∈ min-elems (𝒟 P)›
      by (meson min-elems-charn)
    with assms(3) elem-min-elems tickFree-append-iff
    have ‹trace-hide t' (ev ‘ A) ∈ 𝒟 (P \ A)›
      by (simp add: D-Hiding) (use front-tickFree-Nil in blast)
    from filter-append[of ‹(λx. x ∉ ev ‘ A)› t' t'', folded ‹t = t' @ t''›]
    have ‹trace-hide t (ev ‘ A) = trace-hide t' (ev ‘ A) @ trace-hide t'' (ev ‘ A)› .
    hence ‹s = trace-hide t' (ev ‘ A)›
      by (metis (no-types, lifting) assms(2) ‹s ∈ min-elems (𝒟 (P \ A))›
        min-elems-no ‹trace-hide t' (ev ‘ A) ∈ 𝒟 (P \ A)› less-eq-list-def prefix-def)
    have ‹s ∈ 𝒯 (Q \ A)›
    proof (cases ‹∀ v. trace-hide v (ev ‘ A) = trace-hide t' (ev ‘ A) ⟶ (v, ev ‘ A)
∉ ℱ Q›)
      assume ‹∀ v. trace-hide v (ev ‘ A) = trace-hide t' (ev ‘ A) ⟶ (v, ev ‘ A) ∉
ℱ Q›
      from inf-hidden[OF this] have ‹∃ f. isInfHiddenRun f Q A ∧ t' ∈ range f›
        by (meson ‹t' ∈ min-elems (𝒟 P)› in-mono le-approx-def that)
      thus ‹s ∈ 𝒯 (Q \ A)›
        by (simp add: T-Hiding)
          (use assms(3) ‹s = trace-hide t' (ev ‘ A)› ‹t = t' @ t''›
            front-tickFree-Nil tickFree-append-iff in blast)
    next
      show ‹¬ (∀ v. trace-hide v (ev ‘ A) = trace-hide t' (ev ‘ A)
                ⟶ (v, ev ‘ A) ∉ ℱ Q) ⟹ s ∈ 𝒯 (Q \ A)›
        by (simp add: T-Hiding) (metis ‹s = trace-hide t' (ev ‘ A)›)
    qed
  } note $ = this

  from elem-min-elems[OF ‹s ∈ min-elems (𝒟 (P \ A))›] have ‹s ∈ 𝒟 (P \ A)› .
  then obtain t u
    where ∗ : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ A) @ u›
      ‹t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P A ∧ t ∈ range f)›
    by (simp add: D-Hiding) blast
  have ‹u = []›
  proof (rule ccontr)
    assume ‹u ≠ []›
    have ‹ftF (butlast u)› ‹butlast s = trace-hide t (ev ‘ A) @ butlast u›
      by (use ∗(1) front-tickFree-iff-tickFree-butlast tickFree-imp-front-tickFree in
blast)
        (simp add: ∗(3) ‹u ≠ []› butlast-append)
    with ∗(2, 4) have ‹butlast s ∈ 𝒟 (P \ A)› by (simp add: D-Hiding) blast
```

```
    from min-elems-no[OF ‹s ∈ min-elems (𝒟 (P \ A))› this] ∗(3) ‹u ≠ []›
   show False by (metis Nil-is-append-conv append-butlast-last-id less-self nless-le)
  qed

  from ∗(4) show ‹s ∈ 𝒯 (Q \ A)›
  proof (elim disjE exE)
    show ‹t ∈ 𝒟 P ⟹ s ∈ 𝒯 (Q \ A)› using $ ∗(2, 3) ‹u = []› by blast
  next
    fix f
    assume assm : ‹isInfHiddenRun f P A ∧ t ∈ range f›
    show ‹s ∈ 𝒯 (Q \ A)›
    proof (cases ‹∀ i. f i ∈ 𝒯 Q›)
      from ∗(1, 2, 3) assm show ‹∀ i. f i ∈ 𝒯 Q ⟹ s ∈ 𝒯 (Q \ A)›
        by (simp add: T-Hiding) blast
    next
      assume ‹¬ (∀ i. f i ∈ 𝒯 Q)›
      then obtain i where ‹f i ∉ 𝒯 Q› by blast
      with assm le-approx2T ‹P ⊑ Q› have ‹f i ∈ 𝒟 P› by blast
      moreover have ‹s = trace-hide (f i) (ev ‘ A)›
        by (metis ∗(3) ‹u = []› append.right-neutral assm rangeE)
      moreover have ‹tF (f i)›
        by (metis ∗(2) ∗(3) Hiding-tickFree ‹u = []› append.right-neutral calcula-
tion(2))
      ultimately show ‹s ∈ 𝒯 (Q \ A)› using $ by blast
    qed
  qed
qed


lemma chain-Hiding : ‹chain Y ⟹ chain (λ i. Y i \ A)›
  by (simp add: chain-def mono-Hiding)

lemma KoenigLemma:
  ‹∃ (f::nat ⇒ 'a list). strict-mono f ∧ range f ⊆ {t. ∃ t'∈Tr. t ≤ t'}›
  if ∗ : ‹infinite (Tr::'a list set)› and ∗∗ : ‹∀ i. finite{t. ∃ t'∈Tr. t = take i t'}›
proof −
  define Tr' where ‹Tr' = {t. ∃ t'∈Tr. t ≤ t'}›
  have a : ‹infinite Tr'› by (rule infinite-super[OF - ∗]) (unfold Tr'-def, blast)
  have b : ‹finite {t ∈ Tr'. length t = i}› for i
    by (rule finite-subset[OF - ∗∗[THEN spec, of i]])
      (unfold Tr'-def, simp add: subset-iff, metis prefixE append-eq-conv-conj)
  have c : ‹∃ e. infinite {t' ∈ Tr'. t @ [e] < t'}› if ‹infinite {t' ∈ Tr'. t < t'}› for
t
  proof (rule ccontr)
    assume ‹∄ e. infinite {t' ∈ Tr'. t @ [e] < t'}›
    define E where ‹E ≡ {e |e. t @ [e] ∈ Tr'}›
    have ‹finite E›
      by (rule inj-on-finite[OF - - b[of ‹Suc (length t)›], of ‹λe. t @ [e]›])
        (simp-all add: inj-on-def E-def image-Collect-subsetI)
```

hence ‹*finite* $\{t \mathbin{@} [e] \mid e.\ e \in E\}$› **by** *simp*
have ‹$\{t' \in Tr'.\ t < t'\} = \{t \mathbin{@} [e] \mid e.\ e \in E\} \cup (\bigcup e{\in}E.\ \{t' \in Tr'.\ t \mathbin{@} [e] < t'\})$›
by (*auto simp add*: *E-def Tr′-def less-list-def less-eq-list-def prefix-def*)+
(*metis append-Cons append-self-conv2 neq-Nil-conv self-append-conv*)
with ‹$\nexists e.$ *infinite* $\{t' \in Tr'.\ t \mathbin{@} [e] < t'\}$› ‹*infinite* $\{t' \in Tr'.\ t < t'\}$› ‹*finite* *E*›
show *False* **by** *auto*
qed

define *f* **where** ‹$f \equiv$ *rec-nat* [] $(\lambda i\ t.$ *let* $e =$ *SOME* $e.$ *infinite* $\{t' \in Tr'.\ t \mathbin{@} [e] < t'\}$ *in* $t \mathbin{@} [e])$›
hence ‹*strict-mono f*› **by** (*simp add*: *lift-Suc-mono-less strict-monoI*)
moreover have ‹$f\ n \in Tr' \wedge$ *infinite* $\{t' \in Tr'.\ f\ n < t'\}$› **for** *n*
proof (*induct n*)
show ‹$f\ 0 \in Tr' \wedge$ *infinite* $\{t' \in Tr'.\ f\ 0 < t'\}$›
proof (*rule conjI*)
show ‹$f\ 0 \in Tr'$› **by** (*simp add*: *f-def Tr′-def* $*$ *not-finite-existsD*)
next
show ‹*infinite* $\{t' \in Tr'.\ f\ 0 < t'\}$›
by (*rule infinite-super*[*of* ‹$Tr' - \{f\ 0\}$›])
(*simp add*: *a Tr′-def f-def subset-iff less-list-def*, *simp add*: *a*)
qed
next
fix *n* **assume** *hyp* : ‹$f\ n \in Tr' \wedge$ *infinite* $\{t' \in Tr'.\ f\ n < t'\}$›
have ‹$f\ (Suc\ n) =$ (*let* $e =$ *SOME* $e.$ *infinite* $\{t' \in Tr'.\ f\ n \mathbin{@} [e] < t'\}$ *in* $f\ n \mathbin{@} [e])$›
by (*simp add*: *f-def*)
with *c*[*of* ‹$f\ n$›] **obtain** *e*
where \$: ‹$f\ (Suc\ n) = f\ n \mathbin{@} [e] \wedge$ *infinite* $\{t' \in Tr'.\ f\ n \mathbin{@} [e] < t'\}$›
by (*metis* (*no-types*, *lifting*) *hyp someI-ex*)
then obtain *i* **where** ‹$i \in Tr' \wedge f\ (Suc\ n) < i$› **using** *not-finite-existsD* **by** *auto*
with *dual-order.trans order-less-imp-le* **have** ‹$f\ (Suc\ n) \in Tr'$›
unfolding *Tr′-def* **by** *fastforce*
thus ‹$f\ (Suc\ n) \in Tr' \wedge$ *infinite* $\{t' \in Tr'.\ f\ (Suc\ n) < t'\}$› **by** (*simp add*: \$)
qed
ultimately show ‹$\exists (f{::}nat \Rightarrow 'a\ list).$ *strict-mono* $f \wedge$ *range* $f \subseteq Tr'$› **by** *blast*
qed

lemma *div-Hiding-lub* :
‹*finite* $(A{::}'a\ set) \Longrightarrow$ *chain* $Y \Longrightarrow \mathcal{D}\ (\bigsqcup\ i.\ (Y\ i \setminus A)) \subseteq \mathcal{D}\ ((\bigsqcup\ i.\ Y\ i) \setminus A)$›
for Y :: ‹$nat \Rightarrow ('a, 'r)\ process_{ptick}$›
proof (*auto simp add*:*limproc-is-thelub chain-Hiding D-Hiding T-Hiding D-LUB T-LUB*, *goal-cases*)

case (*1 x*)
{ fix *xa t u f*
assume *a*:*ftF u* ∧ *tF t* ∧ *x* = *trace-hide t* (*ev ‘ A*) @ *u* ∧
isInfHiddenRun f (*Y xa*) *A* ∧ (∀ *i*. *f i* ∉ $\mathcal{D}$ (*Y xa*)) ∧ *t* ∈ *range f*
hence ∀ *i n*. *f i* ∈ $\mathcal{T}$ (*Y n*) **using** *1*(*2*) *D-T chain-lemma le-approx2T* **by** *blast*
with *a* **have** *?case* **by** *blast*
} note *aa*=*this*
{ fix *xa t u f j*
assume *a*:*ftF u* ∧ *tF t* ∧ *x* = *trace-hide t* (*ev ‘ A*) @ *u* ∧
isInfHiddenRun f (*Y xa*) *A* ∧ (*f j* ∈ $\mathcal{D}$ (*Y xa*)) ∧ *t* ∈ *range f*
hence ∃ *t u*. *ftF u* ∧ *tF t* ∧ *x* = *trace-hide t* (*ev ‘ A*) @ *u* ∧ *t* ∈ $\mathcal{D}$ (*Y xa*)
apply(*rule-tac x*=*f j* **in** *exI*, *rule-tac x*=*u* **in** *exI*)
by (*metis Hiding-tickFree rangeE*)
} note *bb*=*this*
have *cc*: ∀ *xa*. ∃ *t u*. *ftF u* ∧ *tF t* ∧ *x* = *trace-hide t* (*ev ‘ A*) @ *u* ∧ *t* ∈ $\mathcal{D}$(*Y xa*)
⟹ *?case* (**is** ∀ *xa*. ∃ *t*. *?S t xa* ⟹ *?case*)
proof −
assume *dd*:∀ *xa*. ∃ *t u*. *ftF u* ∧ *tF t* ∧ *x* = *trace-hide t* (*ev ‘ A*) @ *u* ∧ *t* ∈ $\mathcal{D}$(*Y xa*)
(**is** ∀ *xa*. ∃ *t*. *?S t xa*)
define *f* :: ‹*nat* ⇒ (*'a*, *'r*) *event*$_{ptick}$ *list*› **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*:*ftF u* ∧ *x* = *trace-hide t* (*ev ‘ A*) @ *u* ∧ *tF t*
using *f-def dd*[*rule-format*, *of k*] *some-eq-ex*[*of* λ*t*. *?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* ∈ $\mathcal{D}$ (*Y n*) **using** *f-def dd*[*rule-format*, *of n*] *some-eq-ex*[*of* λ*t*. *?S t n*] **by** *auto*
with *gg 1*(*2*) **have** *t* ∈ $\mathcal{D}$ (*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*:*ftF u* ∧ *x* = *trace-hide t* (*ev ‘ A*) @ *u* ∧ *tF t*
using *f-def dd*[*rule-format*, *of k*] *some-eq-ex*[*of* λ*t*. *?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, 'r) event_ptick 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-TR)
  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 less-eq-list-def prefix-def mono-def strict-mono-mono)
  { fix n
    from gg obtain k r where f k = f' n @ r
      by (metis (lifting) ii less-eq-list-def prefix-def not-finite-existsD)
    hence trace-hide (f' n) (ev ‘ A) ≤ x
      using f-def dd[rule-format, of k] some-eq-ex[of λt. ?S t k]
      by (metis (no-types, lifting) prefixI prefix-prefix filter-append)
  } 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)))
          (λi 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* (*no-types*, *lifting*) *dual-order.eq-iff le-same-imp-eq-or-less ll or-*
der.strict-implies-order)
with *gg* **obtain** *k* **where** *nn*:*f k* ≥ *f'* *m* **by** *blast*
then obtain *u* **where** *oo*:*ftF u* ∧ *x* = *trace-hide* (*f'* *m*) (*ev* ‘ *A*) @ *u* ∧ *tF*
(*f'* *m*)
using *f-def dd*[*rule-format*, *of k*] *some-eq-ex*[*of* λ*t*. *?S t k*]

apply (*auto simp add*: *prefix-def tickFree-def disjoint-iff*)
by (*smt* (*z3*) *Prefix-Order.prefixE append.assoc disjoint-iff filter-append*
filter-is-subset front-tickFree-append subset-iff tickFree-append-iff tickFree-def)

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*=λ*n*. *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*. *ftF u* ∧ *tF 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*. *ftF u* ∧ *tF 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**
ftF u ∧ *tF t* ∧ *x* = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧

$(t \in \mathcal{D}\ (Y\ xa) \vee (\exists f.\ isInfHiddenRun\ f\ (Y\ xa)\ A \wedge t \in range\ f))$
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 D-T chain-lemma le-approx2T* **by** *blast*
qed
hence *ee*:$\forall xa.\ \exists t\ u.\ ftF\ u \wedge tF\ t \wedge x = trace\text{-}hide\ t\ (ev\ `\ A)\ @\ u \wedge t \in \mathcal{D}(Y\ xa)$
using *bb* **by** *blast*
with *cc* **show** *?thesis* **by** *simp*
qed
qed

lemma *Cont-Hiding-prem* : ‹$(\bigsqcup\ i.\ Y\ i) \setminus A = (\bigsqcup\ i.\ (Y\ i \setminus A))$› **if** ‹*finite A*› ‹*chain Y*›
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$s \in \mathcal{D}\ ((\bigsqcup\ i.\ Y\ i) \setminus A) \Longrightarrow s \in \mathcal{D}\ (\bigsqcup\ i.\ (Y\ i \setminus A))$› **for** *s*
by (*simp add*: *limproc-is-thelub chain-Hiding* ‹*chain Y*› *D-Hiding D-LUB T-LUB*) *blast*
next
show ‹$s \in \mathcal{D}\ (\bigsqcup\ i.\ (Y\ i \setminus A)) \Longrightarrow s \in \mathcal{D}\ ((\bigsqcup\ i.\ Y\ i) \setminus A)$› **for** *s*
using *div-Hiding-lub*[*OF* ‹*finite A*› ‹*chain Y*›] **by** *auto*
next
show ‹$(s,\ X) \in \mathcal{F}\ ((\bigsqcup\ i.\ Y\ i) \setminus A) \Longrightarrow (s,\ X) \in \mathcal{F}\ (\bigsqcup\ i.\ (Y\ i \setminus A))$› **for** *s X*
by (*simp add*: *limproc-is-thelub chain-Hiding* ‹*chain Y*› *F-Hiding F-LUB D-LUB T-LUB*) *blast*
next
assume *same-div* : ‹$\mathcal{D}\ ((\bigsqcup\ i.\ Y\ i) \setminus A) = \mathcal{D}\ (\bigsqcup\ i.\ (Y\ i \setminus A))$›
fix *s X* **assume** ‹$(s,\ X) \in \mathcal{F}\ (\bigsqcup\ i.\ (Y\ i \setminus A))$›
show ‹$(s,\ X) \in \mathcal{F}\ ((\bigsqcup\ i.\ Y\ i) \setminus A)$›
proof (*cases* ‹$s \in \mathcal{D}\ (\bigsqcup\ i.\ (Y\ i \setminus A))$›)
show ‹$s \in \mathcal{D}\ (\bigsqcup i.\ Y\ i \setminus A) \Longrightarrow (s,\ X) \in \mathcal{F}\ ((\bigsqcup i.\ Y\ i) \setminus A)$›
by (*simp add*: *is-processT8 same-div*)
next
assume ‹$s \notin \mathcal{D}\ (\bigsqcup i.\ Y\ i \setminus A)$›
then obtain *j* **where** ‹$s \notin \mathcal{D}\ (Y\ j \setminus A)$›
by (*auto simp add*: *limproc-is-thelub chain-Hiding* ‹*chain Y*› *D-LUB*)
moreover from ‹$(s,\ X) \in \mathcal{F}\ (\bigsqcup\ i.\ (Y\ i \setminus A))$› **have** ‹$(s,\ X) \in \mathcal{F}\ (Y\ j \setminus A)$›
by (*simp add*: *limproc-is-thelub chain-Hiding* ‹*chain Y*› *F-LUB*)
ultimately show ‹$(s,\ X) \in \mathcal{F}\ ((\bigsqcup i.\ Y\ i) \setminus A)$›
by (*fact le-approx2*[*OF mono-Hiding*[*OF is-ub-thelub*[*OF* ‹*chain Y*›]], *THEN iffD2*])
qed
qed

lemma *Hiding-cont*[*simp*]: ‹*cont* $(\lambda a.\ f\ a \setminus A)$› **if** ‹*finite A*› **and** ‹*cont f*›

proof (*rule cont-compose*[**where** $f = f$])
show ‹*cont* ($\lambda a.\ a \setminus A$)›
proof (*rule contI2*)
show ‹*monofun* ($\lambda a.\ a \setminus A$)› **by** (*simp add*: *mono-Hiding monofunI*)
next
show ‹*chain* $Y \Longrightarrow (\bigsqcup i.\ Y\ i) \setminus A \sqsubseteq (\bigsqcup i.\ Y\ i \setminus A)$›
for Y :: ‹*nat* ⇒ ('*a*, '*r*) $process_{ptick}$›
by (*simp add*: *Cont-Hiding-prem*[*OF* ‹*finite A*›])
qed
next
from ‹*cont f*› **show** ‹*cont f*› **.**
qed

lemma *length-strict-mono*: ‹*strict-mono f* $\Longrightarrow$ *i* + *length* (*f* 0) ≤ *length* (*f i*)›
for *f* :: ‹*nat* ⇒ '*a list*›
by (*induct i*)
(*simp-all add*: *Suc-le-eq less-length-mono order-le-less-trans strict-mono-Suc-iff*)

lemma *mono-trace-hide*: ‹*a* ≤ *b* $\Longrightarrow$ *trace-hide a* (*ev* ‘ *A*) ≤ *trace-hide b* (*ev* ‘ *A*)›
by (*metis prefixE prefixI filter-append*)

lemma *mono-constant*:
assumes *mono* (*f*::*nat* ⇒ ('*a*, '*r*) $event_{ptick}$ *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 (*intro exI*[*of* - *im*], *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

We can actually optimize the divergences of the (\) operator.

lemma *mono-take* : ‹*s* ≤ *t* $\Longrightarrow$ *take n s* ≤ *take n t*›
unfolding *less-eq-list-def prefix-def* **by** *auto*

lemma *shift-isInfHiddenRun* : ‹∃ *f*. *isInfHiddenRun f P A* ∧ *t* = *f* 0›

```
  if ‹isInfHiddenRun f P A ∧ t ∈ range f›
proof −
  from that obtain i where ‹t = f i› by blast
  hence ‹t = f (i + 0)› by simp
  moreover have ‹isInfHiddenRun (λj. f (i + j)) P A›
    by (metis add-Suc-right strict-mono-Suc-iff that)
  ultimately show ‹∃f. isInfHiddenRun f P A ∧ t = f 0› by blast
qed

lemma uniformize-length-isInfHiddenRun :
  assumes ∗ : ‹isInfHiddenRun f P A› ‹t = f 0›
  defines ‹g ≡ λi. take (i + length t) (f i)›
  shows ‹isInfHiddenRun g P A ∧ (∀ i. length (g i) = i + length t) ∧ t = g 0›
proof (intro conjI allI)
  show ‹strict-mono g›
  proof (unfold strict-mono-Suc-iff, rule allI)
    fix i
    from ∗(1) have ‹strict-mono f› by blast
    from length-strict-mono[of f ‹Suc i›, OF ‹strict-mono f›]
    have $ : ‹take (i + length (f 0)) (f (Suc i)) < take ((Suc i) + length (f 0)) (f
(Suc i))›
      by (simp add: take-Suc-conv-app-nth)
    from ‹strict-mono f›[THEN strict-monoD, of i ‹Suc i›, simplified]
    obtain u where ‹f (Suc i) = f i @ u› by (meson strict-prefixE′)
    with length-strict-mono[of f i, OF ‹strict-mono f›]
     have ‹take (i + length (f 0)) (f i) = take (i + length (f 0)) (f (Suc i))› by
simp
    with $ ∗(2) show ‹g i < g (Suc i)› unfolding g-def by presburger
  qed
next
  show ‹g i ∈ 𝒯 P› for i unfolding g-def
    by (metis ∗(1) prefixI append-take-drop-id is-processT3-TR)
next
  show ‹trace-hide (g i) (ev ‘ A) = trace-hide (g 0) (ev ‘ A)› for i
  proof (rule order-antisym)
    have ‹f 0 ≤ f i› by (simp add: ∗(1) strict-mono-less-eq)
    hence ‹f 0 ≤ take (i + length t) (f i)›
      by (metis ∗(2) prefixE prefixI le-add2 take-all-iff take-append)
    from mono-trace-hide[OF this]
    show ‹trace-hide (g i) (ev ‘ A) ≤ trace-hide (g 0) (ev ‘ A)›
      unfolding g-def
      by (metis ∗ prefixI append-take-drop-id filter-append le-add2 take-all-iff)
  next
   have $ : ‹take (i + length (f 0)) (f i) ≤ f i› by (metis prefixI append-take-drop-id)
    have ‹take (length t) (f 0) ≤ take (i + length t) (f 0)›
      ‹take (i + length t) (f 0) ≤ take (i + length t) (f i)›
      by (simp add: ∗(2))
        (meson ∗(1) le-add2 le-add-same-cancel2 mono-take strict-mono-less-eq)
    from order-trans[OF this] have ‹g 0 ≤ g i› unfolding g-def by simp
```

```
  thus ‹trace-hide (g 0) (ev ‘ A) ≤ trace-hide (g i) (ev ‘ A)› by (fact mono-trace-hide)
 qed
next
 show ‹length (g i) = i + length t› for i by (simp add: ∗ g-def length-strict-mono)
next
 show ‹t = g 0› by (simp add: ∗(2) g-def)
qed


abbreviation isInfHiddenRun-optimized ::
 ‹[nat ⇒ ('a, 'r) event_ptick list, ('a, 'r) process_ptick, 'a set, ('a, 'r) trace_ptick] ⇒
bool›
 where ‹isInfHiddenRun-optimized f P A t ≡
       strict-mono f ∧ (∀ i. f i ∈ 𝒯 P) ∧ (∀ i. f i ∉ 𝒟 P) ∧
       (∀ i. trace-hide (f i) (ev ‘ A) = trace-hide (f 0) (ev ‘ A)) ∧
       (∀ i. length (f i) = i + length t) ∧ t = f 0›

abbreviation div-hide-optimized :: ‹('a, 'r) process_ptick ⇒ 'a set ⇒ ('a, 'r) event_ptick
list set›
 where ‹div-hide-optimized P A ≡
       {s. ∃ t u. ftF u ∧ tF t ∧
                  s = trace-hide t (ev ‘ A) @ u ∧
                  (t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun-optimized f P A t))}›


lemma D-Hiding-optimized : ‹𝒟 (P \ A) = div-hide-optimized P A›
proof (unfold D-Hiding, intro subset-antisym subsetI)
 show ‹s ∈ div-hide-optimized P A ⟹ s ∈ div-hide P A› for s by blast
next
 fix s assume ‹s ∈ div-hide P A›
 then obtain t u
   where ∗ : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ A) @ u›
     ‹t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P A ∧ t ∈ range f)› by blast
 from ∗(4) show ‹s ∈ div-hide-optimized P A›
 proof (elim disjE exE)
   from ∗(1, 2, 3) show ‹t ∈ 𝒟 P ⟹ s ∈ div-hide-optimized P A› by blast
 next
   fix f assume ‹isInfHiddenRun f P A ∧ t ∈ range f›
   with shift-isInfHiddenRun obtain f where ∗∗ : ‹isInfHiddenRun f P A ∧ t =
f 0› by blast
   show ‹s ∈ div-hide-optimized P A›
   proof (cases ‹∃ i. f i ∈ 𝒟 P›)
     assume ‹∃ i. f i ∈ 𝒟 P›
     then obtain i where ‹f i ∈ 𝒟 P› ..
     have ‹trace-hide t (ev ‘ A) = trace-hide (f i) (ev ‘ A)› by (metis ∗∗)
     moreover from calculation have ‹tF (f i)› by (metis ∗(2) Hiding-tickFree)
     ultimately show ‹s ∈ div-hide-optimized P A›
       using ∗(1, 3) ‹f i ∈ 𝒟 P› by blast
   next
```

assume ‹$\nexists i.\ f\ i \in \mathcal{D}\ P$›
from ** *uniformize-length-isInfHiddenRun*[*of f P A t*]
have ‹*isInfHiddenRun* ($\lambda i.$ *take* (i + *length t*) (*f i*)) *P A* ∧
($\forall i.$ *length* (*take* (i + *length t*) (*f i*)) = i + *length t*) ∧
t = *take* (*0* + *length t*) (*f 0*)› **by** *blast*
moreover from ‹$\nexists i.\ f\ i \in \mathcal{D}\ P$› **have** ‹$\forall i.$ *take* (i + *length t*) (*f i*) $\notin \mathcal{D}\ P$›
by (*metis* ** *append-Nil2 append-take-drop-id*
front-tickFree-nonempty-append-imp is-processT2-TR is-processT7)
ultimately show ‹*s* ∈ *div-hide-optimized P A*› **using** *(1, 2, 3) **by** *blast*
qed
qed
qed

lemma *T-Hiding-optimized* :
‹$\mathcal{T}$ (*P* \ *A*) = {*trace-hide t* (*ev* ‘ *A*) |*t*. (*t*, *ev* ‘ *A*) ∈ $\mathcal{F}$ *P*} ∪ *div-hide-optimized P A*›
by (*unfold T-Hiding*, *fold D-Hiding*, *unfold D-Hiding-optimized*) (*fact refl*)

Actually, *f i* can be rewritten as *t* @ *map x* [*0*..<*i*] where *x* is the sequence such that *f* (*Suc i*) = *f i* @ [*x i*].

definition *seqRun* :: ‹[('*a*, '*r*) *trace*$_{ptick}$, *nat* ⇒ ('*a*, '*r*) *event*$_{ptick}$] ⇒ *nat* ⇒ ('*a*, '*r*) *trace*$_{ptick}$›
where ‹*seqRun t x i* ≡ *t* @ *map x* [*0*..<*i*]›

lemma *seqRun-is-rec-nat* : ‹*seqRun t x* = *rec-nat t* ($\lambda i\ t.\ t$ @ [*x i*])›
proof (*rule ext*)
show ‹*seqRun t x i* = *rec-nat t* ($\lambda i\ t.\ t$ @ [*x i*]) *i*› **for** *i*
by (*induct i*) (*simp-all add*: *seqRun-def*)
qed

lemma *seqRun-0* [*simp*] : ‹*seqRun t x 0* = *t*›
and *seqRun-Suc* [*simp*] : ‹*seqRun t x* (*Suc i*) = *seqRun t x i* @ [*x i*]›
and *seqRun-Nil* [*simp*] : ‹*seqRun* [] *x i* = *map x* [*0*..<*i*]›
and *seqRun-Cons* [*simp*] : ‹*seqRun* (*a* # *t*) *x i* = *a* # *seqRun t x i*›
by (*simp-all add*: *seqRun-def*)

lemma *strict-mono-seqRun* [*simp*] : ‹*strict-mono* (*seqRun t x*)›
by (*simp add*: *strict-mono-Suc-iff*)

lemma *length-seqRun* [*simp*] : ‹*length* (*seqRun t x i*) = *i* + *length t*›
by (*simp add*: *seqRun-def*)

lemma *t-le-seqRun* [*simp*] : ‹*t* ≤ *seqRun t x i*› **by** (*simp add*: *seqRun-def*)

lemma *take-t-le-seqRun* [*simp*] : ‹*take i t* ≤ *seqRun t x j*›
by (*induct t*, *simp-all add*: *seqRun-def less-eq-list-def prefix-def*)
(*metis append.assoc append-Cons append-take-drop-id*)

lemma *nth-seqRun* :
‹$j < i + length\ t \Longrightarrow$
$seqRun\ t\ x\ i\ !\ j = (if\ j < length\ t\ then\ t\ !\ j\ else\ x\ (j - length\ t))$›
by (*simp add*: *seqRun-def nth-append*)

lemma *take-seqRun* [*simp*] :
‹$take\ j\ (seqRun\ t\ x\ i) = (if\ j \leq length\ t\ then\ take\ j\ t\ else\ seqRun\ t\ x\ (min\ i\ (j - length\ t)))$›
by (*simp add*: *seqRun-def min-def take-map*)

lemma *seqRun-eq-iff* [*simp*] : ‹$seqRun\ t\ x\ i = seqRun\ t\ x\ j \longleftrightarrow i = j$›
by (*metis add-right-cancel length-seqRun*)

lemma *seqRun-le-iff* [*simp*] : ‹$seqRun\ t\ x\ i \leq seqRun\ t\ x\ j \longleftrightarrow i \leq j$›
by (*simp add*: *strict-mono-less-eq*)

lemma *seqRun-less-iff* [*simp*] : ‹$seqRun\ t\ x\ i < seqRun\ t\ x\ j \longleftrightarrow i < j$›
by (*simp add*: *strict-mono-less*)

lemma *trace-hide-is-Nil-iff* : ‹$trace\text{-}hide\ s\ A = [] \longleftrightarrow set\ s \subseteq A$›
by (*simp add*: *filter-empty-conv subset-code*(*1*))

lemma *trace-hide-seqRun-eq-iff* :
‹$trace\text{-}hide\ (seqRun\ t\ x\ i)\ A = trace\text{-}hide\ t\ A \longleftrightarrow (\forall j{<}i.\ x\ j \in A)$›
by (*simp add*: *seqRun-def trace-hide-is-Nil-iff subset-iff image-iff*)
(*use atLeastLessThan-iff* **in** *blast*)

abbreviation *isInfHidden-seqRun* ::
‹$[nat \Rightarrow ('a, 'r)\ event_{ptick}, ('a, 'r)\ process_{ptick}, 'a\ set, ('a, 'r)\ trace_{ptick}] \Rightarrow bool$›
where ‹$isInfHidden\text{-}seqRun\ x\ P\ A\ t \equiv \forall i.\ seqRun\ t\ x\ i \in \mathcal{T}\ P \wedge x\ i \in ev\ `\ A$›

abbreviation *isInfHidden-seqRun-strong* ::
‹$[nat \Rightarrow ('a, 'r)\ event_{ptick}, ('a, 'r)\ process_{ptick}, 'a\ set, ('a, 'r)\ trace_{ptick}] \Rightarrow bool$›
where ‹$isInfHidden\text{-}seqRun\text{-}strong\ x\ P\ A\ t \equiv$
$\forall i.\ seqRun\ t\ x\ i \in \mathcal{T}\ P \wedge seqRun\ t\ x\ i \notin \mathcal{D}\ P \wedge x\ i \in ev\ `\ A$›

abbreviation *div-hide-seqRun* :: ‹$('a, 'r)\ process_{ptick} \Rightarrow 'a\ set \Rightarrow ('a, 'r)\ event_{ptick}\ list\ set$›
where ‹$div\text{-}hide\text{-}seqRun\ P\ A \equiv$
$\{s.\ \exists t\ u.\ ftF\ u \wedge tF\ t \wedge s = trace\text{-}hide\ t\ (ev\ `\ A)\ @\ u \wedge$
$(t \in \mathcal{D}\ P \vee (\exists x.\ isInfHidden\text{-}seqRun\ x\ P\ A\ t))\}$›

lemma *D-Hiding-seqRun* : ‹$\mathcal{D}\ (P \setminus A) = div\text{-}hide\text{-}seqRun\ P\ A$›
proof (*intro subset-antisym subsetI*)

```
  fix s assume ‹s ∈ 𝒟 (P \ A)›
  then obtain t u
    where ∗ : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ A) @ u›
      ‹t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun-optimized f P A t)›
    unfolding D-Hiding-optimized by blast
  show ‹s ∈ div-hide-seqRun P A›
  proof (clarify, intro exI conjI)
    show ‹ftF u› ‹tF t›
      ‹s = trace-hide t (ev ‘ A) @ u› by (fact ∗(1, 2, 3))+
  next
    from ∗(4) show ‹t ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun x P A t)›
    proof (elim disjE exE)
      show ‹t ∈ 𝒟 P ⟹ t ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun x P A t)› ..
    next
      fix f assume $ : ‹isInfHiddenRun-optimized f P A t›
      define x where ‹x i ≡ f (Suc i) ! (i + length t)› for i
      have $$ : ‹seqRun t x i = f i› for i
      proof (induct i)
        show ‹seqRun t x 0 = f 0› by (simp add: $)
      next
        fix i assume ‹seqRun t x i = f i›
        from $ have ‹length (f (Suc i)) = Suc i + length t›
          ‹length (f i) = i + length t› ‹f i ≤ f (Suc i)›
          by (blast, blast, simp add: strict-mono-less-eq)
        then obtain y where ‹f (Suc i) = f i @ [y]›
          by (simp add: less-eq-list-def prefix-def)
            (metis append-eq-append-conv length-Suc-conv-rev)
        with ‹length (f i) = i + length t› have ‹f (Suc i) = f i @ [x i]›
          unfolding x-def by (metis nth-append-length)
        thus ‹seqRun t x (Suc i) = f (Suc i)› by (simp add: ‹seqRun t x i = f i›)
      qed
      have ‹isInfHidden-seqRun x P A t›
      proof (intro allI conjI)
        from $ $$ show ‹seqRun t x i ∈ 𝒯 P› for i by presburger+
      next
        from trace-hide-seqRun-eq-iff[of ‹ev ‘ A› t x, unfolded $$] $
        show ‹x i ∈ ev ‘ A› for i by blast
      qed
      thus ‹t ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun x P A t)› by blast
    qed
  qed
next
  fix s assume ‹s ∈ div-hide-seqRun P A›
  then obtain t u
    where ∗ : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ A) @ u›
      ‹t ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun x P A t)› by blast

  show ‹s ∈ 𝒟 (P \ A)›
  proof (unfold D-Hiding-optimized)
```

```
    from ∗(4) show ‹s ∈ div-hide-optimized P A›
    proof (elim disjE exE)
      from ∗(1, 2, 3) show ‹t ∈ 𝒟 P ⟹ s ∈ div-hide-optimized P A› by blast
    next
      fix x assume $ : ‹isInfHidden-seqRun x P A t›
      show ‹s ∈ div-hide-optimized P A›
      proof (cases ‹∃ i. seqRun t x i ∈ 𝒟 P›)
        assume ‹∃ i. seqRun t x i ∈ 𝒟 P›
        then obtain i where ‹seqRun t x i ∈ 𝒟 P› ..
        show ‹s ∈ div-hide-optimized P A›
        proof (clarify, intro exI conjI)
          show ‹ftF u› by (fact ∗(1))
        next
          show ‹tF (seqRun t x i)›
           by (metis $ append-T-imp-tickFree length-Cons n-not-Suc-n seqRun-Suc)
        next
          show ‹s = trace-hide (seqRun t x i) (ev ‘ A) @ u›
            by (metis $ ∗(3) trace-hide-seqRun-eq-iff)
        next
          from ‹seqRun t x i ∈ 𝒟 P›
         show ‹seqRun t x i ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun-optimized f P A (seqRun
t x i))› ..
        qed
      next
        assume ‹∄ i. seqRun t x i ∈ 𝒟 P›
        hence ‹isInfHiddenRun-optimized (seqRun t x) P A t›
          by (simp add: trace-hide-seqRun-eq-iff $)
        with ∗(1, 2, 3) show ‹s ∈ div-hide-optimized P A› by blast
      qed
    qed
  qed
qed


lemma T-Hiding-seqRun :
  ‹𝒯 (P \ A) = {trace-hide t (ev ‘ A) |t. (t, ev ‘ A) ∈ ℱ P} ∪ div-hide-seqRun P
A›
  by (unfold T-Hiding, fold D-Hiding-seqRun D-Hiding) (fact refl)

lemma F-Hiding-seqRun :
  ‹ℱ (P \ A) =
   {(s, X). ∃ t. s = trace-hide t (ev ‘ A) ∧ (t, X ∪ ev ‘ A) ∈ ℱ P} ∪
   {(s, X). s ∈ div-hide-seqRun P A}›
  by (unfold F-Hiding, fold D-Hiding-seqRun D-Hiding) (fact refl)


lemma D-Hiding-seqRunI :
  ‹⟦ftF u; tF t; s = trace-hide t (ev ‘ A) @ u;
    t ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun x P A t)⟧ ⟹ s ∈ 𝒟 (P \ A)›
```

```
  unfolding D-Hiding-seqRun by blast

lemma D-Hiding-seqRunE :
  assumes ‹s ∈ 𝒟 (P \ A)›
  obtains t u where ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ A) @ u›
    ‹t ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun-strong x P A t)›
proof −
  from ‹s ∈ 𝒟 (P \ A)› obtain t u
    where ∗ : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ A) @ u›
      ‹t ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun x P A t)› unfolding D-Hiding-seqRun
by blast
  from ∗(4) show thesis
  proof (elim disjE exE)
    from ∗(1, 2, 3) that show ‹t ∈ 𝒟 P ⟹ thesis› by blast
  next
    fix x assume $ : ‹isInfHidden-seqRun x P A t›
    show thesis
    proof (cases ‹∃ i. seqRun t x i ∈ 𝒟 P›)
      assume ‹∃ i. seqRun t x i ∈ 𝒟 P›
      then obtain i where ‹seqRun t x i ∈ 𝒟 P› ..
      show thesis
      proof (rule that)
        show ‹ftF u› by (fact ∗(1))
      next
        show ‹tF (seqRun t x i)›
          by (metis $ append-T-imp-tickFree neq-Nil-conv seqRun-Suc)
      next
        show ‹s = trace-hide (seqRun t x i) (ev ‘ A) @ u›
          by (simp add: ∗(3)) (metis $ trace-hide-seqRun-eq-iff)
      next
        from ‹seqRun t x i ∈ 𝒟 P›
        show ‹seqRun t x i ∈ 𝒟 P ∨
              (∃ y. isInfHidden-seqRun-strong y P A (seqRun t x i))› ..
      qed
    next
      from ∗(1, 2, 3) $ that show ‹∄ i. seqRun t x i ∈ 𝒟 P ⟹ thesis› by blast
    qed
  qed
qed


lemma T-Hiding-seqRunE :
  assumes ‹s ∈ 𝒯 (P \ A)›
  obtains t where ‹s = trace-hide t (ev ‘ A)› ‹(t, ev ‘ A) ∈ ℱ P›
  | t u where ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ A) @ u›
    ‹t ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun-strong x P A t)›
proof (cases ‹s ∈ 𝒟 (P \ A)›)
  assume ‹s ∉ 𝒟 (P \ A)›
  with ‹s ∈ 𝒯 (P \ A)› have ‹s ∈ {trace-hide t (ev ‘ A) |t. (t, ev ‘ A) ∈ ℱ P}›
```

unfolding *D-Hiding T-Hiding* **by** *blast*
with *that(1)* **show** *thesis* **by** *blast*
qed (*elim D-Hiding-seqRunE*, *presburger*)

lemma *butlast-seqRun* : ‹*butlast* (*seqRun t x i*) = (*case i of 0* ⇒ *butlast t*
| *Suc j* ⇒ *seqRun t x j*)›
by (*cases i*) *simp-all*

lemma *isInfHidden-seqRun-imp-tickFree* : ‹*isInfHidden-seqRun x P A t* ⟹ *tF t*›
by (*metis append-T-imp-tickFree not-Cons-self2 seqRun-0 seqRun-Suc*)

lemma *tickFree-seqRun-iff* : ‹*tF* (*seqRun t x i*) ⟷ *tF t* ∧ (∀ *j*<*i*. *is-ev* (*x j*))›
by (*induct i*; *simp*) (*metis less-Suc-eq*)

lemma *front-tickFree-seqRun-iff* :
‹*ftF* (*seqRun t x i*) ⟷ (*case i of 0* ⇒ *ftF t* | *Suc j* ⇒ *tF t* ∧ (∀ *k*<*j*. *is-ev* (*x k*)))›
by (*cases i*) (*simp-all add*: *front-tickFree-iff-tickFree-butlast tickFree-seqRun-iff*)

lemmas *Hiding-seqRun-projs* = *F-Hiding-seqRun D-Hiding-seqRun T-Hiding-seqRun*

Chapter 11

The CSP Refinements

11.1 Definitions and first Properties

11.1.1 Definitions

definition *trace-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*›
(**infix** ‹$\sqsubseteq_T$› *50*)
where ‹$P \sqsubseteq_T Q \equiv \mathcal{T}\ Q \subseteq \mathcal{T}\ P$›

definition *failure-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*›
(**infix** ‹$\sqsubseteq_F$› *50*)
where ‹$P \sqsubseteq_F Q \equiv \mathcal{F}\ Q \subseteq \mathcal{F}\ P$›

definition *divergence-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*›
(**infix** ‹$\sqsubseteq_D$› *50*)
where ‹$P \sqsubseteq_D Q \equiv \mathcal{D}\ Q \subseteq \mathcal{D}\ P$›

abbreviation *failure-divergence-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$
⇒ *bool*› (**infix** ‹$\sqsubseteq_{FD}$› *50*)
where ‹$P \sqsubseteq_{FD} Q \equiv P \leq Q$›

definition *trace-divergence-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒
bool› (**infix** ‹$\sqsubseteq_{DT}$› *50*)

where ‹$P \sqsubseteq_{DT} Q \equiv P \sqsubseteq_T Q \wedge P \sqsubseteq_D Q$›

lemma *failure-divergence-refine-def* : ‹$P \sqsubseteq_{FD} Q \longleftrightarrow P \sqsubseteq_F Q \wedge P \sqsubseteq_D Q$›
unfolding *less-eq-process*$_{ptick}$*-def failure-refine-def divergence-refine-def* **by** *argo*

lemmas *refine-defs = failure-divergence-refine-def trace-divergence-refine-def*
failure-refine-def divergence-refine-def trace-refine-def

lemma *failure-divergence-refineI* :
‹⟦⋀*s*. *s* ∈ $\mathcal{D}$ *Q* ⟹ *s* ∈ $\mathcal{D}$ *P*; ⋀*s X*. (*s*, *X*) ∈ $\mathcal{F}$ *Q* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *P*⟧ ⟹ *P* $\sqsubseteq_{FD}$ *Q*›
by (*auto simp add*: *refine-defs*)

lemma *failure-divergence-refine-optimizedI* :
‹⟦⋀*s*. *s* ∈ $\mathcal{D}$ *Q* ⟹ *s* ∈ $\mathcal{D}$ *P*; ⋀*s X*. (*s*, *X*) ∈ $\mathcal{F}$ *Q* ⟹ $\mathcal{D}$ *Q* ⊆ $\mathcal{D}$ *P* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *P*⟧ ⟹ *P* $\sqsubseteq_{FD}$ *Q*›
by (*simp add*: *failure-divergence-refineI subsetI*)

lemma *trace-divergence-refineI* :
‹⟦⋀*s*. *s* ∈ $\mathcal{D}$ *Q* ⟹ *s* ∈ $\mathcal{D}$ *P*; ⋀*s*. *s* ∈ $\mathcal{T}$ *Q* ⟹ *s* ∈ $\mathcal{T}$ *P*⟧ ⟹ *P* $\sqsubseteq_{DT}$ *Q*›
by (*auto simp add*: *refine-defs*)

lemma *trace-divergence-refine-optimizedI* :
‹⟦⋀*s*. *s* ∈ $\mathcal{D}$ *Q* ⟹ *s* ∈ $\mathcal{D}$ *P*; ⋀*s*. *s* ∈ $\mathcal{T}$ *Q* ⟹ $\mathcal{D}$ *Q* ⊆ $\mathcal{D}$ *P* ⟹ *s* ∈ $\mathcal{T}$ *P*⟧ ⟹ *P* $\sqsubseteq_{DT}$ *Q*›
by (*simp add*: *trace-divergence-refineI subsetI*)

11.1.2 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*: *refine-defs*)

11.1.3 Some obvious refinements

lemma *BOT-leF* [*simp*] : ‹⊥ $\sqsubseteq_F$ *Q*›
and *BOT-leD* [*simp*] : ‹⊥ $\sqsubseteq_D$ *Q*›
and *BOT-leT* [*simp*] : ‹⊥ $\sqsubseteq_T$ *Q*›
and *BOT-leFD*[*simp*] : ‹⊥ $\sqsubseteq_{FD}$ *Q*›
and *BOT-leDT*[*simp*] : ‹⊥ $\sqsubseteq_{DT}$ *Q*›
by (*simp-all add*: *refine-defs le-approx-lemma-F le-approx-lemma-T le-approx1*)

11.1.4 Antisymmetry

lemma *FD-antisym*: ‹*P* $\sqsubseteq_{FD}$ *Q* ⟹ *Q* $\sqsubseteq_{FD}$ *P* ⟹ *P* = *Q*› **by** *simp*

lemma *DT-antisym*: ‹*P* $\sqsubseteq_{DT}$ *Q* ⟹ *Q* $\sqsubseteq_{DT}$ *P* ⟹ *P* = *Q*›
apply (*simp add*: *trace-divergence-refine-def*)
oops

11.1.5 Transitivity

lemma *trans-F* : ‹*P* $\sqsubseteq_F$ *Q* ⟹ *Q* $\sqsubseteq_F$ *S* ⟹ *P* $\sqsubseteq_F$ *S*›
and *trans-D* : ‹*P* $\sqsubseteq_D$ *Q* ⟹ *Q* $\sqsubseteq_D$ *S* ⟹ *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 (*auto simp add*: *failure-refine-def divergence-refine-def trace-refine-def trace-divergence-refine-def*)

11.1.6 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*: *failure-refine-def trace-refine-def divergence-refine-def trace-divergence-refine-def less-eq-process$_{ptick}$-def*)
(*use T-F-spec* **in** *blast*)

11.1.7 More obvious refinements

lemma *leD-STOP*[*simp*] : ‹$P \sqsubseteq_D STOP$›
and *leT-STOP*[*simp*] : ‹$P \sqsubseteq_T STOP$›
and *leDT-STOP*[*simp*] : ‹$P \sqsubseteq_{DT} STOP$›
by (*simp-all add*: *refine-defs T-STOP D-STOP*)

11.1.8 Admissibility

lemma *le-F-adm* [*simp*] : ‹*adm* $(\lambda x.\ u\ x \sqsubseteq_F v\ x)$› **if** ‹*cont u*› **and** ‹*monofun v*›
proof (*unfold adm-def failure-refine-def*, *safe*)
fix *Y s X* **assume** $*$: ‹*chain Y*› ‹$\forall i.\ \mathcal{F}\ (v\ (Y\ i)) \subseteq \mathcal{F}\ (u\ (Y\ i))$› ‹$(s, X) \in \mathcal{F}\ (v\ (Lub\ Y))$›
have ‹$v\ (Y\ i) \sqsubseteq v\ (\bigsqcup i.\ Y\ i)$› **for** *i* **by** (*simp add*: $*(1)$ *is-ub-thelub monofunE* ‹*monofun v*›)
with $*(2)$ *le-approx-lemma-F* **have** ‹$\mathcal{F}\ (v\ (\bigsqcup i.\ Y\ i)) \subseteq \mathcal{F}\ (u\ (Y\ i))$› **for** *i* **by** *blast*
with $*(3)$ **show** ‹$(s, X) \in \mathcal{F}\ (u\ (Lub\ Y))$›
by (*auto simp add*: *ch2ch-cont* ‹*cont u*› ‹*chain Y*› *F-LUB limproc-is-thelub cont2contlubE*)
qed

lemma *le-T-adm* [*simp*] : ‹*adm* $(\lambda x.\ u\ x \sqsubseteq_T v\ x)$› **if** ‹*cont u*› **and** ‹*monofun v*›
proof (*unfold adm-def trace-refine-def*, *intro allI impI subsetI*)
fix *Y s* **assume** $*$: ‹*chain Y*› ‹$\forall i.\ \mathcal{T}\ (v\ (Y\ i)) \subseteq \mathcal{T}\ (u\ (Y\ i))$› ‹$s \in \mathcal{T}\ (v\ (Lub\ Y))$›
have ‹$v\ (Y\ i) \sqsubseteq v\ (\bigsqcup i.\ Y\ i)$› **for** *i* **by** (*simp add*: $*(1)$ *is-ub-thelub monofunE* ‹*monofun v*›)
with $*(2)$ *le-approx-lemma-T* **have** ‹$\mathcal{T}\ (v\ (\bigsqcup i.\ Y\ i)) \subseteq \mathcal{T}\ (u\ (Y\ i))$› **for** *i* **by** *blast*

with *∗(3)* **show** ‹$s \in \mathcal{T}\ (u\ (Lub\ Y))$›
by (*auto simp add*: *ch2ch-cont* ‹*cont u*› ‹*chain Y*› *T-LUB limproc-is-thelub cont2contlubE*)
qed

lemma *le-D-adm* [*simp*] : ‹*adm* $(\lambda x.\ u\ x \sqsubseteq_D v\ x)$› **if** ‹*cont u*› **and** ‹*monofun v*›
proof (*unfold adm-def divergence-refine-def*, *intro allI impI subsetI*)
fix *Y s* **assume** $*$: ‹*chain Y*› ‹$\forall i.\ \mathcal{D}\ (v\ (Y\ i)) \subseteq \mathcal{D}\ (u\ (Y\ i))$› ‹$s \in \mathcal{D}\ (v\ (Lub\ Y))$›
have ‹$v\ (Y\ i) \sqsubseteq v\ (\bigsqcup i.\ Y\ i)$› **for** *i* **by** (*simp add*: *∗(1) is-ub-thelub monofunE* ‹*monofun v*›)
with *∗(2) le-approx1* **have** ‹$\mathcal{D}\ (v\ (\bigsqcup i.\ Y\ i)) \subseteq \mathcal{D}\ (u\ (Y\ i))$› **for** *i* **by** *blast*
with *∗(3)* **show** ‹$s \in \mathcal{D}\ (u\ (Lub\ Y))$›
by (*auto simp add*: *ch2ch-cont* ‹*cont u*› ‹*chain Y*› *D-LUB limproc-is-thelub cont2contlubE*)
qed

declare *le-FD-adm* [*simp*]

thm *le-FD-adm le-FD-adm-cont*

lemma *le-DT-adm* [*simp*] : ‹*cont* $(u::('b::cpo) \Rightarrow ('a, 'r)\ process_{ptick}) \Longrightarrow monofun\ v \Longrightarrow adm(\lambda x.\ u\ x \sqsubseteq_{DT} v\ x)$›
using *adm-conj*[*OF le-T-adm*[*of u v*] *le-D-adm*[*of u v*]] **by** (*simp add*: *trace-divergence-refine-def*)

lemmas *le-F-adm-cont*[*simp*] = *le-F-adm*[*OF - cont2mono*]
and *le-T-adm-cont*[*simp*] = *le-T-adm*[*OF - cont2mono*]
and *le-D-adm-cont*[*simp*] = *le-D-adm*[*OF - cont2mono*]
and *le-DT-adm-cont*[*simp*] = *le-DT-adm*[*OF - cont2mono*]

11.2 Monotonies

11.2.1 Straight Monotony

Non Deterministic Choice

lemma *mono-Ndet-FD* : ‹$P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{FD} P' \sqcap Q'$›
and *mono-Ndet-DT* : ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{DT} P' \sqcap Q'$›
and *mono-Ndet-F* : ‹$P \sqsubseteq_{F} P' \Longrightarrow Q \sqsubseteq_{F} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{F} P' \sqcap Q'$›
and *mono-Ndet-D* : ‹$P \sqsubseteq_{D} P' \Longrightarrow Q \sqsubseteq_{D} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{D} P' \sqcap Q'$›
and *mono-Ndet-T* : ‹$P \sqsubseteq_{T} P' \Longrightarrow Q \sqsubseteq_{T} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{T} P' \sqcap Q'$›
by (*auto simp add*: *refine-defs Ndet-projs*)

lemmas *monos-Ndet* = *mono-Ndet mono-Ndet-FD mono-Ndet-DT*
mono-Ndet-F mono-Ndet-D mono-Ndet-T

Global Non Deterministic Choice

lemma *mono-GlobalNdet-FD* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{FD} \sqcap a \in A.\ Q\ a$›
and *mono-GlobalNdet-DT* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{DT} \sqcap a \in A.\ Q\ a$›
and *mono-GlobalNdet-F* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{F} \sqcap a \in A.\ Q\ a$›
and *mono-GlobalNdet-D* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{D} \sqcap a \in A.\ Q\ a$›
and *mono-GlobalNdet-T* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{T} \sqcap a \in A.\ Q\ a$›
by (*auto simp add*: *refine-defs GlobalNdet-projs*)

lemmas *monos-GlobalNdet = mono-GlobalNdet mono-GlobalNdet-FD mono-GlobalNdet-DT mono-GlobalNdet-F mono-GlobalNdet-D mono-GlobalNdet-T*

lemma *GlobalNdet-FD-subset* : ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{FD} (\sqcap a \in A.\ P\ a)$›
and *GlobalNdet-DT-subset* : ‹$A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{DT} (\sqcap a \in A.\ P\ a)$›
and *GlobalNdet-F-subset* : ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{F} (\sqcap a \in A.\ P\ a)$›
and *GlobalNdet-T-subset* : ‹$A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{T} (\sqcap a \in A.\ P\ a)$›
and *GlobalNdet-D-subset* : ‹$A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{D} (\sqcap a \in A.\ P\ a)$›
by (*auto simp add*: *refine-defs GlobalNdet-projs*)

lemmas *GlobalNdet-le-subset =*
GlobalNdet-FD-subset GlobalNdet-DT-subset
GlobalNdet-F-subset GlobalNdet-T-subset GlobalNdet-D-subset

Deterministic Choice

lemma *mono-Det-FD* : ‹$P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \sqsubset Q \sqsubseteq_{FD} P' \sqsubset Q'$›
proof (*rule trans-FD*)
show ‹$P \sqsubseteq_{FD} P' \Longrightarrow P \sqsubset Q \sqsubseteq_{FD} P' \sqsubset Q$›
unfolding *refine-defs F-Det D-Det* **using** *F-T T-F* **by** *blast*
next
show ‹$Q \sqsubseteq_{FD} Q' \Longrightarrow P' \sqsubset Q \sqsubseteq_{FD} P' \sqsubset Q'$›
unfolding *refine-defs F-Det D-Det* **using** *F-T T-F* **by** *blast*
qed

lemma *mono-Det-D* : ‹$P \sqsubseteq_{D} P' \Longrightarrow Q \sqsubseteq_{D} Q' \Longrightarrow P \sqsubset Q \sqsubseteq_{D} P' \sqsubset Q'$›
by (*metis D-Det Un-mono divergence-refine-def*)

lemma *mono-Det-T* : ‹$P \sqsubseteq_{T} P' \Longrightarrow Q \sqsubseteq_{T} Q' \Longrightarrow P \sqsubset Q \sqsubseteq_{T} P' \sqsubset Q'$›
by (*metis T-Det Un-mono trace-refine-def*)

corollary *mono-Det-DT* : ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \sqsubset Q \sqsubseteq_{DT} P' \sqsubset Q'$›
by (*simp-all add*: *trace-divergence-refine-def mono-Det-D mono-Det-T*)

Deterministic choice monotony doesn't hold for ($\sqsubseteq_F$).

lemmas *monos-Det* = *mono-Det mono-Det-FD mono-Det-DT mono-Det-D mono-Det-T*

Sliding choice

lemma *mono-Sliding-FD* : ‹$P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \rhd Q \sqsubseteq_{FD} P' \rhd Q'$›
and *mono-Sliding-DT* : ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \rhd Q \sqsubseteq_{DT} P' \rhd Q'$›
and *mono-Sliding-D* : ‹$P \sqsubseteq_{D} P' \Longrightarrow Q \sqsubseteq_{D} Q' \Longrightarrow P \rhd Q \sqsubseteq_{D} P' \rhd Q'$›
and *mono-Sliding-T* : ‹$P \sqsubseteq_{T} P' \Longrightarrow Q \sqsubseteq_{T} Q' \Longrightarrow P \rhd Q \sqsubseteq_{T} P' \rhd Q'$›
unfolding *Sliding-def* **by** (*simp-all add*: *monos-Det monos-Ndet*)

Sliding choice monotony doesn't hold for ($\sqsubseteq_F$).

lemma *mono-Sliding-F-right* : ‹$Q \sqsubseteq_F Q' \Longrightarrow P \rhd Q \sqsubseteq_F P \rhd Q'$›
by (*auto simp add*: *failure-refine-def F-Sliding*)

lemmas *monos-Sliding* = *mono-Sliding mono-Sliding-FD mono-Sliding-DT mono-Sliding-D mono-Sliding-T mono-Sliding-F-right*

Synchronization

lemma *mono-Sync-FD* : ‹$[\![P \sqsubseteq_{FD} P'; Q \sqsubseteq_{FD} Q']\!] \Longrightarrow P \ [\![S]\!] \ Q \sqsubseteq_{FD} P' \ [\![S]\!] \ Q'$›
for $P\ P'\ Q\ Q'$ :: ‹$('a, 'r)$ $process_{ptick}$›
proof (*rule trans-FD*[*of* - ‹$P' \ [\![S]\!] \ Q$›])
show ‹$P \ [\![S]\!] \ Q \sqsubseteq_{FD} P' \ [\![S]\!] \ Q$› **if** ‹$P \sqsubseteq_{FD} P'$› **for** $P\ P'\ Q$ :: ‹$('a, 'r)$ $process_{ptick}$›
proof (*rule failure-divergence-refine-optimizedI*)
from *le-ref1 le-ref2T* ‹$P \sqsubseteq_{FD} P'$›
show ‹$s \in \mathcal{D}\ (P' \ [\![S]\!] \ Q) \Longrightarrow s \in \mathcal{D}\ (P \ [\![S]\!] \ Q)$› **for** s
unfolding *D-Sync* **by** *blast*
next
assume *subset-div* : ‹$\mathcal{D}\ (P' \ [\![S]\!] \ Q) \subseteq \mathcal{D}\ (P \ [\![S]\!] \ Q)$›
fix $s\ Z$ **assume** ‹$(s, Z) \in \mathcal{F}\ (P' \ [\![S]\!] \ Q)$›
then consider ‹$s \in \mathcal{D}\ (P' \ [\![S]\!] \ Q)$›
| $(F)\ t\ u\ X\ Y$ **where** ‹$(t, X) \in \mathcal{F}\ P'$› ‹$(u, Y) \in \mathcal{F}\ Q$›
‹s *setinterleaves* $((t, u), range\ tick \cup ev\ `\ S)$›
‹$Z = (X \cup Y) \cap (range\ tick \cup ev\ `\ S) \cup X \cap Y$›
unfolding *F-Sync D-Sync* **by** *blast*
thus ‹$(s, Z) \in \mathcal{F}\ (P \ [\![S]\!] \ Q)$›
proof *cases*
from *subset-div D-F* **show** ‹$s \in \mathcal{D}\ (P' \ [\![S]\!] \ Q) \Longrightarrow (s, Z) \in \mathcal{F}\ (P \ [\![S]\!] \ Q)$› **by** *blast*
next
case *F*
with *le-ref2*[*OF* ‹$P \sqsubseteq_{FD} P'$›] **show** ‹$(s, Z) \in \mathcal{F}\ (P \ [\![S]\!] \ Q)$› **unfolding** *F-Sync* **by** *blast*
qed
qed
thus ‹$Q \sqsubseteq_{FD} Q' \Longrightarrow P' \ [\![S]\!] \ Q \sqsubseteq_{FD} P' \ [\![S]\!] \ Q'$› **by** (*metis Sync-commute*)
qed

lemma *mono-Sync-DT* : ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \llbracket S \rrbracket Q \sqsubseteq_{DT} P' \llbracket S \rrbracket Q'$›
by (*simp add*: *refine-defs T-Sync D-Sync*) *blast*

lemmas *mono-Par* = *mono-Sync*[**where** *A* = *UNIV*]
and *mono-Par-FD* = *mono-Sync-FD*[**where** *S* = *UNIV*]
and *mono-Par-DT* = *mono-Sync-DT*[**where** *S* = *UNIV*]
and *mono-Inter* = *mono-Sync*[**where** *A* = ‹{}›]
and *mono-Inter-FD* = *mono-Sync-FD*[**where** *S* = ‹{}›]
and *mono-Inter-DT* = *mono-Sync-DT*[**where** *S* = ‹{}›]

lemmas *monos-Sync* = *mono-Sync mono-Sync-FD mono-Sync-DT*
and *monos-Par* = *mono-Par mono-Par-FD mono-Par-DT*
and *monos-Inter* = *mono-Inter mono-Inter-FD mono-Inter-DT*

Sequential composition

lemma *mono-Seq-FD* : ‹$P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P ; Q \sqsubseteq_{FD} P' ; Q'$›
by (*simp add*: *refine-defs Seq-projs subset-iff*) (*metis T-F-spec*)

lemma *mono-Seq-DT* : ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P ; Q \sqsubseteq_{DT} P' ; Q'$›
by (*simp add*: *refine-defs Seq-projs subset-iff*) *blast*

lemma *mono-Seq-F-right* : ‹$Q \sqsubseteq_F Q' \Longrightarrow P ; Q \sqsubseteq_F P ; Q'$›
by (*simp add*: *failure-refine-def F-Seq*) *blast*

lemma *mono-Seq-D-right* : ‹$Q \sqsubseteq_D Q' \Longrightarrow P ; Q \sqsubseteq_D P ; Q'$›
by (*simp add*: *divergence-refine-def D-Seq*) *blast*

lemma *mono-Seq-T-right* : ‹$Q \sqsubseteq_T Q' \Longrightarrow P ; Q \sqsubseteq_T P ; Q'$›
by (*simp add*: *trace-refine-def T-Seq*) *blast*

Left Sequence monotony doesn't hold for ($\sqsubseteq_F$), ($\sqsubseteq_D$) and ($\sqsubseteq_T$).

lemmas *monos-Seq* = *mono-Seq mono-Seq-FD mono-Seq-DT*
mono-Seq-F-right mono-Seq-D-right mono-Seq-T-right

Renaming

lemma *mono-Renaming-D* : ‹$P \sqsubseteq_D Q \Longrightarrow$ *Renaming P f g* $\sqsubseteq_D$ *Renaming Q f g*›
unfolding *divergence-refine-def D-Renaming* **by** *blast*

lemma *mono-Renaming-FD* : ‹$P \sqsubseteq_{FD} Q \Longrightarrow$ *Renaming P f g* $\sqsubseteq_{FD}$ *Renaming Q f g*›
using *mono-Renaming-D* **unfolding** *refine-defs F-Renaming D-Renaming* **by** *blast*

lemma *mono-Renaming-DT* : ‹$P \sqsubseteq_{DT} Q \Longrightarrow$ *Renaming* $P\ f\ g \sqsubseteq_{DT}$ *Renaming* $Q\ f\ g$›
using *mono-Renaming-D* **unfolding** *refine-defs T-Renaming D-Renaming* **by** *blast*

lemmas *monos-Renaming* = *mono-Renaming mono-Renaming-FD mono-Renaming-DT mono-Renaming-D*

Hiding

lemma *mono-Hiding-leT-imp-leD* : ‹$P \setminus A \sqsubseteq_D Q \setminus A$› **if** ‹$A \neq \{\}$› **and** ‹$P \sqsubseteq_T Q$›
proof (*unfold divergence-refine-def*, *rule subsetI*)
fix s **assume** ‹$s \in \mathcal{D}\ (Q \setminus A)$›
then obtain $t\ u$
where $*$: ‹*front-tickFree* u› ‹*tickFree* t›
‹$s =$ *trace-hide* t (ev ‘ A) @ u›
‹$t \in \mathcal{D}\ Q \vee (\exists x.$ *isInfHidden-seqRun* $x\ Q\ A\ t)$›
unfolding *D-Hiding-seqRun* **by** *blast*
from $*(4)$ **have** ‹$\exists x.$ *isInfHidden-seqRun* $x\ P\ A\ t$›
proof (*elim disjE exE*)
assume ‹$t \in \mathcal{D}\ Q$›
from ‹$A \neq \{\}$› **obtain** a **where** ‹$a \in A$› **by** *blast*
have ‹*isInfHidden-seqRun* ($\lambda i.$ $ev\ a$) $P\ A\ t$›
proof (*intro allI conjI*)
have ‹*seqRun* t ($\lambda i.$ $ev\ a$) $i \in \mathcal{D}\ Q$› **for** i
by (*induct i*) (*simp-all add*: ‹$t \in \mathcal{D}\ Q$› *is-processT7 tickFree-seqRun-iff* $*(2)$)
thus ‹*seqRun* t ($\lambda i.$ $ev\ a$) $i \in \mathcal{T}\ P$› **for** i
by (*meson D-T* ‹$P \sqsubseteq_T Q$› *trace-refine-def subset-iff*)
next
from ‹$a \in A$› **show** ‹$ev\ a \in ev$ ‘ A› **by** *simp*
qed
thus ‹$\exists x.$ *isInfHidden-seqRun* $x\ P\ A\ t$› **by** *blast*
next
show ‹*isInfHidden-seqRun* $x\ Q\ A\ t \Longrightarrow \exists x.$ *isInfHidden-seqRun* $x\ P\ A\ t$› **for** x
by (*meson* ‹$P \sqsubseteq_T Q$› *trace-refine-def subset-iff*)
qed
with $*(1, 2, 3)$ **show** ‹$s \in \mathcal{D}\ (P \setminus A)$› **unfolding** *D-Hiding-seqRun* **by** *blast*
qed

lemma *mono-Hiding-F* : ‹$P \setminus A \sqsubseteq_F Q \setminus A$› **if** ‹$P \sqsubseteq_F Q$›
proof (*cases* ‹$A = \{\}$›)
from ‹$P \sqsubseteq_F Q$› **show** ‹$A = \{\} \Longrightarrow P \setminus A \sqsubseteq_F Q \setminus A$›
by (*auto simp add*: *failure-refine-def F-Hiding-seqRun subset-iff*
intro: *is-processT7 is-processT8*)
next
show ‹$P \setminus A \sqsubseteq_F Q \setminus A$› **if** ‹$A \neq \{\}$›
proof (*unfold failure-refine-def*, *safe*)

```
    fix s X assume ‹(s, X) ∈ ℱ (Q \ A)›
    then consider ‹s ∈ 𝒟 (Q \ A)›
      | t where ‹s = trace-hide t (ev ‘ A)› ‹(t, X ∪ ev ‘ A) ∈ ℱ Q›
      unfolding F-Hiding-seqRun D-Hiding-seqRun by blast
    thus ‹(s, X) ∈ ℱ (P \ A)›
    proof cases
      fix t assume ‹s = trace-hide t (ev ‘ A)› ‹(t, X ∪ ev ‘ A) ∈ ℱ Q›
      from ‹P ⊑F Q› ‹(t, X ∪ ev ‘ A) ∈ ℱ Q› have ‹(t, X ∪ ev ‘ A) ∈ ℱ P›
        unfolding failure-refine-def by blast
      with ‹s = trace-hide t (ev ‘ A)› show ‹(s, X) ∈ ℱ (P \ A)›
        unfolding F-Hiding by blast
    next
     from mono-Hiding-leT-imp-leD[OF ‹A ≠ {}› ‹P ⊑F Q›[THEN leF-imp-leT]]
      show ‹s ∈ 𝒟 (Q \ A) ⟹ (s, X) ∈ ℱ (P \ A)›
        by (simp add: divergence-refine-def in-mono is-processT8)
    qed
  qed
qed

lemma mono-Hiding-T : ‹P \ A ⊑T Q \ A› if ‹P ⊑T Q›
proof (cases ‹A = {}›)
  from ‹P ⊑T Q› show ‹A = {} ⟹ P \ A ⊑T Q \ A›
    by (auto simp add: trace-refine-def T-Hiding-seqRun
        subset-iff F-T T-F D-T is-processT7)
next
  show ‹P \ A ⊑T Q \ A› if ‹A ≠ {}›
  proof (unfold trace-refine-def, rule subsetI)
    fix s assume ‹s ∈ 𝒯 (Q \ A)›
    then consider ‹s ∈ 𝒟 (Q \ A)›
      | t where ‹s = trace-hide t (ev ‘ A)› ‹(t, ev ‘ A) ∈ ℱ Q›
      unfolding T-Hiding-seqRun D-Hiding-seqRun by blast
    thus ‹s ∈ 𝒯 (P \ A)›
    proof cases
      fix t assume ‹s = trace-hide t (ev ‘ A)› ‹(t, ev ‘ A) ∈ ℱ Q›
      from ‹(t, ev ‘ A) ∈ ℱ Q› have ‹t ∈ 𝒯 P›
        by (meson F-T ‹P ⊑T Q› in-mono trace-refine-def)
      with inf-hidden consider
       t' where ‹trace-hide t' (ev ‘ A) = trace-hide t (ev ‘ A)› ‹(t', ev ‘ A) ∈ ℱ P›
      | f where ‹isInfHiddenRun f P A ∧ t ∈ range f› by blast
      thus ‹s ∈ 𝒯 (P \ A)›
      proof cases
        show ‹trace-hide t' (ev ‘ A) = trace-hide t (ev ‘ A) ⟹
              (t', ev ‘ A) ∈ ℱ P ⟹ s ∈ 𝒯 (P \ A)› for t'
          by (simp add: T-Hiding-seqRun) (metis ‹s = trace-hide t (ev ‘ A)›)
      next
        show ‹isInfHiddenRun f P A ∧ t ∈ range f ⟹ s ∈ 𝒯 (P \ A)› for f
          by (simp add: T-Hiding)
            (metis T-nonTickFree-imp-decomp ‹s = trace-hide t (ev ‘ A)›
              ‹t ∈ 𝒯 P› append-self-conv front-tickFree-Nil tick-T-F)
```

```
    qed
   next
    from mono-Hiding-leT-imp-leD[OF ‹A ≠ {}› ‹P ⊑_T Q›]
    show ‹s ∈ 𝒟 (Q \ A) ⟹ s ∈ 𝒯 (P \ A)›
     by (simp add: divergence-refine-def in-mono D-T)
   qed
  qed
qed

lemma mono-Hiding-FD : ‹P \ A ⊑_FD Q \ A› if ‹P ⊑_FD Q›
proof (cases ‹A = {}›)
  from ‹P ⊑_FD Q› show ‹A = {} ⟹ P \ A ⊑_FD Q \ A›
   by (simp add: refine-defs F-Hiding-seqRun D-Hiding-seqRun) blast
next
  show ‹A ≠ {} ⟹ P \ A ⊑_FD Q \ A›
   by (unfold failure-divergence-refine-def)
   (simp add: ‹P ⊑_FD Q› leFD-imp-leF mono-Hiding-F leF-imp-leT mono-Hiding-leT-imp-leD)
qed

lemma mono-Hiding-DT : ‹P \ A ⊑_DT Q \ A› if ‹P ⊑_DT Q›
proof (cases ‹A = {}›)
  from ‹P ⊑_DT Q› show ‹A = {} ⟹ P \ A ⊑_DT Q \ A›
    by (auto simp add: refine-defs T-Hiding-seqRun D-Hiding-seqRun F-T T-F
in-mono)
next
  show ‹A ≠ {} ⟹ P \ A ⊑_DT Q \ A›
   by (unfold trace-divergence-refine-def)
   (simp add: ‹P ⊑_DT Q› leDT-imp-leT mono-Hiding-T mono-Hiding-leT-imp-leD)
qed
```

Obviously, Hide monotony doesn't hold for $(\sqsubseteq_D)$.

```
lemmas monos-Hiding = mono-Hiding mono-Hiding-FD mono-Hiding-DT
  mono-Hiding-F mono-Hiding-T
```

Prefixes

```
lemma mono-Mprefix-FD : ‹(⋀a. a ∈ A ⟹ P a ⊑_FD Q a) ⟹ Mprefix A P
⊑_FD Mprefix A Q›
 and mono-Mprefix-DT : ‹(⋀a. a ∈ A ⟹ P a ⊑_DT Q a) ⟹ Mprefix A P ⊑_DT
Mprefix A Q›
  and mono-Mprefix-F  : ‹(⋀a. a ∈ A ⟹ P a ⊑_F Q a) ⟹ Mprefix A P ⊑_F
Mprefix A Q›
  and mono-Mprefix-T  : ‹(⋀a. a ∈ A ⟹ P a ⊑_T Q a) ⟹ Mprefix A P ⊑_T
Mprefix A Q›
  and mono-Mprefix-D  : ‹(⋀a. a ∈ A ⟹ P a ⊑_D Q a) ⟹ Mprefix A P ⊑_D
Mprefix A Q›
  by (simp add: refine-defs Mprefix-projs, blast)+

lemmas monos-Mprefix = mono-Mprefix mono-Mprefix-FD mono-Mprefix-DT
```

mono-Mprefix-F mono-Mprefix-T mono-Mprefix-D

corollary *mono-write0-FD* : ‹$P \sqsubseteq_{FD} Q \Longrightarrow (a \rightarrow P) \sqsubseteq_{FD} (a \rightarrow Q)$›
and *mono-write0-DT* : ‹$P \sqsubseteq_{DT} Q \Longrightarrow (a \rightarrow P) \sqsubseteq_{DT} (a \rightarrow Q)$›
and *mono-write0-F* : ‹$P \sqsubseteq_{F} Q \Longrightarrow (a \rightarrow P) \sqsubseteq_{F} (a \rightarrow Q)$›
and *mono-write0-T* : ‹$P \sqsubseteq_{T} Q \Longrightarrow (a \rightarrow P) \sqsubseteq_{T} (a \rightarrow Q)$›
and *mono-write0-D* : ‹$P \sqsubseteq_{D} Q \Longrightarrow (a \rightarrow P) \sqsubseteq_{D} (a \rightarrow Q)$›
unfolding *write0-def* **by** (*simp-all add*: *monos-Mprefix*)

lemmas *monos-write0* = *mono-write0 mono-write0-FD mono-write0-DT mono-write0-F mono-write0-T mono-write0-D*

corollary *mono-write-FD* : ‹$P \sqsubseteq_{FD} Q \Longrightarrow (c!a \rightarrow P) \sqsubseteq_{FD} (c!a \rightarrow Q)$›
and *mono-write-DT* : ‹$P \sqsubseteq_{DT} Q \Longrightarrow (c!a \rightarrow P) \sqsubseteq_{DT} (c!a \rightarrow Q)$›
and *mono-write-F* : ‹$P \sqsubseteq_{F} Q \Longrightarrow (c!a \rightarrow P) \sqsubseteq_{F} (c!a \rightarrow Q)$›
and *mono-write-T* : ‹$P \sqsubseteq_{T} Q \Longrightarrow (c!a \rightarrow P) \sqsubseteq_{T} (c!a \rightarrow Q)$›
and *mono-write-D* : ‹$P \sqsubseteq_{D} Q \Longrightarrow (c!a \rightarrow P) \sqsubseteq_{D} (c!a \rightarrow Q)$›
by (*simp-all add*: *write-is-write0 monos-write0*)

lemmas *monos-write* = *mono-write mono-write-FD mono-write-DT mono-write-F mono-write-T mono-write-D*

corollary *mono-read-FD* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD} Q\ a) \Longrightarrow (c?a{\in}A \rightarrow P\ a) \sqsubseteq_{FD} (c?a{\in}A \rightarrow Q\ a)$›
and *mono-read-DT* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT} Q\ a) \Longrightarrow (c?a{\in}A \rightarrow P\ a) \sqsubseteq_{DT} (c?a{\in}A \rightarrow Q\ a)$›
and *mono-read-F* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F} Q\ a) \Longrightarrow (c?a{\in}A \rightarrow P\ a) \sqsubseteq_{F} (c?a{\in}A \rightarrow Q\ a)$›
and *mono-read-T* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T} Q\ a) \Longrightarrow (c?a{\in}A \rightarrow P\ a) \sqsubseteq_{T} (c?a{\in}A \rightarrow Q\ a)$›
and *mono-read-D* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D} Q\ a) \Longrightarrow (c?a{\in}A \rightarrow P\ a) \sqsubseteq_{D} (c?a{\in}A \rightarrow Q\ a)$›
unfolding *read-def* **by** (*simp-all add*: *monos-Mprefix inv-into-into*)

lemmas *monos-read* = *mono-read mono-read-FD mono-read-DT mono-read-F mono-read-T mono-read-D*

lemma *mono-Mndetprefix-FD* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD} Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{FD}$ *Mndetprefix A Q*›
and *mono-Mndetprefix-DT* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT} Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{DT}$ *Mndetprefix A Q*›
and *mono-Mndetprefix-F* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F} Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{F}$ *Mndetprefix A Q*›
and *mono-Mndetprefix-T* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T} Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{T}$ *Mndetprefix A Q*›
and *mono-Mndetprefix-D* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D} Q\ a) \Longrightarrow$ *Mndetprefix A*

$P \sqsubseteq_D$ *Mndetprefix A Q*›
by (*simp add*: *refine-defs Mndetprefix-projs*, *blast*)+

lemmas *monos-Mndetprefix* = *mono-Mndetprefix mono-Mndetprefix-FD mono-Mndetprefix-DT mono-Mndetprefix-F mono-Mndetprefix-T mono-Mndetprefix-D*

corollary *mono-ndet-write-FD* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD} Q\ a) \Longrightarrow (c!!a{\in}A \rightarrow P\ a) \sqsubseteq_{FD} (c!!a{\in}A \rightarrow Q\ a)$›
and *mono-ndet-write-DT* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT} Q\ a) \Longrightarrow (c!!a{\in}A \rightarrow P\ a) \sqsubseteq_{DT} (c!!a{\in}A \rightarrow Q\ a)$›
and *mono-ndet-write-F* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F} Q\ a) \Longrightarrow (c!!a{\in}A \rightarrow P\ a) \sqsubseteq_{F} (c!!a{\in}A \rightarrow Q\ a)$›
and *mono-ndet-write-T* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T} Q\ a) \Longrightarrow (c!!a{\in}A \rightarrow P\ a) \sqsubseteq_{T} (c!!a{\in}A \rightarrow Q\ a)$›
and *mono-ndet-write-D* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D} Q\ a) \Longrightarrow (c!!a{\in}A \rightarrow P\ a) \sqsubseteq_{D} (c!!a{\in}A \rightarrow Q\ a)$›
unfolding *ndet-write-def* **by** (*simp-all add*: *monos-Mndetprefix inv-into-into*)

lemmas *monos-ndet-write* = *mono-ndet-write mono-ndet-write-FD mono-ndet-write-DT mono-ndet-write-F mono-ndet-write-T mono-ndet-write-D*

lemma *Mndetprefix-FD-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{FD}$ *Mprefix A P*›
and *Mndetprefix-DT-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{DT}$ *Mprefix A P*›
and *Mndetprefix-F-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{F}$ *Mprefix A P*›
and *Mndetprefix-T-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{T}$ *Mprefix A P*›
and *Mndetprefix-D-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{D}$ *Mprefix A P*›
by (*simp-all add*: *refine-defs Mprefix-projs Mndetprefix-projs*) *blast*+

lemmas *Mndetprefix-le-Mprefix* =
Mndetprefix-FD-Mprefix Mndetprefix-DT-Mprefix
Mndetprefix-F-Mprefix Mndetprefix-T-Mprefix Mndetprefix-D-Mprefix

corollary *ndet-write-FD-read* : ‹*ndet-write c A P* $\sqsubseteq_{FD}$ *read c A P*›
and *ndet-write-DT-read* : ‹*ndet-write c A P* $\sqsubseteq_{DT}$ *read c A P*›
and *ndet-write-F-read* : ‹*ndet-write c A P* $\sqsubseteq_{F}$ *read c A P*›
and *ndet-write-T-read* : ‹*ndet-write c A P* $\sqsubseteq_{T}$ *read c A P*›
and *ndet-write-D-read* : ‹*ndet-write c A P* $\sqsubseteq_{D}$ *read c A P*›
by (*simp-all add*: *read-def ndet-write-def Mndetprefix-le-Mprefix*)

lemmas *ndet-write-le-read* =
ndet-write-FD-read ndet-write-DT-read
ndet-write-F-read ndet-write-T-read ndet-write-D-read

lemma *Mndetprefix-FD-subset* : ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_{FD}$ *Mndetprefix A P*›

and *Mndetprefix-DT-subset* : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix* $B\ P \sqsubseteq_{DT}$ *Mndetprefix* $A\ P$›
and *Mndetprefix-F-subset* : ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow$ *Mndetprefix* $B\ P \sqsubseteq_{F}$ *Mndetprefix* $A\ P$›
and *Mndetprefix-T-subset* : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix* $B\ P \sqsubseteq_{T}$ *Mndetprefix* $A\ P$›
and *Mndetprefix-D-subset* : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix* $B\ P \sqsubseteq_{D}$ *Mndetprefix* $A\ P$›
by (*auto simp add*: *refine-defs Mndetprefix-projs*)

lemmas *Mndetprefix-le-subset* =
Mndetprefix-FD-subset Mndetprefix-DT-subset
Mndetprefix-F-subset Mndetprefix-T-subset Mndetprefix-D-subset

lemma *ndet-write-FD-subset* : ‹$A \neq \{\} \Longrightarrow (c!!b{\in}B \to P\ b) \sqsubseteq_{FD} (c!!a{\in}A \to P\ a)$›
and *ndet-write-DT-subset* : ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{DT} (c!!a{\in}A \to P\ a)$›
and *ndet-write-F-subset* : ‹$A \neq \{\} \Longrightarrow (c!!b{\in}B \to P\ b) \sqsubseteq_{F} (c!!a{\in}A \to P\ a)$›
and *ndet-write-T-subset* : ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{T} (c!!a{\in}A \to P\ a)$›
and *ndet-write-D-subset* : ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{D} (c!!a{\in}A \to P\ a)$›
if ‹*inj-on c B*› **and** ‹$A \subseteq B$›
proof −
have ∗ : ‹$(P \circ$ *inv-into* $B\ c)\ x = (P \circ$ *inv-into* $A\ c)\ x$› **if** ‹$x \in c\ `\ A$› **for** x
proof −
from ‹$x \in c\ `\ A$› **obtain** a **where** ‹$a \in A$› ‹$x = c\ a$› **by** *blast*
from ‹$a \in A$› ‹$A \subseteq B$› **have** ‹$a \in B$› **by** *blast*
from ‹$A \subseteq B$› ‹*inj-on c B*› *inj-on-subset* **have** ‹*inj-on c A*› **by** *blast*
from ‹$a \in A$› ‹$x = c\ a$› ‹*inj-on c A*› **have** ‹(*inv-into A c*) $x = a$› **by** *simp*
moreover from ‹$a \in B$› ‹$x = c\ a$› ‹*inj-on c B*› **have** ‹(*inv-into B c*) $x = a$› **by** *simp*
ultimately show ‹$(P \circ$ *inv-into* $B\ c)\ x = (P \circ$ *inv-into* $A\ c)\ x$› **by** *simp*
qed
show ‹$A \neq \{\} \Longrightarrow (c!!b{\in}B \to P\ b) \sqsubseteq_{FD} (c!!a{\in}A \to P\ a)$›
by (*metis* ‹$A \subseteq B$› ∗ *Mndetprefix-FD-subset empty-is-image*
image-mono mono-Mndetprefix-eq ndet-write-def)
show ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{DT} (c!!a{\in}A \to P\ a)$›
by (*metis* ‹$A \subseteq B$› ∗ *Mndetprefix-DT-subset*
image-mono mono-Mndetprefix-eq ndet-write-def)
show ‹$A \neq \{\} \Longrightarrow (c!!b{\in}B \to P\ b) \sqsubseteq_{F} (c!!a{\in}A \to P\ a)$›
by (*metis* ‹$A \subseteq B$› ∗ *Mndetprefix-F-subset empty-is-image*
image-mono mono-Mndetprefix-eq ndet-write-def)
show ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{T} (c!!a{\in}A \to P\ a)$›
by (*metis* ‹$A \subseteq B$› ∗ *Mndetprefix-T-subset*
image-mono mono-Mndetprefix-eq ndet-write-def)
show ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{D} (c!!a{\in}A \to P\ a)$›
by (*metis* ‹$A \subseteq B$› ∗ *Mndetprefix-D-subset*
image-mono mono-Mndetprefix-eq ndet-write-def)
qed

lemmas *ndet-write-le-subset* =

ndet-write-FD-subset ndet-write-DT-subset
ndet-write-F-subset ndet-write-T-subset ndet-write-D-subset

lemma *Mndetprefix-FD-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{FD}$ (*a* → *P a*)›
and *Mndetprefix-DT-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{DT}$ (*a* → *P a*)›
and *Mndetprefix-F-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{F}$ (*a* → *P a*)›
and *Mndetprefix-T-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{T}$ (*a* → *P a*)›
and *Mndetprefix-D-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{D}$ (*a* → *P a*)› **if** ‹*a* ∈ *A*›
by (*rule Mndetprefix-le-subset*[*of* ‹{*a*}›, *simplified*, *OF* ‹*a* ∈ *A*›])+

lemmas *Mndetprefix-le-write0* =
Mndetprefix-FD-write0 Mndetprefix-DT-write0
Mndetprefix-F-write0 Mndetprefix-T-write0 Mndetprefix-D-write0

lemma *Mprefix-DT-subset* : ‹*A* ⊆ *B* ⟹ *Mprefix B P* $\sqsubseteq_{DT}$ *Mprefix A P*›
and *Mprefix-T-subset* : ‹*A* ⊆ *B* ⟹ *Mprefix B P* $\sqsubseteq_{T}$ *Mprefix A P*›
and *Mprefix-D-subset* : ‹*A* ⊆ *B* ⟹ *Mprefix B P* $\sqsubseteq_{D}$ *Mprefix A P*›
by (*auto simp add*: *refine-defs Mprefix-projs*)

lemmas *Mprefix-le-subset* = *Mprefix-DT-subset Mprefix-T-subset Mprefix-D-subset*

Mprefix set monotony doesn't hold for ($\sqsubseteq_F$) and ($\sqsubseteq_{FD}$) where it holds for deterministic choice.

lemma *read-DT-subset* : ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{DT}$ (*c*?*a*∈*A* → *P a*)›
and *read-T-subset* : ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{T}$ (*c*?*a*∈*A* → *P a*)›
and *read-D-subset* : ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{D}$ (*c*?*a*∈*A* → *P a*)›
if ‹*inj-on c B*› **and** ‹*A* ⊆ *B*›
proof −
have ∗ : ‹(*P* ∘ *inv-into B c*) *x* = (*P* ∘ *inv-into A c*) *x*› **if** ‹*x* ∈ *c* ‘ *A*› **for** *x*
proof −
from ‹*x* ∈ *c* ‘ *A*› **obtain** *a* **where** ‹*a* ∈ *A*› ‹*x* = *c a*› **by** *blast*
from ‹*a* ∈ *A*› ‹*A* ⊆ *B*› **have** ‹*a* ∈ *B*› **by** *blast*
from ‹*A* ⊆ *B*› ‹*inj-on c B*› *inj-on-subset* **have** ‹*inj-on c A*› **by** *blast*
from ‹*a* ∈ *A*› ‹*x* = *c a*› ‹*inj-on c A*› **have** ‹(*inv-into A c*) *x* = *a*› **by** *simp*
moreover from ‹*a* ∈ *B*› ‹*x* = *c a*› ‹*inj-on c B*› **have** ‹(*inv-into B c*) *x* = *a*›
by *simp*
ultimately show ‹(*P* ∘ *inv-into B c*) *x* = (*P* ∘ *inv-into A c*) *x*› **by** *simp*
qed
show ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{DT}$ (*c*?*a*∈*A* → *P a*)›
by (*metis* ‹*A* ⊆ *B*› ∗ *Mprefix-DT-subset image-mono mono-Mprefix-eq read-def*)
show ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{T}$ (*c*?*a*∈*A* → *P a*)›
by (*metis* ‹*A* ⊆ *B*› ∗ *Mprefix-T-subset image-mono mono-Mprefix-eq read-def*)
show ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{D}$ (*c*?*a*∈*A* → *P a*)›
by (*metis* ‹*A* ⊆ *B*› ∗ *Mprefix-D-subset image-mono mono-Mprefix-eq read-def*)
qed

11.2.2 Monotony Properties

Non Deterministic Choice Operator Laws

lemma *Ndet-FD-self-left* : $\langle P \sqcap Q \sqsubseteq_{FD} P \rangle$
and *Ndet-DT-self-left* : $\langle P \sqcap Q \sqsubseteq_{DT} P \rangle$
and *Ndet-F-self-left* : $\langle P \sqcap Q \sqsubseteq_{F} P \rangle$
and *Ndet-T-self-left* : $\langle P \sqcap Q \sqsubseteq_{T} P \rangle$
and *Ndet-D-self-left* : $\langle P \sqcap Q \sqsubseteq_{D} P \rangle$
and *Ndet-FD-self-right* : $\langle P \sqcap Q \sqsubseteq_{FD} Q \rangle$
and *Ndet-DT-self-right* : $\langle P \sqcap Q \sqsubseteq_{DT} Q \rangle$
and *Ndet-F-self-right* : $\langle P \sqcap Q \sqsubseteq_{F} Q \rangle$
and *Ndet-T-self-right* : $\langle P \sqcap Q \sqsubseteq_{T} Q \rangle$
and *Ndet-D-self-right* : $\langle P \sqcap Q \sqsubseteq_{D} Q \rangle$
by (*simp-all add*: *refine-defs Ndet-projs*)

lemmas *Ndet-le-self-left* = *Ndet-FD-self-left Ndet-DT-self-left*
Ndet-F-self-left Ndet-T-self-left Ndet-D-self-left
and *Ndet-le-self-right* = *Ndet-FD-self-right Ndet-DT-self-right*
Ndet-F-self-right Ndet-T-self-right Ndet-D-self-right

lemma *Ndet-FD-Det* : $\langle P \sqcap Q \sqsubseteq_{FD} P \,\Box\, Q \rangle$
and *Ndet-DT-Det* : $\langle P \sqcap Q \sqsubseteq_{DT} P \,\Box\, Q \rangle$
and *Ndet-F-Det* : $\langle P \sqcap Q \sqsubseteq_{F} P \,\Box\, Q \rangle$
and *Ndet-T-Det* : $\langle P \sqcap Q \sqsubseteq_{T} P \,\Box\, Q \rangle$
and *Ndet-D-Det* : $\langle P \sqcap Q \sqsubseteq_{D} P \,\Box\, Q \rangle$
by (*auto simp add*: *refine-defs Det-projs Ndet-projs*
intro: *is-processT8 is-processT6-TR-notin*)

lemmas *Ndet-le-Det* = *Ndet-FD-Det Ndet-DT-Det Ndet-F-Det Ndet-T-Det Ndet-D-Det*

lemma *Ndet-FD-Sliding* : $\langle P \sqcap Q \sqsubseteq_{FD} P \rhd Q \rangle$
and *Ndet-DT-Sliding* : $\langle P \sqcap Q \sqsubseteq_{DT} P \rhd Q \rangle$
and *Ndet-F-Sliding* : $\langle P \sqcap Q \sqsubseteq_{F} P \rhd Q \rangle$
and *Ndet-T-Sliding* : $\langle P \sqcap Q \sqsubseteq_{T} P \rhd Q \rangle$
and *Ndet-D-Sliding* : $\langle P \sqcap Q \sqsubseteq_{D} P \rhd Q \rangle$
by (*auto simp add*: *refine-defs Ndet-projs Sliding-projs*
intro: *is-processT8 is-processT6-TR-notin*)

lemmas *Ndet-le-Sliding* = *Ndet-FD-Sliding Ndet-DT-Sliding*
Ndet-F-Sliding Ndet-T-Sliding Ndet-D-Sliding

lemma *GlobalNdet-refine-FD* : $\langle a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_{FD} P\ a \rangle$
using *GlobalNdet-FD-subset*[*of* $\langle \{a\} \rangle$ *A*] **by** *simp*

lemma *GlobalNdet-refine-DT* : $\langle a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_{DT} P\ a \rangle$
using *GlobalNdet-DT-subset*[*of* $\langle \{a\} \rangle$ *A*] **by** *simp*

lemma *GlobalNdet-refine-F* : ‹$a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_F P\ a$›
by (*simp add*: *GlobalNdet-refine-FD leFD-imp-leF*)

lemma *GlobalNdet-refine-D* : ‹$a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_D P\ a$›
by (*simp add*: *GlobalNdet-refine-DT leDT-imp-leD*)

lemma *GlobalNdet-refine-T* : ‹$a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_T P\ a$›
by (*simp add*: *GlobalNdet-refine-DT leDT-imp-leT*)

lemmas *GlobalNdet-refine-le* = *GlobalNdet-refine-FD GlobalNdet-refine-DT GlobalNdet-refine-F GlobalNdet-refine-D GlobalNdet-refine-T*

lemma *mono-GlobalNdet-FD-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_{FD} Q\ a) \Longrightarrow P \sqsubseteq_{FD} \sqcap a \in A.\ Q\ a$›
by (*metis GlobalNdet-id mono-GlobalNdet-FD*)

lemma *mono-GlobalNdet-DT-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_{DT} Q\ a) \Longrightarrow P \sqsubseteq_{DT} \sqcap a \in A.\ Q\ a$›
by (*metis GlobalNdet-id mono-GlobalNdet-DT*)

lemma *mono-GlobalNdet-F-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_F Q\ a) \Longrightarrow P \sqsubseteq_F \sqcap a \in A.\ Q\ a$›
by (*metis GlobalNdet-id mono-GlobalNdet-F*)

lemma *mono-GlobalNdet-D-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_D Q\ a) \Longrightarrow P \sqsubseteq_D \sqcap a \in A.\ Q\ a$›
by (*metis GlobalNdet-id mono-GlobalNdet-D*)

lemma *mono-GlobalNdet-T-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_T Q\ a) \Longrightarrow P \sqsubseteq_T \sqcap a \in A.\ Q\ a$›
by (*metis GlobalNdet-id mono-GlobalNdet-T*)

lemmas *mono-GlobalNdet-le-const* = *mono-GlobalNdet-FD-const mono-GlobalNdet-DT-const mono-GlobalNdet-F-const mono-GlobalNdet-D-const mono-GlobalNdet-T-const*

Refinements and Constant Processes

lemma *STOP-T-iff*: ‹$STOP \sqsubseteq_T P \longleftrightarrow P = STOP$›
by (*metis STOP-iff-T insert-absorb is-processT1-TR subset-singleton-iff trace-refine-def*)

lemma *STOP-F-iff*: ‹$STOP \sqsubseteq_F P \longleftrightarrow P = STOP$›
using *STOP-T-iff idem-F leF-imp-leT* **by** *blast*

lemma *STOP-FD-iff*: ‹$STOP \sqsubseteq_{FD} P \longleftrightarrow P = STOP$›
using *STOP-F-iff leFD-imp-leF* **by** *blast*

lemma *STOP-DT-iff*: ‹$STOP \sqsubseteq_{DT} P \longleftrightarrow P = STOP$›
using *STOP-T-iff idem-DT leDT-imp-leT* **by** *blast*

lemma *SKIP-FD-iff*: ‹*SKIP r* $\sqsubseteq_{FD}$ *P* $\longleftrightarrow$ *P* = *SKIP r*› **for** *P* :: ‹('*a*, '*r*) *process*$_{ptick}$›
proof (*rule iffI*)
show ‹*P* = *SKIP r* $\Longrightarrow$ *SKIP r* $\sqsubseteq_{FD}$ *P*› **by** *simp*
next
show ‹*P* = *SKIP r*› **if** ‹*SKIP r* $\sqsubseteq_{FD}$ *P*›
proof (*rule FD-antisym*[*OF* - ‹*SKIP r* $\sqsubseteq_{FD}$ *P*›])
show ‹*P* $\sqsubseteq_{FD}$ *SKIP r*›
proof (*rule failure-divergence-refineI*)
from ‹*SKIP r* $\sqsubseteq_{FD}$ *P*› **show** ‹*s* $\in$ $\mathcal{D}$ (*SKIP r*) $\Longrightarrow$ *s* $\in$ $\mathcal{D}$ *P*› **for** *s*
by (*simp add*: *refine-defs D-SKIP*)
next
fix *s* :: ‹('*a*, '*r*) *trace*$_{ptick}$› **and** *X* **assume** ‹(*s*, *X*) $\in$ $\mathcal{F}$ (*SKIP r*)›
then consider ‹*s* = []› ‹✓(*r*) $\notin$ *X*› | ‹*s* = [✓(*r*)]› **unfolding** *F-SKIP* **by** *blast*
thus ‹(*s*, *X*) $\in$ $\mathcal{F}$ *P*›
proof *cases*
from ‹*SKIP r* $\sqsubseteq_{FD}$ *P*› **show** ‹*s* = [] $\Longrightarrow$ ✓(*r*) $\notin$ *X* $\Longrightarrow$ (*s*, *X*) $\in$ $\mathcal{F}$ *P*›
by (*simp add*: *refine-defs F-SKIP D-SKIP subset-iff*)
(*metis T-F append-Nil is-processT1-TR is-processT5-S7 is-processT6-TR-notin not-Cons-self2*)
next
from ‹*SKIP r* $\sqsubseteq_{FD}$ *P*› **show** ‹*s* = [✓(*r*)] $\Longrightarrow$ (*s*, *X*) $\in$ $\mathcal{F}$ *P*›
by (*simp add*: *refine-defs F-SKIP D-SKIP subset-iff*)
(*metis* (*no-types*) *Int-insert-right-if0 is-processT1 list.simps*(15) *non-tickFree-tick tickFree-Nil tickFree-def tick-T-F trace-tick-continuation-or-all-tick-failuresE*)
qed
qed
qed
qed

lemma *SKIP-F-iff*: ‹*SKIP r* $\sqsubseteq_{F}$ *P* $\longleftrightarrow$ *P* = *SKIP r*›
proof (*rule iffI*)
show ‹*P* = *SKIP r* $\Longrightarrow$ *SKIP r* $\sqsubseteq_{F}$ *P*› **by** *simp*
next
assume ‹*SKIP r* $\sqsubseteq_{F}$ *P*›
hence ‹*SKIP r* $\sqsubseteq_{D}$ *P*›
by (*simp add*: *failure-refine-def divergence-refine-def subset-iff F-SKIP D-SKIP*)
(*metis append-Nil equals0I insertI1 is-processT8 is-processT9 list.distinct*(1))
with ‹*SKIP r* $\sqsubseteq_{F}$ *P*› **have** ‹*SKIP r* $\sqsubseteq_{FD}$ *P*› **by** (*unfold failure-divergence-refine-def*) *simp*
thus ‹*P* = *SKIP r*› **by** (*fact SKIP-FD-iff*[*THEN iffD1*])
qed

Chapter 12

Algebraic Rules of CSP

12.1 The Non-Deterministic Distributivity

CSP operators are distributive over non deterministic choice.

12.1.1 Global Distributivity

lemma *Mndetprefix-distrib-GlobalNdet* :
‹$B \neq \{\} \Longrightarrow (\sqcap a \in A \rightarrow \sqcap b \in B.\ P\ a\ b) = \sqcap b \in B.\ \sqcap a \in A \rightarrow P\ a\ b$›
by (*simp add*: *Mndetprefix-GlobalNdet GlobalNdet-sets-commute*[*of A*] *write0-GlobalNdet*)

lemma *Det-distrib-GlobalNdet-left* :
‹$P \ \Box\ (\sqcap a{\in}A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P\ \mathit{else}\ \sqcap a{\in}A.\ P \ \Box\ Q\ a)$›
by (*auto simp add*: *Process-eq-spec Det-projs GlobalNdet-projs*
intro: *is-processT8 is-processT6-TR-notin*)

corollary *Det-distrib-GlobalNdet-right* :
‹$(\sqcap a{\in}A.\ P\ a) \ \Box\ Q = (\mathit{if}\ A = \{\}\ \mathit{then}\ Q\ \mathit{else}\ \sqcap a{\in}A.\ P\ a \ \Box\ Q)$›
by (*simp add*: *Det-commute Det-distrib-GlobalNdet-left*)

lemma *Sliding-distrib-GlobalNdet-left* :
‹$P \rhd (\sqcap a{\in}A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P \sqcap \mathit{STOP}\ \mathit{else}\ \sqcap a{\in}A.\ P \rhd Q\ a)$›
by (*auto simp add*: *Process-eq-spec GlobalNdet-projs*
Sliding-projs Ndet-projs STOP-projs)

lemma *Sliding-distrib-GlobalNdet-right* :
‹$(\sqcap a{\in}A.\ P\ a) \rhd Q = (\mathit{if}\ A = \{\}\ \mathit{then}\ Q\ \mathit{else}\ \sqcap a{\in}A.\ P\ a \rhd Q)$›
by (*auto simp add*: *Process-eq-spec GlobalNdet-projs Sliding-projs*)

lemma *Sync-distrib-GlobalNdet-left* :
‹$P \ [\![S]\!]\ (\sqcap\ a{\in}A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P\ [\![S]\!]\ \mathit{STOP}\ \mathit{else} \sqcap\ a{\in}A.\ (P\ [\![S]\!]\ Q\ a))$›
proof (*split if-split*, *intro conjI impI*)
show ‹$A = \{\} \Longrightarrow P\ [\![S]\!]\ (\sqcap\ a{\in}A.\ Q\ a) = P\ [\![S]\!]\ \mathit{STOP}$›

by (*simp add*: *GlobalNdet.abs-eq STOP.abs-eq*)
next
show ‹$A \neq \{\} \Longrightarrow P \llbracket S \rrbracket (\sqcap a{\in}A.\ Q\ a) = \sqcap a{\in}A.\ (P \llbracket S \rrbracket Q\ a)$›
by (*simp add*: *Process-eq-spec Sync-projs F-GlobalNdet D-GlobalNdet T-GlobalNdet*)
(*safe*; *simp*; *use front-tickFree-Nil* **in** *blast*) — quicker than auto proof
qed

corollary *Sync-distrib-GlobalNdet-right* :
‹$(\sqcap a{\in}A.\ P\ a) \llbracket S \rrbracket Q = (\mathit{if}\ A = \{\}\ \mathit{then}\ \mathit{STOP} \llbracket S \rrbracket Q\ \mathit{else} \sqcap a{\in}A.\ (P\ a \llbracket S \rrbracket Q))$›
by (*simp add*: *Sync-commute Sync-distrib-GlobalNdet-left*)

lemmas *Inter-distrib-GlobalNdet-left* = *Sync-distrib-GlobalNdet-left*[**where** $S = \langle\{\}\rangle$]
and *Par-distrib-GlobalNdet-left* = *Sync-distrib-GlobalNdet-left*[**where** $S = UNIV$]
and *Inter-distrib-GlobalNdet-right* = *Sync-distrib-GlobalNdet-right*[**where** $S = \langle\{\}\rangle$]
and *Par-distrib-GlobalNdet-right* = *Sync-distrib-GlobalNdet-right*[**where** $S = UNIV$]

lemma *Seq-distrib-GlobalNdet-left* :
‹$P \mathbin{;} \sqcap a{\in}A.\ Q\ a = (\mathit{if}\ A = \{\}\ \mathit{then}\ P \mathbin{;} \mathit{STOP}\ \mathit{else} \sqcap a{\in}A.\ (P \mathbin{;} Q\ a))$›
by (*simp add*: *Process-eq-spec GlobalNdet-projs STOP-projs Seq-projs*) *blast*

lemma *Seq-distrib-GlobalNdet-right* : ‹$(\sqcap a{\in}A.\ P\ a) \mathbin{;} Q = \sqcap a{\in}A.\ (P\ a \mathbin{;} Q)$›
by (*simp add*: *Process-eq-spec GlobalNdet-projs STOP-projs Seq-projs*)
(*safe*; *simp*; *blast*) — quicker than auto proof

lemma *Renaming-distrib-GlobalNdet* : ‹*Renaming* $(\sqcap a{\in}A.\ P\ a)\ f\ g = \sqcap a{\in}A.$ *Renaming* $(P\ a)\ f\ g$›
by (*simp add*: *Process-eq-spec Renaming-projs GlobalNdet-projs*)
(*safe*; *simp*; *blast*) — quicker than auto proof

The $(\backslash)$ operator does not distribute the *GlobalNdet* in general. We give a finite version derived from the binary version below.

12.1.2 Binary Distributivity

lemma *Mndetprefix-distrib-Ndet* :
‹$(\sqcap a \in A \rightarrow P\ a \sqcap Q\ a) = (\sqcap a \in A \rightarrow P\ a) \sqcap (\sqcap a \in A \rightarrow Q\ a)$›
by (*simp add*: *Process-eq-spec Mndetprefix-projs Ndet-projs*, *safe*) *auto*

lemma *Det-distrib-Ndet-left* : ‹$P \mathbin{\Box} Q \sqcap R = (P \mathbin{\Box} Q) \sqcap (P \mathbin{\Box} R)$›
by (*auto simp add*: *Process-eq-spec Det-projs Ndet-projs*)

corollary *Det-distrib-Ndet-right* : ‹$P \sqcap Q \mathbin{\Box} R = (P \mathbin{\Box} R) \sqcap (Q \mathbin{\Box} R)$›
by (*simp add*: *Det-commute Det-distrib-Ndet-left*)

lemma *Ndet-distrib-Det-left* : ‹$P \sqcap (Q \mathbin{\square} R) = P \sqcap Q \mathbin{\square} P \sqcap R$›
by (*auto simp add*: *Process-eq-spec Det-projs Ndet-projs is-processT8 is-processT6-TR-notin*)

corollary *Ndet-distrib-Det-right* : ‹$(P \mathbin{\square} Q) \sqcap R = P \sqcap R \mathbin{\square} Q \sqcap R$›
by (*simp add*: *Ndet-commute Ndet-distrib-Det-left*)

lemma *Sliding-distrib-Ndet-left* : ‹$P \rhd (Q \sqcap R) = (P \rhd Q) \sqcap (P \rhd R)$›
and *Sliding-distrib-Ndet-right* : ‹$(P \sqcap Q) \rhd R = (P \rhd R) \sqcap (Q \rhd R)$›
by (*auto simp add*: *Process-eq-spec Ndet-projs Sliding-projs*)

lemma *Sync-distrib-Ndet-left* : ‹$M \llbracket S \rrbracket P \sqcap Q = (M \llbracket S \rrbracket P) \sqcap (M \llbracket S \rrbracket Q)$›
by (*auto simp*: *Process-eq-spec*, *simp-all add*: *Sync-projs Ndet-projs*) *blast*+

corollary *Sync-distrib-Ndet-right* : ‹$P \sqcap Q \llbracket S \rrbracket M = (P \llbracket S \rrbracket M) \sqcap (Q \llbracket S \rrbracket M)$›
by (*metis Sync-commute Sync-distrib-Ndet-left*)

lemma *Seq-distrib-Ndet-left* : ‹$P \mathbin{;} Q \sqcap R = (P \mathbin{;} Q) \sqcap (P \mathbin{;} R)$›
by (*fact Seq-distrib-GlobalNdet-left*[*of* P ‹$\{0 :: nat, 1\}$› ‹$\lambda n.$ *if* $n = 0$ *then* Q *else if* $n = 1$ *then* R *else undefined*›, *simplified GlobalNdet-distrib-unit-bis*, *simplified*])

lemma *Seq-distrib-Ndet-right* : ‹$P \sqcap Q \mathbin{;} R = (P \mathbin{;} R) \sqcap (Q \mathbin{;} R)$›
by (*fact Seq-distrib-GlobalNdet-right*[*of* ‹$\{0 :: nat, 1\}$› ‹$\lambda n.$ *if* $n = 0$ *then* P *else if* $n = 1$ *then* Q *else undefined*› R, *simplified GlobalNdet-distrib-unit-bis*, *simplified*])

lemma *Renaming-distrib-Ndet* : ‹*Renaming* $(P \sqcap Q)$ f g = *Renaming* P f $g \sqcap$ *Renaming* Q f g›
by (*fact Renaming-distrib-GlobalNdet*[*of* ‹$\{0 :: nat, 1\}$› ‹$\lambda n.$ *if* $n = 0$ *then* P *else if* $n = 1$ *then* Q *else undefined*›, *simplified GlobalNdet-distrib-unit-bis*, *simplified*])

lemma *Hiding-distrib-Ndet* : ‹$P \sqcap Q \setminus S = (P \setminus S) \sqcap (Q \setminus S)$›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix s **assume** ‹$s \in \mathcal{D}\ (P \sqcap Q \setminus S)$›
then obtain t u
where $*$: ‹*front-tickFree* u› ‹*tickFree* t› ‹s = *trace-hide* t (*ev* ‘ S) @ u›
‹$t \in \mathcal{D}\ (P \sqcap Q) \vee (\exists f.$ *isInfHiddenRun* f $(P \sqcap Q)$ $S \wedge t \in$ *range* $f)$›
unfolding *D-Hiding* **by** *blast*

```
  from ∗(4) show ‹s ∈ 𝒟 ((P \ S) ⊓ (Q \ S))›
  proof (elim disjE exE)
    from ∗(1, 2) show ‹t ∈ 𝒟 (P ⊓ Q) ⟹ s ∈ 𝒟 ((P \ S) ⊓ (Q \ S))›
      unfolding D-Ndet D-Hiding ∗(3) by blast
  next
    fix f assume ‹isInfHiddenRun f (P ⊓ Q) S ∧ t ∈ range f›
    hence ‹isInfHiddenRun f P S ∨ isInfHiddenRun f Q S›
    by (simp add: T-Ndet) (meson is-processT3-TR linorder-le-cases strict-mono-less-eq)
    with ∗(1, 2) ‹isInfHiddenRun f (P ⊓ Q) S ∧ t ∈ range f›
    show ‹s ∈ 𝒟 ((P \ S) ⊓ (Q \ S))› unfolding D-Ndet D-Hiding ∗(3) by blast
  qed
next
  show ‹s ∈ 𝒟 ((P \ S) ⊓ (Q \ S)) ⟹ s ∈ 𝒟 (P ⊓ Q \ S)› for s
    unfolding Ndet-projs D-Hiding by blast
next
  assume same-div : ‹𝒟 (P ⊓ Q \ S) = 𝒟 ((P \ S) ⊓ (Q \ S))›
  fix s X assume ‹(s, X) ∈ ℱ (P ⊓ Q \ S)›
  then consider ‹s ∈ 𝒟 (P ⊓ Q \ S)›
    | t where ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (P ⊓ Q)›
    by (simp add: F-Hiding D-Hiding) blast
  thus ‹(s, X) ∈ ℱ ((P \ S) ⊓ (Q \ S))›
  proof cases
    from same-div D-F show ‹s ∈ 𝒟 (P ⊓ Q \ S) ⟹ (s, X) ∈ ℱ ((P \ S) ⊓ (Q
\ S))› by blast
  next
    show ‹s = trace-hide t (ev ‘ S) ⟹ (t, X ∪ ev ‘ S) ∈ ℱ (P ⊓ Q)
          ⟹ (s, X) ∈ ℱ ((P \ S) ⊓ (Q \ S))› for t
      by (auto simp add: F-Ndet F-Hiding)
  qed
next
  show ‹(s, X) ∈ ℱ ((P \ S) ⊓ (Q \ S)) ⟹ (s, X) ∈ ℱ (P ⊓ Q \ S)› for s X
    unfolding Ndet-projs F-Hiding by blast
qed


lemma Hiding-distrib-finite-GlobalNdet :
  ‹finite A ⟹ (⊓a ∈ A. P a) \ S = ⊓a ∈ A. (P a \ S)›
proof (induct A rule: finite-induct)
  show ‹(⊓a ∈ {}. P a) \ S = ⊓a ∈ {}. (P a \ S)›
    by (auto simp add: Process-eq-spec GlobalNdet-projs
        Hiding-seqRun-projs seqRun-def)
next
  fix A a assume ‹finite A› ‹a ∉ A› and hyp : ‹(⊓a ∈ A. P a) \ S = ⊓a ∈ A. (P
a \ S)›
  show ‹(⊓a ∈ insert a A. P a) \ S = ⊓a ∈ insert a A. (P a \ S)› (is ‹?lhs =
?rhs›)
  proof (cases ‹A = {}›)
    show ‹A = {} ⟹ ?lhs = ?rhs› by simp
  next
```

```
    assume ‹A ≠ {}›
    have ‹?lhs = P a ⊓ (⊓a ∈ A. P a) \ S›
      by (simp add: GlobalNdet-distrib-unit ‹A ≠ {}› ‹a ∉ A›)
     also have ‹... = (P a \ S) ⊓ ((⊓a ∈ A. P a) \ S)› by (simp add: Hid-
ing-distrib-Ndet)
    also have ‹... = (P a \ S) ⊓ (⊓a ∈ A. (P a \ S))› by (simp add: hyp)
    also have ‹... = ?rhs› by (simp add: GlobalNdet-distrib-unit-bis ‹A ≠ {}› ‹a
∉ A›)
    finally show ‹?lhs = ?rhs› .
  qed
qed
```

For the *Mprefix* operator, we obviously do not have a conventional distributivity: $(\sqcap)$ becomes $(\Box)$.

```
lemma Mprefix-distrib-Ndet :
  ‹(□a ∈ A → P ⊓ Q) = (□a ∈ A → P) □ (□a ∈ A → Q)› (is ‹?lhs = ?rhs›)
proof (subst Process-eq-spec, safe)
  show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs›
    and ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› for s
    by (auto simp add: D-Det D-Ndet D-Mprefix)
next
  show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› for s X
    by (cases s) (auto simp add: F-Mprefix F-Det F-Ndet)
next
  show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› for s X
    by (auto simp add: F-Mprefix F-Det F-Ndet)
qed
```

12.2 The Basic Laws

By "basic" laws we mean the behaviour of $\bot$, *STOP* and *SKIP*, plus the associativity of some concerned operators.

12.2.1 The Laws of $\bot$

From the characterization $(?P = \bot) = ([] \in \mathcal{D}\ ?P)$, we can easily derive the behaviour of $\bot$ wrt. *STOP*, *SKIP*, and the operators.

```
lemma BOT-less1 [simp]: ‹(⊥ :: ('a, 'r) process_ptick) ≤ X›
  by (simp add: le-approx-imp-le-ref)

lemma STOP-neq-BOT [simp] : ‹STOP   ≠ ⊥›
  and SKIP-neq-BOT [simp] : ‹SKIP r ≠ ⊥›
  by (simp-all add: BOT-iff-Nil-D D-STOP D-SKIP)
```

lemma *Det-is-BOT-iff* : ‹$P \mathbin{\Box} Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
by (*simp add*: *BOT-iff-Nil-D D-Det D-BOT*)

lemma *Det-BOT* [*simp*] : ‹$P \mathbin{\Box} \bot = \bot$› **and** *BOT-Det* [*simp*] : ‹$\bot \mathbin{\Box} P = \bot$›
by (*simp-all add*: *Det-is-BOT-iff*)

lemma *Ndet-is-BOT-iff* : ‹$P \sqinter Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
by (*simp add*: *BOT-iff-Nil-D D-Ndet D-BOT*)

lemma *Ndet-BOT* [*simp*] : ‹$P \sqinter \bot = \bot$› **and** *BOT-Ndet* [*simp*] : ‹$\bot \sqinter P = \bot$›
by (*simp-all add*: *Ndet-is-BOT-iff*)

lemma *Sliding-is-BOT-iff*: ‹$P \rhd Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
by (*simp add*: *Sliding-def Ndet-is-BOT-iff Det-is-BOT-iff*)

lemma *Sliding-BOT* [*simp*] : ‹$P \rhd \bot = \bot$› **and** *BOT-Sliding* [*simp*] : ‹$\bot \rhd P = \bot$›
by (*simp-all add*: *Sliding-is-BOT-iff*)

lemma *Mprefix-neq-BOT* [*simp*] : ‹$\Box\ a \in A \rightarrow P\ a \neq \bot$›
by (*simp add*: *BOT-iff-Nil-D D-Mprefix*)

lemma *Mndetprefix-neq-BOT* [*simp*] : ‹$\sqcap\ a \in A \rightarrow P\ a \neq \bot$›
by (*simp add*: *BOT-iff-Nil-D D-Mndetprefix write0-def D-Mprefix*)

lemma *Seq-is-BOT-iff* : ‹$P \mathbin{;} Q = \bot \longleftrightarrow P = \bot \vee (\exists\, r.\ [\checkmark(r)] \in \mathcal{T}\ P \wedge Q = \bot)$›
by (*simp add*: *BOT-iff-Nil-D D-Seq*)

lemma *BOT-Seq* [*simp*] : ‹$\bot \mathbin{;} P = \bot$› **by** (*simp add*: *Seq-is-BOT-iff*)

lemma *Hiding-BOT* [*simp*] : ‹$\bot \setminus A = \bot$›
by (*simp add*: *BOT-iff-Nil-D D-Hiding BOT-projs*)
(*meson filter.simps*(*1*) *tickFree-Nil tickFree-imp-front-tickFree*)

lemma *Sync-is-BOT-iff* : ‹$P \llbracket S \rrbracket Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
by (*simp add*: *BOT-iff-Nil-D D-Sync*)
(*metis si-empty1 empty-setinterleaving insertI1 is-processT1-TR*)

lemma *Sync-BOT* [*simp*] : ‹$P \llbracket S \rrbracket \bot = \bot$› **and** *BOT-Sync* [*simp*] : ‹$\bot \llbracket S \rrbracket P = \bot$›
by (*simp-all add*: *Sync-is-BOT-iff*)

lemma *Renaming-is-BOT-iff* : ‹*Renaming P f g* = ⊥ ⟷ *P* = ⊥›
by (*simp add*: *BOT-iff-Nil-D D-Renaming*)

lemma *Renaming-BOT* [*simp*] : ‹*Renaming* ⊥ *f g* = ⊥›
by (*simp add*: *Renaming-is-BOT-iff*)

lemma *GlobalNdet-is-BOT-iff* : ‹(⊓*a*∈*A*. *P a*) = ⊥ ⟷ (∃ *a*∈*A*. *P a* = ⊥)›
by (*simp add*: *BOT-iff-Nil-D D-GlobalNdet*)

lemma *GlobalNdet-BOT* [*simp*] : ‹*a* ∈ *A* ⟹ *P a* = ⊥ ⟹ (⊓*a*∈*A*. *P a*) = ⊥›
by (*auto simp add*: *GlobalNdet-is-BOT-iff*)

12.2.2 The Laws of *STOP*

From the characterization (*?P* = *STOP*) = (*T* *?P* = {[]}), we can easily derive the behaviour of *STOP* wrt. *SKIP* and the operators.

lemma *SKIP-Neq-STOP* : ‹*SKIP r* ≠ *STOP*›
by (*simp add*: *STOP-iff-T T-SKIP*)

lemma *Det-is-STOP-iff* : ‹*P* □ *Q* = *STOP* ⟷ *P* = *STOP* ∧ *Q* = *STOP*›
by (*simp add*: *STOP-iff-T T-Det*) (*use Nil-elem-T* **in** *blast*)

lemma *Det-STOP* [*simp*] : ‹*P* □ *STOP* = *P*› **and** *STOP-Det* [*simp*] : ‹*STOP* □ *P* = *P*›
by (*auto simp add*: *Process-eq-spec F-Det F-STOP D-Det D-STOP T-STOP intro*: *is-processT8 is-processT6-TR-notin*)

lemma *Ndet-is-STOP-iff* : ‹*P* ⊓ *Q* = *STOP* ⟷ *P* = *STOP* ∧ *Q* = *STOP*›
by (*simp add*: *STOP-iff-T T-Ndet*) (*use Nil-elem-T* **in** *blast*)

lemma *Sliding-is-STOP-iff*: ‹*P* ▷ *Q* = *STOP* ⟷ *P* = *STOP* ∧ *Q* = *STOP*›
by (*simp add*: *Sliding-def Ndet-is-STOP-iff Det-is-STOP-iff*)

lemma *Sliding-STOP* [*simp*] : ‹*P* ▷ *STOP* = *P* ⊓ *STOP*› **and** *STOP-Sliding* [*simp*] : ‹*STOP* ▷ *P* = *P*›
by (*simp-all add*: *Sliding-def*)

lemma *Mprefix-is-STOP-iff* : ‹□ *a* ∈ *A* → *P a* = *STOP* ⟷ *A* = {}›
by (*simp add*: *STOP-iff-T T-Mprefix*) (*use Nil-elem-T* **in** *blast*)

lemma *Mprefix-empty* [*simp*] : ‹□ *a* ∈ {} → *P a* = *STOP*›
by (*simp add*: *Mprefix-is-STOP-iff*)

lemma *Mndetprefix-is-STOP-iff* : ‹⊓ *a* ∈ *A* → *P a* = *STOP* ⟷ *A* = {}›

by (*simp add*: *STOP-iff-T T-Mndetprefix'*) (*use Nil-elem-T* **in** *blast*)

lemma *Mndetprefix-empty* [*simp*] : ‹$\sqcap\ a \in \{\} \rightarrow P\ a = STOP$›
by (*simp add*: *Mndetprefix-is-STOP-iff*)

lemma *Seq-is-STOP-iff* :
‹$P\ ;\ Q = STOP \longleftrightarrow \mathcal{T}\ P \subseteq insert\ []\ \{[✓(r)]|\ r.\ True\} \wedge (P \neq STOP \longrightarrow Q = STOP)$›
proof (*intro iffI conjI subsetI*)
show ‹$P\ ;\ Q = STOP \Longrightarrow t \in \mathcal{T}\ P \Longrightarrow t \in insert\ []\ \{[✓(r)]\ |r.\ True\}$› **for** t
by (*simp add*: *STOP-iff-T T-Seq set-eq-iff*)
(*metis T-nonTickFree-imp-decomp append.right-neutral append-Nil is-processT1-TR*)
next
show ‹$P\ ;\ Q = STOP \Longrightarrow P \neq STOP \longrightarrow Q = STOP$›
by (*simp add*: *STOP-iff-T T-Seq set-eq-iff*)
(*metis T-nonTickFree-imp-decomp append-is-Nil-conv is-processT1-TR*)
next
show ‹$\mathcal{T}\ P \subseteq insert\ []\ \{[✓(r)]\ |r.\ True\} \wedge (P \neq STOP \longrightarrow Q = STOP) \Longrightarrow P\ ;\ Q = STOP$›
by (*simp add*: *STOP-iff-T T-Seq subset-iff*, *safe*, *simp-all*)
(((*use D-T* **in** *fastforce*)+)[*3*],
metis D-T $event_{ptick}$*.distinct*(*1*) *front-tickFree-single is-processT9-tick list.sel*(*1*) *not-Cons-self*)
qed

lemma *STOP-Seq* [*simp*] : ‹$STOP\ ;\ P = STOP$› **by** (*simp add*: *Seq-is-STOP-iff T-STOP*)

lemma *Hiding-STOP* [*simp*] : ‹$STOP \setminus A = STOP$›
by (*auto simp add*: *STOP-iff-T T-Hiding F-STOP D-STOP T-STOP strict-mono-def*) *blast*

lemma *STOP-Sync-STOP* [*simp*] : ‹$STOP\ [\![S]\!]\ STOP = STOP$›
by (*simp add*: *STOP-iff-T T-Sync T-STOP D-STOP*)

lemma *Renaming-is-STOP-iff* : ‹$Renaming\ P\ f\ g = STOP \longleftrightarrow P = STOP$›
proof (*rule iffI*)
show ‹$P = STOP \Longrightarrow Renaming\ P\ f\ g = STOP$›
by (*simp add*: *STOP-iff-T T-Renaming subset-iff*)
(*metis D-STOP STOP-iff-T empty-iff*)
next
assume ‹$Renaming\ P\ f\ g = STOP$›
hence ‹$P \neq \bot$› **by** *force*
with ‹$Renaming\ P\ f\ g = STOP$› **show** ‹$P = STOP$›
by (*simp add*: *BOT-iff-Nil-D STOP-iff-T T-Renaming set-eq-iff*) *blast*

qed

lemma *Renaming-STOP* [*simp*] : ‹*Renaming STOP f g* = *STOP*›
by (*simp add*: *Renaming-is-STOP-iff*)

lemma *GlobalNdet-is-STOP-iff* : ‹(⊓*a*∈*A*. *P a*) = *STOP* ⟷ (∀ *a*∈*A*. *P a* = *STOP*)›
by (*simp add*: *STOP-iff-T T-GlobalNdet*) (*use Nil-elem-T* **in** *blast*)

lemma *GlobalNdet-empty* [*simp*] : ‹(⊓*a*∈{}. *P a*) = *STOP*›
by (*simp add*: *GlobalNdet-is-STOP-iff*)

lemma *GlobalNdet-id* [*simp*] : ‹(⊓*a*∈*A*. *P*) = (*if A* = {} *then STOP else P*)›
by *simp*

lemma ‹(⊓*a*∈*A*. *STOP*) = *STOP*› **by** *simp*

More powerful Laws

lemma *Inter-STOP* [*simp*] : ‹*P* ||| *STOP* = *P* ; *STOP*›
proof (*rule Process-eq-optimizedI*)
show ‹*t* ∈ 𝒟 (*P* ||| *STOP*) ⟹ *t* ∈ 𝒟 (*P* ; *STOP*)› **for** *t*
by (*simp add*: *D-Sync D-Seq D-STOP T-STOP*)
(*metis emptyRightProperty is-processT7 self-append-conv*)
next
show ‹*t* ∈ 𝒟 (*P* ; *STOP*) ⟹ *t* ∈ 𝒟 (*P* ||| *STOP*)› **for** *t*
apply (*simp add*: *D-Sync D-Seq D-STOP T-STOP*)
apply (*frule D-imp-front-tickFree*)
apply (*rule exI*[*of* - ‹*if tF t then t else butlast t*›],
rule exI[*of* - ‹*if tF t then t else butlast t*›])
apply (*simp split*: *if-split-asm*)
by (*metis butlast-snoc disjoint-iff emptyLeftSelf front-tickFree-iff-tickFree-butlast front-tickFree-single is-processT9 nonTickFree-n-frontTickFree setinterleaving-sym tickFree-def*)
next
fix *t X* **assume** ‹(*t*, *X*) ∈ ℱ (*P* ||| *STOP*)› ‹*t* ∉ 𝒟 (*P* ||| *STOP*)›
then obtain *t-P X-P X-Q* **where** ‹(*t-P*, *X-P*) ∈ ℱ *P*› ‹*t setinterleaves* ((*t-P*, []), *range tick*)›
‹*X* = (*X-P* ∪ *X-Q*) ∩ *range tick* ∪ *X-P* ∩ *X-Q*›
by (*simp add*: *F-Sync D-Sync F-STOP*) *blast*
from ‹*t setinterleaves* ((*t-P*, []), *range tick*)› *EmptyLeftSync*
have ‹*tF t* ∧ *t* = *t-P*› **by** (*metis inf-commute setinterleaving-sym tickFree-def*)
show ‹(*t*, *X*) ∈ ℱ (*P* ; *STOP*)›
proof (*cases* ‹∃ *r*. *t-P* @ [✓(*r*)] ∈ 𝒯 *P*›)
show ‹∃ *r*. *t-P* @ [✓(*r*)] ∈ 𝒯 *P* ⟹ (*t*, *X*) ∈ ℱ (*P* ; *STOP*)›
using ‹*tF t* ∧ *t* = *t-P*› **by** (*simp add*: *F-Seq F-STOP*)
next
assume ‹∄ *r*. *t-P* @ [✓(*r*)] ∈ 𝒯 *P*›

```
    with ‹(t-P, X-P) ∈ ℱ P› have ‹(t-P, X-P ∪ range tick) ∈ ℱ P›
      by (metis is-processT5-S7' rangeE)
    moreover from ‹X = (X-P ∪ X-Q) ∩ range tick ∪ X-P ∩ X-Q›
    have ‹X ∪ range tick ⊆ X-P ∪ range tick› by auto
    ultimately have ‹(t-P, X ∪ range tick) ∈ ℱ P› by (fact is-processT4)
     with ‹tF t ∧ t = t-P› show ‹(t, X) ∈ ℱ (P ; STOP)› by (auto simp add:
F-Seq)
  qed
next
  fix t X assume ‹(t, X) ∈ ℱ (P ; STOP)› ‹t ∉ 𝒟 (P ; STOP)›
  then consider ‹(t, X ∪ range tick) ∈ ℱ P› ‹tF t› | r where ‹t @ [✓(r)] ∈ 𝒯
P›
    by (simp add: Seq-projs F-STOP) blast
  thus ‹(t, X) ∈ ℱ (P ||| STOP)›
  proof cases
    assume ‹(t, X ∪ range tick) ∈ ℱ P› and ‹tF t›
    from ‹tF t› have ‹t setinterleaves ((t, []), range tick)›
      by (metis disjoint-iff emptyLeftSelf setinterleaving-sym tickFree-def)
    with ‹(t, X ∪ range tick) ∈ ℱ P› show ‹(t, X) ∈ ℱ (P ||| STOP)›
      apply (simp add: F-Sync F-STOP)
      by (metis Int-Un-eq(1) Un-Int-eq(3) is-processT4 sup-ge1)
  next
    fix r assume ‹t @ [✓(r)] ∈ 𝒯 P›
    hence * : ‹(t, X - {✓(r)}) ∈ ℱ P› and ‹tF t›
      by (auto simp add: T-F-spec is-processT6 intro: append-T-imp-tickFree)
    from ‹tF t› have ‹t setinterleaves ((t, []), range tick)›
      by (metis disjoint-iff emptyLeftSelf setinterleaving-sym tickFree-def)
    hence ‹(t, (X - {✓(r)} ∪ UNIV) ∩ range tick ∪ (X - {✓(r)}) ∩ UNIV) ∈
ℱ (P ||| STOP)›
      by (simp add: F-Sync F-STOP) (use * in blast)
   moreover have ‹X ⊆ (X - {✓(r)} ∪ UNIV) ∩ range tick ∪ (X - {✓(r)}) ∩
UNIV› by auto
    ultimately show ‹(t, X) ∈ ℱ (P ||| STOP)› by (fact is-processT4)
  qed
qed

corollary STOP-Inter [simp] : ‹STOP ||| P = P ; STOP›
  by (metis Inter-STOP Sync-commute)


lemma Par-STOP [simp] : ‹P || STOP = (if P = ⊥ then ⊥ else STOP)›
proof (split if-split, intro conjI impI)
  show ‹P = ⊥ ⟹ P || STOP = ⊥› by simp
next
  assume ‹P ≠ ⊥›
  from ‹P ≠ ⊥› show ‹P || STOP = STOP›
    by (simp add: STOP-iff-T T-Sync STOP-projs set-eq-iff BOT-iff-Nil-D)
      (metis EmptyLeftSync si-empty1 inf-top.right-neutral insertI1
        is-processT1-TR set-empty2 setinterleaving-sym)
```

qed

lemma *STOP-Par* [*simp*] : ‹*STOP* || *P* = (*if P* = ⊥ *then* ⊥ *else STOP*)›
by (*metis Par-STOP Sync-commute*)

12.2.3 The Laws of *SKIP*

The definition of SKIPS

definition *SKIPS* :: ‹'*r set* ⇒ ('*a*, '*r*) *process*$_{ptick}$›
where ‹*SKIPS R* ≡ ⊓*r* ∈ *R*. *SKIP r*›

lemma *SKIPS-empty* [*simp*] : ‹*SKIPS* {} = *STOP*›
and *SKIPS-singl-is-SKIP* [*simp*] : ‹*SKIPS* {*r*} = *SKIP r*› **by** (*simp-all add*: *SKIPS-def*)

lemma *F-SKIPS* :
‹*F* (*SKIPS R*) = (*if R* = {} *then* {(*s*, *X*). *s* = []}
else {([], *X*) |*X*. ∃ *r* ∈ *R*. *tick r* ∉ *X*} ∪
{(*s*, *X*). ∃ *r* ∈ *R*. *s* = [*tick r*]})›
by (*auto simp add*: *SKIPS-def F-GlobalNdet F-SKIP*)

lemma *D-SKIPS* : ‹*D* (*SKIPS R*) = {}›
by (*auto simp add*: *SKIPS-def D-GlobalNdet D-SKIP*)

lemma *T-SKIPS* : ‹*T* (*SKIPS R*) = *insert* [] {[*tick r*]| *r*. *r* ∈ *R*}›
by (*auto simp add*: *SKIPS-def T-GlobalNdet T-SKIP*)

lemmas *SKIPS-projs* = *F-SKIPS D-SKIPS T-SKIPS*

Laws

lemma *SKIP-Sliding* [*simp*] : ‹*SKIP r* ▷ *P* = *P* ⊓ *SKIP r*›
by (*auto simp add*: *Process-eq-spec F-Sliding D-Sliding F-Ndet D-Ndet SKIP-projs*)

lemma *Sliding-SKIP* [*simp*] : ‹*P* ▷ *SKIP r* = *P* □ *SKIP r*›
by (*auto simp add*: *Process-eq-spec F-Sliding D-Sliding F-Det D-Det SKIP-projs*)

lemma *Mprefix-neq-SKIP* [*simp*] : ‹□ *a* ∈ *A* → *P a* ≠ *SKIP r*›
by (*auto simp add*: *Process-eq-spec F-Mprefix F-SKIP*)

lemma *Mndetprefix-neq-SKIP* [*simp*] : ‹⊓ *a* ∈ *A* → *P a* ≠ *SKIP r*›
by (*auto simp add*: *Process-eq-spec F-Mndetprefix write0-def F-Mprefix F-SKIP*)

lemma *SKIP-Seq* [*simp*] : ‹*SKIP r* ; *P* = *P*›
by (*simp add*: *Process-eq-spec Seq-projs SKIP-projs*)

lemma *Seq-SKIP* [*simp*] : ‹*P* **;** *SKIP* () = *P*›
— This is only true when the type *'r* of (*'a*, *'r*) $process_{ptick}$ is set to *unit* (morally, we have "only one *tick*").
by (*auto simp add*: *Process-eq-spec Seq-projs SKIP-projs tick-T-F intro*: *is-processT4 is-processT6-TR-notin is-processT8*)
(*metis* (*full-types*) *append.right-neutral is-processT5-S7' old.unit.exhaust rangeE*,
metis (*full-types*) *F-T T-imp-front-tickFree nonTickFree-n-frontTickFree old.unit.exhaust*)

lemma *Hiding-SKIP* [*simp*] : ‹*SKIP r* \ *A* = *SKIP r*›
proof (*subst Process-eq-spec*, *safe*)
show ‹*s* ∈ $\mathcal{D}$ (*SKIP r* \ *A*) ⟹ *s* ∈ $\mathcal{D}$ (*SKIP r*)› **for** *s*
by (*auto simp add*: *D-Hiding D-SKIP T-SKIP*)
(*metis* (*full-types*) *Hiding-tickFree non-tickFree-tick not-less-eq-eq strict-mono-eq*)
next
show ‹*s* ∈ $\mathcal{D}$ (*SKIP r*) ⟹ *s* ∈ $\mathcal{D}$ (*SKIP r* \ *A*)› **for** *s*
by (*simp add*: *D-Hiding D-SKIP T-SKIP*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ (*SKIP r* \ *A*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ (*SKIP r*)› **for** *s X*
by (*auto simp add*: *F-Hiding SKIP-projs image-iff*)
(*metis* (*mono-tags*, *lifting*) *filter.simps*(*1*) *non-tickFree-tick*,
(*metis less-tail list.sel*(*3*) *nil-less strict-mono-Suc-iff*)+)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ (*SKIP r*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ (*SKIP r* \ *A*)› **for** *s X*
by (*simp add*: *F-Hiding SKIP-projs*)
(*metis* $event_{ptick}$.*distinct*(*1*) *filter.simps*(*1*, *2*) *image-iff*)
qed

Synchronization

lemma *SKIP-Sync-SKIP* [*simp*] : ‹*SKIP r* ⟦ *S* ⟧ *SKIP s* = (*if r* = *s then SKIP r else STOP*)›
for *S* :: ‹*'a set*› **and** *r* :: *'r*
proof (*split if-split*, *intro conjI impI*)
show ‹*SKIP r* ⟦*S*⟧ *SKIP s* = *SKIP r*› **if** ‹*r* = *s*›
proof (*fold* ‹*r* = *s*›, *subst Process-eq-spec-optimized*, *safe*)
show ‹*s* ∈ $\mathcal{D}$ (*SKIP r* ⟦*S*⟧ *SKIP r*) ⟹ *s* ∈ $\mathcal{D}$ (*SKIP r*)›
and ‹*s* ∈ $\mathcal{D}$ (*SKIP r*) ⟹ *s* ∈ $\mathcal{D}$ (*SKIP r* ⟦*S*⟧ *SKIP r*)› **for** *s*
by (*simp-all add*: *D-Sync D-SKIP*)
next
assume *same-div* : ‹$\mathcal{D}$ (*SKIP r* ⟦*S*⟧ *SKIP r*) = $\mathcal{D}$ (*SKIP r*)›
fix *s X* **assume** ‹(*s*, *X*) ∈ $\mathcal{F}$ (*SKIP r* ⟦*S*⟧ *SKIP r*)›
then consider ‹*s* ∈ $\mathcal{D}$ (*SKIP r* ⟦*S*⟧ *SKIP r*)›
| *s-P s-Q X-P X-Q* **where**
‹(*s-P*, *X-P*) ∈ $\mathcal{F}$ (*SKIP r*)› ‹(*s-Q*, *X-Q*) ∈ $\mathcal{F}$ (*SKIP r*)›
‹*s setinterleaves* ((*s-P*, *s-Q*), *range tick* ∪ *ev* ' *S*)›
‹*X* = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ' *S*) ∪ *X-P* ∩ *X-Q*›
by (*simp add*: *F-Sync D-Sync*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (*SKIP r*)›

```
    proof cases
      from same-div D-F show ‹s ∈ 𝒟 (SKIP r ⟦S⟧ SKIP r) ⟹ (s, X) ∈ ℱ
(SKIP r)› by blast
    next
      fix s-P s-Q X-P X-Q
      assume ∗ : ‹(s-P, X-P) ∈ ℱ (SKIP r)› ‹(s-Q, X-Q) ∈ ℱ (SKIP r)›
        ‹s setinterleaves ((s-P, s-Q), range tick ∪ ev ‘ S)›
        ‹X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q›
      from ∗(1, 2, 3) consider ‹s = []› ‹s-P = []› ‹s-Q = []›
        | ‹s = [tick r]› ‹s-P = [tick r]› ‹s-Q = [tick r]›
        by (cases s-P; cases s-Q; simp add: image-iff F-SKIP split: if-split-asm)
      thus ‹(s, X) ∈ ℱ (SKIP r)›
      proof cases
        assume ‹s = []› and ‹s-P = []› and ‹s-Q = []›
        from ‹s-P = []› ∗(1) have ‹tick r ∉ X-P› by (simp add: F-SKIP)
       moreover from ‹s-Q = []› ∗(2) have ‹tick r ∉ X-Q› by (simp add: F-SKIP)
        ultimately have ‹tick r ∉ X› by (simp add: ∗(4))
        with ‹s = []› show ‹(s, X) ∈ ℱ (SKIP r)› by (simp add: F-SKIP)
      next
        show ‹s = [tick r] ⟹ (s, X) ∈ ℱ (SKIP r)› by (simp add: F-SKIP)
      qed
    qed
  next
    fix s :: ‹('a, 'r) trace_ptick› and X assume ‹(s, X) ∈ ℱ (SKIP r)›
    then consider ‹s = []› ‹tick r ∉ X› | ‹s = [tick r]› by (simp add: F-SKIP)
blast
    thus ‹(s, X) ∈ ℱ (SKIP r ⟦S⟧ SKIP r)›
    proof cases
      assume ‹s = []› and ‹tick r ∉ X›
      have ‹([], X) ∈ ℱ (SKIP r)› by (simp add: F-SKIP ‹tick r ∉ X›)
      moreover have ‹[] setinterleaves (([], []), range tick ∪ ev ‘ S)› by simp
      moreover have ‹X = (X ∪ X) ∩ (range tick ∪ ev ‘ S) ∪ X ∩ X› by simp
      ultimately show ‹(s, X) ∈ ℱ (SKIP r ⟦S⟧ SKIP r)›
        unfolding F-Sync using ‹s = []› by blast
    next
      assume ‹s = [tick r]›
      have ‹([tick r], X) ∈ ℱ (SKIP r)› by (simp add: F-SKIP)
     moreover have ‹[tick r] setinterleaves (([tick r], [tick r]), range tick ∪ ev ‘
S)› by simp
      moreover have ‹X = (X ∪ X) ∩ (range tick ∪ ev ‘ S) ∪ X ∩ X› by simp
      ultimately show ‹(s, X) ∈ ℱ (SKIP r ⟦S⟧ SKIP r)›
        unfolding F-Sync using ‹s = [tick r]› by blast
    qed
  qed
next
  show ‹r ≠ s ⟹ SKIP r ⟦S⟧ SKIP s = STOP›
    by (force simp add: STOP-iff-T T-Sync T-SKIP D-SKIP)
qed
```

lemma *SKIP-Sync-STOP* [*simp*] : ‹*SKIP r* ⟦*S*⟧ *STOP* = *STOP*›
and *STOP-Sync-SKIP* [*simp*] : ‹*STOP* ⟦*S*⟧ *SKIP r* = *STOP*›
by (*force simp add*: *STOP-iff-T T-Sync T-SKIP D-SKIP T-STOP D-STOP*)+

lemma *Mprefix-Sync-SKIP* : ‹(□*a* ∈ *A* → *P a*) ⟦*S*⟧ *SKIP res* = □*a* ∈ (*A* − *S*) → (*P a* ⟦*S*⟧ *SKIP res*)›
(**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ *D ?lhs*›
then obtain *a t u r v*
where ∗ : ‹*a* ∈ *A*› ‹*front-tickFree v*› ‹*tickFree r* ∨ *v* = []› ‹*s* = *r* @ *v*›
‹*r setinterleaves* ((*ev a* # *t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*t* ∈ *D* (*P a*)› ‹*u* = [] ∨ *u* = [*tick res*]›
by (*simp add*: *D-Sync D-SKIP T-SKIP D-Mprefix image-iff*) *blast*
from ∗(*5*, *7*) **obtain** *r′*
where ∗∗ : ‹*a* ∉ *S*› ‹*r* = *ev a* # *r′*› ‹*r′ setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
by (*elim disjE*; *simp add*: *image-iff split*: *if-split-asm*, *blast*)
show ‹*s* ∈ *D ?rhs*›
by (*simp add*: *D-Mprefix* ∗(*1*, *4*) ∗∗(*1*, *2*) *D-Sync T-SKIP D-SKIP*)
(*use* ∗(*2*, *3*, *6*, *7*) ∗∗(*2*, *3*) *tickFree-Cons-iff* **in** *blast*)
next
fix *s*
assume ‹*s* ∈ *D ?rhs*›
then obtain *s′ a t u r v*
where ∗ : ‹*s* = *ev a* # *s′*› ‹*a* ∈ *A*› ‹*a* ∉ *S*› ‹*front-tickFree v*› ‹*t* ∈ *D* (*P a*)›
‹*tickFree r* ∨ *v* = []› ‹*s′* = *r* @ *v*› ‹*u* = [] ∨ *u* = [*tick res*]›
‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
by (*simp add*: *D-Mprefix D-Sync T-SKIP D-SKIP image-iff*) *blast*
from ∗(*8*, *9*) **have** ‹(*ev a* # *r*) *setinterleaves* ((*ev a* # *t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
by (*elim disjE*; *simp add*: ∗(*3*) *image-iff*)
moreover have ‹*ev a* # *t* ∈ *D* (□*a* ∈ *A* → *P a*)› **by** (*simp add*: *D-Mprefix* ∗(*2*, *5*))
ultimately show ‹*s* ∈ *D ?lhs*›
apply (*simp add*: *D-Sync T-SKIP D-SKIP* ∗(*1*, *7*))
apply (*rule exI*[*of* - ‹*ev a* # *t*›], *rule exI*[*of* - *u*],
rule exI[*of* - ‹*ev a* # *r*›], *simp*)
using ∗(*4*) ∗(*6*) ∗(*8*) **by** *blast*
next
fix *s Z*
assume *same-div*: ‹*D ?lhs* = *D ?rhs*›
assume ‹(*s*, *Z*) ∈ *F ?lhs*›
then consider ‹*s* ∈ *D ?lhs*›
| *t u X Y* **where** ‹(*t*, *X*) ∈ *F* (□*a* ∈ *A* → *P a*)› ‹(*u*, *Y*) ∈ *F* (*SKIP res*)›
‹*s setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*Z* = (*X* ∪ *Y*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X* ∩ *Y*›

```
    by (simp add: F-Sync D-Sync) blast
  thus ‹(s, Z) ∈ ℱ ?rhs›
  proof cases
    from same-div D-F show ‹s ∈ 𝒟 ?lhs ⟹ (s, Z) ∈ ℱ ?rhs› by blast
  next
    fix t u X Y
    assume assms : ‹(t, X) ∈ ℱ (□a ∈ A → P a)› ‹(u, Y) ∈ ℱ (SKIP res)›
      ‹s setinterleaves ((t, u), range tick ∪ ev ‘ S)›
      ‹Z = (X ∪ Y) ∩ (range tick ∪ ev ‘ S) ∪ X ∩ Y›
    from assms(1) have ‹t = [] ∧ X ∩ ev ‘ A = {} ∨
                        (∃a t'. a ∈ A ∧ t = ev a # t' ∧ (t', X) ∈ ℱ (P a))›
      by (simp add: F-Mprefix image-iff) blast
    thus ‹(s, Z) ∈ ℱ ?rhs›
    proof (elim disjE exE conjE)
      assume ‹t = []› and ‹X ∩ ev ‘ A = {}›
      from ‹t = []› assms(2, 3) emptyLeftProperty have ‹s = [] ∧ u = []›
        by (cases s; cases u; simp add: F-SKIP split: if-split-asm)
      with ‹X ∩ ev ‘ A = {}› show ‹(s, Z) ∈ ℱ ?rhs›
        by (auto simp add: F-Mprefix assms(4))
    next
      fix a t'
      assume ‹a ∈ A› ‹t = ev a # t'› ‹(t', X) ∈ ℱ (P a)›
      from assms(2, 3) ‹t = ev a # t'› obtain s'
        where ∗ : ‹a ∉ S› ‹s = ev a # s'› ‹s' setinterleaves ((t', u), range tick ∪
ev ‘ S)›
        by (simp add: F-SKIP, elim disjE; simp split: if-split-asm, blast)
      show ‹(s, Z) ∈ ℱ ?rhs›
        by (simp add: F-Mprefix ‹s = ev a # s'› ‹a ∈ A› ‹a ∉ S› F-Sync)
          (use ∗(3) ‹(t', X) ∈ ℱ (P a)› assms(2, 4) in blast)
    qed
  qed
next
  fix s Z
  assume same-div: ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, Z) ∈ ℱ ?rhs›
  hence ‹s = [] ∧ Z ∩ ev ‘ (A − S) = {} ∨ (∃a s'. a ∈ A ∧ a ∉ S ∧ s = ev a #
s' ∧ (s', Z) ∈ ℱ (P a ⟦S⟧ SKIP res))›
    by (simp add: F-Mprefix image-iff) blast
  thus ‹(s, Z) ∈ ℱ ?lhs›
  proof (elim disjE exE conjE)
    assume ‹s = []› and ‹Z ∩ ev ‘ (A − S) = {}›
    hence ∗ : ‹([], Z − ev ‘ S) ∈ ℱ (□a ∈ A → P a) ∧ ([], Z − {tick res}) ∈ ℱ
(SKIP res) ∧
              s setinterleaves (([], []), range tick ∪ ev ‘ S) ∧
              Z = (Z − ev ‘ S ∪ (Z − {tick res})) ∩ (range tick ∪ ev ‘ S) ∪ (Z −
ev ‘ S) ∩ (Z − {tick res})›
      by (auto simp add: F-Mprefix F-SKIP)
    show ‹(s, Z) ∈ ℱ ?lhs› by (simp add: F-Sync) (metis ∗)
  next
```

fix $a\ s'$
assume ‹$a \in A$› ‹$a \notin S$› ‹$s = ev\ a\ \#\ s'$› ‹$(s', Z) \in \mathcal{F}\ (P\ a\ [\![S]\!]\ SKIP\ res)$›
from ‹$(s', Z) \in \mathcal{F}\ (P\ a\ [\![S]\!]\ SKIP\ res)$› **consider** ‹$s' \in \mathcal{D}\ (P\ a\ [\![S]\!]\ SKIP\ res)$›
| $t\ u\ X\ Y$ **where** ‹$(t, X) \in \mathcal{F}\ (P\ a)$› ‹$(u, Y) \in \mathcal{F}\ (SKIP\ res)$›
‹s' *setinterleaves* $((t, u), range\ tick \cup ev\ `\ S)$›
‹$Z = (X \cup Y) \cap (range\ tick \cup ev\ `\ S) \cup X \cap Y$›
by (*simp add*: *F-Sync D-Sync*) *blast*
thus ‹$(s, Z) \in \mathcal{F}$ *?lhs*›
proof *cases*
assume ‹$s' \in \mathcal{D}\ (P\ a\ [\![S]\!]\ SKIP\ res)$›
with ‹$a \in A$› ‹$a \notin S$› ‹$s = ev\ a\ \#\ s'$› **have** ‹$s \in \mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Mprefix*)
with *same-div D-F* **show** ‹$(s, Z) \in \mathcal{F}$ *?lhs*› **by** *blast*
next
fix $t\ u\ X\ Y$
assume *assms*: ‹$(t, X) \in \mathcal{F}\ (P\ a)$› ‹$(u, Y) \in \mathcal{F}\ (SKIP\ res)$›
‹s' *setinterleaves* $((t, u), range\ tick \cup ev\ `\ S)$›
‹$Z = (X \cup Y) \cap (range\ tick \cup ev\ `\ S) \cup X \cap Y$›
hence ‹$(ev\ a\ \#\ t, X) \in \mathcal{F}\ (\Box a \in A \rightarrow P\ a) \wedge s$ *setinterleaves* $((ev\ a\ \#\ t, u), range\ tick \cup ev\ `\ S)$›
by (*simp add*: *F-SKIP F-Mprefix* ‹$s = ev\ a\ \#\ s'$›, *elim disjE*)
(*simp-all add*: ‹$a \in A$› ‹$a \notin S$› *image-iff*)
with *assms*(*2*, *4*) **show** ‹$(s, Z) \in \mathcal{F}$ *?lhs*› **by** (*simp add*: *F-Sync*) *blast*
qed
qed
qed

lemma *SKIP-Sync-Mprefix* : ‹$SKIP\ res\ [\![S]\!]\ (\Box b \in B \rightarrow P\ b) = \Box b \in (B - S) \rightarrow (SKIP\ res\ [\![S]\!]\ P\ b)$›
by (*metis* (*no-types*, *lifting*) *Mprefix-Sync-SKIP Sync-commute mono-Mprefix-eq*)

lemma *Renaming-SKIP* [*simp*] : ‹$Renaming\ (SKIP\ r)\ f\ g = SKIP\ (g\ r)$›
by (*force simp add*: *Process-eq-spec F-Renaming F-SKIP D-Renaming D-SKIP*)

Associative Operators

lemma *Det-assoc*: ‹$P \Box (Q \Box R) = P \Box Q \Box R$›
by (*auto simp add*: *Process-eq-spec D-Det F-Det T-Det*)

lemma *Ndet-assoc*: ‹$P \sqcap (Q \sqcap R) = P \sqcap Q \sqcap R$›
by (*auto simp add*: *Process-eq-spec D-Ndet F-Ndet*)

lemma *Sliding-assoc*: ‹$P \rhd (Q \rhd R) = P \rhd Q \rhd R$›
by (*auto simp add*: *Process-eq-spec F-Sliding D-Sliding T-Sliding*)

lemma *Seq-assoc* : ‹$P ; (Q ; R) = P ; Q ; R$›
proof (*rule Process-eq-optimizedI*)
show ‹$t \in \mathcal{D}\ (P ; (Q ; R)) \Longrightarrow t \in \mathcal{D}\ (P ; Q ; R)$› **for** t
by (*simp add*: *D-Seq T-Seq-bis*) (*metis append.assoc*)
next
fix t **assume** ‹$t \in \mathcal{D}\ (P ; Q ; R)$›
then consider ‹$t \in \mathcal{D}\ (P ; Q)$›
| $u\ v\ r$ **where** ‹$t = u\ @\ v$› ‹$u\ @\ [✓(r)] \in \mathcal{T}\ (P ; Q)$› ‹$v \in \mathcal{D}\ R$›
by (*simp add*: *D-Seq*) *blast*
thus ‹$t \in \mathcal{D}\ (P ; (Q ; R))$›
proof *cases*
show ‹$t \in \mathcal{D}\ (P ; Q) \Longrightarrow t \in \mathcal{D}\ (P ; (Q ; R))$› **by** (*auto simp add*: *D-Seq*)
next
fix $u\ v\ r$ **assume** ‹$t = u\ @\ v$› ‹$u\ @\ [✓(r)] \in \mathcal{T}\ (P ; Q)$› ‹$v \in \mathcal{D}\ R$›
from ‹$u\ @\ [✓(r)] \in \mathcal{T}\ (P ; Q)$› **consider** ‹$u\ @\ [✓(r)] \in \mathcal{D}\ P$›
| $w\ x\ s$ **where** ‹$u\ @\ [✓(r)] = w\ @\ x$› ‹$w\ @\ [✓(s)] \in \mathcal{T}\ P$› ‹$x \in \mathcal{T}\ Q$›
by (*simp add*: *T-Seq-bis*) *blast*
thus ‹$t \in \mathcal{D}\ (P ; (Q ; R))$›
proof *cases*
from ‹$v \in \mathcal{D}\ R$› **show** ‹$u\ @\ [✓(r)] \in \mathcal{D}\ P \Longrightarrow t \in \mathcal{D}\ (P ; (Q ; R))$›
by (*simp add*: *D-Seq* ‹$t = u\ @\ v$›)
(*metis D-imp-front-tickFree butlast-snoc is-processT7 is-processT9*
front-tickFree-iff-tickFree-butlast)
next
from ‹$v \in \mathcal{D}\ R$› **show** ‹$[\![u\ @\ [✓(r)] = w\ @\ x;\ w\ @\ [✓(s)] \in \mathcal{T}\ P;\ x \in \mathcal{T}\ Q]\!]$
$\Longrightarrow t \in \mathcal{D}\ (P ; (Q ; R))$› **for** $w\ x\ s$
by (*cases x rule*: *rev-cases*, *simp-all add*: *D-Seq* ‹$t = u\ @\ v$›)
(*meson append-T-imp-tickFree non-tickFree-tick not-Cons-self2 tick-*
Free-append-iff, *blast*)
qed
qed
next
fix $t\ X$ **assume** ‹$(t, X) \in \mathcal{F}\ (P ; (Q ; R))$› ‹$t \notin \mathcal{D}\ (P ; Q ; R)$› ‹$t \notin \mathcal{D}\ (P ; Q ;$
$R)$›
then consider ‹$(t, X \cup range\ tick) \in \mathcal{F}\ P$› ‹$tF\ t$›
| $u\ v\ r$ **where** ‹$t = u\ @\ v$› ‹$u\ @\ [✓(r)] \in \mathcal{T}\ P$› ‹$(v, X) \in \mathcal{F}\ (Q ; R)$›
unfolding *Seq-projs* **by** *simp metis*
thus ‹$(t, X) \in \mathcal{F}\ (P ; Q ; R)$›
proof *cases*
show ‹$(t, X \cup range\ tick) \in \mathcal{F}\ P \Longrightarrow tF\ t \Longrightarrow (t, X) \in \mathcal{F}\ (P ; Q ; R)$›
by (*simp add*: *F-Seq*)
next
fix $u\ v\ r$ **assume** ‹$t = u\ @\ v$› ‹$u\ @\ [✓(r)] \in \mathcal{T}\ P$› ‹$(v, X) \in \mathcal{F}\ (Q ; R)$›
with ‹$t \notin \mathcal{D}\ (P ; Q ; R)$› **consider** ‹$(v, X \cup range\ tick) \in \mathcal{F}\ Q$› ‹$tF\ v$›

```
    | w x s where ‹v = w @ x› ‹w @ [✓(s)] ∈ 𝒯 Q› ‹(x, X) ∈ ℱ R›
    unfolding Seq-projs by fast
  thus ‹(t, X) ∈ ℱ (P ; Q ; R)›
  proof cases
    show ‹(v, X ∪ range tick) ∈ ℱ Q ⟹ tF v ⟹ (t, X) ∈ ℱ (P ; Q ; R)›
      by (simp add: ‹t = u @ v› F-Seq)
        (metis ‹u @ [✓(r)] ∈ 𝒯 P› append-T-imp-tickFree list.discI)
  next
    show ‹v = w @ x ⟹ w @ [✓(s)] ∈ 𝒯 Q ⟹ (x, X) ∈ ℱ R ⟹ (t, X) ∈ ℱ
(P ; Q ; R)› for w x s
      by (simp add: ‹t = u @ v› F-Seq T-Seq-bis) (metis ‹u @ [✓(r)] ∈ 𝒯 P›
append-assoc)
  qed
 qed
next
 fix t X assume ‹(t, X) ∈ ℱ (P ; Q ; R)› ‹t ∉ 𝒟 (P ; Q ; R)› ‹t ∉ 𝒟 (P ; (Q ;
R))›
 then consider ‹(t, X ∪ range tick) ∈ ℱ (P ; Q)› ‹tF t›
   | u v r where ‹t = u @ v› ‹u @ [✓(r)] ∈ 𝒯 (P ; Q)› ‹(v, X) ∈ ℱ R›
   unfolding Seq-projs[of ‹P ; Q›] by blast
 thus ‹(t, X) ∈ ℱ (P ; (Q ; R))›
 proof cases
  show ‹(t, X ∪ range tick) ∈ ℱ (P ; Q) ⟹ tF t ⟹ (t, X) ∈ ℱ (P ; (Q ; R))›
    by (simp add: F-Seq) (metis tickFree-append-iff)
 next
   fix u v r assume ‹t = u @ v› ‹u @ [✓(r)] ∈ 𝒯 (P ; Q)› ‹(v, X) ∈ ℱ R›
   with ‹t ∉ 𝒟 (P ; (Q ; R))› obtain w x s
     where ‹u @ [✓(r)] = w @ x› ‹w @ [✓(s)] ∈ 𝒯 P› ‹x ∈ 𝒯 Q›
     by (simp add: D-Seq T-Seq)
       (meson F-imp-front-tickFree ‹u @ [✓(r)] ∈ 𝒯 (P ; Q)›
         append-T-imp-tickFree is-processT7 is-processT9 not-Cons-self2)
   with ‹(v, X) ∈ ℱ R› show ‹(t, X) ∈ ℱ (P ; (Q ; R))›
     by (cases x rule: rev-cases, simp-all add: ‹t = u @ v› F-Seq)
       (metis append-T-imp-tickFree non-tickFree-tick
         not-Cons-self2 tickFree-append-iff, blast)
 qed
qed
```

Synchronization

```
lemma interleave-set: ‹s setinterleaves ((t, u), C) ⟹ set t ∪ set u ⊆ set s›
 by (induct ‹(t, C, u)› arbitrary: t u s rule: setinterleaving.induct)
  (simp split: if-split-asm; meson dual-order.trans list.set-intros set-subset-Cons)+


lemma interleave-order: ‹s setinterleaves ((t1 @ t2, u), C) ⟹ set t2 ⊆ set (drop
(length t1) s)›
proof (induct ‹(t1, C, u)› arbitrary: t1 u s rule: setinterleaving.induct)
 case 1
```

with *emptyRightProperty* **show** *?case* **by** *auto*
next
case (*2 y u*)
with *interleave-set* **show** *?case* **by** *simp blast*
next
case (*3 x t1*)
thus *?case* **by** (*simp split*: *if-split-asm*) (*metis drop-Suc-Cons*)
next
case (*4 x t1 y u*)
thus *?case*
apply (*simp split*: *if-split-asm*)
apply (*metis drop-Suc-Cons*)
apply (*metis* (*mono-tags*, *opaque-lifting*) *drop-Suc-Cons dual-order.refl le-SucI set-drop-subset-set-drop subset-trans*)
apply (*metis* (*no-types*, *opaque-lifting*) *drop-Suc-Cons dual-order.trans le-SucI order-refl set-drop-subset-set-drop*)
by (*metis drop-Suc-Cons*)
qed

lemma *interleave-append-left* :
‹*s setinterleaves* ((*t1* @ *t2*, *u*), *C*) ⟹
∃ *u1 u2 s1 s2*. *u* = *u1* @ *u2* ∧ *s* = *s1* @ *s2* ∧
s1 setinterleaves ((*t1*, *u1*), *C*) ∧ *s2 setinterleaves* ((*t2*, *u2*), *C*)›
proof (*induct* ‹(*t1*, *C*, *u*)› *arbitrary*: *t1 u s rule*: *setinterleaving.induct*)
case *1*
thus *?case* **by** *simp*
next
case (*2 y u*)
thus *?case* **by** (*metis si-empty1 append-Nil singletonI*)
next
case (*3 x t1*)
thus *?case* **by** (*simp split*: *if-split-asm*) (*metis append-Cons*)
next
case (*4 x t1 y u*)
from *4.prems* **consider** *v* **where** ‹*x* ∈ *C*› ‹*y* ∈ *C*› ‹*x* = *y*› ‹*s* = *y* # *v*› ‹*v setinterleaves* ((*t1* @ *t2*, *u*), *C*)›
| *v* **where** ‹*x* ∈ *C*› ‹*y* ∉ *C*› ‹*s* = *y* # *v*› ‹*v setinterleaves* (((*x* # *t1*) @ *t2*, *u*), *C*)›
| *v* **where** ‹*x* ∉ *C*› ‹*y* ∉ *C*› ‹*s* = *x* # *v*› ‹*v setinterleaves* ((*t1* @ *t2*, *y* # *u*), *C*)›
| *v* **where** ‹*x* ∉ *C*› ‹*y* ∉ *C*› ‹*s* = *y* # *v*› ‹*v setinterleaves* (((*x* # *t1*) @ *t2*, *u*), *C*)›
| *v* **where** ‹*x* ∉ *C*› ‹*y* ∈ *C*› ‹*s* = *x* # *v*› ‹*v setinterleaves* ((*t1* @ *t2*, *y* # *u*), *C*)›
by (*auto split*: *if-split-asm*)
thus *?case*
proof *cases*
fix *v* **assume** ∗ : ‹*x* ∈ *C*› ‹*y* ∈ *C*› ‹*x* = *y*› ‹*s* = *y* # *v*› ‹*v setinterleaves* ((*t1*

```
@ t2, u), C)›
   from 4.hyps(1)[OF *(1, 2, 3, 5)] obtain u1 u2 s1 s2
     where ** : ‹u = u1 @ u2› ‹v = s1 @ s2› ‹s1 setinterleaves ((t1, u1), C)›
       ‹s2 setinterleaves ((t2, u2), C)› by blast
   show ?case
     apply (rule exI[of - ‹y # u1›], rule exI[of - u2])
     apply (rule exI[of - ‹y # s1›], rule exI[of - s2])
     by (simp add: *(2, 3, 4) **)
 next
   fix v assume * : ‹x ∈ C› ‹y ∉ C› ‹s = y # v› ‹v setinterleaves (((x # t1) @
t2, u), C)›
   from 4.hyps(2)[OF *(1, 2, 4)] obtain u1 u2 s1 s2
     where ** : ‹u = u1 @ u2› ‹v = s1 @ s2› ‹s1 setinterleaves ((x # t1, u1),
C)›
       ‹s2 setinterleaves ((t2, u2), C)› by blast
   show ?case
     apply (rule exI[of - ‹y # u1›], rule exI[of - u2])
     apply (rule exI[of - ‹y # s1›], rule exI[of - s2])
     by (simp add: *(1, 2, 3) **)
 next
   fix v assume * : ‹x ∉ C› ‹y ∉ C› ‹s = x # v› ‹v setinterleaves ((t1 @ t2, y
# u), C)›
   from 4.hyps(3)[OF *(1, 2, 4)] obtain u1 u2 s1 s2
     where ** : ‹y # u = u1 @ u2› ‹v = s1 @ s2› ‹s1 setinterleaves ((t1, u1),
C)›
       ‹s2 setinterleaves ((t2, u2), C)› by blast
   show ?case
     apply (rule exI[of - ‹u1›],     rule exI[of - u2])
     apply (rule exI[of - ‹x # s1›], rule exI[of - s2])
     by (cases u1) (use ** in ‹simp-all add: *(1, 2, 3)›)
 next
   fix v assume * : ‹x ∉ C› ‹y ∉ C› ‹s = y # v› ‹v setinterleaves (((x # t1) @
t2, u), C)›
   from 4.hyps(4)[OF *(1, 2, 4)] obtain u1 u2 s1 s2
     where ** : ‹u = u1 @ u2› ‹v = s1 @ s2› ‹s1 setinterleaves ((x # t1, u1),
C)›
       ‹s2 setinterleaves ((t2, u2), C)› by blast
   show ?case
     apply (rule exI[of - ‹y # u1›], rule exI[of - u2])
     apply (rule exI[of - ‹y # s1›], rule exI[of - s2])
     by (simp add: *(1, 2, 3) **)
 next
   fix v assume * : ‹x ∉ C› ‹y ∈ C› ‹s = x # v› ‹v setinterleaves ((t1 @ t2, y
# u), C)›
   from 4.hyps(5)[simplified, OF *(1, 2, 4)] obtain u1 u2 s1 s2
     where ** : ‹y # u = u1 @ u2› ‹v = s1 @ s2› ‹s1 setinterleaves ((t1, u1),
C)›
       ‹s2 setinterleaves ((t2, u2), C)› by blast
   show ?case
```

```
    apply (rule exI[of - ‹u1›],      rule exI[of - u2])
    apply (rule exI[of - ‹x # s1›], rule exI[of - s2])
    by (cases u1) (use ** in ‹simp-all add: *(1, 2, 3)›)
  qed
qed


lemma interleave-append-right :
  ‹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 (subst (asm) setinterleaving-sym, drule interleave-append-left)
    (simp add: setinterleaving-sym)

lemma interleave-append-tail-left:
  ‹s setinterleaves ((t1, u), C) ⟹ set t2 ∩ C = {} ⟹ (s @ t2) setinterleaves
((t1 @ t2, u), C)›
  by (induct ‹(t1, C, u)› arbitrary: t1 t2 u s rule: setinterleaving.induct,
      auto split: if-split-asm)
    (metis disjoint-iff emptyLeftSelf setinterleaving-sym,
      metis SyncSingleHeadAdd setinterleaving-sym)


lemma interleave-append-tail-right:
  ‹s setinterleaves ((t, u1), C) ⟹ set u2 ∩ C = {} ⟹ (s @ u2) setinterleaves
((t, u1 @ u2), C)›
  by (metis (no-types) setinterleaving-sym interleave-append-tail-left)



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 ‹(tu, A, v)› arbitrary: t u tu v tuv rule: setinterleaving.induct)
  case 1
  with emptyLeftProperty empty-setinterleaving show ?case by blast
next
  case (2 z v)
  from 2.prems(2) emptyLeftProperty have ‹tuv = z # v› by blast
  with 2.prems empty-setinterleaving setinterleaving-sym show ?case by blast
next
  case (3 y tu)
  from 3.prems(2) emptyRightProperty have ‹tuv = y # tu› by blast
  with 3.prems show ?case
    by (metis (no-types) EmptyLeftSync Un-iff disjoint-iff
        emptyLeftSelf interleave-set setinterleaving-sym subsetD)
next
  case (4 y tu z v)
  consider ‹y ∈ A› ‹z ∈ A› | ‹y ∈ A› ‹z ∉ A› | ‹y ∉ A› ‹z ∈ A› | ‹y ∉ A› ‹z ∉
```

```
A› by blast
 thus ?case
 proof cases
   assume ‹y ∈ A› and ‹z ∈ A›
   with 4.prems(2) obtain tuv'
     where ‹y = z› ‹tuv = y # tuv'› ‹tuv' setinterleaves ((tu, v), A)›
     by (simp split: if-split-asm) blast
   from ‹tuv = y # tuv'› ‹y = z› ‹z ∈ A› 4.prems(1)
     4.hyps(1)[OF ‹y ∈ A› ‹z ∈ A› ‹y = z› - ‹tuv' setinterleaves ((tu, v), A)›]
   show ?case by (induct ‹(t, A, u)› arbitrary: t u tu rule: setinterleaving.induct;
       fastforce split: if-split-asm)
 next
   assume ‹y ∈ A› and ‹z ∉ A›
   with 4.prems(2) obtain tuv'
     where ‹tuv = z # tuv'› ‹tuv' setinterleaves ((y # tu, v), A)›
     by (simp split: if-split-asm) blast
   from ‹y ∈ A› and ‹z ∉ A› ‹tuv = z # tuv'› 4.prems(1)
     4.hyps(2)[OF - - - ‹tuv' setinterleaves ((y # tu, v), A)›]
   show ?case
     apply (cases ‹(t, A, u)› rule: setinterleaving.cases; simp split: if-split-asm)
     by (metis 4.prems(1) SyncHdAdd list.distinct(1) list.sel(1, 3)) fastforce
 next
   assume ‹y ∉ A› and ‹z ∈ A›
   with 4.prems(2) obtain tuv'
     where ‹tuv = y # tuv'› ‹tuv' setinterleaves ((tu, z # v), A)›
     by (simp split: if-split-asm) blast
   from ‹y ∉ A› ‹z ∈ A› ‹tuv = y # tuv'› 4.prems(1)
     4.hyps(5)[OF - - - ‹tuv' setinterleaves ((tu, z # v), A)›]
    show ?case by (cases ‹(t, A, u)› rule: setinterleaving.cases; fastforce split:
if-split-asm)
 next
   assume ‹y ∉ A› and ‹z ∉ A›
   with 4.prems(2) obtain tuv'
     where ‹tuv = y # tuv' ∧ tuv' setinterleaves ((tu, z # v), A) ∨
           tuv = z # tuv' ∧ tuv' setinterleaves ((y # tu, v), A)› by auto
   thus ?case
   proof (elim disjE conjE)
     assume ‹tuv = y # tuv'› ‹tuv' setinterleaves ((tu, z # v), A)›
     show ?case
     proof (cases ‹(t, A, u)› rule: setinterleaving.cases)
       from 4.prems(1) show ‹(t, A, u) = ([], X, []) ⟹ ?case› for X by simp
     next
       show ‹(t, A, u) = ([], X, y' # u') ⟹ ?case› for X y' u'
         by (metis 4.prems(1) 4.prems(2) Pair-inject emptyLeftProperty
             emptyLeftSelf empty-iff empty-set ftf-Sync1 ftf-Sync21)
     next
       show ‹(t, A, u) = (x' # t', X, []) ⟹ ?case› for X x' t'
         by (metis (no-types, lifting) 4.prems Nil-in-shufflesI Pair-inject si-empty1
             emptyLeftNonSync emptyLeftSelf emptyRightProperty ftf-Sync21 shuf-
```

```
flies.simps(1))
    next
      fix X x' t' y' u'
      assume ‹(t, A, u) = (x' # t', X, y' # u')›
      hence ‹t = x' # t'› ‹u = y' # u'› ‹X = A› by simp-all
      consider ‹x' ∈ A› ‹y' ∈ A› | ‹x' ∈ A› ‹y' ∉ A›
        | ‹x' ∉ A› ‹y' ∈ A› | ‹x' ∉ A› ‹y' ∉ A› by blast
      thus ?case
      proof cases
        assume ‹x' ∈ A› ‹y' ∈ A›
        with 4.prems(1) ‹t = x' # t'› ‹u = y' # u'› ‹y ∉ A› have False
          by (metis SyncSameHdTl list.sel(1))
        thus ?case by simp
      next
        assume ‹x' ∈ A› ‹y' ∉ A›
        with 4.prems(1) ‹t = x' # t'› ‹u = y' # u'›
        have ‹y' = y› ‹tu setinterleaves ((x' # t', u'), A)› by simp-all
        from 4.hyps(3)[OF ‹y ∉ A› ‹z ∉ A› this(2) ‹tuv' setinterleaves ((tu, z #
v), A)›]
        obtain uv where ‹uv setinterleaves ((u', z # v), A) ∧
                          tuv' setinterleaves ((x' # t', uv), A)› ..
        hence ‹(y # uv) setinterleaves ((u, z # v), A) ∧ tuv setinterleaves ((t, y
# uv), A)›
          by (simp add: ‹u = y' # u'› ‹z ∉ A› ‹y' = y› ‹y ∉ A› ‹t = x' # t'› ‹tuv
= y # tuv'›)
        thus ?case by blast
      next
        assume ‹x' ∉ A› ‹y' ∈ A›
        with 4.prems(1) ‹t = x' # t'› ‹u = y' # u'›
        have ‹x' = y› ‹tu setinterleaves ((t', y' # u'), A)› by simp-all
        from 4.hyps(3)[OF ‹y ∉ A› ‹z ∉ A› this(2) ‹tuv' setinterleaves ((tu, z #
v), A)›]
        obtain uv where ‹uv setinterleaves ((y' # u', z # v), A) ∧
                          tuv' setinterleaves ((t', uv), A)› ..
        hence ‹uv setinterleaves ((u, z # v), A) ∧ tuv setinterleaves ((t, uv), A)›
          by (simp add: ‹tuv = y # tuv'› ‹u = y' # u'› ‹z ∉ A›
              ‹t = x' # t'› ‹x' = y› ‹y ∉ A› SyncSingleHeadAdd)
        thus ?case by blast
      next
        assume ‹x' ∉ A› ‹y' ∉ A›
        with 4.prems(1) ‹t = x' # t'› ‹u = y' # u'›
        have ‹x' = y ∧ tu setinterleaves ((t', y' # u'), A) ∨
              y' = y ∧ tu setinterleaves ((x' # t', u'), A)› by auto
        thus ?case
        proof (elim disjE conjE)
          assume ‹x' = y› ‹tu setinterleaves ((t', y' # u'), A)›
          from 4.hyps(3)[OF ‹y ∉ A› ‹z ∉ A› this(2) ‹tuv' setinterleaves ((tu, z
# v), A)›]
          obtain uv where ‹uv setinterleaves ((y' # u', z # v), A) ∧
```

```
                    tuv' setinterleaves ((t', uv), A)› ..
          hence ‹uv setinterleaves ((u, z # v), A) ∧ tuv setinterleaves ((t, uv), A)›
            by (simp add: ‹x' = y› ‹y ∉ A› ‹t = x' # t'›
                ‹tuv = y # tuv'› ‹u = y' # u'› SyncSingleHeadAdd)
           thus ?case by blast
         next
           assume ‹y' = y› ‹tu setinterleaves ((x' # t', u'), A)›
           from 4.hyps(3)[OF ‹y ∉ A› ‹z ∉ A› this(2) ‹tuv' setinterleaves ((tu, z
# v), A)›]
           obtain uv where ‹uv setinterleaves ((u', z # v), A) ∧
                    tuv' setinterleaves ((x' # t', uv), A)› ..
           hence ‹(y # uv) setinterleaves ((u, z # v), A) ∧ tuv setinterleaves ((t,
y # uv), A)›
             by (simp add: ‹u = y' # u'› ‹y' = y› ‹y ∉ A› ‹z ∉ A› ‹t = x' # t'›
‹tuv = y # tuv'›)
           thus ?case by blast
         qed
       qed
     qed
   next
     assume ‹tuv = z # tuv'› ‹tuv' setinterleaves ((y # tu, v), A)›
     from 4.hyps(4)[OF ‹y ∉ A› ‹z ∉ A› 4.prems(1) ‹tuv' setinterleaves ((y #
tu, v), A)›]
     obtain uv where ‹uv setinterleaves ((u, v), A) ∧ tuv' setinterleaves ((t, uv),
A)› ..
     hence ‹(z # uv) setinterleaves ((u, z # v), A) ∧ tuv setinterleaves ((t, z #
uv), A)›
       by (cases u; cases t) (simp-all add: ‹z ∉ A› ‹tuv = z # tuv'›)
     thus ?case by blast
   qed
 qed
qed


lemma interleave-assoc-2:
 assumes ∗:‹uv setinterleaves ((u, v), A)› and
   ∗∗:‹tuv setinterleaves ((t, uv), A)›
 shows ‹∃ tu. tu setinterleaves ((t, u), A) ∧ tuv setinterleaves ((tu, v), A)›
 using ∗ ∗∗ setinterleaving-sym interleave-assoc-1 by blast


theorem Sync-assoc: ‹P ⟦S⟧ (Q ⟦S⟧ R) = P ⟦S⟧ Q ⟦S⟧ R› for P Q R :: ‹('a, 'r)
process_ptick›
proof (rule FD-antisym)
 show ‹P ⟦S⟧ (Q ⟦S⟧ R) ⊑_FD P ⟦S⟧ Q ⟦S⟧ R› for P Q R :: ‹('a, 'r) process_ptick›
 proof (rule failure-divergence-refine-optimizedI)
   fix s assume ‹s ∈ 𝒟 (P ⟦S⟧ Q ⟦S⟧ R)›
   from this[simplified D-Sync[of ‹P ⟦S⟧ Q›]] obtain t u r v
```

```
    where ∗ : ‹front-tickFree v› ‹tickFree r ∨ v = []› ‹s = r @ v›
      ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
      ‹t ∈ 𝒟 (P ⟦S⟧ Q) ∧ u ∈ 𝒯 R ∨ t ∈ 𝒟 R ∧ u ∈ 𝒯 (P ⟦S⟧ Q) − 𝒟 (P ⟦S⟧
Q)›
    by simp (metis D-T setinterleaving-sym)
  from ∗(5) show ‹s ∈ 𝒟 (P ⟦S⟧ (Q ⟦S⟧ R))›
  proof (elim disjE conjE)
    assume ‹t ∈ 𝒟 R› and ‹u ∈ 𝒯 (P ⟦S⟧ Q) − 𝒟 (P ⟦S⟧ Q)›
    from ‹u ∈ 𝒯 (P ⟦S⟧ Q) − 𝒟 (P ⟦S⟧ Q)› obtain t1 u1
      where ∗∗ : ‹u setinterleaves ((t1, u1), range tick ∪ ev ‘ S)›
        ‹t1 ∈ 𝒯 P› ‹u1 ∈ 𝒯 Q› by (simp add: D-Sync T-Sync) blast
    from interleave-assoc-1[OF ∗∗(1)] obtain uv
      where ∗∗∗ : ‹uv setinterleaves ((u1, t), range tick ∪ ev ‘ S)›
        ‹r setinterleaves ((t1, uv), range tick ∪ ev ‘ S)›
      using ∗(4) setinterleaving-sym by blast
    from ∗∗∗(1) setinterleaving-sym ‹t ∈ 𝒟 R› ‹u1 ∈ 𝒯 Q› front-tickFree-Nil
    have ‹uv ∈ 𝒟 (Q ⟦S⟧ R)› unfolding D-Sync by blast
    with ∗(1, 2, 3) ∗∗(2) ∗∗∗(2) setinterleaving-sym
    show ‹s ∈ 𝒟 (P ⟦S⟧ (Q ⟦S⟧ R))› by (subst D-Sync) blast
  next
    assume ‹t ∈ 𝒟 (P ⟦S⟧ Q)› and ‹u ∈ 𝒯 R›
    from ‹t ∈ 𝒟 (P ⟦S⟧ Q)› obtain t1 u1 r1 v1
      where ∗∗ : ‹front-tickFree v1› ‹tickFree r1 ∨ v1 = []› ‹t = r1 @ v1›
        ‹r1 setinterleaves ((t1, u1), range tick ∪ ev ‘ S)›
        ‹t1 ∈ 𝒟 P ∧ u1 ∈ 𝒯 Q ∨ t1 ∈ 𝒟 Q ∧ u1 ∈ 𝒯 P›
      by (simp add: D-Sync) blast
    from interleave-append-left[OF ∗(4)[unfolded ∗∗(3)]] obtain r′ r′′ u′ u′′
      where ∗∗∗ : ‹r′ setinterleaves ((r1, u′), range tick ∪ ev ‘ S)›
        ‹r = r′ @ r′′› ‹u = u′ @ u′′› by blast
    have ‹u′ ∈ 𝒯 R› by (metis ∗∗∗(3) prefixI ‹u ∈ 𝒯 R› is-processT3-TR)
    have $ : ‹front-tickFree (r′′ @ v)›
      by (metis ∗(3) ∗∗∗(2) D-imp-front-tickFree ‹s ∈ 𝒟 (P ⟦S⟧ Q ⟦S⟧ R)›
          front-tickFree-append-iff front-tickFree-charn tickFree-append-iff)
    have $$ : ‹tickFree r′ ∨ r′′ @ v = []›
      by (metis ∗(3) ∗∗∗(2) D-imp-front-tickFree ‹s ∈ 𝒟 (P ⟦S⟧ Q ⟦S⟧ R)›
          append.right-neutral front-tickFree-append-iff tickFree-append-iff)
    have $$$ : ‹s = r′ @ r′′ @ v› by (simp add: ∗(3) ∗∗∗(2))

    from ∗∗(5) show ‹s ∈ 𝒟 (P ⟦S⟧ (Q ⟦S⟧ R))›
    proof (elim disjE conjE)
      assume ‹t1 ∈ 𝒟 P› and ‹u1 ∈ 𝒯 Q›
      from interleave-assoc-1[OF ∗∗(4) ∗∗∗(1)] obtain uv
        where ∗∗∗∗ : ‹uv setinterleaves ((u1, u′), range tick ∪ ev ‘ S)›
          ‹r′ setinterleaves ((t1, uv), range tick ∪ ev ‘ S)› by blast
      from ‹u′ ∈ 𝒯 R› ∗∗∗∗(1) ‹u1 ∈ 𝒯 Q› have ‹uv ∈ 𝒯 (Q ⟦S⟧ R)›
        unfolding T-Sync by blast
      with $ $$ $$$ ∗∗∗∗(2) ‹t1 ∈ 𝒟 P›
      show ‹s ∈ 𝒟 (P ⟦S⟧ (Q ⟦S⟧ R))› unfolding D-Sync by blast
    next
```

```
      assume ‹t1 ∈ 𝒟 Q› ‹u1 ∈ ℱ P›
      from interleave-assoc-2[OF **(4)] ***(1) setinterleaving-sym obtain tu
        where **** : ‹tu setinterleaves ((u', t1), range tick ∪ ev ‘ S)›
          ‹r' setinterleaves ((tu, u1), range tick ∪ ev ‘ S)› by blast
      from ‹u' ∈ ℱ R› ****(1)[unfolded setinterleaving-sym] front-tickFree-Nil ‹t1
∈ 𝒟 Q›
      have ‹tu ∈ 𝒟 (Q ⟦S⟧ R)› unfolding D-Sync by blast
      with $ $$ $$$ ****(2) ‹u1 ∈ ℱ P›
      show ‹s ∈ 𝒟 (P ⟦S⟧ (Q ⟦S⟧ R))› unfolding D-Sync by blast
    qed
  qed
 next
  let ?Sync-set = ‹λX Y. (X ∪ Y) ∩ (range tick ∪ ev ‘ S) ∪ X ∩ Y›
  assume subset-div : ‹𝒟 (P ⟦S⟧ Q ⟦S⟧ R) ⊆ 𝒟 (P ⟦S⟧ (Q ⟦S⟧ R))›
  fix s Z assume ‹(s, Z) ∈ ℱ (P ⟦S⟧ Q ⟦S⟧ R)›
  then consider (D) ‹s ∈ 𝒟 (P ⟦S⟧ Q ⟦S⟧ R)›
   | (F) t X u Y where ‹(t, X) ∈ ℱ (P ⟦S⟧ Q)› ‹(u, Y) ∈ ℱ R› ‹Z = ?Sync-set
X Y›
      ‹s setinterleaves ((t, u), range tick ∪ ev ‘ S)›
    unfolding F-Sync D-Sync by blast
  thus ‹(s, Z) ∈ ℱ (P ⟦S⟧ (Q ⟦S⟧ R))›
  proof cases
    from subset-div D-F show ‹s ∈ 𝒟 (P ⟦S⟧ Q ⟦S⟧ R) ⟹ (s, Z) ∈ ℱ (P ⟦S⟧
(Q ⟦S⟧ R))› by blast
  next
    case F
    from F(1) consider (D1) ‹t ∈ 𝒟 (P ⟦S⟧ Q)›
      | (F1) t1 X1 u1 Y1 where ‹(t1, X1) ∈ ℱ P› ‹(u1, Y1) ∈ ℱ Q› ‹X =
?Sync-set X1 Y1›
        ‹t setinterleaves ((t1, u1), range tick ∪ ev ‘ S)›
      unfolding F-Sync D-Sync by blast
    thus ‹(s, Z) ∈ ℱ (P ⟦S⟧ (Q ⟦S⟧ R))›
    proof cases
      assume ‹t ∈ 𝒟 (P ⟦S⟧ Q)›
      with F(2) F(4) F-T front-tickFree-Nil
      have ‹s ∈ 𝒟 (P ⟦S⟧ Q ⟦S⟧ R)› unfolding D-Sync by blast
      with subset-div D-F show ‹(s, Z) ∈ ℱ (P ⟦S⟧ (Q ⟦S⟧ R))› by blast
    next
      case F1
      from interleave-assoc-1[OF F1(4) F(4)] obtain uv
        where * : ‹uv setinterleaves ((u1, u), range tick ∪ ev ‘ S)›
          ‹s setinterleaves ((t1, uv), range tick ∪ ev ‘ S)› by blast
      from *(1) F(2) F1(2) have ‹(uv, ?Sync-set Y1 Y) ∈ ℱ (Q ⟦S⟧ R)›
        unfolding F-Sync by blast
       with *(2) F1(1) have ‹(s, ?Sync-set X1 (?Sync-set Y1 Y)) ∈ ℱ (P ⟦S⟧
(Q ⟦S⟧ R))›
        by (subst F-Sync) blast
       also from F(3) F1(3) have ‹?Sync-set X1 (?Sync-set Y1 Y) = Z› by
blast
```

finally show ‹$(s, Z) \in \mathcal{F}\ (P\ [\![S]\!]\ (Q\ [\![S]\!]\ R))$› **.**
qed
qed
qed
thus ‹$P\ [\![S]\!]\ Q\ [\![S]\!]\ R \sqsubseteq_{FD}\ P\ [\![S]\!]\ (Q\ [\![S]\!]\ R)$› **by** (*metis Sync-commute*)
qed

12.3 The Step-Laws

The step-laws describe the behaviour of the operators wrt. the multi-prefix choice.

12.3.1 Deterministic Choice

lemma *Mprefix-Det-Mprefix*:
‹$(\Box a \in A \to P\ a) \Box (\Box b \in B \to Q\ b) =$
$\Box x \in (A \cup B) \to (\mathit{if}\ x \in A \cap B\ \mathit{then}\ P\ x \sqcap Q\ x\ \mathit{else}\ \mathit{if}\ x \in A\ \mathit{then}\ P\ x\ \mathit{else}\ Q\ x)$›
(**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow s \in \mathcal{D}$ *?rhs*› **for** *s*
by (*cases s*) (*auto simp add*: *D-Det D-Ndet D-Mprefix*)
next
show ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-Det D-Ndet D-Mprefix split*: *if-split-asm*)
next
show ‹$(s, X) \in \mathcal{F}$ *?lhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Det F-Ndet F-Mprefix*)
next
show ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Det F-Ndet F-Mprefix split*: *if-split-asm*)
qed

corollary *Mprefix-Un-distrib*:
‹$\Box a \in (A \cup B) \to P\ a = (\Box a \in A \to P\ a) \Box (\Box b \in B \to P\ b)$›
by (*simp add*: *Mprefix-Det-Mprefix*) (*rule mono-Mprefix-eq*, *simp*)

12.3.2 Non-Deterministic Choice

lemma *Mprefix-Ndet-Mprefix* :
‹$(\Box a \in A \to P\ a) \sqcap (\Box b \in B \to Q\ b) =$
$(\Box a \in (A - B) \to P\ a) \sqcap (\Box b \in (B - A) \to Q\ b) \Box (\Box x \in (A \cap B) \to P\ x \sqcap Q\ x)$›
(**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow s \in \mathcal{D}$ *?rhs*› **and** ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow s \in \mathcal{D}$ *?lhs*› **for** *s*

by (*auto simp add*: *D-Ndet D-Mprefix D-Det*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
and ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **for** *s X*
by (*cases s*; *auto simp add*: *F-Ndet F-Det F-Mprefix D-Mprefix D-Ndet T-Ndet*)+
qed

12.3.3 Sliding Choice

lemma *Mprefix-Sliding-Mprefix* :
‹(□*a*∈ *A* → *P a*) ▷ (□*b* ∈ *B* → *Q b*) =
(□*x* ∈ (*A* ∪ *B*) → (*if x* ∈ *A* ∩ *B then P x* ⊓ *Q x else if x* ∈ *A then P x else Q x*)) ⊓
(□*b* ∈ *B* → (*if b* ∈ *A then P b* ⊓ *Q b else Q b*))›
(**is** ‹*?lhs* = *?rhs*›)
— It is not so much a "step law" as a rewriting.
proof (*subst Process-eq-spec*, *safe*)
show ‹*s* ∈ $\mathcal{D}$ *?lhs* ⟹ *s* ∈ $\mathcal{D}$ *?rhs*›
and ‹*s* ∈ $\mathcal{D}$ *?rhs* ⟹ *s* ∈ $\mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-Sliding D-Mprefix D-Ndet split*: *if-split-asm*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Sliding F-Mprefix F-Ndet*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Sliding F-Mprefix F-Ndet split*: *if-split-asm*)
qed

corollary *Mprefix-Sliding-superset-Mprefix* :
‹(□*a*∈ *A* → *P a*) ▷ (□*b* ∈ *B* → *Q b*) =
□*b* ∈ *B* → (*if b* ∈ *A then P b* ⊓ *Q b else Q b*)› **if** ‹*A* ⊆ *B*›
— This one (with the additional assumption *A* ⊆ *B*) is a "step law".
proof −
have ‹(□*a*∈ *A* → *P a*) ▷ (□*b* ∈ *B* → *Q b*) =
(□*b* ∈ *B* → (*if b* ∈ *A* ∩ *B then P b* ⊓ *Q b else if b* ∈ *A then P b else Q b*)) ⊓
(□*b* ∈ *B* → (*if b* ∈ *A then P b* ⊓ *Q b else Q b*))›
by (*simp add*: *Mprefix-Sliding-Mprefix Un-absorb1* ‹*A* ⊆ *B*›)
also have ‹(□*b* ∈ *B* → (*if b* ∈ *A* ∩ *B then P b* ⊓ *Q b else if b* ∈ *A then P b else Q b*)) =
□*b* ∈ *B* → (*if b* ∈ *A then P b* ⊓ *Q b else Q b*)›
by (*rule mono-Mprefix-eq*) *simp*
finally show ‹(□*a*∈ *A* → *P a*) ▷ (□*b* ∈ *B* → *Q b*) =
□*b* ∈ *B* → (*if b* ∈ *A then P b* ⊓ *Q b else Q b*)› **by** *simp*
qed

corollary *Mprefix-Sliding-same-set-Mprefix* :
‹(□*a*∈ *A* → *P a*) ▷ (□*a* ∈ *A* → *Q a*) = □*a* ∈ *A* → *P a* ⊓ *Q a*›
by (*simp add*: *Mprefix-Sliding-superset-Mprefix*)

(*rule mono-Mprefix-eq*, *simp*)

12.3.4 Sequential Composition

lemma *Mprefix-Seq*: ‹$\Box a \in A \rightarrow P\ a\ ;\ Q = \Box a \in A \rightarrow (P\ a\ ;\ Q)$›
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a\ ;\ Q) \Longrightarrow s \in \mathcal{D}\ (\Box a \in A \rightarrow (P\ a\ ;\ Q))$› **for** *s*
by (*cases s*; *simp add*: *D-Seq D-Mprefix T-Mprefix image-iff*)
(*metis* $event_{ptick}$*.distinct*(*1*) *hd-append list.sel*(*1*, *3*) *tl-append2*)
next
show ‹$s \in \mathcal{D}\ (\Box a{\in}A \rightarrow (P\ a\ ;\ Q)) \Longrightarrow s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a\ ;\ Q)$› **for** *s*
by (*cases s*; *simp add*: *D-Seq D-Mprefix T-Mprefix image-iff*) (*metis append-Cons*)
next
show ‹$(s, X) \in \mathcal{F}\ (\Box a{\in}A \rightarrow (P\ a\ ;\ Q)) \Longrightarrow (s, X) \in \mathcal{F}\ (\Box a \in A \rightarrow P\ a\ ;\ Q)$›
for *s X*
apply (*cases s*; *simp add*: *F-Mprefix T-Mprefix D-Mprefix F-Seq disjoint-iff image-iff*)
by *blast* (*metis Cons-eq-appendI* $event_{ptick}$*.disc*(*1*))
next
assume *same-div*: ‹$\mathcal{D}\ (\Box a \in A \rightarrow P\ a\ ;\ Q) = \mathcal{D}\ (\Box a \in A \rightarrow (P\ a\ ;\ Q))$›
show ‹$(s, X) \in \mathcal{F}\ (\Box a \in A \rightarrow P\ a\ ;\ Q) \Longrightarrow (s, X) \in \mathcal{F}\ (\Box a \in A \rightarrow (P\ a\ ;\ Q))$›
for *s X*
apply (*simp add*: *F-Seq*, *elim disjE exE conjE*)
subgoal by (*cases s*; *simp add*: *F-Mprefix image-iff F-Seq*; *blast*)
subgoal by (*cases s*; *simp add*: *F-Mprefix T-Mprefix F-Seq image-iff*)
(*metis append-self-conv2* $event_{ptick}$*.simps*(*4*) *hd-append2 list.sel*(*1*, *3*) *tl-append-if*)
using *same-div D-F D-Seq* **by** *blast*
qed

12.3.5 Hiding

We use a context to hide the intermediate results.

context begin

Two intermediate Results

private lemma *D-Hiding-Mprefix-dir2*:
‹$s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a \setminus S)$› **if** ‹$s \in \mathcal{D}\ (\Box a \in (A - S) \rightarrow (P\ a \setminus S))$›
proof −
from *that* **obtain** *a s′*
where ∗ : ‹$a \in A$› ‹$a \notin S$› ‹$s = ev\ a\ \#\ s'$› ‹$s' \in \mathcal{D}\ (P\ a \setminus S)$›
by (*auto simp add*: *D-Mprefix*)
from ∗(*4*) **obtain** *t u*
where ∗∗ : ‹*ftF u*› ‹*tF t*› ‹$s' =$ *trace-hide t* (*ev* ‘ *S*) @ *u*›
‹$t \in \mathcal{D}\ (P\ a) \vee (\exists\ f.$ *isInfHiddenRun f* $(P\ a)\ S \wedge t \in$ *range f*)›
by (*simp add*: *D-Hiding*) *blast*
from ∗∗(*4*) **show** ‹$s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a \setminus S)$›
proof (*elim disjE*)

```
    assume ‹t ∈ 𝒟 (P a)›
    hence ‹tF (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
          ev a # t ∈ 𝒟 (Mprefix A P)›
      by (simp add: *(1, 2, 3) **(2, 3) image-iff[of ‹ev a›] D-Mprefix)
    show ‹s ∈ 𝒟 (Mprefix A P \ S)›
      unfolding D-Hiding using **(1) ‹?this› by blast
  next
    assume ‹∃f. isInfHiddenRun f (P a) S ∧ t ∈ range f›
    then obtain f where ***: ‹isInfHiddenRun f (P a) S› ‹t ∈ range f› by blast
    hence ‹tF (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
          isInfHiddenRun (λi. ev a # f i) (Mprefix A P) S ∧
          ev a # t ∈ range (λi. ev a # f i)›
      by (auto simp add: *(1, 2, 3) **(2, 3) image-iff[of ‹ev a›]
          T-Mprefix strict-mono-Suc-iff)
    show ‹s ∈ 𝒟 (Mprefix A P \ S)›
      unfolding D-Hiding using **(1) ‹?this› by blast
  qed
qed


private lemma F-Hiding-Mprefix-dir2:
  ‹(s, X) ∈ ℱ (□a ∈ A → P a \ S)› if ‹s ≠ []› and ‹(s, X) ∈ ℱ (□a ∈ (A − S)
→ (P a \ S))›
proof −
  from that obtain a s'
    where *: ‹a ∈ A› ‹a ∉ S› ‹s = ev a # s'› ‹(s', X) ∈ ℱ (P a \ S)›
    by (auto simp add: F-Mprefix)
  from *(4) consider ‹s' ∈ 𝒟 (P a \ S)›
    | t where ‹s' = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (P a)›
    by (simp add: F-Hiding D-Hiding) blast
  thus ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
  proof cases
    assume ‹s' ∈ 𝒟 (P a \ S)›
    then obtain t u
      where **: ‹ftF u› ‹tF t›
        ‹s' = trace-hide t (ev ‘ S) @ u›
        ‹t ∈ 𝒟 (P a) ∨ (∃ f. isInfHiddenRun f (P a) S ∧ t ∈ range f)›
      by (simp add: D-Hiding) blast
    from **(4) show ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
    proof (elim disjE)
      assume ‹t ∈ 𝒟 (P a)›
      hence ‹tF (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
            ev a # t ∈ 𝒟 (Mprefix A P)›
        by (simp add: *(1, 2, 3) **(2, 3) D-Mprefix image-iff[of ‹ev a›])
      show ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
        unfolding F-Hiding using **(1) ‹?this› by blast
    next
      assume ‹∃f. isInfHiddenRun f (P a) S ∧ t ∈ range f›
      then obtain f where ‹isInfHiddenRun f (P a) S› ‹t ∈ range f› by blast
```

hence ‹*tF* (*ev a # t*) ∧ *s* = *trace-hide* (*ev a # t*) (*ev ' S*) @ *u* ∧
(*isInfHiddenRun* (λ*i*. *ev a # f i*) (*Mprefix A P*) *S* ∧
ev a # t ∈ *range* (λ*i*. *ev a # f i*))›
by (*auto simp add*: ∗(*1*, *2*, *3*) ∗∗(*2*, *3*) *monotone-on-def T-Mprefix*)
show ‹(*s*, *X*) ∈ *F* (*Mprefix A P* \ *S*)›
unfolding *F-Hiding* **using** ∗∗(*1*) ‹*?this*› **by** *blast*
qed
next
fix *t* **assume** ∗∗ : ‹*s'* = *trace-hide t* (*ev ' S*)› ‹(*t*, *X* ∪ *ev ' S*) ∈ *F* (*P a*)›
have ‹*s* = *trace-hide* (*ev a # t*) (*ev ' S*) ∧ (*ev a # t*, *X* ∪ *ev ' S*) ∈ *F* (*Mprefix A P*)›
by (*simp add*: ∗(*1*, *2*, *3*) ∗∗ *F-Mprefix image-iff*)
show ‹(*s*, *X*) ∈ *F* (*Mprefix A P* \ *S*)›
unfolding *F-Hiding* **using** ‹*?this*› **by** *blast*
qed
qed

(\) and *Mprefix* for disjoint Sets

theorem *Hiding-Mprefix-disjoint*:
‹□*a* ∈ *A* → *P a* \ *S* = □*a* ∈ *A* → (*P a* \ *S*)›
(**is** ‹*?lhs* = *?rhs*›) **if** *disjoint*: ‹*A* ∩ *S* = {}›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ *D ?lhs*›
then obtain *t u*
where ∗ : ‹*ftF u*› ‹*tF t*› ‹*s* = *trace-hide t* (*ev ' S*) @ *u*›
‹*t* ∈ *D* (*Mprefix A P*) ∨
(∃ *f*. *isInfHiddenRun f* (*Mprefix A P*) *S* ∧ *t* ∈ *range f*)›
by (*simp add*: *D-Hiding*) *blast*
from ∗(*4*) **show** ‹*s* ∈ *D ?rhs*›
proof (*elim disjE*)
assume ‹*t* ∈ *D* (*Mprefix A P*)›
then obtain *a t'* **where** ∗∗ : ‹*a* ∈ *A*› ‹*a* ∉ *S*› ‹*t* = *ev a # t'*› ‹*t'* ∈ *D* (*P a*)›
by (*simp add*: *D-Mprefix*) (*metis disjoint-iff disjoint*)
have ‹*ftF u* ∧ *tF t'* ∧ *trace-hide t'* (*ev ' S*) @ *u* = *trace-hide t'* (*ev ' S*) @ *u* ∧ *t'* ∈ *D* (*P a*)›
apply (*simp add*: ∗(*1*) ∗∗(*4*))
using ∗(*2*) ∗∗(*3*) *tickFree-Cons-iff* **by** *blast*
show ‹*s* ∈ *D ?rhs*›
apply (*simp add*: *D-Mprefix* ∗(*3*) ∗∗(*1*, *2*, *3*) *image-iff*[*of* ‹*ev* -›] *D-Hiding*)
using ‹*?this*› **by** *blast*
next
assume ‹∃*f*. *isInfHiddenRun f* (*Mprefix A P*) *S* ∧ *t* ∈ *range f*›
then obtain *f* **where** ∗∗ : ‹*isInfHiddenRun f* (*Mprefix A P*) *S*›
‹*t* ∈ *range f*› **by** *blast*
from ∗∗(*1*) *T-Mprefix* **obtain** *a*
where ∗∗∗ : ‹*a* ∈ *A*› ‹*a* ∉ *S*› ‹*f* (*Suc 0*) ≠ []› ‹*hd* (*f* (*Suc 0*)) = *ev a*›
by (*simp add*: *T-Mprefix*)

```
      (metis disjoint disjoint-iff list.sel(1) nil-less strict-mono-Suc-iff)
   from **(1)[THEN conjunct2, THEN conjunct2, rule-format, of 1]
    **(1)[simplified isInfHiddenRun-1] ***(1, 4) disjoint
   have **** : ‹f j ≠ [] ∧ hd (f j) = ev a› for j
    using **(1)[THEN conjunct2, THEN conjunct1, rule-format, of j]
    apply (cases ‹f 1›; simp add: T-Mprefix ***(3) split: if-split-asm)
    by (metis Nil-is-append-conv filter.simps(1)
       hd-append2 list.distinct(1) list.sel(1)) blast
   then obtain t' where ‹t = ev a # t'›
    by (metis **(2) list.exhaust-sel rangeE)
   hence ‹tF t' ∧ trace-hide t' (ev ‘ S) @ u = trace-hide t' (ev ‘ S) @ u ∧
        isInfHiddenRun (λi. tl (f i)) (P a) S ∧ t' ∈ range (λi. tl (f i))›
    apply (simp, intro conjI)
    using *(2) tickFree-Cons-iff apply blast
    apply (meson **(1) **** less-tail strict-mono-Suc-iff)
    using **(1) apply (simp add: T-Mprefix ****)
    apply (metis **** event_ptick.inject(1) list.sel(1, 3))
    apply (subst (1 2) list.collapse[OF ****[THEN conjunct1], symmetric])
     apply (metis (no-types, lifting) **(1) **** filter.simps(2) list.exhaust-sel
list.sel(3))
    by (metis (no-types, lifting) **(2) image-iff list.sel(3))
   show ‹s ∈ 𝒟 ?rhs›
    apply (simp add: D-Mprefix *(3) image-iff[of ‹ev a›] ***(1, 2) D-Hiding ‹t
= ev a # t'›)
    using *(1) ‹?this› by blast
  qed
next
  show ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› for s
   by (simp add: D-Hiding-Mprefix-dir2 Diff-triv that)
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 𝒟 ?lhs›
   | t where ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
   by (simp add: F-Hiding D-Hiding) blast
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
   from same-div D-F show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
  next
   fix t assume * : ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A
P)›
   show ‹(s, X) ∈ ℱ ?rhs›
   proof (cases ‹t = []›)
    from * show ‹t = [] ⟹ (s, X) ∈ ℱ ?rhs›
     by (auto simp add: F-Mprefix)
   next
    assume ‹t ≠ []›
    with *(2) disjoint obtain a t'
```

```
      where ** : ‹a ∈ A› ‹a ∉ S› ‹t = ev a # t'› ‹(t', X ∪ ev ‘ S) ∈ ℱ (P a)›
      by (cases t, auto simp add: F-Mprefix)
    show ‹(s, X) ∈ ℱ ?rhs›
      apply (simp add: F-Mprefix *(1) **(1, 2, 3) image-iff[of ‹ev a›])
      by (simp add: F-Hiding, rule disjI1, auto simp add: **(4))
  qed
 qed
next
 from disjoint show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› for s X
  apply (cases ‹s = []›)
  apply (simp add: F-Mprefix F-Hiding disjoint-iff,
      metis event_ptick.inject(1) filter.simps(1) imageE)
  by (simp add: Diff-triv F-Hiding-Mprefix-dir2)
qed
```

(\) and *Mprefix* for non disjoint Sets

```
theorem Hiding-Mprefix-non-disjoint:
 — Rework this proof
 ‹□a ∈ A → P a \ S = (□a ∈ (A − S) → (P a \ S)) ▷ (⊓a ∈ (A ∩ S). (P a \ S))›
 (is ‹?lhs = ?rhs›) if non-disjoint: ‹A ∩ S ≠ {}›
proof (subst Process-eq-spec-optimized, safe)
 fix s
 assume ‹s ∈ 𝒟 ?lhs›
 then obtain t u
  where * : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ S) @ u›
   ‹t ∈ 𝒟 (Mprefix A P) ∨ (∃ f. isInfHiddenRun f (Mprefix A P) S ∧ t ∈ range f)›
  by (simp add: D-Hiding) blast
 from *(4) show ‹s ∈ 𝒟 ?rhs›
 proof (elim disjE)
  assume ‹t ∈ 𝒟 (Mprefix A P)›
  then obtain a t' where ** : ‹a ∈ A› ‹t = ev a # t'› ‹t' ∈ 𝒟 (P a)›
   by (auto simp add: D-Mprefix)
  have ‹tF t' ∧ (if a ∈ S then s else tl s) = trace-hide t' (ev ‘ S) @ u ∧
      (t' ∈ 𝒟 (P a) ∨ (∃f. isInfHiddenRun f (P a) S ∧ t' ∈ range f))›
   using *(2) **(2, 3) by (auto simp add: *(1, 3) **(2) image-iff)
  with *(1) have *** : ‹(if a ∈ S then s else tl s) ∈ 𝒟 (P a \ S)›
   by (simp add: D-Hiding) blast
  show ‹s ∈ 𝒟 ?rhs›
  proof (cases ‹a ∈ S›)
   assume ‹a ∈ S›
   hence ‹a ∈ A ∩ S› by (simp add: **(1))
   with *** show ‹s ∈ 𝒟 ?rhs›
    by (auto simp add: D-Sliding D-GlobalNdet)
  next
   assume ‹a ∉ S›
   hence ‹a ∈ A − S› by (simp add: **(1))
```

```
    with *** show ‹s ∈ 𝒟 ?rhs›
      by (auto simp add: D-Sliding D-Mprefix *(3) **(2) image-iff)
  qed
next
  assume ‹∃ f. isInfHiddenRun f (Mprefix A P) S ∧ t ∈ range f›
  then obtain f
    where ** : ‹isInfHiddenRun f (Mprefix A P) S› ‹t ∈ range f› by blast
  obtain k where ‹t = f k› using **(2) by blast
  show ‹s ∈ 𝒟 ?rhs›
  proof (cases ‹f 0 = []›)
    assume ‹f 0 = []›
    hence ‹f 1 ≠ []›
      by (metis **(1) One-nat-def monotoneD nil-less zero-less-Suc)
    with **(1)[THEN conjunct2, THEN conjunct1, rule-format, of 1]
    obtain a where *** : ‹a ∈ A› ‹f 1 ≠ []› ‹hd (f 1) = ev a›
      by (auto simp add: T-Mprefix)
    have **** : ‹0 < j ⟹ f j ≠ [] ∧ hd (f j) = ev a› for j
    proof (induct j rule: nat-induct-non-zero)
      from ***(2, 3) show ‹f 1 ≠ [] ∧ hd (f 1) = ev a› by blast
    next
      case (Suc j)
      have ‹j < Suc j› by simp
      from **(1)[THEN conjunct1, THEN strict-monoD, OF this]
      obtain v where ‹f (Suc j) = f j @ v› by (meson strict-prefixE')
      thus ?case by (simp add: Suc.hyps(2))
    qed
    from ***(1) have *****: ‹a ∈ A ∩ S›
      by simp (metis (no-types, lifting) **(1) ***(2, 3)
          ‹f 0 = []› empty-filter-conv event_ptick.inject(1)
          filter.simps(1) image-iff list.set-sel(1))
    have ‹(if i = 0 ∧ t = [] then [] else tl (f (Suc i))) ∈ 𝒯 (P a)› for i
    proof −
      from **(1) ****[of ‹Suc i›] have ‹tl (f (Suc i)) ∈ 𝒯 (P a)›
        by (auto simp add: T-Mprefix)
      thus ‹(if i = 0 ∧ t = [] then [] else tl (f (Suc i))) ∈ 𝒯 (P a)› by simp
    qed
    hence ****** :
      ‹ftF u ∧ tF (tl t) ∧ s = trace-hide (tl t) (ev ‘ S) @ u ∧
       isInfHiddenRun (λi. if i = 0 ∧ t = [] then [] else tl (f (Suc i))) (P a) S ∧
       tl t ∈ range (λi. if i = 0 ∧ t = [] then [] else tl (f (Suc i)))›
      apply (intro conjI)
      apply (use *(1) in blast)
      apply (metis *(2) tickFree-tl)
      apply (metis *(3) **(1) ‹f 0 = []› ‹t = f k› empty-filter-conv
          filter.simps(1) list.sel(2) list.set-sel(2))
      apply (simp add: monotone-on-def,
          metis **(1) strict-prefix-simps(1) Suc-less-eq less-tail
          nil-le nless-le not-less-less-Suc-eq strict-monoD)
      apply blast
```

```
      apply (simp, metis **(1) **** ‹f 0 = []› empty-filter-conv
          filter.simps(1) list.set-sel(2) zero-less-Suc)
      by (simp add: image-iff,
          metis Suc-pred ‹f 0 = []› ‹t = f k› bot-nat-0.not-eq-extremum)
    have ‹a ∈ A ∩ S ∧ s ∈ 𝒟 (P a \ S)›
      apply (simp add: D-Hiding *****[simplified])
      using ****** by blast
    hence ‹s ∈ 𝒟 (⊓a ∈ (A ∩ S). (P a \ S))› by (simp add: D-GlobalNdet) blast
    thus ‹s ∈ 𝒟 ?rhs› by (simp add: D-Sliding)
  next
    assume ‹f 0 ≠ []›
    with **(1)[THEN conjunct2, THEN conjunct1, rule-format, of 0]
    obtain a where *** : ‹a ∈ A› ‹f 0 ≠ []› ‹hd (f 0) = ev a›
      by (auto simp add: T-Mprefix)
    have **** : ‹f j ≠ [] ∧ hd (f j) = ev a› for j
    proof (induct j)
      from ***(2, 3) show ‹f 0 ≠ [] ∧ hd (f 0) = ev a› by blast
    next
      case (Suc j)
      have ‹j < Suc j› by simp
      from **(1)[THEN conjunct1, THEN strict-monoD, OF this]
      obtain v where ‹f (Suc j) = f j @ v› by (meson strict-prefixE')
      thus ?case by (simp add: Suc.hyps(1))
    qed
    show ‹s ∈ 𝒟 ?rhs›
    proof (cases ‹a ∈ S›)
      assume ‹a ∈ S›
      hence ‹a ∈ A ∩ S› by (simp add: ***(1))
      have ‹tF (tl t) ∧ s = trace-hide (tl t) (ev ‘ S) @ u ∧
          isInfHiddenRun (λi. tl (f i)) (P a) S ∧ tl t ∈ range (λi. tl (f i))›
        apply (simp add: *(3), intro conjI)
        apply (metis *(2) tickFree-tl)
        apply (cases t; simp; metis **** ‹a ∈ S› ‹t = f k› image-iff list.sel(1))
        apply (meson **(1) **** less-tail strict-mono-Suc-iff)
        using **(1) apply (simp add: T-Mprefix ****)
        apply (metis **** event_ptick.inject(1) list.sel(1, 3))
        apply (metis (no-types, lifting) **(1) **** filter.simps(2) list.exhaust-sel
list.sel(3))
        using **(2) by blast
      have ‹s ∈ 𝒟 (⊓a ∈ (A ∩ S). (P a \ S))›
        apply (simp add: D-GlobalNdet D-Hiding)
        using *(1) ‹a ∈ A ∩ S› ‹?this› by blast
      thus ‹s ∈ 𝒟 ?rhs› by (simp add: D-Sliding)
    next
      assume ‹a ∉ S›
      have ‹tF (tl t) ∧ trace-hide (tl t) (ev ‘ S) @ u = trace-hide (tl t) (ev ‘ S)
@ u ∧
          isInfHiddenRun (λi. tl (f i)) (P a) S ∧ tl t ∈ range (λi. tl (f i))›
        apply (simp add: *(3), intro conjI)
```

```
        apply (metis ∗(2) tickFree-tl)
        apply (meson ∗∗(1) ∗∗∗∗ less-tail strict-mono-Suc-iff)
        using ∗∗(1) apply (simp add: T-Mprefix ∗∗∗∗)
        apply (metis ∗∗∗∗ event_ptick.inject(1) list.sel(1, 3))
        apply (metis (no-types, lifting) ∗∗(1) ∗∗∗∗ filter.simps(2) list.exhaust-sel
list.sel(3))
        using ∗∗(2) by blast
      from ‹a ∉ S› have ‹s ∈ 𝒟 (□a ∈ (A − S) → (P a \ S))›
        apply (simp add: D-Mprefix ∗(3) ‹t = f k›)
        using ∗∗∗(1) ∗∗∗∗[of k]
        apply (cases ‹f k›; simp add: ‹a ∉ S› image-iff[of ‹ev a›] D-Hiding)
        using ‹?this› by (metis ∗(1) ‹t = f k› list.sel(3))
      thus ‹s ∈ 𝒟 ?rhs› by (simp add: D-Sliding)
    qed
  qed
 qed
next
 fix s
 assume ‹s ∈ 𝒟 ?rhs›
 then consider ‹s ∈ 𝒟 (□a ∈ (A − S) → (P a \ S))›
  | ‹s ∈ 𝒟 (⊓a ∈ (A ∩ S). (P a \ S))› by (simp add: D-Sliding) blast
 thus ‹s ∈ 𝒟 ?lhs›
 proof cases
  show ‹s ∈ 𝒟 (□a ∈ (A − S) → (P a \ S)) ⟹ s ∈ 𝒟 ?lhs›
   by (rule D-Hiding-Mprefix-dir2)
 next
  assume ‹s ∈ 𝒟 (⊓a ∈ (A ∩ S). (P a \ S))›
  then obtain a where ∗ : ‹a ∈ A› ‹a ∈ S› ‹s ∈ 𝒟 (P a \ S)›
   by (auto simp add: D-GlobalNdet)
  from ∗(3) obtain t u
   where ∗∗ : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ S) @ u›
    ‹t ∈ 𝒟 (P a) ∨ (∃ f. isInfHiddenRun f (P a) S ∧ t ∈ range f)›
   by (simp add: D-Hiding) blast
  from ∗∗(4) show ‹s ∈ 𝒟 ?lhs›
  proof (elim disjE)
   assume ‹t ∈ 𝒟 (P a)›
   hence $ : ‹tF (ev a # t) ∧ ev a # t ∈ 𝒟 (Mprefix A P) ∧
            s = trace-hide (ev a # t) (ev ‘ S) @ u›
    by (simp add: ∗(1, 2) ∗∗(2, 3) D-Mprefix)
   show ‹s ∈ 𝒟 ?lhs›
    apply (simp add: D-Hiding)
    using $ ∗∗(1) by blast
  next
   assume ‹∃f. isInfHiddenRun f (P a) S ∧ t ∈ range f›
   then obtain f where ‹isInfHiddenRun f (P a) S› ‹t ∈ range f› by blast
   hence $ : ‹tF (ev a # t) ∧
            s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
            isInfHiddenRun (λi. ev a # f i) (Mprefix A P) S ∧
            ev a # t ∈ range (λi. ev a # f i)›
```

```
      by (auto simp add: ∗(1, 2) ∗∗(2, 3) monotone-on-def T-Mprefix)
    show ‹s ∈ 𝒟 ?lhs›
      apply (simp add: D-Hiding)
      using $ ∗∗(1) by blast
  qed
 qed
next
 fix s X
 assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
 assume ‹(s, X) ∈ ℱ ?lhs›
 then consider ‹s ∈ 𝒟 ?lhs›
  | t where ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
  by (simp add: F-Hiding D-Hiding) blast
 thus ‹(s, X) ∈ ℱ ?rhs›
 proof cases
  from D-F same-div show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
 next
  fix t assume ∗ : ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A
P)›
  from ∗(2) consider ‹t = []› ‹(X ∪ ev ‘ S) ∩ ev ‘ A = {}›
   | a t′ where ‹a ∈ A› ‹t = ev a # t′› ‹(t′, X ∪ ev ‘ S) ∈ ℱ (P a)›
   by (auto simp add: F-Mprefix)
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
   show ‹t = [] ⟹ (X ∪ ev ‘ S) ∩ ev ‘ A = {} ⟹ (s, X) ∈ ℱ ?rhs›
    using ∗(1) by (auto simp add: F-Sliding F-GlobalNdet)
  next
   fix a t′ assume ∗∗ : ‹a ∈ A› ‹t = ev a # t′› ‹(t′, X ∪ ev ‘ S) ∈ ℱ (P a)›
   show ‹(s, X) ∈ ℱ ?rhs›
   proof (cases ‹a ∈ S›)
    show ‹a ∈ S ⟹ (s, X) ∈ ℱ ?rhs›
     using ∗(1) ∗∗ by (auto simp add: F-Sliding F-GlobalNdet F-Hiding)
   next
    show ‹a ∉ S ⟹ (s, X) ∈ ℱ ?rhs›
     using ∗(1) ∗∗(1, 2, 3) by (auto simp add: F-Sliding F-Mprefix F-Hiding)
   qed
  qed
 qed
next
 fix s X
 assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
 assume ‹(s, X) ∈ ℱ ?rhs›
 then consider ‹(s, X) ∈ ℱ (⊓a ∈ (A ∩ S). (P a \ S))›
  | ‹s ≠ []› ‹(s, X) ∈ ℱ (□a ∈ (A − S) → (P a \ S))›
  by (auto simp add: F-Sliding)
 thus ‹(s, X) ∈ ℱ ?lhs›
 proof cases
  assume ‹(s, X) ∈ ℱ (⊓a ∈ (A ∩ S). (P a \ S))›
  then obtain a where ∗ : ‹a ∈ A› ‹a ∈ S› ‹(s, X) ∈ ℱ (P a \ S)›
```

```
    by (simp add: F-GlobalNdet non-disjoint) blast
  from *(3) consider ‹s ∈ 𝒟 (P a \ S)›
    | t where ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (P a)›
    by (simp add: F-Hiding D-Hiding) blast
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    assume ‹s ∈ 𝒟 (P a \ S)›
    then obtain t u
      where ** : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ S) @ u›
        ‹t ∈ 𝒟 (P a) ∨ (∃ f. isInfHiddenRun f (P a) S ∧ t ∈ range f)›
      by (simp add: D-Hiding) blast
    from **(4) show ‹(s, X) ∈ ℱ ?lhs›
    proof (elim disjE)
      assume ‹t ∈ 𝒟 (P a)›
      hence $ : ‹tF (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
                ev a # t ∈ 𝒟 (Mprefix A P)›
        by (simp add: D-Mprefix *(1, 2) **(2, 3) image-iff[of ‹ev a›])
      show ‹(s, X) ∈ ℱ ?lhs›
        apply (simp add: F-Hiding)
        using $ **(1) by blast
    next
      assume ‹∃f. isInfHiddenRun f (P a) S ∧ t ∈ range f›
      then obtain f where ‹isInfHiddenRun f (P a) S› ‹t ∈ range f› by blast
      hence $ : ‹tF (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
                isInfHiddenRun (λi. ev a # f i) (Mprefix A P) S ∧
                ev a # t ∈ range (λi. ev a # f i)›
        by (simp add: T-Mprefix *(1, 2) **(2, 3)
            image-iff[of ‹ev a›] monotone-on-def) blast
      show ‹(s, X) ∈ ℱ ?lhs›
        apply (simp add: F-Hiding)
        using $ **(1) by blast
    qed
  next
    fix t assume ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (P a)›
    hence ‹s = trace-hide (ev a # t) (ev ‘ S) ∧
          (ev a # t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
      by (simp add: *(1, 2) F-Mprefix)
    show ‹(s, X) ∈ ℱ ?lhs›
      apply (simp add: F-Hiding, rule disjI1)
      using ‹?this› by blast
  qed
next
  show ‹s ≠ [] ⟹ (s, X) ∈ ℱ (□a ∈ (A − S) → (P a \ S)) ⟹ (s, X) ∈ ℱ
?lhs›
    by (rule F-Hiding-Mprefix-dir2)
qed
qed

end
```

12.3.6 Synchronization

lemma *Mprefix-Sync-Mprefix-bis* :
‹*Mprefix* $(A \cup A')\ P\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q =$
$(\Box a{\in}A \rightarrow (P\ a\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q))$
$\Box\ (\Box b{\in}B \rightarrow (\mathit{Mprefix}\ (A \cup A')\ P\ [\![S]\!]\ Q\ b))$
$\Box\ (\Box x{\in}(A' \cap B') \rightarrow (P\ x\ [\![S]\!]\ Q\ x))$›
(**is** ‹*?lhs A A' P B B' Q* = *?rhs A A' P B B' Q*›)
if *sets-assms*: ‹$A \cap S = \{\}$› ‹$A' \subseteq S$› ‹$B \cap S = \{\}$› ‹$B' \subseteq S$›
for *P Q* :: ‹$'a \Rightarrow ('a, 'r)$ $process_{ptick}$›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s X*
assume ‹$(s, X) \in \mathcal{F}$ (*?lhs A A' P B B' Q*)›
and *same-div* : ‹$\mathcal{D}$ (*?lhs A A' P B B' Q*) = $\mathcal{D}$ (*?rhs A A' P B B' Q*)›
from *this*(*1*) **consider** (*fail*) *s-P s-Q X-P X-Q*
where ‹(*s-P*, *X-P*) $\in \mathcal{F}$ (*Mprefix* $(A \cup A')\ P$)› ‹(*s-Q*, *X-Q*) $\in \mathcal{F}$ (*Mprefix* $(B \cup B')\ Q$)›
‹*s setinterleaves* ((*s-P*, *s-Q*), *range tick* $\cup$ (*ev* ‘ *S*))›
‹$X = (X\text{-}P \cup X\text{-}Q) \cap (\mathit{range\ tick} \cup (\mathit{ev}\ `\ S)) \cup X\text{-}P \cap X\text{-}Q$›
| ‹$s \in \mathcal{D}$ (*?lhs A A' P B B' Q*)›
by (*simp add*: *F-Sync D-Sync*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ (*?rhs A A' P B B' Q*)›
proof *cases*
case *fail*
show ‹$(s, X) \in \mathcal{F}$ (*?rhs A A' P B B' Q*)›
proof (*cases* ‹$\exists r.\ s = [] \vee \mathit{hd}\ s = \mathit{tick}\ r$›)
case *True*
with *fail*(*1*, *2*, *3*) **have** ** : ‹$s = [] \wedge s\text{-}P = [] \wedge s\text{-}Q = []$›
by (*cases s-P*; *cases s-Q*; *force simp add*: *F-Mprefix subset-iff split*: *if-split-asm*)
with *fail*(*1*, *2*, *4*) *sets-assms*(*1*, *3*) **show** ‹$(s, X) \in \mathcal{F}$ (*?rhs A A' P B B' Q*)›
by (*auto simp add*: *subset-iff F-Det D-Det T-Det F-Mprefix image-Un*)
next
case *False*
then obtain *e* **where** *** : ‹$s \neq []$› ‹*hd s* = *ev e*› **by** (*meson* $event_{ptick}$.*exhaust*)
from *fail*(*1*, *2*, *3*) *** *sets-assms* **consider**
‹$e \in A \wedge s\text{-}P \neq [] \wedge \mathit{hd}\ s\text{-}P = \mathit{ev}\ e \wedge (\mathit{tl}\ s\text{-}P, X\text{-}P) \in \mathcal{F}\ (P\ e) \wedge$
tl s setinterleaves ((*tl s-P*, *s-Q*), *range tick* $\cup$ (*ev* ‘ *S*))› |
‹$e \in B \wedge s\text{-}Q \neq [] \wedge \mathit{hd}\ s\text{-}Q = \mathit{ev}\ e \wedge (\mathit{tl}\ s\text{-}Q, X\text{-}Q) \in \mathcal{F}\ (Q\ e) \wedge$
tl s setinterleaves ((*s-P*, *tl s-Q*), *range tick* $\cup$ (*ev* ‘ *S*))› |
‹$e \in A' \wedge e \in B' \wedge s\text{-}P \neq [] \wedge s\text{-}Q \neq [] \wedge \mathit{hd}\ s\text{-}P = \mathit{ev}\ e \wedge \mathit{hd}\ s\text{-}Q = \mathit{ev}\ e \wedge$
$(\mathit{tl}\ s\text{-}P, X\text{-}P) \in \mathcal{F}\ (P\ e) \wedge$
$(\mathit{tl}\ s\text{-}Q, X\text{-}Q) \in \mathcal{F}\ (Q\ e) \wedge \mathit{tl}\ s\ \mathit{setinterleaves}\ ((\mathit{tl}\ s\text{-}P, \mathit{tl}\ s\text{-}Q), \mathit{range\ tick} \cup (\mathit{ev}\ `\ S))$›
by (*cases s-P*; *cases s-Q*) (*simp-all add*: *F-Mprefix split*: *if-split-asm*, *safe*, *auto*)
thus ‹$(s, X) \in \mathcal{F}$ (*?rhs A A' P B B' Q*)›
apply *cases*
apply (*simp-all add*: *F-Det* ***(*1*))
apply (*rule disjI1*, *simp add*: *F-Mprefix* ***, *simp add*: *F-Sync*)
apply (*rule exI*[*of* - *e*], *rule exI*[*of* - ‹*tl s*›], *simp*, *intro conjI*)

```
      apply (metis ***(1) ***(2) list.exhaust-sel)
      using fail(2, 4) apply blast
     apply (rule disjI2, rule disjI1, simp add: F-Mprefix ***, simp add: F-Sync)
      apply (rule exI[of - e], rule exI[of - ‹tl s›], simp, intro conjI)
      apply (metis ***(1) ***(2) list.exhaust-sel)
      using fail(1, 4) apply blast

      apply (rule disjI2, rule disjI2, simp add: F-Mprefix *** F-Sync)
      apply (rule exI[of - e], rule exI[of - ‹tl s›], simp, intro conjI)
      apply (metis ***(1) ***(2) list.exhaust-sel)
      using fail(4) by blast
   qed
  next
   show ‹s ∈ 𝒟 (?lhs A A′ P B B′ Q) ⟹ (s, X) ∈ ℱ (?rhs A A′ P B B′ Q)›
    using same-div D-F by blast
  qed
next

  fix s X
  { fix P Q A A′ B B′
   assume assms : ‹s ≠ []› ‹(s, X) ∈ ℱ (□x ∈ A → (P x ⟦S⟧ Mprefix (B ∪ B′)
Q))›
     ‹A ∩ S = {}› ‹A′ ⊆ S› ‹B ∩ S = {}› ‹B′ ⊆ S›
    and same-div : ‹𝒟 (?lhs A A′ P B B′ Q) = 𝒟 (?rhs A A′ P B B′ Q)›
   from assms(1, 2) obtain e
     where * : ‹e ∈ A› ‹hd s = ev e› ‹(tl s, X) ∈ ℱ (P e ⟦S⟧ Mprefix (B ∪ B′)
Q)›
     by (auto simp add: F-Mprefix)
   from * assms(1) consider (fail) s-P s-Q X-P X-Q
     where ‹(s-P, X-P) ∈ ℱ (P e)› ‹(s-Q, X-Q) ∈ ℱ (Mprefix (B ∪ B′) Q)›
       ‹tl s setinterleaves ((s-P, s-Q), range tick ∪ (ev ‘ S))›
       ‹X = (X-P ∪ X-Q) ∩ (range tick ∪ (ev ‘ S)) ∪ X-P ∩ X-Q›
     | ‹s ∈ 𝒟 (□x ∈ A → (P x ⟦S⟧ Mprefix (B ∪ B′) Q))›
     by (simp add: F-Sync D-Mprefix D-Sync) (metis (no-types) list.collapse)
   hence ‹(s, X) ∈ ℱ (?lhs A A′ P B B′ Q)›
   proof cases
     case fail
     show ‹(s, X) ∈ ℱ (?lhs A A′ P B B′ Q)›
       apply (simp add: F-Sync, rule disjI1)
       apply (rule exI[of - ‹ev e # s-P›], rule exI[of - s-Q],
           rule exI[of - X-P], intro conjI)
       apply (simp add: F-Mprefix image-iff *(1) fail(1))
       apply (rule exI[of - X-Q], simp add: fail(2, 4))
       apply (subst list.collapse[OF assms(1), symmetric])
       using *(1, 2) fail(3) assms(3) by (cases s-Q) auto
   next
     assume ‹s ∈ 𝒟 (□x∈A → (P x ⟦S⟧ Mprefix (B ∪ B′) Q))›
     hence ‹s ∈ 𝒟 (?lhs A A′ P B B′ Q)› by (simp add: same-div D-Det)
     thus ‹(s, X) ∈ ℱ (?lhs A A′ P B B′ Q)› using D-F by blast
```

```
  qed
 } note ∗ = this

 { assume assms : ‹s ≠ []› ‹(s, X) ∈ ℱ (□x∈(A' ∩ B') → (P x ⟦S⟧ Q x))›
    and same-div : ‹𝒟 (?lhs A A' P B B' Q) = 𝒟 (?rhs A A' P B B' Q)›
  then obtain e where ∗ : ‹e ∈ A'› ‹e ∈ B'› ‹hd s = ev e› ‹(tl s, X) ∈ ℱ (P e
⟦S⟧ Q e)›
    by (auto simp add: F-Mprefix)
  have inside: ‹e ∈ S› using ∗(2) that(4) by blast
  from ∗ assms(1) consider (fail) s-P s-Q X-P X-Q where
    ‹(s-P, X-P) ∈ ℱ (P e)› ‹(s-Q, X-Q) ∈ ℱ (Q e)›
    ‹tl s setinterleaves ((s-P, s-Q), range tick ∪ (ev ‘ S))›
    ‹X = (X-P ∪ X-Q) ∩ (range tick ∪ (ev ‘ S)) ∪ X-P ∩ X-Q›
  | ‹s ∈ 𝒟 (□x∈(A' ∩ B') → (P x ⟦S⟧ Q x))›
    by (simp add: F-Sync D-Mprefix D-Sync) (metis (no-types) list.collapse)
  hence ‹(s, X) ∈ ℱ (?lhs A A' P B B' Q)›
  proof cases
    case fail
    show ‹(s, X) ∈ ℱ (?lhs A A' P B B' Q)›
     apply (subst list.collapse[OF assms(1), symmetric], simp add: F-Sync ∗(3),
rule disjI1)
      apply (rule exI[of - ‹ev e # s-P›], rule exI[of - ‹ev e # s-Q›])
      apply (rule exI[of - X-P], intro conjI, simp add: F-Mprefix ∗(1) fail(1))
       by (rule exI[of - X-Q]) (use fail(3) in ‹simp add: ∗(2) fail(2, 4) inside
F-Mprefix›)
  next
    assume ‹s ∈ 𝒟 (□x∈(A' ∩ B') → (P x ⟦S⟧ Q x))›
    hence ‹s ∈ 𝒟 (?lhs A A' P B B' Q)› by (simp add: same-div D-Det)
    thus ‹(s, X) ∈ ℱ (?lhs A A' P B B' Q)› using D-F by blast
  qed
 } note ∗∗ = this

 have ∗∗∗ : ‹X ∩ ev ‘ (A' ∩ B') = {} ⟹ X ∩ ev ‘ A = {} ⟹ X ∩ ev ‘ B = {}
⟹
          ([], X) ∈ ℱ (?lhs A A' P B B' Q)›
  apply (simp add: F-Sync, rule disjI1)
  apply (rule exI[of - ‹[]›], rule exI[of - ‹[]›], simp add: F-Mprefix)
  apply (rule exI[of - ‹X − (ev ‘ (A' − B'))›], intro conjI, fastforce)
  apply (rule exI[of - ‹X − (ev ‘ (B' − A'))›], intro conjI, fastforce)
  using sets-assms(2, 4) by auto

 assume same-div : ‹𝒟 (?lhs A A' P B B' Q) = 𝒟 (?rhs A A' P B B' Q)›
 hence same-div-sym : ‹𝒟 (?lhs B B' Q A A' P) = 𝒟 (?rhs B B' Q A A' P)›
   by (subst Sync-commute, subst Int-commute, subst Det-commute, simp add:
Sync-commute)
 show ‹(s, X) ∈ ℱ (?rhs A A' P B B' Q) ⟹ (s, X) ∈ ℱ (?lhs A A' P B B' Q)›
  apply (unfold F-Det, safe, simp-all)
  apply (rule ∗∗∗; simp add: F-Mprefix)
  apply (rule ∗; simp add: sets-assms same-div)
```

```
    apply (subst (asm) Sync-commute, subst Sync-commute)
    apply (rule *; simp add: sets-assms same-div-sym)
    apply (erule **, assumption, rule same-div)
    apply (simp add: D-Mprefix D-Det)[1]
    by (simp add: T-Mprefix T-Det)
next

  { fix s t u r v and P Q :: ‹'a ⇒ ('a, 'r) process_ptick› and A A' B B'
    assume assms : ‹ftF v› ‹tF r ∨ v = []› ‹s = r @ v›
      ‹r setinterleaves ((t, u), range tick ∪ (ev ‘ S))›
      ‹t ∈ 𝒟 (Mprefix (A ∪ A') P)› ‹u ∈ 𝒯 (Mprefix (B ∪ B') Q)›
      ‹A ∩ S = {}› ‹A' ⊆ S› ‹B ∩ S = {}› ‹B' ⊆ S›
    from assms(5) obtain e where * : ‹t ≠ []› ‹hd t = ev e› ‹tl t ∈ 𝒟 (P e)› ‹e
∈ A ∨ e ∈ A'›
      by (force simp add: D-Mprefix)
    have nonNil: ‹r ≠ [] ∧ s ≠ []›
      using *(1) assms(3, 4) empty-setinterleaving by blast
    have ‹s ∈ 𝒟 (?rhs A A' P B B' Q)›
    proof (cases ‹u = []›)
      case True
        hence ‹hd s = ev e› by (metis *(2) EmptyLeftSync setinterleaving-sym
assms(3, 4) hd-append nonNil)
      also from *(1, 2, 4) setinterleaving-sym assms(4, 8)[simplified True] have
‹e ∈ A›
        using emptyLeftNonSync hd-in-set by fastforce
      ultimately show ‹s ∈ 𝒟 (?rhs A A' P B B' Q)›
        apply (simp add: D-Det)
        apply (rule disjI1, simp add: D-Mprefix nonNil D-Sync)
        apply (rule exI[of - e], rule exI[of - ‹tl s›], simp, intro conjI)
        apply (metis list.collapse nonNil)
        apply (rule exI[of - ‹tl t›], rule exI[of - ‹[]›],
            rule exI[of - ‹tl r›], rule exI[of - v])
        apply (auto simp add: assms(1, 3, 6)[simplified True] * nonNil)[1]
        apply (metis assms(2) tickFree-tl)
        using setinterleaving-sym SyncTlEmpty True assms(4) by blast
    next
      case False
      with assms(6) obtain e' where ** : ‹hd u = ev e'› ‹tl u ∈ 𝒯 (Q e')› ‹e' ∈
B ∨ e' ∈ B'›
        by (auto simp add: T-Mprefix)
      consider ‹e  ∈ A ∧ hd s = ev e  ∧ tl r setinterleaves ((tl t, u), range tick ∪
ev ‘ S)› |
        ‹e' ∈ B ∧ hd s = ev e' ∧ tl r setinterleaves ((t, tl u), range tick ∪ ev ‘ S)› |
        ‹e = e' ∧ e ∈ A' ∧ e ∈ B' ∧ hd s = ev e ∧
              tl r setinterleaves ((tl t, tl u), range tick ∪ ev ‘ S)›
        using assms(4)
        apply (subst (asm) list.collapse[OF nonNil[THEN conjunct1], symmetric],
            subst (asm) list.collapse[OF *(1), symmetric, simplified *(2)],
            subst (asm) list.collapse[OF False, symmetric, simplified **(1)])
```

```
      apply (simp add: assms(3) nonNil image-iff split: if-split-asm)
      apply (metis (no-types, opaque-lifting) *(4) **(3) Int-iff
          assms(7, 9) empty-iff list.sel(1, 3))
      apply (metis (no-types, opaque-lifting) *(1, 2) **(3) Int-iff
          assms(10) inf.order-iff list.exhaust-sel list.sel(1, 3))
      apply (metis (no-types, opaque-lifting) *(1, 2, 4) **(1, 3) Int-iff
          False assms(8, 10) inf.order-iff list.exhaust-sel list.sel(1, 3))
      by (metis (no-types, opaque-lifting) *(4) **(1) False assms(8)
          list.collapse list.sel(1, 3) subsetD)
    thus ‹s ∈ 𝒟 (?rhs A A' P B B' Q)›
      apply cases
      apply (simp-all add: D-Det)
      apply (rule disjI1, simp add: D-Mprefix nonNil D-Sync)
      apply (rule exI[of - e], rule exI[of - ‹tl s›], simp, intro conjI)
      apply (metis list.exhaust-sel nonNil)
      apply (rule exI[of - ‹tl t›], rule exI[of - u],
          rule exI[of - ‹tl r›], rule exI[of - v])
    apply (simp add: assms(1, 3, 6) *(3) nonNil, use assms(2) nonNil tickFree-tl
in blast)
      apply (rule disjI2, rule disjI1, simp add: D-Mprefix nonNil D-Sync)
      apply (rule exI[of - e'], rule exI[of - ‹tl s›], simp, intro conjI)
      apply (metis list.exhaust-sel nonNil)
      apply (rule exI[of - t], rule exI[of - ‹tl u›],
          rule exI[of - ‹tl r›], rule exI[of - v])
       apply (metis * **(2) assms(1, 2, 3) list.exhaust-sel nonNil tickFree-tl
tl-append2)
      apply (rule disjI2, rule disjI2, auto simp add: D-Mprefix nonNil image-iff)
      apply (simp add: D-Sync)
      apply (rule exI[of - e'], rule exI[of - ‹tl s›], simp, intro conjI)
      apply (metis list.collapse nonNil)
      apply (rule exI[of - ‹tl t›], rule exI[of - ‹tl u›],
          rule exI[of - ‹tl r›], rule exI[of - v])
       by (use *(3) **(2) assms(1, 2, 3) nonNil tickFree-tl in ‹auto simp add:
nonNil›)
  qed
 } note * = this

 fix s
 assume ‹s ∈ 𝒟 (?lhs A A' P B B' Q)›
 then obtain t u r v
   where ** : ‹ftF v› ‹tF r ∨ v = []› ‹s = r @ v›
    ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
    ‹t ∈ 𝒟 (Mprefix (A ∪ A') P) ∧ u ∈ 𝒯 (Mprefix (B ∪ B') Q) ∨
          t ∈ 𝒟 (Mprefix (B ∪ B') Q) ∧ u ∈ 𝒯 (Mprefix (A ∪ A') P)›
   by (simp add: D-Sync) blast
 have same-div : ‹𝒟 (?rhs A A' P B B' Q) = 𝒟 (?rhs B B' Q A A' P)›
   by (subst Det-commute, subst Int-commute, simp add: Sync-commute)
 from **(5) show ‹s ∈ 𝒟 (?rhs A A' P B B' Q)›
   apply (rule disjE)
```

```
    by (rule *[OF **(1, 2, 3, 4)]; simp add: sets-assms)
      (subst same-div, rule *[OF **(1, 2, 3, 4)]; simp add: sets-assms)
next

  { fix A A' B B' P Q s
    assume set-assm : ‹A ∩ S = {}›
    have ‹s ∈ 𝒟 (?lhs A A' P B B' Q)› if ‹s ∈ 𝒟 (□x∈A → (P x ⟦S⟧ Mprefix (B
∪ B') Q))›
    proof −
      from that obtain e s' where assms: ‹s = ev e # s'› ‹e ∈ A› ‹s' ∈ 𝒟 (P e
⟦S⟧ Mprefix (B ∪ B') Q)›
        by (simp add: D-Mprefix) blast
      from assms(3) obtain t u r v
        where * : ‹ftF v› ‹tF r ∨ v = []› ‹s' = r @ v›
          ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
          ‹t ∈ 𝒟 (P e) ∧ u ∈ 𝒯 (Mprefix (B ∪ B') Q) ∨
                t ∈ 𝒟 (Mprefix (B ∪ B') Q) ∧ u ∈ 𝒯 (P e)› by (simp add: D-Sync)
blast
      have notin: ‹e ∉ S› using assms(2) set-assm by blast
      show ‹s ∈ 𝒟 (?lhs A A' P B B' Q)›
        apply (simp add: assms(1) D-Sync)
        using *(5) apply (elim disjE)

        apply (rule exI[of - ‹ev e # t›], rule exI[of - u],
            rule exI[of - ‹ev e # r›], rule exI[of - v])
        apply (simp add: *(1, 2, 3, 4))
        apply (cases u; simp add: notin image-iff T-Mprefix D-Mprefix)
        using *(4) assms(2) apply (blast+)[2]

        apply (rule exI[of - t], rule exI[of - ‹ev e # u›],
            rule exI[of - ‹ev e # r›], rule exI[of - v])
        apply (simp add: *(1, 2, 3, 4))
        apply (cases t; simp add: notin image-iff T-Mprefix D-Mprefix)
        using *(4) assms(2) by blast
    qed
  } note * = this

  show ‹s ∈ 𝒟 (?rhs A A' P B B' Q) ⟹ s ∈ 𝒟 (?lhs A A' P B B' Q)› for s
    apply (simp add: D-Det)
    apply (erule disjE, rule *, simp-all add: sets-assms(1))
     apply (erule disjE, subst Sync-commute, rule *, simp-all add: sets-assms(3)
Sync-commute)
    subgoal
    proof −
      assume ‹s ∈ 𝒟 (□x ∈ (A' ∩ B') → (P x ⟦S⟧ Q x))›
      then obtain e t u r v
        where * : ‹e ∈ A'› ‹e ∈ B'› ‹s ≠ []› ‹hd s = ev e› ‹ftF v› ‹tF r ∨ v = []›
          ‹tl s = r @ v› ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
          ‹t ∈ 𝒟 (P e) ∧ u ∈ 𝒯 (Q e) ∨ t ∈ 𝒟 (Q e) ∧ u ∈ 𝒯 (P e)›
```

```
      by (simp add: D-Mprefix D-Sync) force
    have inside: ‹e ∈ S› using ∗(1) sets-assms(2) by blast
    show ‹s ∈ 𝒟 (?lhs A A' P B B' Q)›
      apply (subst list.collapse[OF ∗(3), symmetric], simp add: D-Sync)
      apply (rule exI[of - ‹ev e # t›], rule exI[of - ‹ev e # u›],
          rule exI[of - ‹ev e # r›], rule exI[of - v])
      by (simp add: ∗ inside D-Mprefix T-Mprefix image-iff)
  qed .
qed

corollary Mprefix-Sync-Mprefix:
  — This version is easier to use.
  ‹□a∈A → P a ⟦S⟧ □b∈B → Q b =
   (□a∈(A − S) → (P a ⟦S⟧ □b∈B → Q b)) □
   (□b∈(B − S) → (□a∈A → P a ⟦S⟧ Q b)) □
   (□x∈(A ∩ B ∩ S) → (P x ⟦S⟧ Q x))›
  by (subst Mprefix-Sync-Mprefix-bis[of ‹A − S› S ‹A ∩ S› ‹B − S› ‹B ∩ S›,
simplified Un-Diff-Int])
    (simp-all add: Int-commute inf-left-commute)
```

Renaming

```
lemma Renaming-Mprefix:
  ‹Renaming (□a ∈ A → P a) f g =
   □y ∈ f ‘ A → ⊓a ∈ {x ∈ A. y = f x}. Renaming (P a) f g› (is ‹?lhs = ?rhs›)
proof (subst Process-eq-spec-optimized, safe)
  show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs› for s
    by (auto simp add: D-Renaming D-Mprefix D-GlobalNdet)
      (use list.map-sel(2) tickFree-tl in blast)
next
  fix s
  assume ‹s ∈ 𝒟 ?rhs›
  then obtain a s' where ∗ : ‹a ∈ A› ‹s = map-event_ptick f g (ev a) # s'›
    ‹s' ∈ 𝒟 (Renaming (P a) f g)›
    by (auto simp add: D-Mprefix D-GlobalNdet split: if-split-asm)
  from ∗(3) obtain s1 s2
    where ∗∗ : ‹tF s1› ‹ftF s2›
      ‹s' = map (map-event_ptick f g) s1 @ s2› ‹s1 ∈ 𝒟 (P a)›
    by (simp add: D-Renaming) blast
  have ∗∗∗ : ‹tF (ev a # s1) ∧
            s = map-event_ptick f g (ev a) # map (map-event_ptick f g) s1 @ s2 ∧
ev a # s1 ∈ 𝒟 (Mprefix A P)›
    by (simp add: D-Mprefix ∗(1, 2) ∗∗(1, 3, 4))
  show ‹s ∈ 𝒟 ?lhs›
    by (simp add: D-Renaming)
      (metis ∗∗(2) ∗∗∗ append-Cons list.simps(9))
next
  fix s X
  assume same-div: ‹𝒟 ?lhs = 𝒟 ?rhs›
```

assume ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
then consider ‹$s \in \mathcal{D}$ *?lhs*›
| t **where** ‹$(t,\ \mathit{map\text{-}event}_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ (\mathit{Mprefix}\ A\ P)$› ‹$s = \mathit{map}\ (\mathit{map\text{-}event}_{ptick}\ f\ g)\ t$›
by (*simp add*: *F-Renaming D-Renaming*) *blast*
thus ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
proof *cases*
from *D-F same-div* **show** ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow (s,\ X) \in \mathcal{F}$ *?rhs*› **by** *blast*
next
fix t **assume** $*$: ‹$(t,\ \mathit{map\text{-}event}_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ (\mathit{Mprefix}\ A\ P)$›
‹$s = \mathit{map}\ (\mathit{map\text{-}event}_{ptick}\ f\ g)\ t$›
show ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
proof (*cases* ‹$t = []$›)
from $*$ **show** ‹$t = [] \Longrightarrow (s,\ X) \in \mathcal{F}$ *?rhs*›
by (*auto simp add*: *F-Mprefix disjoint-iff-not-equal*)
next
assume ‹$t \neq []$›
with $*(1)$ **obtain** $a\ t'$
where $**$: ‹$a \in A$› ‹$t = ev\ a\ \#\ t'$› ‹$(t',\ \mathit{map\text{-}event}_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ (P\ a)$›
by (*auto simp add*: *F-Mprefix*)
have $***$: ‹$s \neq [] \wedge \mathit{hd}\ s \in ev\ `\ f\ `\ A$›
using $*(2)$ $**(1,\ 2)$ **by** *simp*
with $**$ **have** ‹$ev\ (f\ a) = \mathit{hd}\ s \wedge$
$(\mathit{tl}\ s,\ X) \in \mathcal{F}\ (\sqcap a \in \{x \in A.\ f\ a = f\ x\}.\ \mathit{Renaming}\ (P\ a)\ f\ g)$›
by (*auto simp add*: *F-GlobalNdet* $*(2)$ *F-Renaming*)
with $***$ **show** ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Mprefix image-iff*) (*metis* (*no-types*, *lifting*) $**(1)$ *list.collapse*)
qed
qed
next
show ‹$(s,\ X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (s,\ X) \in \mathcal{F}$ *?lhs*› **for** $s\ X$
proof (*cases* s)
show ‹$s = [] \Longrightarrow (s,\ X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (s,\ X) \in \mathcal{F}$ *?lhs*›
by (*auto simp add*: *F-Mprefix F-Renaming*)
next
fix $b\ s'$
assume ‹$s = b\ \#\ s'$› ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
then obtain a
where $*$: ‹$a \in A$› ‹$s = (\mathit{map\text{-}event}_{ptick}\ f\ g)\ (ev\ a)\ \#\ s'$›
‹$(s',\ X) \in \mathcal{F}\ (\sqcap a \in \{x \in A.\ f\ a = f\ x\}.\ \mathit{Renaming}\ (P\ a)\ f\ g)$›
by (*auto simp add*: *F-Mprefix*)
from $*(1,\ 3)$ **obtain** a'
where $**$: ‹$a' \in A$› ‹$f\ a' = f\ a$› ‹$(s',\ X) \in \mathcal{F}\ (\mathit{Renaming}\ (P\ a')\ f\ g)$›
by (*auto simp add*: *F-GlobalNdet split*: *if-split-asm*)
from $**(3)$ **consider**
t **where** ‹$(t,\ \mathit{map\text{-}event}_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ (P\ a')$› ‹$s' = \mathit{map}\ (\mathit{map\text{-}event}_{ptick}\ f\ g)\ t$›
| *s1 s2* **where** ‹*tF s1*› ‹*ftF s2*›

```
    ‹s' = map (map-event_ptick f g) s1 @ s2› ‹s1 ∈ 𝒟 (P a')›
    by (simp add: F-Renaming) blast
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    fix t assume *** : ‹(t, map-event_ptick f g −‘ X) ∈ ℱ (P a')›
      ‹s' = map (map-event_ptick f g) t›
    have **** : ‹(ev a' # t, map-event_ptick f g −‘ X) ∈ ℱ (Mprefix A P) ∧
                s = map (map-event_ptick f g) (ev a' # t)›
      by (auto simp add: F-Mprefix *(2) **(1, 2) ***(1, 2))
    with **** show ‹(s, X) ∈ ℱ ?lhs› by (auto simp add: F-Renaming)
  next
    fix s1 s2 assume *** : ‹tF s1› ‹ftF s2›
      ‹s' = map (map-event_ptick f g) s1 @ s2› ‹s1 ∈ 𝒟 (P a')›
    have ‹tF (ev a' # s1) ∧
          s = map-event_ptick f g (ev a') # map (map-event_ptick f g) s1 @ s2 ∧
          ev a' # s1 ∈ 𝒟 (Mprefix A P)›
      by (auto simp add: *(2) **(1, 2) *** D-Mprefix)
    with ***(2) show ‹(s, X) ∈ ℱ ?lhs› by (auto simp add: F-Renaming)
  qed
 qed
qed
```

12.4 Extension of the Step-Laws

12.4.1 Derived step-laws for *Sync*

lemma *Mprefix-Inter-Mprefix* :
‹□a∈A → P a ||| □b∈B → Q b = (□a∈A → (P a ||| □b∈B → Q b)) □ (□b∈B → (□a ∈ A → P a ||| Q b))›
by (*fact Mprefix-Sync-Mprefix*[**where** *S* = ‹{}›, *simplified*])

lemma *Mprefix-Par-Mprefix* : ‹□a∈A → P a || □b∈B → Q b = □x∈(A ∩ B) → (P x || Q x)›
by (*fact Mprefix-Sync-Mprefix*[**where** *S* = ‹*UNIV*›, *simplified*])

lemma *Mprefix-Sync-Mprefix-subset* :
‹⟦A ⊆ S; B ⊆ S⟧ ⟹ *Mprefix A P* ⟦S⟧ *Mprefix B Q* = □x∈(A ∩ B) → (P x ⟦S⟧ Q x)›
by (*fact Mprefix-Sync-Mprefix-bis*[*of* ‹{}› *S A* ‹{}› *B*, *simplified*])

lemma *Mprefix-Sync-Mprefix-indep* :
‹A ∩ S = {} ⟹ B ∩ S = {} ⟹
Mprefix A P ⟦S⟧ *Mprefix B Q* = (□a∈A → (P a ⟦S⟧ *Mprefix B Q*)) □ (□b∈B → (*Mprefix A P* ⟦S⟧ Q b))›
by (*fact Mprefix-Sync-Mprefix-bis*[*of A S* ‹{}› *B* ‹{}›, *simplified*])

lemma *Mprefix-Sync-Mprefix-left* :

‹$A \cap S = \{\} \Longrightarrow B \subseteq S \Longrightarrow$ *Mprefix A P* ⟦S⟧ *Mprefix B Q* = $\Box a{\in}A \rightarrow$ (*P a* ⟦S⟧ *Mprefix B Q*)›
by (*fact Mprefix-Sync-Mprefix-bis*[*of A S* ‹{}› ‹{}› *B*, *simplified*])

lemma *Mprefix-Sync-Mprefix-right* :
‹$A \subseteq S \Longrightarrow B \cap S = \{\} \Longrightarrow$ *Mprefix A P* ⟦S⟧ *Mprefix B Q* = $\Box b{\in}B \rightarrow$ (*Mprefix A P* ⟦S⟧ *Q b*)›
by (*fact Mprefix-Sync-Mprefix-bis*[*of* ‹{}› *S A B* ‹{}›, *simplified*])

lemma *Mprefix-Sync-STOP* : ‹($\Box a \in A \rightarrow P\ a$) ⟦$S$⟧ *STOP* = $\Box a \in (A - S) \rightarrow$ (*P a* ⟦S⟧ *STOP*)›
by (*subst Mprefix-empty*[*symmetric*], *subst Mprefix-Sync-Mprefix*, *simp*)

lemma *STOP-Sync-Mprefix* : ‹*STOP* ⟦S⟧ ($\Box b \in B \rightarrow Q\ b$) = $\Box b \in (B - S) \rightarrow$ (*STOP* ⟦S⟧ *Q b*)›
by (*metis* (*no-types*, *lifting*) *Mprefix-Sync-STOP Sync-commute mono-Mprefix-eq*)

Mixing deterministic and non deterministic prefix choices **lemma** *Mndetprefix-Sync-Mprefix* :
‹($\sqcap a \in A \rightarrow P\ a$) ⟦$S$⟧ ($\Box b \in B \rightarrow Q\ b$) =
(*if* $A = \{\}$ *then STOP* ⟦S⟧ ($\Box b \in B \rightarrow Q\ b$)
else $\sqcap a{\in}A$. (*if* $a \in S$ *then STOP else* ($a \rightarrow$ (*P a* ⟦S⟧ ($\Box b \in B \rightarrow Q\ b$)))) $\Box$
($\Box b{\in}(B - S) \rightarrow$ (($a \rightarrow P\ a$) ⟦S⟧ *Q b*)) $\Box$
(*if* $a \in B \cap S$ *then* ($a \rightarrow$ (*P a* ⟦S⟧ *Q a*)) *else STOP*))›
by (*unfold Mndetprefix-GlobalNdet Sync-distrib-GlobalNdet-right write0-def Mprefix-Sync-Mprefix*)
(*auto simp add*: *Mprefix-singl insert-Diff-if Int-insert-left intro*: *mono-GlobalNdet-eq arg-cong2*[**where** f = ‹($\Box$)›] *mono-Mprefix-eq*)

lemma *Mprefix-Sync-Mndetprefix* :
‹($\Box a \in A \rightarrow P\ a$) ⟦$S$⟧ ($\sqcap b \in B \rightarrow Q\ b$) =
(*if* $B = \{\}$ *then* ($\Box a \in A \rightarrow P\ a$) ⟦$S$⟧ *STOP*
else $\sqcap b{\in}B$. (*if* $b \in S$ *then STOP else* ($b \rightarrow$ (($\Box a \in A \rightarrow P\ a$) ⟦$S$⟧ *Q b*))) $\Box$
($\Box a{\in}(A - S) \rightarrow$ (*P a* ⟦S⟧ ($b \rightarrow Q\ b$))) $\Box$
(*if* $b \in A \cap S$ *then* ($b \rightarrow$ (*P b* ⟦S⟧ *Q b*)) *else STOP*))›
by (*subst* (*1 2 3 4 5*) *Sync-commute*) (*simp add*: *Mndetprefix-Sync-Mprefix*)

In particular, we can obtain the theorem for *Mndetprefix* synchronized with *STOP*.

lemma *Mndetprefix-Sync-STOP* :
‹($\sqcap a \in A \rightarrow P\ a$) ⟦$S$⟧ *STOP* =
(*if* $A \cap S = \{\}$ *then* $\sqcap a \in A \rightarrow$ (*P a* ⟦S⟧ *STOP*)
else ($\sqcap a \in (A - S) \rightarrow$ (*P a* ⟦S⟧ *STOP*)) $\sqcap$ *STOP*)›
(**is** ‹*?lhs* = (*if* $A \cap S = \{\}$ *then ?rhs1 else ?rhs2* $\sqcap$ *STOP*)›)
proof −
have ‹($\sqcap a \in A \rightarrow P\ a$) ⟦$S$⟧ *STOP* =
$\sqcap a{\in}A$. (*if* $a \in S$ *then STOP else* ($a \rightarrow$ (*P a* ⟦S⟧ *STOP*)))› (**is** ‹*?lhs* = *?rhs′*›)
by (*subst Mndetprefix-Sync-Mprefix*[**where** B = ‹{}›, *simplified*])

```
    (auto intro: mono-GlobalNdet-eq)
  also have ‹?rhs' = (if A ∩ S = {} then ?rhs1 else ?rhs2 ⊓ STOP)›
  proof (split if-split, intro conjI impI)
    show ‹A ∩ S = {} ⟹ ?rhs' = ?rhs1›
      by (auto simp add: Mndetprefix-GlobalNdet intro!: mono-GlobalNdet-eq)
  next
    show ‹?rhs' = ?rhs2 ⊓ STOP› if ‹A ∩ S ≠ {}›
    proof (cases ‹A − S = {}›)
      show ‹?rhs' = ?rhs2 ⊓ STOP› if ‹A − S = {}›
        by (simp add: ‹A − S = {}› GlobalNdet-is-STOP-iff)
          (use ‹A − S = {}› in blast)
    next
      show ‹?rhs' = ?rhs2 ⊓ STOP› if ‹A − S ≠ {}›
      proof (subst Int-Diff-Un[symmetric],
          subst GlobalNdet-factorization-union
          [OF ‹A ∩ S ≠ {}› ‹A − S ≠ {}›, symmetric])
        have ‹(⊓a∈(A ∩ S). (if a ∈ S then STOP else (a → (P a ⟦S⟧ STOP))))
            = STOP› (is ‹?fact1 = STOP›)
          by (simp add: GlobalNdet-is-STOP-iff)
        moreover have ‹(⊓a∈(A − S). (if a ∈ S then STOP else (a → (P a ⟦S⟧
STOP))))
                    = ?rhs2› (is ‹?fact2 = ?rhs2›)
          by (auto simp add: Mndetprefix-GlobalNdet intro: mono-GlobalNdet-eq)
        ultimately show ‹?fact1 ⊓ ?fact2 = ?rhs2 ⊓ STOP› by (simp add:
Ndet-commute)
      qed
    qed
  qed
  finally show ‹?lhs = (if A ∩ S = {} then ?rhs1 else ?rhs2 ⊓ STOP)› .
qed

corollary STOP-Sync-Mndetprefix :
  ‹STOP ⟦S⟧ (⊓b ∈ B → Q b) =
   ( if B ∩ S = {} then ⊓b ∈ B → (STOP ⟦S⟧ Q b)
    else (⊓b ∈ (B − S) → (STOP ⟦S⟧ Q b)) ⊓ STOP)›
  by (subst (1 2 3) Sync-commute) (simp add: Mndetprefix-Sync-STOP)



corollary Mndetprefix-Sync-Mprefix-subset :
  ‹(⊓a ∈ A → P a) ⟦S⟧ (□b ∈ B → Q b) =
   ( if A ⊆ B then ⊓ a ∈ A → (P a ⟦S⟧ Q a)
    else (⊓a ∈ (A ∩ B) → (P a ⟦S⟧ Q a)) ⊓ STOP)›
  (is ‹?lhs = (if A ⊆ B then ?rhs1 else ?rhs2)›) if ‹A ⊆ S› ‹B ⊆ S›
proof (cases ‹A = {}›)
  show ‹A = {} ⟹ ?lhs = (if A ⊆ B then ?rhs1 else ?rhs2)›
    by (simp add: Mprefix-is-STOP-iff STOP-Sync-Mprefix ‹B ⊆ S›)
next
  from ‹A ⊆ S› have ∗ : ‹a ∈ A ⟹ a ∈ S› for a by blast
```

from ‹$B \subseteq S$› **have** $**$: ‹$B - S = \{\}$› ‹$a \in B \wedge a \in S \longleftrightarrow a \in B$› **for** a **by** *auto*
assume ‹$A \neq \{\}$›
have ‹*?lhs* = $\sqcap a{\in}A.$ (*if* $a \in B$ *then* $(a \rightarrow (P\ a\ [\![S]\!]\ Q\ a))$ *else* $STOP$)› (**is** ‹*?lhs* = *?rhs'*›)
by (*auto simp add*: *Mndetprefix-Sync-Mprefix* $*$ $**$ ‹$A \neq \{\}$› *intro*: *mono-GlobalNdet-eq*)
also have ‹*?rhs'* = (*if* $A \subseteq B$ *then ?rhs1 else ?rhs2*)›
proof (*split if-split*, *intro conjI impI*)
show ‹$A \subseteq B \Longrightarrow$ *?rhs'* = $\sqcap a{\in}A \rightarrow (P\ a\ [\![S]\!]\ Q\ a)$›
by (*auto simp add*: *Mndetprefix-GlobalNdet intro*!: *mono-GlobalNdet-eq*)
next
show ‹*?rhs'* = $(\sqcap a{\in}(A \cap B) \rightarrow (P\ a\ [\![S]\!]\ Q\ a)) \sqcap STOP$› **if** ‹$\neg\ A \subseteq B$›
proof (*cases* ‹$A \cap B = \{\}$›)
show ‹$A \cap B = \{\} \Longrightarrow$ *?rhs'* = $(\sqcap a{\in}(A \cap B) \rightarrow (P\ a\ [\![S]\!]\ Q\ a)) \sqcap STOP$›
by (*auto simp add*: *GlobalNdet-is-STOP-iff*)
next
assume ‹$A \cap B \neq \{\}$›
from ‹$\neg\ A \subseteq B$› **have** ‹$A - B \neq \{\}$› **by** *blast*
show ‹*?rhs'* = $(\sqcap a{\in}(A \cap B) \rightarrow (P\ a\ [\![S]\!]\ Q\ a)) \sqcap STOP$›
by (*auto simp add*: *Mndetprefix-GlobalNdet GlobalNdet-is-STOP-iff*
simp flip: *GlobalNdet-factorization-union*
[*OF* ‹$A \cap B \neq \{\}$› ‹$A - B \neq \{\}$›, *unfolded Int-Diff-Un*]
intro!: *arg-cong2*[**where** f = ‹$(\sqcap)$›] *mono-GlobalNdet-eq*)
qed
qed
finally show ‹*?lhs* = (*if* $A \subseteq B$ *then ?rhs1 else ?rhs2*)› **by** *simp*
qed

corollary *Mprefix-Sync-Mndetprefix-subset* :
‹$A \subseteq S \Longrightarrow B \subseteq S \Longrightarrow$
$\Box a \in A \rightarrow P\ a\ [\![S]\!]\ \sqcap b \in B \rightarrow Q\ b =$
(*if* $B \subseteq A$ *then* $\sqcap b \in B \rightarrow (P\ b\ [\![S]\!]\ Q\ b)$
else $(\sqcap b \in (A \cap B) \rightarrow (P\ b\ [\![S]\!]\ Q\ b)) \sqcap STOP$)›
by (*subst* (*1 2 3*) *Sync-commute*) (*simp add*: *Mndetprefix-Sync-Mprefix-subset Int-commute*)

corollary *Mndetprefix-Sync-Mprefix-indep* :
‹$(\sqcap a \in A \rightarrow P\ a)\ [\![S]\!]\ (\Box b \in B \rightarrow Q\ b) =$
(*if* $A = \{\}$ *then* $\Box b{\in}B \rightarrow (STOP\ [\![S]\!]\ Q\ b)$
else $\sqcap a{\in}A.\ (a \rightarrow (P\ a\ [\![S]\!]\ (\Box b \in B \rightarrow Q\ b))) \Box$
$(\Box b{\in}B \rightarrow ((a \rightarrow P\ a)\ [\![S]\!]\ Q\ b)))$›
if ‹$A \cap S = \{\}$› **and** ‹$B \cap S = \{\}$›
proof (*cases* ‹$A = \{\}$›)
show ‹$A = \{\} \Longrightarrow$ *?thesis*›
by (*simp add*: *Diff-triv STOP-Sync-Mprefix* ‹$B \cap S = \{\}$›)
next

from *that(1)* **have** $*$: ‹$a \in A \Longrightarrow a \notin S$› **for** a **by** *blast*
from *that(2)* **have** $**$: ‹$a \in B \wedge a \in S \longleftrightarrow False$› **for** a **by** *blast*
from *that(2)* **have** $***$: ‹$B - S = B$› **by** *blast*
show *?thesis* **if** ‹$A \neq \{\}$›
by (*simp add*: *Mndetprefix-Sync-Mprefix* ‹$A \neq \{\}$›)
(*rule mono-GlobalNdet-eq*, *simp add*: $* ** ***$)
qed

corollary *Mprefix-Sync-Mndetprefix-indep* :
‹$A \cap S = \{\} \Longrightarrow B \cap S = \{\} \Longrightarrow$
$(\Box a \in A \rightarrow P\ a)\ [\![S]\!]\ (\sqcap b \in B \rightarrow Q\ b) =$
(*if* $B = \{\}$ *then* $\Box a \in A \rightarrow (P\ a\ [\![S]\!]\ STOP)$
else $\sqcap b \in B.\ (b \rightarrow ((\Box a \in A \rightarrow P\ a)\ [\![S]\!]\ Q\ b)) \Box$
$(\Box a \in A \rightarrow (P\ a\ [\![S]\!]\ (b \rightarrow Q\ b))))$›
by (*subst* (*1 2 3 4*) *Sync-commute*) (*simp add*: *Mndetprefix-Sync-Mprefix-indep*)

corollary *Mndetprefix-Sync-Mprefix-left* :
‹$(\sqcap a \in A \rightarrow P\ a)\ [\![S]\!]\ (\Box b \in B \rightarrow Q\ b) =$
(*if* $A = \{\}$ *then* $STOP\ [\![S]\!]\ (\Box b \in B \rightarrow Q\ b)$
else $\sqcap a \in A \rightarrow (P\ a\ [\![S]\!]\ (\Box b \in B \rightarrow Q\ b))$)›
if ‹$A \cap S = \{\}$› **and** ‹$B \subseteq S$›
proof (*cases* ‹$A = \{\}$›)
show ‹$A = \{\} \Longrightarrow$ *?thesis*› **by** *simp*
next
from *that(1)* **have** $*$: ‹$a \in A \Longrightarrow a \notin S$› **for** a **by** *blast*
from *that(2)* **have** $**$: ‹$B - S = \{\}$› **by** *blast*
show *?thesis* **if** ‹$A \neq \{\}$›
by (*simp add*: *Mndetprefix-Sync-Mprefix* ‹$A \neq \{\}$›, *unfold Mndetprefix-GlobalNdet*)
(*rule mono-GlobalNdet-eq*, *simp add*: $* **$)
qed

corollary *Mndetprefix-Sync-Mprefix-right* :
‹$(\sqcap a \in A \rightarrow P\ a)\ [\![S]\!]\ (\Box b \in B \rightarrow Q\ b) =$
(*if* $A = \{\}$ *then* $STOP\ [\![S]\!]\ (\Box b \in B \rightarrow Q\ b)$
else $\Box b \in B \rightarrow ((\sqcap a \in A \rightarrow P\ a)\ [\![S]\!]\ Q\ b)$)›
if ‹$A \subseteq S$› **and** ‹$B \cap S = \{\}$›
proof (*cases* ‹$A = \{\}$›)
show ‹$A = \{\} \Longrightarrow$ *?thesis*› **by** *simp*
next
from *that(1)* **have** $*$: ‹$a \in A \Longrightarrow a \in S$› **for** a **by** *blast*
from *that(2)* **have** $**$: ‹$B - S = B$› **by** *blast*
show *?thesis* **if** ‹$A \neq \{\}$›
by (*simp add*: *Mndetprefix-Sync-Mprefix* ‹$A \neq \{\}$›,
simp add: *Mndetprefix-GlobalNdet Sync-distrib-GlobalNdet-right* ‹$A \neq \{\}$›
flip: *GlobalNdet-Mprefix-distr*)
(*rule mono-GlobalNdet-eq*, *use* $* **$ **in** *auto*)
qed

corollary *Mprefix-Sync-Mndetprefix-left* :
‹$A \cap S = \{\} \implies B \subseteq S \implies$
$(\Box a \in A \rightarrow P\ a)\ [\![S]\!]\ (\sqcap b \in B \rightarrow Q\ b) =$
$($ *if* $B = \{\}$ *then* $(\Box a \in A \rightarrow P\ a)\ [\![S]\!]\ STOP$
else $\Box a \in A \rightarrow (P\ a\ [\![S]\!]\ (\sqcap b \in B \rightarrow Q\ b))$)›
by (*subst* (*1 2 3*) *Sync-commute*) (*simp add*: *Mndetprefix-Sync-Mprefix-right*)

corollary *Mprefix-Sync-Mndetprefix-right* :
‹$A \subseteq S \implies B \cap S = \{\} \implies$
$(\Box a \in A \rightarrow P\ a)\ [\![S]\!]\ (\sqcap b \in B \rightarrow Q\ b) =$
$($ *if* $B = \{\}$ *then* $(\Box a \in A \rightarrow P\ a)\ [\![S]\!]\ STOP$
else $\sqcap b \in B \rightarrow ((\Box a \in A \rightarrow P\ a)\ [\![S]\!]\ Q\ b)$)›
by (*subst* (*1 2 3*) *Sync-commute*) (*simp add*: *Mndetprefix-Sync-Mprefix-left*)

corollary *Mndetprefix-Par-Mprefix* :
‹$\sqcap a \in A \rightarrow P\ a \mid\mid \Box b \in B \rightarrow Q\ b =$
(*if* $A \subseteq B$ *then* $\sqcap a \in A \rightarrow (P\ a \mid\mid Q\ a)$ *else* $(\sqcap a \in (A \cap B) \rightarrow (P\ a \mid\mid Q\ a)) \sqcap STOP$)›
by (*simp add*: *Mndetprefix-Sync-Mprefix-subset*)

corollary *Mprefix-Par-Mndetprefix* :
‹$\Box a \in A \rightarrow P\ a \mid\mid \sqcap b \in B \rightarrow Q\ b =$
(*if* $B \subseteq A$ *then* $\sqcap b \in B \rightarrow (P\ b \mid\mid Q\ b)$ *else* $(\sqcap b \in (A \cap B) \rightarrow (P\ b \mid\mid Q\ b)) \sqcap STOP$)›
by (*simp add*: *Mprefix-Sync-Mndetprefix-subset*)

corollary *Mndetprefix-Inter-Mprefix* :
‹$(\sqcap a \in A \rightarrow P\ a) \mid\mid\mid (\Box b \in B \rightarrow Q\ b) =$
$($ *if* $A = \{\}$ *then* $(\Box b \in B \rightarrow Q\ b\ ;\ STOP)$
else $\sqcap a \in A.\ (a \rightarrow (P\ a \mid\mid\mid (\Box b \in B \rightarrow Q\ b))) \Box$
$(\Box b \in B \rightarrow ((a \rightarrow P\ a) \mid\mid\mid Q\ b))$)›
by (*simp add*: *Mndetprefix-Sync-Mprefix-indep Mprefix-Seq*)

corollary *Mprefix-Inter-Mndetprefix* :
‹$(\Box a \in A \rightarrow P\ a) \mid\mid\mid (\sqcap b \in B \rightarrow Q\ b) =$
$($ *if* $B = \{\}$ *then* $(\Box a \in A \rightarrow P\ a\ ;\ STOP)$
else $\sqcap b \in B.\ (b \rightarrow ((\Box a \in A \rightarrow P\ a) \mid\mid\mid Q\ b)) \Box$
$(\Box a \in A \rightarrow (P\ a \mid\mid\mid (b \rightarrow Q\ b)))$)›
by (*simp add*: *Mprefix-Sync-Mndetprefix-indep Mprefix-Seq*)

12.4.2 Non deterministic step-laws

The *Mndetprefix* operator slightly differs from *Mprefix* so we can often adapt the step-laws.

lemma *Mndetprefix-Ndet-Mndetprefix* :

— We assume $A \cap B \neq \{\}$, otherwise this rewriting rule is not interesting.
‹$(\sqcap a \in A \to P\ a) \sqcap (\sqcap b \in B \to Q\ b) =$
(*if* $A = B$ *then* $\sqcap b \in B \to P\ b \sqcap Q\ b$
else if $A \subseteq B$ *then* $(\sqcap b \in (B - A) \to Q\ b) \sqcap (\sqcap x \in A \to P\ x \sqcap Q\ x)$
else if $B \subseteq A$ *then* $(\sqcap a \in (A - B) \to P\ a) \sqcap (\sqcap x \in B \to P\ x \sqcap Q\ x)$
else $(\sqcap a \in (A - B) \to P\ a) \sqcap (\sqcap b \in (B - A) \to Q\ b) \sqcap (\sqcap x \in (A \cap B) \to P\ x \sqcap Q\ x)$)›
(**is** ‹*?lhs* = (*if* $A = B$ *then ?rhs1 else if* $A \subseteq B$ *then ?rhs2 else if* $B \subseteq A$ *then ?rhs3 else ?rhs4*)›)
if ‹$A \cap B \neq \{\}$›
proof (*split if-split*, *intro conjI impI*)
show ‹$A = B \Longrightarrow$ *?lhs* = *?rhs1*› **by** (*simp add*: *Mndetprefix-distrib-Ndet*)
next
show ‹*?lhs* = (*if* $A \subseteq B$ *then ?rhs2 else if* $B \subseteq A$ *then ?rhs3 else ?rhs4*)› **if** ‹$A \neq B$›
proof (*split if-split*, *intro conjI impI*)
from ‹$A \cap B \neq \{\}$› ‹$A \neq B$› **show** ‹$A \subseteq B \Longrightarrow$ *?lhs* = *?rhs2*›
by (*simp add*: *Process-eq-spec Ndet-projs STOP-projs Mndetprefix-projs*, *safe*, *auto*)
next
show ‹*?lhs* = (*if* $B \subseteq A$ *then ?rhs3 else ?rhs4*)› **if** ‹$\neg\ A \subseteq B$›
proof (*split if-split*, *intro conjI impI*)
from ‹$A \neq B$› **show** ‹$B \subseteq A \Longrightarrow$ *?lhs* = *?rhs3*›
by (*simp add*: *Process-eq-spec Ndet-projs STOP-projs Mndetprefix-projs*, *safe*, *auto*)
next
from ‹$A \cap B \neq \{\}$› ‹$\neg\ A \subseteq B$› **show** ‹$\neg\ B \subseteq A \Longrightarrow$ *?lhs* = *?rhs4*›
by (*simp add*: *Process-eq-spec Ndet-projs STOP-projs Mndetprefix-projs*, *safe*, *auto*)
qed
qed
qed

lemma *Mndetprefix-Det-Mndetprefix* :
‹$(\sqcap a \in A \to P\ a) \ \Box\ (\sqcap b \in B \to Q\ b) =$
(*if* $A = \{\}$ *then* $\sqcap b \in B \to Q\ b$ *else if* $B = \{\}$ *then* $\sqcap a \in A \to P\ a$
else $\sqcap a \in A.\ \sqcap b \in B.$ (*if* $a = b$ *then* $a \to P\ a \sqcap Q\ a$ *else* $(a \to P\ a) \ \Box\ (b \to Q\ b)$))›
(**is** ‹*?lhs* = (*if* $A = \{\}$ *then ?rhs1 else if* $B = \{\}$ *then ?rhs2 else ?rhs3*)›)
proof (*split if-split*, *intro conjI impI*)
show ‹$A = \{\} \Longrightarrow$ *?lhs* = *?rhs1*› **by** *simp*
next
show ‹*?lhs* = (*if* $B = \{\}$ *then ?rhs2 else ?rhs3*)› **if** ‹$A \neq \{\}$›
proof (*split if-split*, *intro conjI impI*)
show ‹$B = \{\} \Longrightarrow$ *?lhs* = *?rhs2*› **by** *simp*
next
from ‹$A \neq \{\}$› **show** ‹$B \neq \{\} \Longrightarrow$ *?lhs* = *?rhs3*›
by (*auto simp add*: *Mndetprefix-GlobalNdet Det-distrib-GlobalNdet-left Det-distrib-GlobalNdet-right GlobalNdet-sets-commute*[*of A*]

intro!: *mono-GlobalNdet-eq split*: *if-split-asm*)
(*simp add*: *Process-eq-spec Det-projs write0-def Mprefix-projs Ndet-projs*, *safe*, *auto*)
qed
qed

lemma *FD-Mndetprefix-Det-Mndetprefix* :
‹$\sqcap x{\in}(A \cup B) \rightarrow$ (*if* $x \in A \cap B$ *then* $P\ x \sqcap Q\ x$ *else if* $x \in A$ *then* $P\ x$ *else* $Q\ x$)
$\sqsubseteq_{FD}$ $(\sqcap a{\in}\ A \rightarrow P\ a) \ \Box\ (\sqcap b \in B \rightarrow Q\ b)$›
(**is** ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›)
proof (*rule failure-divergence-refineI*)
show ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-Det D-Ndet D-Mndetprefix'*)
next
show ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow$ $(s, X) \in \mathcal{F}$ *?lhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Det F-Ndet Mndetprefix-projs split*: *if-split-asm*)
qed

lemma *FD-Mndetprefix-Sliding-Mndetprefix* :
‹($\sqcap x \in (A \cup B) \rightarrow$ (*if* $x \in A \cap B$ *then* $P\ x \sqcap Q\ x$ *else if* $x \in A$ *then* $P\ x$ *else* $Q\ x$)) $\sqcap$
($\sqcap b \in B \rightarrow$ (*if* $b \in A$ *then* $P\ b \sqcap Q\ b$ *else* $Q\ b$)) $\sqsubseteq_{FD}$
$(\sqcap a{\in}\ A \rightarrow P\ a) \triangleright (\sqcap b \in B \rightarrow Q\ b)$› (**is** ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›)
proof (*rule failure-divergence-refineI*)
show ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-Sliding D-Ndet D-Mndetprefix'*)
next
show ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow$ $(s, X) \in \mathcal{F}$ *?lhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Sliding F-Ndet Mndetprefix-projs split*: *if-split-asm*)
qed

lemma *Mndetprefix-Sliding-superset-Mndetprefix* :
‹$(\sqcap a{\in}\ A \rightarrow P\ a) \triangleright (\sqcap b \in B \rightarrow Q\ b) =$
$\sqcap b \in B \rightarrow$ (*if* $b \in A$ *then* $P\ b \sqcap Q\ b$ *else* $Q\ b$)›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹$A \subseteq B$›
proof (*subst Process-eq-spec*, *safe*)
from ‹$A \subseteq B$› **show** ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?rhs*› **for** *s*
by (*auto simp add*: *D-Sliding D-Ndet D-Mndetprefix'*)
next
show ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-Sliding D-Ndet D-Mndetprefix' split*: *if-split-asm*)
next
from ‹$A \subseteq B$› **show** ‹$(s, X) \in \mathcal{F}$ *?lhs* $\Longrightarrow$ $(s, X) \in \mathcal{F}$ *?rhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Sliding F-Ndet Mndetprefix-projs split*: *if-split-asm*)
next
show ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow$ $(s, X) \in \mathcal{F}$ *?lhs*› **for** *s X*

by (*cases s*) (*auto simp add*: *F-Sliding F-Ndet Mndetprefix-projs split*: *if-split-asm*)
qed

corollary *Mndetprefix-Sliding-same-set-Mndetprefix* :
‹(⊓*a*∈ *A* → *P a*) ▷ (⊓*a* ∈ *A* → *Q a*) = ⊓*a* ∈ *A* → *P a* ⊓ *Q a*›
by (*simp add*: *Mndetprefix-Sliding-superset-Mndetprefix*)
(*rule mono-Mndetprefix-eq*, *simp*)

lemma *Renaming-Mndetprefix* :
‹*Renaming* (⊓*a* ∈ *A* → *P a*) *f g* = ⊓*b* ∈ *f* ‘ *A* → ⊓*a* ∈ {*a* ∈ *A*. *b* = *f a*}. *Renaming* (*P a*) *f g*›
proof −
have ∗ : ‹*A* = (⋃ *b* ∈ *f* ‘ *A*. {*a* ∈ *A*. *b* = *f a*})› **by** *auto*
have ∗∗ : ‹{*b* ∈ *f* ‘ *A*. {*a* ∈ *A*. *b* = *f a*} ≠ {}} = *f* ‘ *A*› **by** *auto*
have ‹*Renaming* (⊓*a* ∈ *A* → *P a*) *f g* = ⊓*a* ∈ *A*. (*f a* → *Renaming* (*P a*) *f g*)›
by (*auto simp add*: *Mndetprefix-GlobalNdet Renaming-distrib-GlobalNdet write0-def Renaming-Mprefix*
intro!: *mono-GlobalNdet-eq mono-Mprefix-eq*)
also have ‹. . . = ⊓*b* ∈ *f* ‘ *A*. ⊓*a* ∈ {*a* ∈ *A*. *b* = *f a*}. (*f a* → *Renaming* (*P a*) *f g*)›
by (*subst* ∗, *subst GlobalNdet-Union*, *subst* ∗∗, *fact refl*)
also have ‹. . . = ⊓*b* ∈ *f* ‘ *A*. (*b* → (⊓*a* ∈ {*a* ∈ *A*. *b* = *f a*}. *Renaming* (*P a*) *f g*))›
by (*rule mono-GlobalNdet-eq*, *subst write0-GlobalNdet*)
(*auto intro*: *mono-GlobalNdet-eq*)
also have ‹. . . = ⊓*b* ∈ *f* ‘ *A* → ⊓*a* ∈ {*a* ∈ *A*. *b* = *f a*}. *Renaming* (*P a*) *f g*›
by (*simp add*: *Mndetprefix-GlobalNdet*)
finally show ‹*Renaming* (⊓*a* ∈ *A* → *P a*) *f g* = . . .› **.**
qed

lemma *Mndetprefix-Seq* : ‹⊓*a* ∈ *A* → *P a* **;** *Q* = ⊓*a* ∈ *A* → (*P a* **;** *Q*)›
by (*simp add*: *Mndetprefix-GlobalNdet Seq-distrib-GlobalNdet-right*)
(*rule mono-GlobalNdet-eq*, *simp add*: *write0-def Mprefix-Seq*)

For (\), we can not use the distributivity of *GlobalNdet* since it is only working for finite non determinism. But we can still reuse some previous results in the following proof.

theorem *Hiding-Mndetprefix-disjoint* :
‹⊓*a* ∈ *A* → *P a* \ *S* = ⊓*a* ∈ *A* → (*P a* \ *S*)› (**is** ‹*?lhs* = *?rhs*›) **if** ‹*A* ∩ *S* = {}›
proof (*subst Process-eq-spec-optimized*, *safe*)
have ‹*D ?lhs* = *D* (□*a* ∈ *A* → *P a* \ *S*)›
by (*simp add*: *D-Hiding-seqRun Mndetprefix-projs Mprefix-projs*)
also have ‹. . . = *D* (□*a* ∈ *A* → (*P a* \ *S*))›
by (*simp add*: *Hiding-Mprefix-disjoint* ‹*A* ∩ *S* = {}›)

```
  also have ‹... = 𝒟 ?rhs›
    by (simp add: D-Mndetprefix' D-Mprefix)
  finally show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs›
    and ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› for s by simp-all
next
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  fix s X assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 𝒟 ?lhs›
    | t where ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (□a ∈ A → P a)›
    unfolding F-Hiding-seqRun D-Hiding-seqRun by blast
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
    from same-div D-F show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
  next
    fix t assume ∗ : ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (□a ∈ A →
P a)›
    show ‹(s, X) ∈ ℱ ?rhs›
    proof (cases ‹t = []›)
      from ∗ show ‹t = [] ⟹ (s, X) ∈ ℱ ?rhs›
        by (auto simp add: F-Mndetprefix' split: if-split-asm)
    next
      assume ‹t ≠ []›
      with ∗(2) have ‹(t, X ∪ ev ‘ S) ∈ ℱ (□a ∈ A → P a)›
        by (simp add: F-Mprefix F-Mndetprefix' split: if-split-asm)
      with ∗(1) have ‹(s, X) ∈ ℱ (□a ∈ A → P a \ S)›
        unfolding F-Hiding by blast
      hence ‹(s, X) ∈ ℱ (□a ∈ A → (P a \ S))›
        by (simp add: Hiding-Mprefix-disjoint ‹A ∩ S = {}›)
      thus ‹(s, X) ∈ ℱ ?rhs› by (auto simp add: F-Mprefix F-Mndetprefix')
    qed
  qed
next
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  show ‹(s, X) ∈ ℱ ?lhs› if ‹(s, X) ∈ ℱ ?rhs› for s X
  proof (cases ‹s = []›)
    assume ‹s = []›
    with ‹(s, X) ∈ ℱ ?rhs› have ‹A = {} ∨ (∃ a ∈ A. ev a ∉ X)›
      by (simp add: F-Mndetprefix' split: if-split-asm)
    with ‹A ∩ S = {}› show ‹s = [] ⟹ (s, X) ∈ ℱ ?lhs›
      by (simp add: Mndetprefix-projs F-Hiding-seqRun disjoint-iff)
        (metis (no-types, lifting) event_ptick.inject(1) filter.simps(1) imageE)
  next
    assume ‹s ≠ []›
    with ‹(s, X) ∈ ℱ ?rhs› have ‹(s, X) ∈ ℱ (□a ∈ A → (P a \ S))›
      by (simp add: F-Mprefix F-Mndetprefix' split: if-split-asm)
    hence ‹(s, X) ∈ ℱ (□a ∈ A → P a \ S)›
      by (simp add: Hiding-Mprefix-disjoint ‹A ∩ S = {}›)
    then consider ‹s ∈ 𝒟 (□a ∈ A → P a \ S)›
      | t where ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (□a ∈ A → P a)›
```

```
    unfolding F-Hiding-seqRun D-Hiding-seqRun by blast
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    assume ‹s ∈ 𝒟 (□a ∈ A → P a \ S)›
    hence ‹s ∈ 𝒟 ?lhs› by (simp add: D-Hiding-seqRun Mprefix-projs Mndetpre-
fix-projs)
    with D-F show ‹(s, X) ∈ ℱ ?lhs› by blast
  next
    fix t assume ∗ : ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (□a ∈ A →
P a)›
    from ‹s ≠ []› ∗(1) filter.simps(1) have ‹t ≠ []› by blast
    with ∗ show ‹(s, X) ∈ ℱ ?lhs›
      by (simp add: F-Hiding F-Mprefix F-Mndetprefix′) blast
  qed
 qed
qed

theorem Hiding-Mndetprefix-subset :
 ‹⊓a ∈ A → P a \ S = ⊓a ∈ A. (P a \ S)› (is ‹?lhs = ?rhs›) if ‹A ⊆ S›
proof (cases ‹A = {}›)
 show ‹A = {} ⟹ ?lhs = ?rhs› by simp
next
 assume ‹A ≠ {}›
 hence ‹A ∩ S ≠ {}› by (simp add: inf-absorb1 ‹A ⊆ S›)
 show ‹?lhs = ?rhs›
 proof (subst Process-eq-spec-optimized, safe)
  have ‹𝒟 ?lhs = 𝒟 (□a ∈ A → P a \ S)›
    by (simp add: D-Hiding-seqRun Mndetprefix-projs Mprefix-projs)
  also have ‹... = 𝒟 ((□a ∈ (A − S) → (P a \ S)) ▷ (⊓a ∈ (A ∩ S). (P a \
S)))›
    by (simp add: Hiding-Mprefix-non-disjoint ‹A ∩ S ≠ {}›)
  also have ‹... = 𝒟 ?rhs›
    by (use ‹A ⊆ S› in ‹auto simp add: D-GlobalNdet D-Mprefix D-Sliding›)
  finally show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs›
    and ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› for s by simp-all
 next
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  fix s X assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 𝒟 ?lhs›
    | t where ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (⊓a ∈ A → P a)›
    unfolding F-Hiding D-Hiding by blast
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
    from same-div D-F show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
  next
    fix t assume ∗ : ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (⊓a ∈ A →
P a)›
    from ∗(2) ‹A ⊆ S› obtain a t′ where ‹a ∈ A› ‹t = ev a # t′› ‹(t′, X ∪ ev
‘ S) ∈ ℱ (P a)›
```

```
      by (auto simp add: F-Mndetprefix' ‹A ≠ {}› subset-iff)
    hence ‹(t, X ∪ ev ‘ S) ∈ ℱ (□a ∈ A → P a)› by (simp add: F-Mprefix)
    with ∗(1) have ‹(s, X) ∈ ℱ (□a ∈ A → P a \ S)›
      by (auto simp add: F-Hiding-seqRun)
    hence ‹(s, X) ∈ ℱ ((□a ∈ (A − S) → (P a \ S)) ▷ (⊓a ∈ (A ∩ S). (P a \
S)))›
      by (simp add: Hiding-Mprefix-non-disjoint ‹A ∩ S ≠ {}›)
    also have ‹(□a ∈ (A − S) → (P a \ S)) = STOP›
      by (simp add: Mprefix-is-STOP-iff ‹A ⊆ S›)
    also from ‹A ⊆ S› have ‹A ∩ S = A› by blast
    finally show ‹(s, X) ∈ ℱ (⊓a ∈ A. (P a \ S))› by simp
  qed
 next
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  fix s X assume ‹(s, X) ∈ ℱ ?rhs›
  then obtain a where ‹a ∈ A› ‹(s, X) ∈ ℱ (P a \ S)›
    by (auto simp add: F-GlobalNdet ‹A ≠ {}›)
  from ‹(s, X) ∈ ℱ (P a \ S)› consider ‹s ∈ 𝒟 (P a \ S)›
    | t where ‹s = trace-hide t (ev ‘ S)› ‹(t, X ∪ ev ‘ S) ∈ ℱ (P a)›
    unfolding F-Hiding D-Hiding by blast
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    assume ‹s ∈ 𝒟 (P a \ S)›
    with ‹a ∈ A› have ‹s ∈ 𝒟 ?rhs› by (auto simp add: D-GlobalNdet)
    with same-div D-F show ‹(s, X) ∈ ℱ ?lhs› by blast
  next
    from ‹a ∈ A› ‹A ⊆ S›
    show ‹s = trace-hide t (ev ‘ S) ⟹ (t, X ∪ ev ‘ S) ∈ ℱ (P a)
          ⟹ (s, X) ∈ ℱ ?lhs› for t
      by (simp add: F-Hiding F-Mndetprefix' ‹A ≠ {}› subset-iff)
        (metis (no-types, lifting) filter.simps(2) image-eqI)
  qed
 qed
qed

theorem Hiding-Mndetprefix-non-disjoint-not-subset :
 ‹⊓a ∈ A → P a \ S = (⊓a ∈ (A − S) → (P a \ S)) ⊓ (⊓a ∈ (A ∩ S). (P a \
S))›
 (is ‹?lhs = ?rhs1 ⊓ ?rhs2›) if ‹A ∩ S ≠ {}› and ‹¬ A ⊆ S›
proof −
 from ‹¬ A ⊆ S› have ‹A − S ≠ {}› by blast
 moreover have ‹A − S ∪ A ∩ S = A› by blast
 ultimately have ‹(⊓a ∈ A → P a) = (⊓a ∈ (A − S) → P a) ⊓ (⊓a ∈ (A ∩ S)
→ P a)›
   by (metis Mndetprefix-Un-distrib ‹A ∩ S ≠ {}›)
 hence ‹?lhs = (⊓a ∈ (A − S) → P a \ S) ⊓ (⊓a ∈ (A ∩ S) → P a \ S)›
   by (simp add: Hiding-distrib-Ndet)
 also have ‹⊓a ∈ (A − S) → P a \ S = ?rhs1›
   by (simp add: Hiding-Mndetprefix-disjoint inf.commute)
```

also have ‹$\sqcap a \in (A \cap S) \to P\ a \setminus S = ?rhs2$›
by (*simp add*: *Hiding-Mndetprefix-subset*)
finally show ‹*?lhs* = *?rhs1* ⊓ *?rhs2*› **.**
qed

12.5 Read and Write Laws

12.5.1 Projections

read

lemma *F-read* :
‹$\mathcal{F}\ (c?a{\in}A \to P\ a) = \{([], X)\ |X.\ X \cap ev\ `\ c\ `\ A = \{\}\} \cup$
$\{(ev\ a\ \#\ s, X)\ |a\ s\ X.\ a \in c\ `\ A \wedge (s, X) \in \mathcal{F}\ ((P \circ inv\text{-}into\ A\ c)\ a)\}$›
by (*simp add*: *read-def F-Mprefix*)

lemma *F-read-inj-on* :
‹*inj-on c A* ⟹
$\mathcal{F}\ (c?a{\in}A \to P\ a) = \{([], X)\ |X.\ X \cap ev\ `\ c\ `\ A = \{\}\} \cup$
$\{(ev\ (c\ a)\ \#\ s, X)\ |a\ s\ X.\ a \in A \wedge (s, X) \in \mathcal{F}\ (P\ a)\}$›
by (*auto simp add*: *F-read*)

lemma *D-read* :
‹$\mathcal{D}\ (c?a{\in}A \to P\ a) = \{ev\ a\ \#\ s\ |a\ s.\ a \in c\ `\ A \wedge s \in \mathcal{D}\ ((P \circ inv\text{-}into\ A\ c)\ a)\}$›
by (*simp add*: *read-def D-Mprefix*)

lemma *D-read-inj-on* :
‹*inj-on c A* ⟹ $\mathcal{D}\ (c?a{\in}A \to P\ a) = \{ev\ (c\ a)\ \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{D}\ (P\ a)\}$›
by (*auto simp add*: *D-read*)

lemma *T-read* :
‹$\mathcal{T}\ (c?a{\in}A \to P\ a) = insert\ []\ \{ev\ a\ \#\ s\ |a\ s.\ a \in c\ `\ A \wedge s \in \mathcal{T}\ ((P \circ inv\text{-}into\ A\ c)\ a)\}$›
by (*simp add*: *read-def T-Mprefix*)

lemma *T-read-inj-on* :
‹*inj-on c A* ⟹ $\mathcal{T}\ (c?a{\in}A \to P\ a) = insert\ []\ \{ev\ (c\ a)\ \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{T}\ (P\ a)\}$›
by (*auto simp add*: *T-read*)

lemmas *read-projs* = *F-read D-read T-read*
and *read-inj-on-projs* = *F-read-inj-on D-read-inj-on T-read-inj-on*

ndet-write

lemma *F-ndet-write* :
‹$\mathcal{F}$ (c!!a∈A → P a) =
(*if* A = {} *then* {(s, X). s = []}
else {([], X) |X. ∃ a∈A. *ev* (c a) ∉ X} ∪
{(*ev* a # s, X) |a s X. a ∈ c ‘ A ∧ (s, X) ∈ $\mathcal{F}$ ((P ∘ *inv-into* A c) a)})›
by (*simp add*: *ndet-write-def F-Mndetprefix′*)

lemma *F-ndet-write-inj-on* :
‹*inj-on* c A ⟹
$\mathcal{F}$ (c!!a∈A → P a) =
(*if* A = {} *then* {(s, X). s = []}
else {([], X) |X. ∃ a∈A. *ev* (c a) ∉ X} ∪
{(*ev* (c a) # s, X) |a s X. a ∈ A ∧ (s, X) ∈ $\mathcal{F}$ (P a)})›
by (*auto simp add*: *F-ndet-write*)

lemma *D-ndet-write* :
‹$\mathcal{D}$ (c!!a∈A → P a) = {*ev* a # s |a s. a ∈ c ‘ A ∧ s ∈ $\mathcal{D}$ ((P ∘ *inv-into* A c) a)}›
by (*simp add*: *ndet-write-def D-Mndetprefix′*)

lemma *D-ndet-write-inj-on* :
‹*inj-on* c A ⟹ $\mathcal{D}$ (c!!a∈A → P a) = {*ev* (c a) # s |a s. a ∈ A ∧ s ∈ $\mathcal{D}$ (P a)}›
by (*auto simp add*: *D-ndet-write*)

lemma *T-ndet-write* :
‹$\mathcal{T}$ (c!!a∈A → P a) = *insert* [] {*ev* a # s |a s. a ∈ c ‘ A ∧ s ∈ $\mathcal{T}$ ((P ∘ *inv-into* A c) a)}›
by (*simp add*: *ndet-write-def T-Mndetprefix′*)

lemma *T-ndet-write-inj-on* :
‹*inj-on* c A ⟹ $\mathcal{T}$ (c!!a∈A → P a) = *insert* [] {*ev* (c a) # s |a s. a ∈ A ∧ s ∈ $\mathcal{T}$ (P a)}›
by (*auto simp add*: *T-ndet-write*)

lemmas *ndet-write-projs* = *F-ndet-write D-ndet-write T-ndet-write*
and *ndet-write-inj-on-projs* = *F-ndet-write-inj-on D-ndet-write-inj-on T-ndet-write-inj-on*

write **and** (→)

lemma *F-write* :
‹$\mathcal{F}$ (c!a → P) = {([], X) |X. *ev* (c a) ∉ X} ∪ {(*ev* (c a) # s, X) |s X. (s, X) ∈ $\mathcal{F}$ P}›
by (*simp add*: *write-def F-Mprefix*)

lemma *F-write0* :
‹$\mathcal{F}$ (a → P) = {([], X) |X. *ev* a ∉ X} ∪ {(*ev* a # s, X) |s X. (s, X) ∈ $\mathcal{F}$ P}›

by (*simp add*: *write0-def F-Mprefix*)

lemma *D-write* : ‹$\mathcal{D}$ (*c*!*a* → *P*) = {*ev* (*c a*) # *s* |*s*. *s* ∈ $\mathcal{D}$ *P*}›
by (*simp add*: *write-def D-Mprefix*)

lemma *D-write0* : ‹$\mathcal{D}$ (*a* → *P*) = {*ev a* # *s* |*s*. *s* ∈ $\mathcal{D}$ *P*}›
by (*simp add*: *write0-def D-Mprefix*)

lemma *T-write* : ‹$\mathcal{T}$ (*c*!*a* → *P*) = *insert* [] {*ev* (*c a*) # *s* |*s*. *s* ∈ $\mathcal{T}$ *P*}›
by (*simp add*: *write-def T-Mprefix*)

lemma *T-write0* : ‹$\mathcal{T}$ (*a* → *P*) = *insert* [] {*ev a* # *s* |*s*. *s* ∈ $\mathcal{T}$ *P*}›
by (*simp add*: *write0-def T-Mprefix*)

lemmas *write-projs* = *F-write D-write T-write*
and *write0-projs* = *F-write0 D-write0 T-write0*

12.5.2 Equality with Constant Process

STOP

lemma *read-is-STOP-iff* : ‹*c*?*a*∈*A* → *P a* = *STOP* ⟷ *A* = {}›
by (*simp add*: *read-def Mprefix-is-STOP-iff*)

lemma *read-empty* [*simp*] : ‹*c*?*a*∈{} → *P a* = *STOP*› **by** (*simp add*: *read-def*)

lemma *ndet-write-is-STOP-iff* : ‹*c*!!*a*∈*A* → *P a* = *STOP* ⟷ *A* = {}›
by (*simp add*: *ndet-write-def Mndetprefix-is-STOP-iff*)

lemma *ndet-write-empty* [*simp*] : ‹*c*!!*a*∈{} → *P a* = *STOP*› **by** (*simp add*: *ndet-write-def*)

lemma *write0-neq-STOP* [*simp*] : ‹*a* → *P* ≠ *STOP*› **by** (*simp add*: *write0-def Mprefix-is-STOP-iff*)

lemma *write-neq-STOP* [*simp*] : ‹*c*!*a* → *P* ≠ *STOP*› **by** (*simp add*: *write-is-write0*)

SKIP

lemma *read-neq-SKIP* [*simp*] : ‹*c*?*a*∈*A* → *P a* ≠ *SKIP r*› **by** (*simp add*: *read-def*)

lemma *ndet-write-neq-SKIP* [*simp*] : ‹*c*!!*a*∈*A* → *P a* ≠ *SKIP r*› **by** (*simp add*: *ndet-write-def*)

lemma *write0-neq-SKIP* [*simp*] : ‹*a* → *P* ≠ *SKIP r*› **by** (*simp add*: *write0-def*)

lemma *write-neq-SKIP* [*simp*] : ‹*c*!*a* → *P* ≠ *SKIP r*› **by** (*simp add*: *write-is-write0*)

$\bot$

lemma *read-neq-BOT* [*simp*] : ‹$c?a{\in}A \to P\ a \neq \bot$› **by** (*simp add*: *read-def*)

lemma *ndet-write-neq-BOT* [*simp*] : ‹$c!!a{\in}A \to P\ a \neq \bot$› **by** (*simp add*: *ndet-write-def*)

lemma *write0-neq-BOT* [*simp*] : ‹$a \to P \neq \bot$› **by** (*simp add*: *write0-def*)

lemma *write-neq-BOT* [*simp*] : ‹$c!a \to P \neq \bot$› **by** (*simp add*: *write-is-write0*)

12.5.3 Extensions of Step-Laws

Monotony for equality

lemma *mono-read-eq* :
‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a = Q\ a) \Longrightarrow read\ c\ A\ P = read\ c\ A\ Q$›
by (*auto simp add*: *read-def inv-into-into intro*!: *mono-Mprefix-eq*)

lemma *mono-ndet-write-eq* :
‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a = Q\ a) \Longrightarrow ndet\text{-}write\ c\ A\ P = ndet\text{-}write\ c\ A\ Q$›
by (*auto simp add*: *ndet-write-def inv-into-into intro*!: *mono-Mndetprefix-eq*)

($\Box$) and ($\sqcap$)

lemma *read-Ndet-read* :
‹$(c?a{\in}A \to P\ a) \sqcap (c?b{\in}B \to Q\ b) =$
$(c?a{\in}(A - B) \to P\ a) \sqcap (c?b{\in}(B - A) \to Q\ b) \Box (c?x{\in}(A \cap B) \to P\ x \sqcap Q\ x)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹*inj-on c* $(A \cup B)$›
proof −
have $*$: ‹$c\ `\ A - c\ `\ B = c\ `\ (A - B)$›
by (*metis Diff-subset inj-on-image-set-diff le-supI1 sup.cobounded2* ‹*inj-on c* $(A \cup B)$›)
have $**$: ‹$c\ `\ B - c\ `\ A = c\ `\ (B - A)$›
by (*metis Diff-subset Un-upper1 inj-on-image-set-diff le-supI2* ‹*inj-on c* $(A \cup B)$›)
have $***$: ‹$c\ `\ A \cap c\ `\ B = c\ `\ (A \cap B)$›
by (*metis inf-sup-ord*(*3*) *inj-on-image-Int sup-ge2* ‹*inj-on c* $(A \cup B)$›)
from ‹*inj-on c* $(A \cup B)$› **have** ‹*inj-on c* $(A - B)$› **by** (*simp add*: *inj-on-Un inj-on-diff*)
with ‹*inj-on c* $(A \cup B)$› **have** \$: ‹$a \in A - B \Longrightarrow inv\text{-}into\ A\ c\ (c\ a) = inv\text{-}into\ (A - B)\ c\ (c\ a)$› **for** *a*
by (*auto simp add*: *inj-on-Un*)
from ‹*inj-on c* $(A \cup B)$› **have** ‹*inj-on c* $(B - A)$› **by** (*simp add*: *inj-on-Un inj-on-diff*)
with ‹*inj-on c* $(A \cup B)$› **have** \$\$: ‹$b \in B - A \Longrightarrow inv\text{-}into\ B\ c\ (c\ b) = inv\text{-}into\ (B - A)\ c\ (c\ b)$› **for** *b*
by (*auto simp add*: *inj-on-Un*)
have \$\$\$: ‹*inv-into A c* $(c\ a) = inv\text{-}into\ (A \cap B)\ c\ (c\ a)$›
‹*inv-into B c* $(c\ a) = inv\text{-}into\ (A \cap B)\ c\ (c\ a)$› **if** ‹$a \in A \cap B$› **for** *a*

```
   using ‹a ∈ A ∩ B› ‹inj-on c (A ∪ B)› by (auto simp add: inj-on-Un inj-on-Int)
  show ‹?lhs = ?rhs›
    by (unfold read-def, subst Mprefix-Ndet-Mprefix)
      (auto simp add: ∗ ∗∗ ∗∗∗ $ $$ $$$
       intro!: mono-Mprefix-eq arg-cong[where f = P] arg-cong2[where f = ‹(□)›]
        arg-cong2[where f = ‹(⊓)›])
qed

lemma read-Det-read :
  ‹(c?a∈A → P a) □ (c?b∈B → Q b) =
   c?a∈(A ∪ B) → (if a ∈ A ∩ B then P a ⊓ Q a else if a ∈ A then P a else Q a)›
  (is ‹?lhs = ?rhs›) if ‹inj-on c (A ∪ B)›
proof −
  have ∗ : ‹c ‘ A ∪ c ‘ B = c ‘ (A ∪ B)› by blast
  from ‹inj-on c (A ∪ B)›
  have $ : ‹a ∈ A ⟹ inv-into A c (c a) = inv-into (A ∪ B) c (c a)›
    ‹b ∈ B ⟹ inv-into B c (c b) = inv-into (A ∪ B) c (c b)› for a b
    by (simp-all add: inj-on-Un)
  from ‹inj-on c (A ∪ B)› show ‹?lhs = ?rhs›
    by (auto simp add: read-def Mprefix-Det-Mprefix ∗ $
        intro!: mono-Mprefix-eq) (metis Un-iff inj-onD)+
qed


lemma ndet-write-Det-ndet-write :
  ‹(c!!a∈A → P a) □ (c!!b∈B → Q b) =
   (  if A = {} then (c!!b∈B → Q b)
    else  if B = {} then (c!!a∈A → P a)
        else ⊓a∈A. ⊓b∈B. (if a = b then c!a → P a ⊓ Q a else (c!a → P a) □ (c!b
→ Q b)))›
  if ‹inj-on c (A ∪ B)›
proof −
  have ∗ : ‹a ∈ A ⟹ inv-into A c (c a) = inv-into (A ∪ B) c (c a)›
    ‹b ∈ B ⟹ inv-into B c (c b) = inv-into (A ∪ B) c (c b)› for a b
    using ‹inj-on c (A ∪ B)› by (auto simp add: inj-on-Un)
  from ‹inj-on c (A ∪ B)› show ?thesis
   by (auto simp add: ndet-write-def Mndetprefix-Det-Mndetprefix ∗ write-is-write0
inj-on-eq-iff
       split: if-split-asm intro!: mono-GlobalNdet-eq2)
qed

lemma ndet-write-Ndet-ndet-write :
  ‹(c!!a∈A → P a) ⊓ (c!!b∈B → Q b) =
   (  if A = B then (c!!b∈B → P b ⊓ Q b)
    else  if A ⊆ B
        then (c!!b∈(B − A) → Q b) ⊓ (c!!a∈A → P a ⊓ Q a)
        else  if B ⊆ A
            then (c!!a∈(A − B) → P a) ⊓ (c!!b∈B → P b ⊓ Q b)
            else (c!!a∈(A − B) → P a) ⊓ (c!!b∈(B − A) → Q b) ⊓ (c!!a∈(A ∩ B)
```

$\rightarrow P\ a \sqcap Q\ a))$›
if ‹$A \cap B \neq \{\}$› ‹*inj-on* $c\ (A \cup B)$›
proof −
have $*$: ‹$a \in A \Longrightarrow$ *inv-into* $A\ c\ (c\ a) =$ *inv-into* $(A \cup B)\ c\ (c\ a)$›
‹$b \in B \Longrightarrow$ *inv-into* $B\ c\ (c\ b) =$ *inv-into* $(A \cup B)\ c\ (c\ b)$›
‹$a \in A - B \Longrightarrow$ *inv-into* $(A - B)\ c\ (c\ a) =$ *inv-into* $A\ c\ (c\ a)$›
‹$b \in B - A \Longrightarrow$ *inv-into* $(B - A)\ c\ (c\ b) =$ *inv-into* $B\ c\ (c\ b)$›
‹$x \in A \cap B \Longrightarrow$ *inv-into* $(A \cup B)\ c\ (c\ x) =$ *inv-into* $(A \cap B)\ c\ (c\ x)$› **for** $a\ b\ x$
using ‹*inj-on* $c\ (A \cup B)$› **by** (*auto simp add*: *inj-on-Un inj-on-diff inj-on-Int*)
from ‹*inj-on* $c\ (A \cup B)$› **have** $: ‹c ‘ $A \subseteq c$ ‘ $B \longleftrightarrow A \subseteq B$› ‹$c$ ‘ $B \subseteq c$ ‘ $A \longleftrightarrow B \subseteq A$›
by (*auto simp add*: *inj-on-eq-iff*)
hence $$: ‹c ‘ $A = c$ ‘ $B \longleftrightarrow A = B$› **by** *blast*
from ‹*inj-on* $c\ (A \cup B)$›
have $$$: ‹c ‘ $A - c$ ‘ $B = c$ ‘ $(A - B)$› ‹c ‘ $B - c$ ‘ $A = c$ ‘ $(B - A)$› ‹c ‘ $A \cap c$ ‘ $B = c$ ‘ $(A \cap B)$›
by (*auto simp add*: *inj-on-eq-iff*)
from ‹$A \cap B \neq \{\}$› **have** ‹c ‘ $A \cap c$ ‘ $B \neq \{\}$› **by** *blast*
show *?thesis*
by (*auto simp add*: *Mndetprefix-Ndet-Mndetprefix*[*OF* ‹c ‘ $A \cap c$ ‘ $B \neq \{\}$›]
$ $$ $$$ $*$ *comp-def ndet-write-def*
intro!: *mono-Mndetprefix-eq arg-cong2*[**where** $f = ‹(\sqcap)›$])
qed

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

lemma *write0-Det-write0-is-write0-Ndet-write0* : ‹$(a \rightarrow P) \Box (a \rightarrow Q) = (a \rightarrow P) \sqcap (a \rightarrow Q)$›
by (*simp add*: *write0-def Mprefix-Det-Mprefix*) (*simp add*: *Mprefix-singl write0-Ndet-write0*)

lemma *write-Ndet-write* : ‹$(c!a \rightarrow P) \sqcap (c!a \rightarrow Q) = c!a \rightarrow P \sqcap Q$›
by (*simp add*: *write0-Ndet-write0 write-is-write0*)

lemma *write-Det-write-is-write-Ndet-write*: ‹$(c!a \rightarrow P) \Box (c!a \rightarrow Q) = (c!a \rightarrow P) \sqcap (c!a \rightarrow Q)$›
by (*simp add*: *write0-Det-write0-is-write0-Ndet-write0 write-is-write0*)

lemma *write-Ndet-read* :
‹$(c!a \rightarrow P) \sqcap (c?b{\in}B \rightarrow Q\ b) =$
(*if* $a \in B$ *then STOP else* $c!a \rightarrow P) \sqcap (c?b{\in}(B - \{a\}) \rightarrow Q\ b) \Box$ (*if* $a \in B$ *then* $c!a \rightarrow P \sqcap Q\ a$ *else STOP*)›
if ‹*inj-on* $c\ (\{a\} \cup B)$›
by (*subst read-Ndet-read*[*OF* ‹*inj-on* $c\ (\{a\} \cup B)$›, *simplified*])

(*auto simp add*: *insert-Diff-if intro*: *arg-cong2*[**where** $f = \langle(\Box)\rangle$] *arg-cong2*[**where** $f = \langle(\sqcap)\rangle$])

lemma *read-Ndet-write* :
$\langle$*inj-on c* $(A \cup \{b\}) \Longrightarrow (c?a{\in}A \to P\ a) \sqcap (c!b \to Q) =$
(*if* $b \in A$ *then STOP else* $c!b \to Q$) $\sqcap$ ($c?a{\in}(A - \{b\}) \to P\ a$) $\Box$ (*if* $b \in A$ *then* $c!b \to P\ b \sqcap Q$ *else STOP*)$\rangle$
by (*subst Ndet-commute*, *subst write-Ndet-read*) (*simp-all add*: *Ndet-commute*)

lemma *write0-Ndet-read* :
$\langle(a \to P) \sqcap (id?b{\in}B \to Q\ b) =$
(*if* $a \in B$ *then STOP else* $a \to P$) $\sqcap$ ($id?b{\in}(B - \{a\}) \to Q\ b$) $\Box$ (*if* $a \in B$ *then* $a \to P \sqcap Q\ a$ *else STOP*)$\rangle$
by (*subst write-Ndet-read*[**where** $c = id$, *unfolded write-is-write0*, *simplified*]) *simp*

lemma *read-Ndet-write0* :
$\langle(id?a{\in}A \to P\ a) \sqcap (b \to Q) =$
(*if* $b \in A$ *then STOP else* $b \to Q$) $\sqcap$ ($id?a{\in}(A - \{b\}) \to P\ a$) $\Box$ (*if* $b \in A$ *then* $b \to P\ b \sqcap Q$ *else STOP*)$\rangle$
by (*subst read-Ndet-write*[**where** $c = id$, *unfolded write-is-write0*, *simplified*]) *simp*

lemma *write-Det-read* :
$\langle$*inj-on c* (*insert a B*) $\Longrightarrow$ ($c!a \to P$) $\Box$ ($c?b{\in}B \to Q\ b$) =
$c?b{\in}$(*insert a B*) $\to$ (*if* $b = a \wedge a \in B$ *then* $P \sqcap Q\ a$ *else if* $b = a$ *then* P *else* $Q\ b$)$\rangle$
by (*subst read-Det-read*[**where** $A = \langle\{a\}\rangle$, *simplified*]) (*auto intro*: *mono-read-eq*)

lemma *read-Det-write* :
$\langle$*inj-on c* (*insert b A*) $\Longrightarrow$ ($c?a{\in}A \to P\ a$) $\Box$ ($c!b \to Q$) =
$c?a{\in}$(*insert b A*) $\to$ (*if* $a = b \wedge b \in A$ *then* $P\ a \sqcap Q$ *else if* $a = b$ *then* Q *else* $P\ a$)$\rangle$
by (*subst read-Det-read*[**where** $B = \langle\{b\}\rangle$, *simplified*]) (*auto intro*: *mono-read-eq*)

lemma *write0-Det-read* :
$\langle(a \to P) \Box (id?b{\in}B \to Q\ b) =$
$id?b{\in}$(*insert a B*) $\to$ (*if* $b = a \wedge a \in B$ *then* $P \sqcap Q\ a$ *else if* $b = a$ *then* P *else* $Q\ b$)$\rangle$
by (*subst write-Det-read*[**where** $c = id$, *unfolded write-is-write0*, *simplified*]) *simp*

lemma *read-Det-write0* :
$\langle(id?a{\in}A \to P\ a) \Box (b \to Q) =$
$id?a{\in}$(*insert b A*) $\to$ (*if* $a = b \wedge b \in A$ *then* $P\ a \sqcap Q$ *else if* $a = b$ *then* Q *else* $P\ a$)$\rangle$
by (*subst read-Det-write*[**where** $c = id$, *unfolded write-is-write0*, *simplified*]) *simp*

Sliding

lemma *write0-Sliding-write0* :
‹$(a \to P) \rhd (b \to Q) =$
$(\Box x \in \{a, b\} \to (\mathit{if}\ a = b\ \mathit{then}\ P \sqcap Q\ \mathit{else}\ \mathit{if}\ x = a\ \mathit{then}\ P\ \mathit{else}\ Q)) \sqcap$
$(b \to (\mathit{if}\ a = b\ \mathit{then}\ P \sqcap Q\ \mathit{else}\ Q))$›
by (*auto simp add*: *Process-eq-spec write0-def*
Sliding-projs Ndet-projs Mprefix-projs)

lemma *write-Sliding-write* :
‹$(c!a \to P) \rhd (d!b \to Q) =$
$(\Box x \in \{c\ a, d\ b\} \to (\mathit{if}\ c\ a = d\ b\ \mathit{then}\ P \sqcap Q\ \mathit{else}\ \mathit{if}\ x = c\ a\ \mathit{then}\ P\ \mathit{else}\ Q)) \sqcap$
$(d!b \to (\mathit{if}\ c\ a = d\ b\ \mathit{then}\ P \sqcap Q\ \mathit{else}\ Q))$›
by (*simp add*: *write-is-write0 write0-Sliding-write0*)

lemma *write0-Sliding-write* :
‹$(a \to P) \rhd (d!b \to Q) =$
$(\Box x \in \{a, d\ b\} \to (\mathit{if}\ a = d\ b\ \mathit{then}\ P \sqcap Q\ \mathit{else}\ \mathit{if}\ x = a\ \mathit{then}\ P\ \mathit{else}\ Q)) \sqcap$
$(d!b \to (\mathit{if}\ a = d\ b\ \mathit{then}\ P \sqcap Q\ \mathit{else}\ Q))$›
by (*simp add*: *write-is-write0 write0-Sliding-write0*)

lemma *write-Sliding-write0* :
‹$(c!a \to P) \rhd (b \to Q) =$
$(\Box x \in \{c\ a, b\} \to (\mathit{if}\ c\ a = b\ \mathit{then}\ P \sqcap Q\ \mathit{else}\ \mathit{if}\ x = c\ a\ \mathit{then}\ P\ \mathit{else}\ Q)) \sqcap$
$(b \to (\mathit{if}\ c\ a = b\ \mathit{then}\ P \sqcap Q\ \mathit{else}\ Q))$›
by (*simp add*: *write-is-write0 write0-Sliding-write0*)

lemma *read-Sliding-superset-read* :
— Not really interesting without the additional assumptions.
‹$A \subseteq B \Longrightarrow$ *inj-on c B* $\Longrightarrow$
$(c?a{\in}A \to P\ a) \rhd (c?b{\in}B \to Q\ b) = c?b{\in}B \to (\mathit{if}\ b \in A\ \mathit{then}\ P\ b \sqcap Q\ b\ \mathit{else}\ Q\ b)$›
by (*unfold read-def*, *subst Mprefix-Sliding-superset-Mprefix*)
(*auto simp add*: *inj-on-eq-iff subset-iff intro*!: *mono-Mprefix-eq*, *metis inj-on-subset inv-into-f-f subset-eq*)

lemma *read-Sliding-same-set-read* :
‹*inj-on c A* $\Longrightarrow (c?a{\in}A \to P\ a) \rhd (c?a{\in}A \to Q\ a) = c?a{\in}A \to P\ a \sqcap Q\ a$›
by (*unfold read-def Mprefix-Sliding-same-set-Mprefix*)
(*auto simp add*: *inj-on-eq-iff subset-iff intro*: *mono-Mprefix-eq*)

lemma *ndet-write-Sliding-superset-ndet-write* :
‹$A \subseteq B \Longrightarrow$ *inj-on c B* $\Longrightarrow$
$(c!!a{\in}A \to P\ a) \rhd (c!!b{\in}B \to Q\ b) = c!!b{\in}B \to (\mathit{if}\ b \in A\ \mathit{then}\ P\ b \sqcap Q\ b\ \mathit{else}\ Q\ b)$›
by (*unfold ndet-write-def*, *subst Mndetprefix-Sliding-superset-Mndetprefix*)
(*auto simp add*: *inj-on-eq-iff subset-iff intro*!: *mono-Mndetprefix-eq*, *metis inj-on-subset inv-into-f-f subset-eq*)

lemma *ndet-write-Sliding-same-set-ndet-write* :
‹*inj-on c A* ⟹ (*c*!!*a*∈*A* → *P a*) ▷ (*c*!!*a*∈*A* → *Q a*) = *c*!!*a*∈*A* → *P a* ⊓ *Q a*›
by (*unfold ndet-write-def Mndetprefix-Sliding-same-set-Mndetprefix*)
(*auto simp add*: *inj-on-eq-iff subset-iff intro*: *mono-Mndetprefix-eq*)

lemma *write-Sliding-superset-read* :
‹*a* ∈ *B* ⟹ *inj-on c B* ⟹
(*c*!*a* → *P*) ▷ (*c*?*b*∈*B* → *Q b*) = *c*?*b*∈*B* → (*if b* = *a then P* ⊓ *Q b else Q b*)›
by (*subst read-Sliding-superset-read*[**where** *A* = ‹{*a*}›, *simplified*]) *simp-all*

lemma *write0-Sliding-superset-read* :
‹*a* ∈ *B* ⟹ (*a* → *P*) ▷ (*id*?*b*∈*B* → *Q b*) = *id*?*b*∈*B* → (*if b* = *a then P* ⊓ *Q b else Q b*)›
by (*subst read-Sliding-superset-read*
[**where** *A* = ‹{*a*}› **and** *c* = *id*, *simplified*, *unfolded write-is-write0*, *simplified*])
simp-all

lemma *write-Sliding-superset-ndet-write* :
‹*a* ∈ *B* ⟹ *inj-on c B* ⟹
(*c*!*a* → *P*) ▷ (*c*!!*b*∈*B* → *Q b*) = *c*!!*b*∈*B* → (*if b* = *a then P* ⊓ *Q b else Q b*)›
by (*subst ndet-write-Sliding-superset-ndet-write*[**where** *A* = ‹{*a*}›, *simplified*])
simp-all

lemma *write0-Sliding-superset-ndet-write* :
‹*a* ∈ *B* ⟹ (*a* → *P*) ▷ (*id*!!*b*∈*B* → *Q b*) = *id*!!*b*∈*B* → (*if b* = *a then P* ⊓ *Q b else Q b*)›
by (*subst ndet-write-Sliding-superset-ndet-write*
[**where** *A* = ‹{*a*}› **and** *c* = *id*, *simplified*, *unfolded write-is-write0*, *simplified*])
simp-all

Seq

lemma *read-Seq* : ‹*c*?*a*∈*A* → *P a* ; *Q* = *c*?*a*∈*A* → (*P a* ; *Q*)›
by (*simp add*: *read-def Mprefix-Seq comp-def*)

lemma *write0-Seq* : ‹*a* → *P* ; *Q* = *a* → (*P* ; *Q*)›
by (*simp add*: *write0-def Mprefix-Seq*)

lemma *ndet-write-Seq* : ‹*c*!!*a*∈*A* → *P a* ; *Q* = *c*!!*a*∈*A* → (*P a* ; *Q*)›
by (*simp add*: *ndet-write-is-GlobalNdet-write0 Seq-distrib-GlobalNdet-right write0-Seq*)

lemma *write-Seq* : ‹*c*!*a* → *P* ; *Q* = *c*!*a* → (*P* ; *Q*)›
by (*simp add*: *write0-Seq write-is-write0*)

Renaming

lemma *Renaming-read* :
‹*Renaming* (*c*?*a*∈*A* → *P a*) *f g* = (*f* ∘ *c*)?*a*∈*A* → *Renaming* (*P a*) *f g*›

```
  if ‹inj-on c A› and ‹inj-on f (c ‘ A)›
proof −
  have ‹f ‘ c ‘ A = (f ∘ c) ‘ A› by auto
  have ‹inj-on (f ∘ c) A› by (simp add: comp-inj-on ‹inj-on c A› ‹inj-on f (c ‘
A)›)
  have ∗ : ‹y ∈ (λx. f (c x)) ‘ A ⟹ {x ∈ c ‘ A. y = f x} = {inv-into (c ‘ A) f
y}› for y
    using ‹inj-on f (c ‘ A)› by auto
  show ‹Renaming (c?a∈A → P a) f g = (f ∘ c)?a∈A → Renaming (P a) f g›
 proof (unfold read-def Renaming-Mprefix ‹f ‘ c ‘ A = (f ∘ c) ‘ A›, rule mono-Mprefix-eq)
    from ‹inj-on (f ∘ c) A›
    show ‹a ∈ (f ∘ c) ‘ A ⟹
        ⊓a∈{x ∈ c ‘ A. a = f x}. Renaming ((P ∘ inv-into A c) a) f g =
        ((λa. Renaming (P a) f g) ∘ inv-into A (f ∘ c)) a› for a
      by (auto simp add: ∗ inv-into-f-eq ‹inj-on c A› ‹inj-on f (c ‘ A)›
        intro: arg-cong[where f = ‹λa. Renaming (P a) f g›])
  qed
qed

lemma Renaming-write :
  ‹Renaming (c!a → P) f g = (f ∘ c)!a → Renaming P f g›
  by (fact Renaming-read[where A = ‹{a}›, simplified])

lemma Renaming-write0 :
  ‹Renaming (a → P) f g = f a → Renaming P f g›
  by (fact Renaming-write[where c = id, unfolded write-is-write0, simplified])

lemma Renaming-ndet-write :
  ‹Renaming (c!!a∈A → P a) f g = (f ∘ c)!!a∈A → Renaming (P a) f g›
  if ‹inj-on c A› and ‹inj-on f (c ‘ A)›
proof −
  have ‹inj-on (f ∘ c) A› by (simp add: comp-inj-on ‹inj-on c A› ‹inj-on f (c ‘
A)›)
  have ‹(λx. f (c x)) ‘ A = f ‘ (c ‘ A)› by auto
  show ‹Renaming (c!!a∈A → P a) f g = (f ∘ c)!!a∈A → Renaming (P a) f g›
   by (simp add: ‹?this› ndet-write-is-GlobalNdet-write0 Renaming-distrib-GlobalNdet
Renaming-write0, rule sym)
      (use ‹inj-on (f ∘ c) A› ‹inj-on c A› in
        ‹auto simp add: inv-into-f-eq
              intro: mono-GlobalNdet-eq2 arg-cong[where f = ‹λa. - → Renaming
(P a) f g›]›)
qed
```

(\)

```
lemma Hiding-read-disjoint :
  ‹c ‘ A ∩ S = {} ⟹ c?a∈A → P a \ S = c?a∈A → (P a \ S)›
  by (unfold read-def, subst Hiding-Mprefix-disjoint)
    (auto simp add: disjoint-iff image-iff intro: mono-Mprefix-eq)
```

lemma ‹$A \subseteq B \Longrightarrow a \in A \Longrightarrow (\mathit{inv\text{-}into}\ A\ c)\ a = (\mathit{inv\text{-}into}\ B\ c)\ a$›
oops — Not provable, therefore we need the injectivity for the non disjoint case.

lemma *Hiding-read-non-disjoint* :
‹$c?a{\in}A \to P\ a \setminus S = (c?a{\in}(A - c -' S) \to (P\ a \setminus S)) \rhd (\sqcap a{\in}(A \cap c -' S).\ (P\ a \setminus S))$›
if ‹$\mathit{inj\text{-}on}\ c\ A$› **and** ‹$c\ `\ A \cap S \neq \{\}$›
proof −
have $*$: ‹$(P \circ \mathit{inv\text{-}into}\ A\ c)\ x = (P \circ \mathit{inv\text{-}into}\ (A - c -' S)\ c)\ x$›
if ‹$x \in c\ `\ (A - c -' S)$› **for** x
proof −
from ‹$x \in c\ `\ (A - c -' S)$› **obtain** a **where** ‹$a \in A - c -' S$› ‹$x = c\ a$› **by** *blast*
from ‹$a \in A - c -' S$› **have** ‹$a \in A$› **by** *blast*
from ‹$\mathit{inj\text{-}on}\ c\ A$› *inj-on-subset* **have** ‹$\mathit{inj\text{-}on}\ c\ (A - c -' S)$› **by** *force*
from ‹$a \in A - c -' S$› ‹$x = c\ a$› ‹$\mathit{inj\text{-}on}\ c\ (A - c -' S)$›
have ‹$(\mathit{inv\text{-}into}\ (A - c -' S)\ c)\ x = a$› **by** *simp*
moreover from ‹$a \in A$› ‹$x = c\ a$› ‹$\mathit{inj\text{-}on}\ c\ A$› **have** ‹$(\mathit{inv\text{-}into}\ A\ c)\ x = a$› **by** *simp*
ultimately show ‹$(P \circ \mathit{inv\text{-}into}\ A\ c)\ x = (P \circ \mathit{inv\text{-}into}\ (A - c -' S)\ c)\ x$› **by** *simp*
qed
have $**$: ‹$c\ `\ (A - c -' S) = c\ `\ A - S$› **by** *blast*
have $***$: ‹$c\ `\ A \cap S = c\ `\ (A \cap (c -' S))$› **by** *blast*
have ‹$(c?a{\in}A \to P\ a) \setminus S = (c?a{\in}(A - c -' S) \to (P\ a \setminus S)) \rhd (\sqcap a{\in}(c\ `\ A \cap S).\ (P\ (\mathit{inv\text{-}into}\ A\ c\ a) \setminus S))$›
proof (*unfold read-def comp-def*, *subst Hiding-Mprefix-non-disjoint*, *use* ‹$c\ `\ A \cap S \neq \{\}$› **in** *blast*, *rule arg-cong*[**where** $f = \langle\lambda P.\ P \rhd \text{-}\rangle$])
show ‹$\Box a{\in}(c\ `\ A - S) \to (P\ (\mathit{inv\text{-}into}\ A\ c\ a) \setminus S) = \Box x{\in}c\ `\ (A - c -' S) \to (P\ (\mathit{inv\text{-}into}\ (A - c -' S)\ c\ x) \setminus S)$›
by (*use* $*$ **in** ‹*force simp add*: $**$ *intro* : *mono-Mprefix-eq*›)
qed
also have ‹$(\sqcap a{\in}(c\ `\ A \cap S).\ (P\ (\mathit{inv\text{-}into}\ A\ c\ a) \setminus S)) = \sqcap a{\in}(A \cap c -' S).\ (P\ a \setminus S)$›
by (*subst* $***$, *rule mono-GlobalNdet-eq2*) (*simp add*: ‹$\mathit{inj\text{-}on}\ c\ A$›)
finally show ‹$(c?a{\in}A \to P\ a) \setminus S = (c?a{\in}(A - c -' S) \to (P\ a \setminus S)) \rhd (\sqcap a{\in}(A \cap c -' S).\ (P\ a \setminus S))$› .
qed

lemma *Hiding-read-subset* :
‹$c?a{\in}A \to P\ a \setminus S = \sqcap a{\in}(c\ `\ A \cap S).\ (P\ (\mathit{inv\text{-}into}\ A\ c\ a) \setminus S)$›
if ‹$\mathit{inj\text{-}on}\ c\ A$› **and** ‹$c\ `\ A \subseteq S$›
proof (*cases* ‹$A = \{\}$›)
show ‹$A = \{\} \Longrightarrow c?a{\in}A \to P\ a \setminus S = \sqcap a{\in}(c\ `\ A \cap S).\ (P\ (\mathit{inv\text{-}into}\ A\ c\ a) \setminus S)$›
by (*auto simp add*: *Process-eq-spec GlobalNdet-projs F-Hiding-seqRun D-Hiding-seqRun read-def Mprefix-projs*)
next

assume ‹$A \neq \{\}$›
with ‹c ‘ $A \subseteq S$› **have** ‹c ‘ $A \cap S \neq \{\}$› ‹c ‘ $A - S = \{\}$› **by** *auto*
show ‹$c?a{\in}A \rightarrow P\ a \setminus S = \sqcap a{\in}(c$ ‘ $A \cap S).\ (P\ (\mathit{inv\text{-}into}\ A\ c\ a) \setminus S)$›
by (*simp add*: *read-def Hiding-Mprefix-non-disjoint*[*OF* ‹c ‘ $A \cap S \neq \{\}$›] ‹c ‘ $A - S = \{\}$›
Process-eq-spec Sliding-projs GlobalNdet-projs Mprefix-projs)
qed

lemma *Hiding-ndet-write-disjoint* :
‹c ‘ $A \cap S = \{\} \Longrightarrow (c!!a{\in}A \rightarrow P\ a) \setminus S = (c!!a{\in}A \rightarrow (P\ a \setminus S))$›
by (*unfold ndet-write-def*, *subst Hiding-Mndetprefix-disjoint*)
(*auto simp add*: *disjoint-iff image-iff intro*: *mono-Mndetprefix-eq*)

lemma *Hiding-ndet-write-subset* :
‹c ‘ $A \subseteq S \Longrightarrow (c!!a{\in}A \rightarrow P\ a) \setminus S = \sqcap a{\in}c$ ‘ $A.\ (P\ (\mathit{inv\text{-}into}\ A\ c\ a) \setminus S)$›
by (*unfold ndet-write-def*, *subst Hiding-Mndetprefix-subset*)
(*auto simp add*: *disjoint-iff image-iff intro*: *mono-GlobalNdet-eq*)

lemma *Hiding-ndet-write-subset-bis* :
— With the injectivity...
‹*inj-on c A* $\Longrightarrow$ c ‘ $A \subseteq S \Longrightarrow (c!!a{\in}A \rightarrow P\ a) \setminus S = \sqcap a{\in}A.\ (P\ a \setminus S)$›
by (*simp add*: *Hiding-ndet-write-subset mono-GlobalNdet-eq2*)

lemma ‹$A \subseteq B \Longrightarrow a \in A \Longrightarrow (\mathit{inv\text{-}into}\ A\ c)\ a = (\mathit{inv\text{-}into}\ B\ c)\ a$›
oops — Not provable, therefore we need the injectivity for the non disjoint case.

lemma *Hiding-ndet-write-non-disjoint-not-subset* :
‹$(c!!a{\in}A \rightarrow P\ a) \setminus S =$
$(c!!a{\in}(A - c -‘ S) \rightarrow (P\ a \setminus S)) \sqcap (\sqcap a{\in}(A \cap c -‘ S).\ (P\ a \setminus S))$›
if ‹*inj-on c A*› **and** ‹c ‘ $A \cap S \neq \{\}$› **and** ‹$\neg\ c$ ‘ $A \subseteq S$›
proof −
have $*$: ‹$(P \circ \mathit{inv\text{-}into}\ A\ c)\ x = (P \circ \mathit{inv\text{-}into}\ (A - c -‘ S)\ c)\ x$›
if ‹$x \in c$ ‘ $(A - c -‘ S)$› **for** x
proof −
from ‹$x \in c$ ‘ $(A - c -‘ S)$› **obtain** a **where** ‹$a \in A - c -‘ S$› ‹$x = c\ a$› **by** *blast*
from ‹$a \in A - c -‘ S$› **have** ‹$a \in A$› **by** *blast*
from ‹*inj-on c A*› *inj-on-subset* **have** ‹*inj-on c* $(A - c -‘ S)$› **by** *force*
from ‹$a \in A - c -‘ S$› ‹$x = c\ a$› ‹*inj-on c* $(A - c -‘ S)$›
have ‹$(\mathit{inv\text{-}into}\ (A - c -‘ S)\ c)\ x = a$› **by** *simp*
moreover from ‹$a \in A$› ‹$x = c\ a$› ‹*inj-on c A*› **have** ‹$(\mathit{inv\text{-}into}\ A\ c)\ x = a$› **by** *simp*
ultimately show ‹$(P \circ \mathit{inv\text{-}into}\ A\ c)\ x = (P \circ \mathit{inv\text{-}into}\ (A - c -‘ S)\ c)\ x$› **by** *simp*
qed
have $**$: ‹c ‘ $(A - c -‘ S) = c$ ‘ $A - S$› **by** *blast*
have $***$: ‹c ‘ $A \cap S = c$ ‘ $(A \cap c -‘ S)$› **by** *blast*

have ‹(c!!a∈A → P a) \ S = (c!!a∈(A − c −‘ S) → (P a \ S)) ⊓
(⊓a∈(c ‘ A ∩ S). (P (inv-into A c a) \ S))›
proof (*unfold ndet-write-def comp-def*
Hiding-Mndetprefix-non-disjoint-not-subset
[*OF* ‹c ‘ A ∩ S ≠ {}› ‹¬ c ‘ A ⊆ S›])
show ‹(⊓a∈(c ‘ A − S) → (P (*inv-into* A c a) \ S)) ⊓
(⊓a∈(c ‘ A ∩ S). (P (*inv-into* A c a) \ S)) =
(⊓x∈c ‘ (A − c −‘ S) → (P (*inv-into* (A − c −‘ S) c x) \ S)) ⊓
(⊓a∈(c ‘ A ∩ S). (P (*inv-into* A c a) \ S))›
by (*rule arg-cong*[**where** *f* = ‹λP. P ⊓ -›])
(*use* ∗ **in** ‹*force simp add*: ∗∗ *intro* : *mono-Mndetprefix-eq*›)
qed
also have ‹(⊓a∈(c ‘ A ∩ S). (P (*inv-into* A c a) \ S)) = ⊓a∈(A ∩ c −‘ S). (P a \ S)›
by (*auto simp add*: ∗∗∗ ‹*inj-on* c A› *intro*: *mono-GlobalNdet-eq2*)
finally show ‹c!!a∈A → P a \ S =
(c!!a∈(A − c −‘ S) → (P a \ S)) ⊓ (⊓a∈(A ∩ c −‘ S). (P a \ S))› **.**
qed

lemma *Hiding-write0-disjoint* :
‹a ∉ S ⟹ a → P \ S = a → (P \ S)›
by (*simp add*: *write0-def Hiding-Mprefix-disjoint*)

lemma *Hiding-write0-non-disjoint* :
‹a ∈ S ⟹ a → P \ S = P \ S›
by (*simp add*: *write0-def Hiding-Mprefix-non-disjoint*)

lemma *Hiding-write-disjoint* :
‹c a ∉ S ⟹ c!a → P \ S = c!a → (P \ S)›
by (*simp add*: *Hiding-write0-disjoint write-is-write0*)

lemma *Hiding-write-subset* :
‹c a ∈ S ⟹ c!a → P \ S = P \ S›
by (*simp add*: *Hiding-write0-non-disjoint write-is-write0*)

Sync

read **lemma** *read-Sync-read* :
— This is the general case.
‹c?a∈A → P a ⟦S⟧ d?b∈B → Q b =
(c?a∈(A − c −‘ S) → (P a ⟦S⟧ d?b∈B → Q b)) □
(d?b∈(B − d −‘ S) → (c?a∈A → P a ⟦S⟧ Q b)) □
(□x∈(c ‘ A ∩ d ‘ B ∩ S) → (P (*inv-into* A c x) ⟦S⟧ Q (*inv-into* B d x)))›
(**is** ‹*?lhs* = *?rhs1* □ *?rhs2* □ *?rhs3*›)
if ‹c ‘ A ∩ S = {} ∨ c ‘ A ⊆ S ∨ *inj-on* c A›
‹d ‘ B ∩ S = {} ∨ d ‘ B ⊆ S ∨ *inj-on* d B›
— Assumptions may seem strange, but the motivation is that when A − c −‘ S

$\neq \{\}$ (which is equivalent to $\neg\ c\ `\ A \subseteq S$), we need to ensure that *inv-into* $(A - c -` S)$ c is equal to *inv-into* A c. This requires $A - c -` S = A$ (which is equivalent to $c\ `\ A \cap S = \{\}$) or *inj-on* c A. We need obviously a similar assumption for B.

proof −
have $* : \langle \bigwedge e\ X.\ e\ `\ (X - e -` S) = e\ `\ X - S \rangle$ **by** *auto*
have $\langle ?lhs = (\Box a{\in}(c\ `\ A - S) \to (P\ (\mathit{inv\text{-}into}\ A\ c\ a)\ [\![S]\!]\ (\Box x{\in}d\ `\ B \to Q\ (\mathit{inv\text{-}into}\ B\ d\ x)))) \Box$
$(\Box b{\in}(d\ `\ B - S) \to ((\Box x{\in}c\ `\ A \to P\ (\mathit{inv\text{-}into}\ A\ c\ x))\ [\![S]\!]\ Q\ (\mathit{inv\text{-}into}\ B\ d\ b))) \Box$
$(\Box x{\in}(c\ `\ A \cap d\ `\ B \cap S) \to (P\ (\mathit{inv\text{-}into}\ A\ c\ x)\ [\![S]\!]\ Q\ (\mathit{inv\text{-}into}\ B\ d\ x)))\rangle$
(**is** $\langle ?lhs = ?rhs1' \Box ?rhs2' \Box ?rhs3 \rangle$)
by (*simp add*: *read-def Mprefix-Sync-Mprefix comp-def*)
also from *that*(1) **have** $\langle ?rhs1' = ?rhs1 \rangle$
proof (*elim disjE*)
assume $\langle c\ `\ A \cap S = \{\} \rangle$
hence $\langle A - c -` S = A \wedge c\ `\ A - S = c\ `\ A \rangle$ **by** *fast*
thus $\langle ?rhs1' = ?rhs1 \rangle$ **by** (*simp add*: *read-def comp-def*)
next
assume $\langle c\ `\ A \subseteq S \rangle$
hence $\langle A - c -` S = \{\} \wedge c\ `\ A - S = \{\} \rangle$ **by** *fast*
show $\langle ?rhs1' = ?rhs1 \rangle$ **by** (*simp add*: $\langle ?this \rangle$)
next
assume $\langle \mathit{inj\text{-}on}\ c\ A \rangle$
hence $\langle \mathit{inj\text{-}on}\ c\ (A - c -` S) \rangle$ **by** (*simp add*: *inj-on-diff*)
with $\langle \mathit{inj\text{-}on}\ c\ A \rangle$ **show** $\langle ?rhs1' = ?rhs1 \rangle$
by (*auto simp add*: *read-def comp-def* $*$ *intro*: *mono-Mprefix-eq*)
qed
also from *that*(2) **have** $\langle ?rhs2' = ?rhs2 \rangle$
proof (*elim disjE*)
assume $\langle d\ `\ B \cap S = \{\} \rangle$
hence $\langle B - d -` S = B \wedge d\ `\ B - S = d\ `\ B \rangle$ **by** *fast*
thus $\langle ?rhs2' = ?rhs2 \rangle$ **by** (*simp add*: *read-def comp-def*)
next
assume $\langle d\ `\ B \subseteq S \rangle$
hence $\langle B - d -` S = \{\} \wedge d\ `\ B - S = \{\} \rangle$ **by** *fast*
show $\langle ?rhs2' = ?rhs2 \rangle$ **by** (*simp add*: $\langle ?this \rangle$)
next
assume $\langle \mathit{inj\text{-}on}\ d\ B \rangle$
hence $\langle \mathit{inj\text{-}on}\ d\ (B - d -` S) \rangle$ **by** (*simp add*: *inj-on-diff*)
with $\langle \mathit{inj\text{-}on}\ d\ B \rangle$ **show** $\langle ?rhs2' = ?rhs2 \rangle$
by (*auto simp add*: *read-def comp-def* $*$ *intro*: *mono-Mprefix-eq*)
qed
finally show $\langle ?lhs = ?rhs1 \Box ?rhs2 \Box ?rhs3 \rangle$.
qed

Enforce read **lemma** *read-Sync-read-forced-read-left* :
$\langle c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$(c?a{\in}(A - c -` S) \to (P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b)) \Box$

```
   (d?b∈(B − d −‘ S) → (c?a∈A → P a ⟦S⟧ Q b)) □
   (c?x∈(A ∩ c −‘ (d ‘ B ∩ S)) → (P x ⟦S⟧ Q x))›
  (is ‹?lhs = ?rhs1 □ ?rhs2 □ ?rhs3›)
  if ‹c ‘ A ∩ S = {} ∨ inj-on c A›
    ‹d ‘ B ∩ S = {} ∨ inj-on d B›
    ‹⋀a b. a ∈ A ⟹ b ∈ B ⟹ c a = d b ⟹ d b ∈ S ⟹ a = b›
proof −
  let ?rhs3′ = ‹(□x∈(c ‘ A ∩ d ‘ B ∩ S) → (P (inv-into A c x) ⟦S⟧ Q (inv-into
B d x)))›
  have ∗ : ‹c ‘ (A ∩ c −‘ (d ‘ B ∩ S)) = c ‘ A ∩ d ‘ B ∩ S› by blast
  have ∗∗ : ‹c ‘ (A ∩ c −‘ d ‘ B) = c ‘ A ∩ d ‘ B› by blast
  from that(1, 2) consider ‹c ‘ A ∩ S = {} ∨ d ‘ B ∩ S = {}›
    | ‹inj-on c A› ‹inj-on d B› by blast
  hence ‹?rhs3′ = ?rhs3›
  proof cases
    assume ‹c ‘ A ∩ S = {} ∨ d ‘ B ∩ S = {}›
    hence ‹c ‘ A ∩ d ‘ B ∩ S = {} ∧ A ∩ c −‘ (d ‘ B ∩ S) = {}› by blast
    thus ‹?rhs3′ = ?rhs3› by simp
  next
    assume ‹inj-on c A› ‹inj-on d B›
    show ‹?rhs3′ = ?rhs3›
    proof (unfold read-def ∗ comp-def,
        intro mono-Mprefix-eq arg-cong2[where f = ‹λP Q. P ⟦S⟧ Q›])
      fix x assume ‹x ∈ c ‘ A ∩ d ‘ B ∩ S›
      moreover from ‹inj-on c A› inj-on-Int
      have ‹inj-on c A ∧ inj-on c (A ∩ c −‘ (d ‘ B ∩ S))› by blast
      ultimately show ‹P (inv-into A c x) = P (inv-into (A ∩ c −‘ (d ‘ B ∩ S))
c x)›
        by (simp add: image-iff, elim conjE bexE, simp)
    next
      fix x assume $ : ‹x ∈ c ‘ A ∩ d ‘ B ∩ S›
      then obtain a b where $$ : ‹x = c a› ‹a ∈ A› ‹x = d b› ‹b ∈ B› by blast
      from ‹inj-on c A› inj-on-Int have $$$ : ‹inj-on c (A ∩ c −‘ (d ‘ B ∩ S))›
by blast
      have ‹inv-into B d x = b› by (simp add: $$(3, 4) ‹inj-on d B›)
      also have ‹b = a› by (metis $ $$ Int-iff that(3))
      also have ‹a = inv-into (A ∩ c −‘ (d ‘ B ∩ S)) c x›
        by (metis $ $$(1, 2) $$$ ∗ Int-lower1
            ‹inj-on c A› inj-on-image-mem-iff inv-into-f-eq)
      finally have ‹inv-into B d x = inv-into (A ∩ c −‘ (d ‘ B ∩ S)) c x› .
      thus ‹Q (inv-into B d x) = Q (inv-into (A ∩ c −‘ (d ‘ B ∩ S)) c x)› by simp
    qed
  qed
  moreover have ‹?lhs = ?rhs1 □ ?rhs2 □ ?rhs3′›
    using that(1, 2) by (subst read-Sync-read) auto
  ultimately show ‹?lhs = ?rhs1 □ ?rhs2 □ ?rhs3› by argo
qed
```

lemma *read-Sync-read-forced-read-right*:
‹⟦c ‘ $A \cap S = \{\} \vee$ *inj-on* c A; d ‘ $B \cap S = \{\} \vee$ *inj-on* d B;
$\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow d\ b \in S \Longrightarrow a = b$⟧ ⟹
$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$(c?a{\in}(A - c -‘ S) \to (P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b)) \Box$
$(d?b{\in}(B - d -‘ S) \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)) \Box$
$(d?x{\in}(B \cap d -‘ (c ‘ A \cap S)) \to (P\ x\ [\![S]\!]\ Q\ x))$›
by (*subst Sync-commute*, *subst Det-commute*, *subst read-Sync-read-forced-read-left*)
(*blast*, *blast*, *metis*, *auto simp add*: *Sync-commute intro*: *arg-cong2*[**where** $f =$ ‹$(\Box)$›])

Special Cases **lemma** *read-Sync-read-subset* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$\Box x{\in}(c ‘ A \cap d ‘ B) \to (P\ (\textit{inv-into}\ A\ c\ x)\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ x))$›
if ‹c ‘ $A \subseteq S$› ‹d ‘ $B \subseteq S$›
proof −
from *that* **have** $*$: ‹$A - c -‘ S = \{\}$› ‹$B - d -‘ S = \{\}$› **by** *auto*
from *that*(*1*) **have** $**$: ‹$c ‘ A \cap d ‘ B \cap S = c ‘ A \cap d ‘ B$› **by** *blast*
show *?thesis* **by** (*subst read-Sync-read*)
(*use that* **in** ‹*simp-all add*: $* **$›)
qed

lemma *read-Sync-read-subset-forced-read-left* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = c?x{\in}(A \cap c -‘ d ‘ B) \to (P\ x\ [\![S]\!]\ Q\ x)$›
if ‹c ‘ $A \subseteq S$› ‹d ‘ $B \subseteq S$› ‹*inj-on* c A› ‹*inj-on* d B›
‹$\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow d\ b \in S \Longrightarrow a = b$›
proof −
from *that* **have** $*$: ‹$A - c -‘ S = \{\}$› ‹$B - d -‘ S = \{\}$› **by** *auto*
from *that*(*1*) **have** $**$: ‹$A \cap (c -‘ d ‘ B \cap c -‘ S) = A \cap c -‘ d ‘ B$› **by** *blast*
show *?thesis* **by** (*subst read-Sync-read-forced-read-left*)
(*use that*(*3*, *4*, *5*) **in** ‹*simp-all add*: $* **$›)
qed

lemma *read-Sync-read-subset-forced-read-right* :
‹⟦c ‘ $A \subseteq S$; d ‘ $B \subseteq S$; *inj-on* c A; *inj-on* d B;
$\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow d\ b \in S \Longrightarrow a = b$⟧ ⟹
$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = d?x{\in}(B \cap d -‘ c ‘ A) \to (P\ x\ [\![S]\!]\ Q\ x)$›
by (*subst Sync-commute*, *subst read-Sync-read-subset-forced-read-left*)
(*simp-all add*: *Sync-commute*, *metis*)

lemma *read-Sync-read-indep* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$(c?a{\in}A \to (P\ a\ [\![S]\!]\ (d?b{\in}B \to Q\ b))) \Box (d?b{\in}B \to ((c?a{\in}A \to P\ a)\ [\![S]\!]\ Q\ b))$›
if ‹c ‘ $A \cap S = \{\}$› ‹d ‘ $B \cap S = \{\}$›
proof −
from *that* **have** $*$: ‹$A - c -‘ S = A$› ‹$B - d -‘ S = B$› **by** *auto*
from *that*(*1*) **have** $**$: ‹$c ‘ A \cap d ‘ B \cap S = \{\}$› **by** *blast*

show *?thesis* **by** (*subst read-Sync-read*) (*use that* **in** ‹*simp-all add*: ∗ ∗∗›)
qed

lemma *read-Sync-read-left* :
‹*c*?*a*∈*A* → *P a* ⟦*S*⟧ *d*?*b*∈*B* → *Q b* = *c*?*a*∈*A* → (*P a* ⟦*S*⟧ (*d*?*b*∈*B* → *Q b*))›
if ‹*c* ‘ *A* ∩ *S* = {}› ‹*d* ‘ *B* ⊆ *S*›
proof −
from *that*(*1*) **have** ∗ : ‹*A* − *c* −‘ *S* = *A*› ‹*c* ‘ *A* ∩ *d* ‘ *B* ∩ *S* = {}› **by** *auto*
from *that*(*2*) **have** ∗∗ : ‹*B* − *d* −‘ *S* = {}› **by** *blast*
show *?thesis* **by** (*subst read-Sync-read*)
(*use that* **in** ‹*simp-all add*: ∗ ∗∗›)
qed

lemma *read-Sync-read-right* :
‹*c* ‘ *A* ⊆ *S* ⟹ *d* ‘ *B* ∩ *S* = {} ⟹
c?*a*∈*A* → *P a* ⟦*S*⟧ *d*?*b*∈*B* → *Q b* = *d*?*b*∈*B* → ((*c*?*a*∈*A* → *P a*) ⟦*S*⟧ *Q b*)›
by (*subst Sync-commute*, *subst read-Sync-read-left*)
(*simp-all add*: *Sync-commute*)

corollary *read-Par-read* :
‹*c*?*a*∈*A* → *P a* || *d*?*b*∈*B* → *Q b* =
□*x*∈(*c* ‘ *A* ∩ *d* ‘ *B*) → (*P* (*inv-into A c x*) || *Q* (*inv-into B d x*))›
by (*simp add*: *read-Sync-read-subset*)

corollary *read-Par-read-forced-read-left* :
‹⟦*inj-on c A*; *inj-on d B*; ⋀*a b*. *a* ∈ *A* ⟹ *b* ∈ *B* ⟹ *c a* = *d b* ⟹ *a* = *b*⟧ ⟹
c?*a*∈*A* → *P a* || *d*?*b*∈*B* → *Q b* = *c*?*x*∈(*A* ∩ *c* −‘ *d* ‘ *B*) → (*P x* || *Q x*)›
by (*subst read-Sync-read-forced-read-left*) *simp-all*

corollary *read-Par-read-forced-read-right* :
‹⟦*inj-on c A*; *inj-on d B*; ⋀*a b*. *a* ∈ *A* ⟹ *b* ∈ *B* ⟹ *c a* = *d b* ⟹ *a* = *b*⟧ ⟹
c?*a*∈*A* → *P a* || *d*?*b*∈*B* → *Q b* = *d*?*x*∈(*B* ∩ *d* −‘ *c* ‘ *A*) → (*P x* || *Q x*)›
by (*subst read-Sync-read-forced-read-right*) *simp-all*

corollary *read-Inter-read* :
‹⟦*inj-on c A*; *inj-on d B*; ⋀*a b*. *a* ∈ *A* ⟹ *b* ∈ *B* ⟹ *c a* = *d b* ⟹ *a* = *b*⟧ ⟹
c?*a*∈*A* → *P a* ||| *d*?*b*∈*B* → *Q b* =
(*c*?*a*∈*A* → (*P a* ||| *d*?*b*∈*B* → *Q b*)) □ (*d*?*b*∈*B* → (*c*?*a*∈*A* → *P a* ||| *Q b*))›
by (*simp add*: *read-Sync-read*)

Same channel Some results can be rewritten when we have the same channel.

lemma *read-Sync-read-forced-read-same-chan* :
‹*c*?*a*∈*A* → *P a* ⟦*S*⟧ *c*?*b*∈*B* → *Q b* =
(*c*?*a*∈(*A* − *c* −‘ *S*) → (*P a* ⟦*S*⟧ *c*?*b*∈*B* → *Q b*)) □
(*c*?*b*∈(*B* − *c* −‘ *S*) → (*c*?*a*∈*A* → *P a* ⟦*S*⟧ *Q b*)) □

$(c?x{\in}(A \cap B \cap c -` S) \rightarrow (P\ x\ [\![S]\!]\ Q\ x))\rangle$
(**is** $\langle ?lhs = ?rhs1 \ \Box\ ?rhs2 \ \Box\ ?rhs3\rangle$)
if $\langle c ` A \cap S = \{\} \vee inj\text{-}on\ c\ A\rangle$ $\langle c ` B \cap S = \{\} \vee inj\text{-}on\ c\ B\rangle$
$\langle \bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = c\ b \Longrightarrow c\ b \in S \Longrightarrow a = b\rangle$
proof −
— Actually, the third assumption is equivalent to the following (we of course do not use that(3) in the proof of equivalence).
from *that*(*1*, *2*)
have $\langle inj\text{-}on\ c\ ((A \cup B) \cap c -` S) \longleftrightarrow$
$(\forall a\ b.\ a \in A \longrightarrow b \in B \longrightarrow c\ a = c\ b \longrightarrow c\ b \in S \longrightarrow a = b)\rangle$
by (*elim disjE*, *simp-all add*: *inj-on-def*)
((*auto*)[*3*], *metis Int-iff Un-iff vimageE vimageI*)

from *that*(*3*) **have** $* : \langle A \cap (c -` c ` B \cap c -` S) = A \cap B \cap c -` S\rangle$ **by** *auto blast*
show *?thesis* **by** (*simp add*: *read-Sync-read-forced-read-left that* $*$)
qed

lemma *read-Sync-read-forced-read-same-chan-weaker* :
— Easier with a stronger assumption.
$\langle inj\text{-}on\ c\ (A \cup B) \Longrightarrow$
$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ c?b{\in}B \rightarrow Q\ b =$
$(c?a{\in}(A - c -` S) \rightarrow (P\ a\ [\![S]\!]\ c?b{\in}B \rightarrow Q\ b)) \ \Box$
$(c?b{\in}(B - c -` S) \rightarrow (c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ Q\ b)) \ \Box$
$(c?x{\in}(A \cap B \cap c -` S) \rightarrow (P\ x\ [\![S]\!]\ Q\ x))\rangle$
by (*rule read-Sync-read-forced-read-same-chan*)
(*simp-all add*: *inj-on-Un*, *metis Un-iff inj-onD inj-on-Un*)

lemma *read-Sync-read-subset-forced-read-same-chan* :
— In the subset case, the assumption *inj-on c* $(A \cup B)$ is equivalent. The result is not weaker anymore.
$\langle c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ c?b{\in}B \rightarrow Q\ b = c?x{\in}(A \cap B) \rightarrow (P\ x\ [\![S]\!]\ Q\ x)\rangle$
if $\langle c ` A \subseteq S\rangle$ $\langle c ` B \subseteq S\rangle$ $\langle inj\text{-}on\ c\ (A \cup B)\rangle$
proof −
from *that*(*3*) **have** $\langle A \cap c -` c ` B = A \cap B\rangle$ **by** (*auto simp add*: *inj-on-def*)
with *that*(*3*) **show** *?thesis*
by (*subst read-Sync-read-subset-forced-read-left*)
(*simp-all add*: *that*(*1*, *2*) *inj-on-Un*, *meson Un-iff inj-on-contraD that*(*3*))
qed

read **and** *ndet-write*. **lemma** *ndet-write-Sync-read* :
$\langle c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b =$
(*if* $A = \{\}$ *then STOP* $[\![S]\!]\ d?b{\in}B \rightarrow Q\ b$
else $\sqcap a{\in}c ` A.$ (*if* $a \in S$ *then STOP else* $a \rightarrow (P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b)) \ \Box$
$(\Box b{\in}(d ` B - S) \rightarrow (a \rightarrow P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ b))) \ \Box$
(*if* $a \in d ` B \cap S$ *then* $a \rightarrow (P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ Q\ (inv\text{-}into\ B$

d a)) else STOP))›
by (*auto simp add*: *ndet-write-def read-def Mndetprefix-Sync-Mprefix*
intro: *mono-GlobalNdet-eq arg-cong2*[**where** *f* = ‹(□)›])

lemma *read-Sync-ndet-write* :
‹*c*?*a*∈*A* → *P a* ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* =
(*if B* = {} *then c*?*a*∈*A* → *P a* ⟦*S*⟧ *STOP*
else ⊓*b*∈*d* ‘ *B*. (*if b* ∈ *S then STOP else b* → (*c*?*a*∈*A* → *P a* ⟦*S*⟧ *Q* (*inv-into B d b*))) □
(□*a*∈(*c* ‘ *A* − *S*) → (*P* (*inv-into A c a*) ⟦*S*⟧ *b* → *Q* (*inv-into B d b*))) □
(*if b* ∈ *c* ‘ *A* ∩ *S then b* → (*P* (*inv-into A c b*) ⟦*S*⟧ *Q* (*inv-into B d b*)) *else STOP*))›
by (*auto simp add*: *ndet-write-def read-def Mprefix-Sync-Mndetprefix*
intro: *mono-GlobalNdet-eq arg-cong2*[**where** *f* = ‹(□)›])

lemma *ndet-write-Sync-read-subset* :
‹*c* ‘ *A* ⊆ *S* ⟹ *d* ‘ *B* ⊆ *S* ⟹
c!!*a*∈*A* → *P a* ⟦*S*⟧ *d*?*b*∈*B* → *Q b* =
(*if c* ‘ *A* ⊆ *d* ‘ *B then* ⊓*a*∈*c* ‘ *A* → (*P* (*inv-into A c a*) ⟦*S*⟧ *Q* (*inv-into B d a*))
else (⊓*a*∈(*c* ‘ *A* ∩ *d* ‘ *B*) → (*P* (*inv-into A c a*) ⟦*S*⟧ *Q* (*inv-into B d a*))) ⊓ *STOP*)›
by (*simp add*: *read-def ndet-write-def Mndetprefix-Sync-Mprefix-subset*)

lemma *read-Sync-ndet-write-subset* :
‹*c* ‘ *A* ⊆ *S* ⟹ *d* ‘ *B* ⊆ *S* ⟹
c?*a*∈*A* → *P a* ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* =
(*if d* ‘ *B* ⊆ *c* ‘ *A then* ⊓*b*∈*d* ‘ *B* → (*P* (*inv-into A c b*) ⟦*S*⟧ *Q* (*inv-into B d b*))
else (⊓*b*∈(*c* ‘ *A* ∩ *d* ‘ *B*) → (*P* (*inv-into A c b*) ⟦*S*⟧ *Q* (*inv-into B d b*))) ⊓ *STOP*)›
by (*simp add*: *read-def ndet-write-def Mprefix-Sync-Mndetprefix-subset*)

— If we have the same injective channel, it's better.
lemma *ndet-write-Sync-read-subset-same-chan*:
‹*c*!!*a*∈*A* → *P a* ⟦*S*⟧ *c*?*b*∈*B* → *Q b* =
(*if A* ⊆ *B then c*!!*a*∈*A* → (*P a* ⟦*S*⟧ *Q a*) *else* (*c*!!*a*∈(*A* ∩ *B*) → (*P a* ⟦*S*⟧ *Q a*)) ⊓ *STOP*)›
if ‹*c* ‘ *A* ⊆ *S*› ‹*c* ‘ *B* ⊆ *S*› ‹*inj-on c* (*A* ∪ *B*)›
proof −
from ‹*inj-on c* (*A* ∪ *B*)› **have** ∗ : ‹*c* ‘ *A* ⊆ *c* ‘ *B* ⟷ *A* ⊆ *B*›
by (*auto simp add*: *inj-on-eq-iff*)
from ‹*inj-on c* (*A* ∪ *B*)› **have** ∗∗ : ‹*c* ‘ *A* ∩ *c* ‘ *B* = *c* ‘ (*A* ∩ *B*)›
by (*auto simp add*: *inj-on-Un*)
from ‹*inj-on c* (*A* ∪ *B*)› **show** *?thesis*
by (*unfold ndet-write-Sync-read-subset*[*OF* ‹*c* ‘ *A* ⊆ *S*› ‹*c* ‘ *B* ⊆ *S*›] ∗ ∗∗)
(*auto simp add*: *ndet-write-def inj-on-Un inj-on-Int*
intro!: *mono-Mndetprefix-eq arg-cong2*[**where** *f* = ‹(⊓)›])

qed

lemma *read-Sync-ndet-write-subset-same-chan*:
‹c ‘ $A \subseteq S \Longrightarrow c$ ‘ $B \subseteq S \Longrightarrow$ *inj-on* $c\ (A \cup B) \Longrightarrow$
$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ c!!b{\in}B \rightarrow Q\ b =$
(*if* $B \subseteq A$ *then* $c!!b{\in}B \rightarrow (P\ b\ [\![S]\!]\ Q\ b)$ *else* $(c!!b{\in}(A \cap B) \rightarrow (P\ b\ [\![S]\!]\ Q\ b))$
$\sqcap$ *STOP*)›
by (*subst* (*1 2 3*) *Sync-commute*)
(*simp add*: *Int-commute Un-commute ndet-write-Sync-read-subset-same-chan*)

lemma *ndet-write-Sync-read-indep* :
‹c ‘ $A \cap S = \{\} \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow$
$c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b =$
(*if* $A = \{\}$ *then* $d?b{\in}B \rightarrow$ $(STOP\ [\![S]\!]\ Q\ b)$
else $\sqcap a{\in}c$ ‘ $A.\ (a \rightarrow (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b)) \ \Box$
$(d?b{\in}B \rightarrow (a \rightarrow P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ Q\ b)))$›
by (*auto simp add*: *ndet-write-def read-def Mndetprefix-Sync-Mprefix-indep comp-def*
intro: *mono-GlobalNdet-eq arg-cong2*[**where** $f = \langle(\Box)\rangle$])

lemma *read-Sync-ndet-write-indep* :
‹c ‘ $A \cap S = \{\} \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow$
$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b =$
(*if* $B = \{\}$ *then* $c?a{\in}A \rightarrow (P\ a\ [\![S]\!]\ STOP)$
else $\sqcap b{\in}d$ ‘ $B.\ (b \rightarrow (c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ b))) \ \Box$
$(c?a{\in}A \rightarrow (P\ a\ [\![S]\!]\ b \rightarrow Q\ (\textit{inv-into}\ B\ d\ b))))$›
by (*auto simp add*: *ndet-write-def read-def Mprefix-Sync-Mndetprefix-indep comp-def*
intro: *mono-GlobalNdet-eq arg-cong2*[**where** $f = \langle(\Box)\rangle$])

lemma *ndet-write-Sync-read-left* :
‹$c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b = c!!a{\in}A \rightarrow (P\ a\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹c ‘ $A \cap S = \{\}$› ‹d ‘ $B \subseteq S$›
proof −
from *that* **have** ‹*?lhs* = (*if* $A = \{\}$ *then* $STOP\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b$
else $c!!a{\in}A \rightarrow (P\ a\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b)$)›
by (*auto simp add*: *ndet-write-def read-def Mndetprefix-Sync-Mprefix-left comp-def*
intro: *mono-GlobalNdet-eq arg-cong2*[**where** $f = \langle(\Box)\rangle$])
also have ‹. . . = *?rhs*› **by** (*simp add*: *read-def ndet-write-def Mprefix-is-STOP-iff*
STOP-Sync-Mprefix that(*2*))
finally show ‹*?lhs* = *?rhs*› **.**
qed

lemma *read-Sync-ndet-write-left* :
‹$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b = c?a{\in}A \rightarrow (P\ a\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹c ‘ $A \cap S = \{\}$› ‹d ‘ $B \subseteq S$›
proof −
from *that* **have** ‹*?lhs* = (*if* $B = \{\}$ *then* $(c?a{\in}A \rightarrow P\ a)\ [\![S]\!]\ STOP$ *else ?rhs*)›
by (*auto simp add*: *ndet-write-def read-def Mprefix-Sync-Mndetprefix-left comp-def*

intro: *mono-GlobalNdet-eq arg-cong2*[**where** $f = \langle(\Box)\rangle$])
also have $\langle \ldots = ?rhs\rangle$
by (*simp add*: *read-def comp-def*)
(*use Mprefix-Sync-Mprefix-left that*(*1*) **in** *force*)
finally show $\langle ?lhs = ?rhs\rangle$ **.**
qed

lemma *ndet-write-Sync-read-right* :
$\langle c \mathbin{`} A \subseteq S \Longrightarrow d \mathbin{`} B \cap S = \{\} \Longrightarrow$
$c!!a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = d?b{\in}B \to (c!!a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)\rangle$
by (*subst* (*1 2*) *Sync-commute*) (*simp add*: *read-Sync-ndet-write-left*)

lemma *read-Sync-ndet-write-right* :
$\langle c \mathbin{`} A \subseteq S \Longrightarrow d \mathbin{`} B \cap S = \{\} \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b = d!!b{\in}B \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)\rangle$
by (*subst* (*1 2*) *Sync-commute*) (*simp add*: *ndet-write-Sync-read-left*)

read **and** *write***.** **lemma** *write-Sync-read* :
$\langle c!a \to P\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
(*if* $c\ a \in S$ *then STOP else* $c!a \to (P\ [\![S]\!]\ d?b{\in}B \to Q\ b))\ \Box$
$(\Box b{\in}(d \mathbin{`} B - S) \to (c!a \to P\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ b)))\ \Box$
(*if* $c\ a \in d \mathbin{`} B \cap S$ *then* $c!a \to (P\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ (c\ a)))$ *else STOP*)$\rangle$
by (*subst ndet-write-Sync-read*[**where** $A = \langle\{a\}\rangle$, *simplified*])
(*simp add*: *write-is-write0 image-iff*)

lemma *read-Sync-write* :
$\langle c?a{\in}A \to P\ a\ [\![S]\!]\ d!b \to Q =$
(*if* $d\ b \in S$ *then STOP else* $d!b \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q))\ \Box$
$(\Box a{\in}(c \mathbin{`} A - S) \to (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ d!b \to Q))\ \Box$
(*if* $d\ b \in c \mathbin{`} A \cap S$ *then* $d!b \to (P\ (\textit{inv-into}\ A\ c\ (d\ b))\ [\![S]\!]\ Q)$ *else STOP*)$\rangle$
by (*subst read-Sync-ndet-write*[**where** $B = \langle\{b\}\rangle$, *simplified*])
(*simp add*: *write-is-write0 image-iff*)

lemma *write-Sync-read-subset* :
$\langle c\ a \in S \Longrightarrow d \mathbin{`} B \subseteq S \Longrightarrow$
$c!a \to P\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
(*if* $c\ a \in d \mathbin{`} B$ *then* $c!a \to (P\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ (c\ a)))$ *else STOP*)$\rangle$
by (*simp add*: *write-Sync-read*)
(*metis Det-STOP Det-commute Diff-eq-empty-iff Mprefix-empty*)

lemma *read-Sync-write-subset* :
$\langle c \mathbin{`} A \subseteq S \Longrightarrow d\ b \in S \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d!b \to Q =$
(*if* $d\ b \in c \mathbin{`} A$ *then* $d!b \to (P\ (\textit{inv-into}\ A\ c\ (d\ b))\ [\![S]\!]\ Q)$ *else STOP*)$\rangle$
by (*metis Sync-commute write-Sync-read-subset*)

— If we have the same injective channel, it's better.
lemma *write-Sync-read-subset-same-chan*:
$\langle c\ a \in S \Longrightarrow c\ `\ B \subseteq S \Longrightarrow$ *inj-on* c (*insert* $a\ B$) $\Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ c?b{\in}B \rightarrow Q\ b = (\textit{if}\ a \in B\ \textit{then}\ c!a \rightarrow (P\ [\![S]\!]\ Q\ a)\ \textit{else}\ STOP)\rangle$
by (*subst ndet-write-Sync-read-subset-same-chan*[**where** $A = \langle\{a\}\rangle$, *simplified*])
simp-all

lemma *read-Sync-write-subset-same-chan*:
$\langle c\ `\ A \subseteq S \Longrightarrow c\ b \in S \Longrightarrow$ *inj-on* c (*insert* $b\ A$) $\Longrightarrow$
$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ c!b \rightarrow Q = (\textit{if}\ b \in A\ \textit{then}\ c!b \rightarrow (P\ b\ [\![S]\!]\ Q)\ \textit{else}\ STOP)\rangle$
by (*metis Sync-commute write-Sync-read-subset-same-chan*)

lemma *write-Sync-read-indep* :
$\langle c\ a \notin S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b =$
$(c!a \rightarrow (P\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b)) \Box (d?b{\in}B \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q\ b))\rangle$
by (*subst ndet-write-Sync-read-indep*[**where** $A = \langle\{a\}\rangle$, *simplified*])
(*simp-all add*: *write-is-write0*)

lemma *read-Sync-write-indep* :
$\langle c\ `\ A \cap S = \{\} \Longrightarrow d\ b \notin S \Longrightarrow$
$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!b \rightarrow Q =$
$(d!b \rightarrow (c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ Q)) \Box (c?a{\in}A \rightarrow (P\ a\ [\![S]\!]\ d!b \rightarrow Q))\rangle$
by (*subst read-Sync-ndet-write-indep*[**where** $B = \langle\{b\}\rangle$, *simplified*])
(*simp-all add*: *write-is-write0*)

lemma *write-Sync-read-left* :
$\langle c\ a \notin S \Longrightarrow d\ `\ B \subseteq S \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b = c!a \rightarrow (P\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b)\rangle$
by (*subst ndet-write-Sync-read-left*[**where** $A = \langle\{a\}\rangle$, *simplified*]) *simp-all*

lemma *read-Sync-write-left* :
$\langle c\ `\ A \cap S = \{\} \Longrightarrow d\ b \in S \Longrightarrow$
$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!b \rightarrow Q = c?a{\in}A \rightarrow (P\ a\ [\![S]\!]\ d!b \rightarrow Q)\rangle$
by (*subst read-Sync-ndet-write-left*[**where** $B = \langle\{b\}\rangle$, *simplified*]) *simp-all*

lemma *write-Sync-read-right* :
$\langle c\ a \in S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b = d?b{\in}B \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q\ b)\rangle$
by (*subst* (*1 2*) *Sync-commute*) (*simp add*: *read-Sync-write-left*)

lemma *read-Sync-write-right* :
$\langle c\ `\ A \subseteq S \Longrightarrow d\ b \notin S \Longrightarrow$
$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!b \rightarrow Q = d!b \rightarrow (c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ Q)\rangle$
by (*subst* (*1 2*) *Sync-commute*) (*simp add*: *write-Sync-read-left*)

ndet-write **and** *ndet-write* **lemma** *ndet-write-Sync-ndet-write* :
‹*c*!!*a*∈*A* → *P a* ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* =
(*if A* = {} *then if d* ‘ *B* ∩ *S* = {} *then d*!!*b*∈*B* → (*STOP* ⟦*S*⟧ *Q b*)
else (⊓*x*∈*d* ‘ (*B* − *d* −‘ *S*) → (*STOP* ⟦*S*⟧ *Q* (*inv-into B d x*))) ⊓ *STOP*
else if B = {} *then if c* ‘ *A* ∩ *S* = {} *then c*!!*a*∈*A* → (*P a* ⟦*S*⟧ *STOP*)
else (⊓*x*∈*c* ‘ (*A* − *c* −‘ *S*) → (*P* (*inv-into A c x*) ⟦*S*⟧ *STOP*)) ⊓ *STOP*
else ⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*.
(*if a* ∈ *S then STOP else a* → (*P* (*inv-into A c a*) ⟦*S*⟧ *b* → *Q* (*inv-into B d b*))) □
(*if b* ∈ *S then STOP else b* → (*a* → *P* (*inv-into A c a*) ⟦*S*⟧ *Q* (*inv-into B d b*))) □
(*if a* = *b* ∧ *b* ∈ *S then b* → (*P* (*inv-into A c a*) ⟦*S*⟧ *Q* (*inv-into B d b*)) *else STOP*))›
(**is** ‹*?lhs* = (*if A* = {} *then if d* ‘ *B* ∩ *S* = {} *then ?mv-right else ?mv-right′* ⊓ *STOP*
else if B = {} *then if c* ‘ *A* ∩ *S* = {} *then ?mv-left else ?mv-left′* ⊓ *STOP*
else ?huge-mess)›)
proof (*split if-split*, *intro conjI impI*)
have ‹*d* ‘ (*B* − *d* −‘ *S*) = *d* ‘ *B* − *S*› **by** *blast*
thus ‹*A* = {} ⟹ *?lhs* = (*if d* ‘ *B* ∩ *S* = {} *then ?mv-right else ?mv-right′* ⊓ *STOP*)›
by (*auto simp add*: *ndet-write-def STOP-Sync-Mndetprefix comp-def*
intro: *mono-Mndetprefix-eq*)
next
show ‹*?lhs* = (*if B* = {} *then if c* ‘ *A* ∩ *S* = {} *then ?mv-left else ?mv-left′* ⊓ *STOP else ?huge-mess*)› **if** ‹*A* ≠ {}›
proof (*split if-split*, *intro conjI impI*)
have ‹*c* ‘ (*A* − *c* −‘ *S*) = *c* ‘ *A* − *S*› **by** *blast*
thus ‹*B* = {} ⟹ *?lhs* = (*if c* ‘ *A* ∩ *S* = {} *then ?mv-left else ?mv-left′* ⊓ *STOP*)›
by (*auto simp add*: *ndet-write-def Mndetprefix-Sync-STOP comp-def*
intro: *mono-Mndetprefix-eq*)
next
assume ‹*B* ≠ {}›
have ‹*?lhs* = (⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. (*a* → *P* (*inv-into A c a*) ⟦*S*⟧ (*b* → *Q* (*inv-into B d b*))))›
by (*simp add*: *ndet-write-def Mndetprefix-GlobalNdet* ‹*A* ≠ {}› ‹*B* ≠ {}›
Sync-distrib-GlobalNdet-left Sync-distrib-GlobalNdet-right)
also have ‹. . . = *?huge-mess*›
by (*auto simp add*: *write0-def Mprefix-Sync-Mprefix Diff-triv Mprefix-is-STOP-iff disjoint-iff*
intro!: *mono-GlobalNdet-eq arg-cong2*[**where** *f* = ‹(□)›])
finally show ‹*?lhs* = *?huge-mess*› **.**
qed
qed

lemma *ndet-write-Sync-ndet-write-subset* :
‹*c*!!*a*∈*A* → *P a* ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* =
(*if* ∃ *b*. *c* ‘ *A* = {*b*} ∧ *d* ‘ *B* = {*b*}
then (*THE b*. *d* ‘ *B* = {*b*}) → (*P* (*inv-into A c* (*THE a*. *c* ‘ *A* = {*a*})) ⟦*S*⟧ *Q* (*inv-into B d* (*THE b*. *d* ‘ *B* = {*b*})))
else (⊓*x*∈(*c* ‘ *A* ∩ *d* ‘ *B*) → (*P* (*inv-into A c x*) ⟦*S*⟧ *Q* (*inv-into B d x*))) ⊓ *STOP*)›
(**is** ‹*?lhs* = (*if ?cond then ?rhs1 else ?rhs2*)›) **if** ‹*c* ‘ *A* ⊆ *S*› ‹*d* ‘ *B* ⊆ *S*›
proof (*split if-split*, *intro conjI impI*)
show ‹*?cond* ⟹ *?lhs* = *?rhs1*›
by (*elim exE*, *simp add*: *ndet-write-is-GlobalNdet-write0 write0-def*)
(*subst Mprefix-Sync-Mprefix-subset*, *use* ‹*c* ‘ *A* ⊆ *S*› **in** *simp-all*)
next
assume ‹¬ *?cond*›
let *?term* = ‹λ*a b*. (*b* → (*P* (*inv-into A c a*) ⟦*S*⟧ *Q* (*inv-into B d b*)))›
have ‹*?lhs* = ⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. (*if a* = *b then ?term a b else STOP*)›
(**is** ‹*?lhs* = ⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*›)
proof (*cases* ‹*A* = {} ∨ *B* = {}›)
from ‹*c* ‘ *A* ⊆ *S*› ‹*d* ‘ *B* ⊆ *S*› **show** ‹*A* = {} ∨ *B* = {} ⟹ *?lhs* = (⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*)›
by (*elim disjE*) (*simp-all add*: *ndet-write-def Mndetprefix-Sync-STOP STOP-Sync-Mndetprefix Int-absorb2 Mndetprefix-is-STOP-iff Ndet-is-STOP-iff*)
next
show ‹¬ (*A* = {} ∨ *B* = {}) ⟹ *?lhs* = (⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*)›
by (*simp add*: *ndet-write-Sync-ndet-write*)
(*intro mono-GlobalNdet-eq*, *use* ‹*c* ‘ *A* ⊆ *S*› ‹*d* ‘ *B* ⊆ *S*› **in** *auto*)
qed

also have ‹(⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*) = *?rhs2*›
proof (*cases* ‹*d* ‘ *B* ∩ *c* ‘ *A* = {}›)
assume ‹*d* ‘ *B* ∩ *c* ‘ *A* = {}›
hence ‹*c* ‘ *A* ∩ *d* ‘ *B* = {}› **by** *blast*
hence ‹(⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*) = *STOP*› **by** (*auto simp add*: *Global-Ndet-is-STOP-iff*)
thus ‹(⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*) = *?rhs2*›
by (*auto simp add*: ‹*c* ‘ *A* ∩ *d* ‘ *B* = {}›)
next
show ‹(⊓*b*∈*d* ‘ *B*. ⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*) = *?rhs2*› **if** ‹*d* ‘ *B* ∩ *c* ‘ *A* ≠ {}›
proof (*cases* ‹*d* ‘ *B* − *c* ‘ *A* = {}›)
assume ‹*d* ‘ *B* − *c* ‘ *A* = {}›
hence ‹*c* ‘ *A* ∩ *d* ‘ *B* = *d* ‘ *B*› **by** *blast*
have ‹(⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*) = (*if c* ‘ *A* = {*b*} *then ?term b b else ?term b b* ⊓ *STOP*)›
(**is** ‹(⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*) = *?rhs*′′ *b*›) **if** ‹*b* ∈ *d* ‘ *B*› **for** *b*
proof (*cases* ‹*c* ‘ *A* ∩ {*b*} = {}›)
from ‹*d* ‘ *B* − *c* ‘ *A* = {}› ‹*b* ∈ *d* ‘ *B*›
show ‹*c* ‘ *A* ∩ {*b*} = {} ⟹ (⊓*a*∈*c* ‘ *A*. *?rhs*′ *b a*) = *?rhs*′′ *b*› **by** *auto*
next

```
      show ‹(⊓a∈c ‘ A. ?rhs' b a) = ?rhs'' b› if ‹c ‘ A ∩ {b} ≠ {}›
      proof (cases ‹c ‘ A − {b} = {}›)
        show ‹c ‘ A − {b} = {} ⟹ (⊓a∈c ‘ A. ?rhs' b a) = ?rhs'' b›
          using ‹c ‘ A ∩ {b} ≠ {}› by auto
      next
        show ‹(⊓a∈c ‘ A. ?rhs' b a) = ?rhs'' b› if ‹c ‘ A − {b} ≠ {}›
          using ‹c ‘ A − {b} ≠ {}› ‹c ‘ A ∩ {b} ≠ {}›
          by (auto simp add: GlobalNdet-is-STOP-iff
              simp flip: GlobalNdet-factorization-union
              [OF ‹c ‘ A ∩ {b} ≠ {}› ‹c ‘ A − {b} ≠ {}›, unfolded Int-Diff-Un]
              intro: arg-cong2[where f = ‹(⊓)›])
      qed
    qed
    hence ‹(⊓b ∈ d ‘ B. ⊓a∈c ‘ A. ?rhs' b a) = ⊓b ∈ d ‘ B. ?rhs'' b›
      by (fact mono-GlobalNdet-eq)
    also have ‹(⊓b ∈ d ‘ B. ?rhs'' b) = ?rhs2›
    proof −
      from ‹¬ ?cond› have ‹(⊓b ∈ d ‘ B. ?rhs'' b) = ⊓b ∈ d ‘ B. ?term b b ⊓
STOP›
        by (metis Diff-eq-empty-iff Int-commute ‹c ‘ A ∩ d ‘ B = d ‘ B›
            ‹d ‘ B − c ‘ A = {}› subset-singleton-iff ‹d ‘ B ∩ c ‘ A ≠ {}›)
      also have ‹... = (⊓b ∈ d ‘ B. ?term b b) ⊓ STOP›
        by (simp add: Process-eq-spec Ndet-projs GlobalNdet-projs STOP-projs)
      finally show ‹(⊓b ∈ d ‘ B. ?rhs'' b) = ?rhs2›
        by (simp add: Mndetprefix-GlobalNdet ‹c ‘ A ∩ d ‘ B = d ‘ B›)
    qed
    finally show ‹(⊓b ∈ d ‘ B. ⊓a∈c ‘ A. ?rhs' b a) = ?rhs2› .
  next
    assume ‹d ‘ B − c ‘ A ≠ {}›
    have ‹⊓a∈c ‘ A. (if a = b then ?term a b else STOP) =
        (if b ∈ c ‘ A then if c ‘ A = {b} then ?term b b else (?term b b) ⊓ STOP
else STOP)›
      if ‹b ∈ d ‘ B› for b
    proof (split if-split, intro conjI impI)
      show ‹⊓a∈c ‘ A. (if a = b then ?term a b else STOP) =
          (if c ‘ A = {b} then ?term b b else (?term b b) ⊓ STOP)› if ‹b ∈ c ‘ A›
      proof (split if-split, intro conjI impI)
        show ‹c ‘ A = {b} ⟹ ⊓a∈c ‘ A. (if a = b then ?term a b else STOP) =
?term b b› by simp
      next
        assume ‹c ‘ A ≠ {b}›
        with ‹b ∈ c ‘ A› have ‹insert b (c ‘ A) = c ‘ A› ‹c ‘ A − {b} ≠ {}› by
auto
        show ‹c ‘ A ≠ {b} ⟹ ⊓a∈c ‘ A. (if a = b then ?term a b else STOP) =
?term b b ⊓ STOP›
          by (auto simp add: GlobalNdet-is-STOP-iff intro!: arg-cong2[where f =
‹(⊓)›]
              simp flip: GlobalNdet-factorization-union
              [of ‹{b}›, OF - ‹c ‘ A − {b} ≠ {}›, simplified, unfolded ‹insert b (c ‘
```

```
A) = c ‘ A›])
      qed
    next
      show ‹b ∉ c ‘ A ⟹ ⊓a∈c ‘ A. (if a = b then ?term a b else STOP) =
STOP›
        by (auto simp add: GlobalNdet-is-STOP-iff)
    qed

    hence ‹⊓b ∈ d ‘ B. ⊓a∈c ‘ A. ?rhs' b a =
          ⊓b ∈ d ‘ B. (if b ∈ c ‘ A then if c ‘ A = {b} then ?term b b else (?term
b b) ⊓ STOP else STOP)›
      by (fact mono-GlobalNdet-eq)
    also from ‹d ‘ B − c ‘ A ≠ {}› have ‹... = (⊓b ∈ d ‘ B. (if b ∈ c ‘ A then
?term b b else STOP)) ⊓ STOP›
      by (simp add: Process-eq-spec GlobalNdet-projs, safe)
        (simp-all add: GlobalNdet-projs STOP-projs Ndet-projs split: if-split-asm,
auto)
    also have ‹... = ?rhs2›
    proof (fold GlobalNdet-factorization-union
        [OF ‹d ‘ B ∩ c ‘ A ≠ {}› ‹d ‘ B − c ‘ A ≠ {}›, unfolded Int-Diff-Un])
      have ‹⊓b∈(d ‘ B ∩ c ‘ A). (if b ∈ c ‘ A then ?term b b else STOP) =
          ⊓b∈(d ‘ B ∩ c ‘ A). ?term b b› by (auto intro: mono-GlobalNdet-eq)
       moreover have ‹⊓b∈(d ‘ B − c ‘ A). (if b ∈ c ‘ A then ?term b b else
STOP) = STOP›
        by (simp add: GlobalNdet-is-STOP-iff)
       ultimately show ‹(⊓b∈(d ‘ B ∩ c ‘ A). (if b ∈ c ‘ A then ?term b b else
STOP)) ⊓
                  (⊓b∈(d ‘ B − c ‘ A). (if b ∈ c ‘ A then ?term b b else STOP))
⊓ STOP = ?rhs2›
        by (metis Mndetprefix-GlobalNdet Int-commute Ndet-assoc Ndet-id)
    qed
    finally show ‹(⊓b ∈ d ‘ B. ⊓a∈c ‘ A. ?rhs' b a) = ?rhs2› .
  qed
 qed
 finally show ‹?lhs = ?rhs2› .
qed


lemma ndet-write-Sync-ndet-write-indep :
 ‹c ‘ A ∩ S = {} ⟹ d ‘ B ∩ S = {} ⟹
  c!!a∈A → P a ⟦S⟧ d!!b∈B → Q b =
  ( if A = {} then d!!b∈B → (STOP ⟦S⟧ Q b)
   else if B = {} then c!!a∈A → (P a ⟦S⟧ STOP)
        else ⊓b∈d ‘ B. ⊓a∈c ‘ A.
            ((a → (P (inv-into A c a) ⟦S⟧ b → Q (inv-into B d b)))) □
            ((b → (a → P (inv-into A c a) ⟦S⟧ Q (inv-into B d b)))))›
 by (auto simp add: ndet-write-Sync-ndet-write disjoint-iff intro!: mono-GlobalNdet-eq)
```

lemma *ndet-write-Sync-ndet-write-left* :
‹c**!!**$a{\in}A \to P\ a\ [\![S]\!]\ d$**!!**$b{\in}B \to Q\ b = c$**!!**$a{\in}A \to (P\ a\ [\![S]\!]\ d$**!!**$b{\in}B \to Q\ b)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹c ‘ $A \cap S = \{\}$› ‹d ‘ $B \subseteq S$›
proof −
let *?rhs′* = ‹$\sqcap b{\in}d$ ‘ $B.\ \sqcap a{\in}c$ ‘ $A.\ a \to (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ b \to Q\ (\textit{inv-into}\ B\ d\ b))$›
have ‹*?lhs* = (*if* $A = \{\}$ *then* *if* d ‘ $B \cap S = \{\}$ *then* d**!!**$b{\in}B \to (STOP\ [\![S]\!]\ Q\ b)$
else $(\sqcap x{\in}d$ ‘ $(B - d -‘ S) \to (STOP\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ x))) \sqcap STOP$
else *if* $B = \{\}$ *then* *if* c ‘ $A \cap S = \{\}$ *then* c**!!**$a{\in}A \to (P\ a\ [\![S]\!]\ STOP)$
else $(\sqcap x{\in}c$ ‘ $(A - c -‘ S) \to (P\ (\textit{inv-into}\ A\ c\ x)\ [\![S]\!]\ STOP)) \sqcap STOP$
else $\sqcap b{\in}d$ ‘ $B.\ \sqcap a{\in}c$ ‘ $A.$
(*if* $a \in S$ *then* *STOP* *else* $(a \to (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ b \to Q\ (\textit{inv-into}\ B\ d\ b)))) \ \Box$
(*if* $b \in S$ *then* *STOP* *else* $(b \to (a \to P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ b)))) \ \Box$
(*if* $a = b \wedge b \in S$ *then* $b \to (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ b))$ *else* *STOP*))›
(**is** ‹*?lhs* = (*if* $A = \{\}$ *then* *?rhs1* *else if* $B = \{\}$ *then* *?rhs2* *else* *?rhs3*)›)
by (*fact ndet-write-Sync-ndet-write*)
also from ‹d ‘ $B \subseteq S$› **have** ‹*?rhs1* = *STOP*›
by (*auto simp add*: *Ndet-is-STOP-iff ndet-write-def Mndetprefix-GlobalNdet GlobalNdet-is-STOP-iff*)
also from ‹c ‘ $A \cap S = \{\}$› **have** ‹*?rhs2* = c**!!**$a{\in}A \to (P\ a\ [\![S]\!]\ STOP)$› **by** *presburger*
also from ‹c ‘ $A \cap S = \{\}$› ‹d ‘ $B \subseteq S$›
have ‹*?rhs3* = $\sqcap b{\in}d$ ‘ $B.\ \sqcap a{\in}c$ ‘ $A.\ a \to (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ b \to Q\ (\textit{inv-into}\ B\ d\ b))$›
by (*intro mono-GlobalNdet-eq*) *auto*
finally have ‹*?lhs* = (*if* $A = \{\}$ *then* *STOP*
else *if* $B = \{\}$ *then* c**!!**$a{\in}A \to (P\ a\ [\![S]\!]\ STOP)$
else *?rhs′*)› **.**
moreover have ‹$B \neq \{\} \Longrightarrow$ *?rhs′* = $\sqcap a{\in}c$ ‘ $A.\ a \to (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ \sqcap b{\in}d$ ‘ $B.\ b \to Q\ (\textit{inv-into}\ B\ d\ b))$›
by (*subst GlobalNdet-sets-commute*)
(*simp add*: *Sync-distrib-GlobalNdet-left write0-GlobalNdet*)
moreover have ‹$\ldots = c$**!!**$a{\in}A \to (P\ a\ [\![S]\!]\ d$**!!**$b{\in}B \to Q\ b)$›
by (*simp add*: *ndet-write-is-GlobalNdet-write0*)
ultimately show ‹*?lhs* = *?rhs*› **by** *simp*
qed

lemma *ndet-write-Sync-ndet-write-right* :
‹c ‘ $A \subseteq S \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow$
c**!!**$a{\in}A \to P\ a\ [\![S]\!]\ d$**!!**$b{\in}B \to Q\ b = d$**!!**$b{\in}B \to (c$**!!**$a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)$›
by (*subst* (*1 2*) *Sync-commute*) (*simp add*: *ndet-write-Sync-ndet-write-left*)

ndet-write **and** *write* **lemma** *write-Sync-ndet-write* :
‹$c!a \to P \ [\![S]\!] \ d!!b{\in}B \to Q \ b =$
(*if* $B = \{\}$ *then* $c!a \to P \ [\![S]\!] \ STOP$
else $\sqcap b{\in}d \ `\ B$. (*if* $b \in S$ *then* $STOP$ *else* $b \to (c!a \to P \ [\![S]\!] \ Q \ (inv\text{-}into \ B \ d \ b))) \ \Box$
(*if* $c \ a \in S$ *then* $STOP$ *else* $c!a \to (P \ [\![S]\!] \ b \to Q \ (inv\text{-}into \ B \ d \ b))) \ \Box$
(*if* $b = c \ a \wedge c \ a \in S$ *then* $c!a \to (P \ [\![S]\!] \ Q \ (inv\text{-}into \ B \ d \ (c \ a)))$ *else* $STOP$))›
by (*subst read-Sync-ndet-write*[**where** $A = \langle\{a\}\rangle$, *simplified*],
auto simp add: *write-def Mprefix-singl split*: *if-split-asm*
intro!: *mono-GlobalNdet-eq arg-cong2*[**where** $f = \langle(\Box)\rangle$] *mono-Mprefix-eq*)
(*simp add*: *insert-Diff-if write0-def*)

lemma *ndet-write-Sync-write* :
‹$c!!a{\in}A \to P \ a \ [\![S]\!] \ d!b \to Q =$
(*if* $A = \{\}$ *then* $STOP \ [\![S]\!] \ d!b \to Q$
else $\sqcap a{\in}c \ `\ A$. (*if* $a \in S$ *then* $STOP$ *else* $a \to (P \ (inv\text{-}into \ A \ c \ a) \ [\![S]\!] \ d!b \to Q)) \ \Box$
(*if* $d \ b \in S$ *then* $STOP$ *else* $d!b \to (a \to P \ (inv\text{-}into \ A \ c \ a) \ [\![S]\!] \ Q)) \ \Box$
(*if* $a = d \ b \wedge d \ b \in S$ *then* $d!b \to (P \ (inv\text{-}into \ A \ c \ a) \ [\![S]\!] \ Q)$ *else* $STOP$))›
by (*subst* (*1 2 3 4 5*) *Sync-commute*)
(*auto simp add*: *write-Sync-ndet-write intro*: *mono-GlobalNdet-eq*)

lemma *write-Sync-ndet-write-subset* :
‹$c!a \to P \ [\![S]\!] \ d!!b{\in}B \to Q \ b =$
(*if* $c \ a \notin d \ `\ B$ *then* $STOP$ *else if* $d \ `\ B = \{c \ a\}$ *then* $c!a \to (P \ [\![S]\!] \ Q \ (inv\text{-}into \ B \ d \ (c \ a)))$
else $(c!a \to (P \ [\![S]\!] \ Q \ (inv\text{-}into \ B \ d \ (c \ a)))) \sqcap STOP$)› **if** ‹$c \ a \in S$› ‹$d \ `\ B \subseteq S$›
proof (*subst read-Sync-ndet-write-subset*[**where** $A = \langle\{a\}\rangle$, *simplified*])
from ‹$c \ a \in S$› **show** ‹$c \ a \in S$› **.**
next
from ‹$d \ `\ B \subseteq S$› **show** ‹$d \ `\ B \subseteq S$› **.**
next
show ‹(*if* $d \ `\ B \subseteq \{c \ a\}$ *then* $\sqcap b{\in}d \ `\ B \to (P \ [\![S]\!] \ Q \ (inv\text{-}into \ B \ d \ b))$
else $(\sqcap b{\in}(c \ `\ \{a\} \cap d \ `\ B) \to (P \ [\![S]\!] \ Q \ (inv\text{-}into \ B \ d \ b))) \sqcap STOP) =$
(*if* $c \ a \notin d \ `\ B$ *then* $STOP$ *else if* $d \ `\ B = \{c \ a\}$ *then* $c!a \to (P \ [\![S]\!] \ Q \ (inv\text{-}into \ B \ d \ (c \ a)))$
else $(c!a \to (P \ [\![S]\!] \ Q \ (inv\text{-}into \ B \ d \ (c \ a)))) \sqcap STOP$)›
(**is** ‹*?lhs* = (*if* $c \ a \notin d \ `\ B$ *then* $STOP$ *else if* $d \ `\ B = \{c \ a\}$ *then ?rhs else ?rhs* $\sqcap \ STOP$)›)
proof (*split if-split*, *intro conjI impI*)
show ‹$c \ a \notin d \ `\ B \Longrightarrow$ *?lhs* = $STOP$›
by (*auto simp add*: *GlobalNdet-is-STOP-iff image-subset-iff image-iff*)
next

show ‹¬ *c a* ∉ *d* ‘ *B* ⟹ *?lhs* = (*if d* ‘ *B* = {*c a*} *then ?rhs else ?rhs* ⊓ *STOP*)›
by (*auto simp add*: *image-subset-iff Ndet-is-STOP-iff write-is-write0*)
qed
qed

lemma *ndet-write-Sync-write-subset* :
‹*c* ‘ *A* ⊆ *S* ⟹ *d b* ∈ *S* ⟹
(*c*!!*a*∈*A* → *P a*) ⟦*S*⟧ (*d*!*b* → *Q*) =
(*if d b* ∉ *c* ‘ *A then STOP else if c* ‘ *A* = {*d b*} *then d*!*b* → (*P* (*inv-into A c* (*d b*)) ⟦*S*⟧ *Q*)
else (*d*!*b* → (*P* (*inv-into A c* (*d b*)) ⟦*S*⟧ *Q*)) ⊓ *STOP*)›
by (*subst* (*1 2 3*) *Sync-commute*) (*simp add*: *write-Sync-ndet-write-subset*)

lemma *write-Sync-ndet-write-indep* :
‹*c a* ∉ *S* ⟹ *d* ‘ *B* ∩ *S* = {} ⟹
c!*a* → *P* ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* =
(*if B* = {} *then c*!*a* → (*P* ⟦*S*⟧ *STOP*)
else ⊓*b*∈*d* ‘ *B*. (*c*!*a* → (*P* ⟦*S*⟧ *b* → *Q* (*inv-into B d b*))) □
(*b* → (*c*!*a* → *P* ⟦*S*⟧ *Q* (*inv-into B d b*))))›
by (*subst ndet-write-Sync-ndet-write-indep*[**where** *A* = ‹{*a*}›, *simplified*])
(*auto simp add*: *write-is-write0 intro*: *mono-GlobalNdet-eq*)

lemma *ndet-write-Sync-write-indep* :
‹*c* ‘ *A* ∩ *S* = {} ⟹ *d b* ∉ *S* ⟹
c!!*a*∈*A* → *P a* ⟦*S*⟧ *d*!*b* → *Q* =
(*if A* = {} *then d*!*b* → (*STOP* ⟦*S*⟧ *Q*)
else ⊓*a*∈*c* ‘ *A*. (*a* → (*P* (*inv-into A c a*) ⟦*S*⟧ *d*!*b* → *Q*)) □
(*d*!*b* → (*a* → *P* (*inv-into A c a*) ⟦*S*⟧ *Q*)))›
by (*subst* (*1 2 3 4*) *Sync-commute*, *subst Det-commute*)
(*simp add*: *write-Sync-ndet-write-indep*)

lemma *write-Sync-ndet-write-left* :
‹*c a* ∉ *S* ⟹ *d* ‘ *B* ⊆ *S* ⟹ *c*!*a* → *P* ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* = *c*!*a* → (*P* ⟦*S*⟧ *d*!!*b*∈*B* → *Q b*)›
by (*subst ndet-write-Sync-ndet-write-left*[**where** *A* = ‹{*a*}›, *simplified*]) *simp-all*

lemma *ndet-write-Sync-write-left* :
‹*c* ‘ *A* ∩ *S* = {} ⟹ *d b* ∈ *S* ⟹ *c*!!*a*∈*A* → *P a* ⟦*S*⟧ *d*!*b* → *Q* = *c*!!*a*∈*A* → (*P a* ⟦*S*⟧ *d*!*b* → *Q*)›
by (*subst ndet-write-Sync-ndet-write-left*[**where** *B* = ‹{*b*}›, *simplified*]) *simp-all*

lemma *write-Sync-ndet-write-right* :
‹*c a* ∈ *S* ⟹ *d* ‘ *B* ∩ *S* = {} ⟹ *c*!*a* → *P* ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* = *d*!!*b*∈*B* → (*c*!*a* → *P* ⟦*S*⟧ *Q b*)›
by (*subst ndet-write-Sync-ndet-write-right*[**where** *A* = ‹{*a*}›, *simplified*]) *simp-all*

lemma *ndet-write-Sync-write-right* :
‹c ‘ $A \subseteq S \Longrightarrow d\ b \notin S \Longrightarrow c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!b \rightarrow Q = d!b \rightarrow (c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ Q)$›
by (*subst ndet-write-Sync-ndet-write-right*[**where** $B = \langle\{b\}\rangle$, *simplified*]) *simp-all*

write **and** *write* **lemma** *write-Sync-write* :
‹$c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q =$
($\mathit{if}\ d\ b \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ d!b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q)) \Box$
($\mathit{if}\ c\ a \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ c!a \rightarrow (P\ [\![S]\!]\ d!b \rightarrow Q)) \Box$
($\mathit{if}\ c\ a = d\ b \wedge d\ b \in S\ \mathit{then}\ c!a \rightarrow (P\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP}$)›
by (*subst read-Sync-read*[**where** $A = \langle\{a\}\rangle$ **and** $B = \langle\{b\}\rangle$, *simplified*])
(*simp add*: *write-def insert-Diff-if Det-commute Int-insert-right*)

lemma *write-Inter-write* :
‹$c!a \rightarrow P\ |||\ d!b \rightarrow Q = (c!a \rightarrow (P\ |||\ d!b \rightarrow Q)) \Box (d!b \rightarrow (c!a \rightarrow P\ |||\ Q))$›
by (*simp add*: *write-Sync-write Det-commute*)

lemma *write-Par-write* :
‹$c!a \rightarrow P\ ||\ d!b \rightarrow Q = (\mathit{if}\ c\ a = d\ b\ \mathit{then}\ c!a \rightarrow (P\ ||\ Q)\ \mathit{else}\ \mathit{STOP})$›
by (*simp add*: *write-Sync-write*)

lemma *write-Sync-write-subset* :
‹$c\ a \in S \Longrightarrow d\ b \in S \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = (\mathit{if}\ c\ a = d\ b\ \mathit{then}\ c!a \rightarrow (P\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})$›
by (*simp add*: *write-Sync-write*)

lemma *write-Sync-write-indep* :
‹$c\ a \notin S \Longrightarrow d\ b \notin S \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = (c!a \rightarrow (P\ [\![S]\!]\ d!b \rightarrow Q)) \Box (d!b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q))$›
by (*simp add*: *Det-commute write-Sync-write*)

lemma *write-Sync-write-left* :
‹$c\ a \notin S \Longrightarrow d\ b \in S \Longrightarrow c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = c!a \rightarrow (P\ [\![S]\!]\ d!b \rightarrow Q)$›
by (*auto simp add*: *write-Sync-write*)

lemma *write-Sync-write-right* :
‹$c\ a \in S \Longrightarrow d\ b \notin S \Longrightarrow c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = d!b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q)$›
by (*auto simp add*: *write-Sync-write*)

read **and** $(\rightarrow)$**.** **lemma** *write0-Sync-read* :
‹$a \rightarrow P\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b =$
($\mathit{if}\ a \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ a \rightarrow (P\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b)) \Box$
($\Box b{\in}(d\ ‘\ B - S) \rightarrow (a \rightarrow P\ [\![S]\!]\ Q\ (\mathit{inv\text{-}into}\ B\ d\ b))) \Box$
($\mathit{if}\ a \in d\ ‘\ B \cap S\ \mathit{then}\ a \rightarrow (P\ [\![S]\!]\ Q\ (\mathit{inv\text{-}into}\ B\ d\ a))\ \mathit{else}\ \mathit{STOP}$)›
by (*simp add*: *write-Sync-read*[**where** $c = id$, *unfolded write-is-write0*, *simpli-*

fied])

lemma *read-Sync-write0* :
‹*c*?*a*∈*A* → *P a* ⟦*S*⟧ *b* → *Q* =
(*if b* ∈ *S then STOP else b* → (*c*?*a*∈*A* → *P a* ⟦*S*⟧ *Q*)) □
(□*a*∈(*c* ‘ *A* − *S*) → (*P* (*inv-into A c a*) ⟦*S*⟧ *b* → *Q*)) □
(*if b* ∈ *c* ‘ *A* ∩ *S then b* → (*P* (*inv-into A c b*) ⟦*S*⟧ *Q*) *else STOP*)›
by (*simp add*: *read-Sync-write*[**where** *d* = *id*, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-read-subset* :
‹*a* ∈ *S* ⟹ *d* ‘ *B* ⊆ *S* ⟹
a → *P* ⟦*S*⟧ *d*?*b*∈*B* → *Q b* =
(*if a* ∈ *d* ‘ *B then a* → (*P* ⟦*S*⟧ *Q* (*inv-into B d a*)) *else STOP*)›
by (*simp add*: *write-Sync-read-subset*[**where** *c* = *id*, *unfolded write-is-write0*, *simplified*])

lemma *read-Sync-write0-subset* :
‹*c* ‘ *A* ⊆ *S* ⟹ *b* ∈ *S* ⟹
c?*a*∈*A* → *P a* ⟦*S*⟧ *b* → *Q* =
(*if b* ∈ *c* ‘ *A then b* → (*P* (*inv-into A c b*) ⟦*S*⟧ *Q*) *else STOP*)›
by (*simp add*: *read-Sync-write-subset*[**where** *d* = ‹λ*x*. *x*›, *unfolded write-is-write0*])

lemma *write0-Sync-read-subset-same-chan*:
‹*a* ∈ *S* ⟹ *B* ⊆ *S* ⟹
a → *P* ⟦*S*⟧ *id*?*b*∈*B* → *Q b* = (*if a* ∈ *B then a* → (*P* ⟦*S*⟧ *Q a*) *else STOP*)›
by (*simp add*: *write-Sync-read-subset-same-chan*
[**where** *c* = *id*, *unfolded write-is-write0*, *simplified*])

lemma *read-Sync-write0-subset-same-chan*:
‹*A* ⊆ *S* ⟹ *b* ∈ *S* ⟹
id?*a*∈*A* → *P a* ⟦*S*⟧ *b* → *Q* = (*if b* ∈ *A then b* → (*P b* ⟦*S*⟧ *Q*) *else STOP*)›
by (*simp add*: *read-Sync-write-subset-same-chan*
[**where** *c* = *id*, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-read-indep* :
‹*a* ∉ *S* ⟹ *d* ‘ *B* ∩ *S* = {} ⟹
a → *P* ⟦*S*⟧ *d*?*b*∈*B* → *Q b* =
(*a* → (*P* ⟦*S*⟧ *d*?*b*∈*B* → *Q b*)) □ (*d*?*b*∈*B* → (*a* → *P* ⟦*S*⟧ *Q b*))›
by (*simp add*: *write-Sync-read-indep*[**where** *c* = *id*, *unfolded write-is-write0*, *simplified*])

lemma *read-Sync-write0-indep* :
‹*c* ‘ *A* ∩ *S* = {} ⟹ *b* ∉ *S* ⟹
c?*a*∈*A* → *P a* ⟦*S*⟧ *b* → *Q* =
(*b* → (*c*?*a*∈*A* → *P a* ⟦*S*⟧ *Q*)) □ (*c*?*a*∈*A* → (*P a* ⟦*S*⟧ *b* → *Q*))›
by (*simp add*: *read-Sync-write-indep*[**where** *d* = *id*, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-read-left* :
‹$a \notin S \Longrightarrow d$ ‘ $B \subseteq S \Longrightarrow a \rightarrow P \llbracket S \rrbracket\ d?b{\in}B \rightarrow Q\ b = a \rightarrow (P \llbracket S \rrbracket\ d?b{\in}B \rightarrow Q\ b)$›
by (*simp add*: *write-Sync-read-left*[**where** $c = id$, *unfolded write-is-write0*, *simplified*])

lemma *read-Sync-write0-left* :
‹c ‘ $A \cap S = \{\} \Longrightarrow b \in S \Longrightarrow c?a{\in}A \rightarrow P\ a \llbracket S \rrbracket\ b \rightarrow Q = c?a{\in}A \rightarrow (P\ a \llbracket S \rrbracket\ b \rightarrow Q)$›
by (*simp add*: *read-Sync-write-left*[**where** $d = id$, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-read-right* :
‹$a \in S \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow a \rightarrow P \llbracket S \rrbracket\ d?b{\in}B \rightarrow Q\ b = d?b{\in}B \rightarrow (a \rightarrow P \llbracket S \rrbracket\ Q\ b)$›
by (*simp add*: *write-Sync-read-right*[**where** $c = id$, *unfolded write-is-write0*, *simplified*])

lemma *read-Sync-write0-right* :
‹c ‘ $A \subseteq S \Longrightarrow b \notin S \Longrightarrow c?a{\in}A \rightarrow P\ a \llbracket S \rrbracket\ b \rightarrow Q = b \rightarrow (c?a{\in}A \rightarrow P\ a \llbracket S \rrbracket\ Q)$›
by (*simp add*: *read-Sync-write-right*[**where** $d = id$, *unfolded write-is-write0*, *simplified*])

ndet-write **and** $(\rightarrow)$ **lemma** *write0-Sync-ndet-write* :
‹$a \rightarrow P \llbracket S \rrbracket\ d!!b{\in}B \rightarrow Q\ b =$
(*if* $B = \{\}$ *then* $a \rightarrow P \llbracket S \rrbracket$ *STOP*
else $\sqcap b{\in}d$ ‘ B. (*if* $b \in S$ *then STOP else* $b \rightarrow (a \rightarrow P \llbracket S \rrbracket\ Q$ (*inv-into* $B\ d\ b$))) $\Box$
(*if* $a \in S$ *then STOP else* $a \rightarrow (P \llbracket S \rrbracket\ b \rightarrow Q$ (*inv-into* $B\ d\ b$))) $\Box$
(*if* $b = a \wedge a \in S$ *then* $a \rightarrow (P \llbracket S \rrbracket\ Q$ (*inv-into* $B\ d\ a$)) *else STOP*)›
by (*simp add*: *write-Sync-ndet-write*[**where** $c = \langle \lambda x.\ x \rangle$, *unfolded write-is-write0*, *simplified*])

lemma *ndet-write-Sync-write0* :
‹$c!!a{\in}A \rightarrow P\ a \llbracket S \rrbracket\ b \rightarrow Q =$
(*if* $A = \{\}$ *then STOP* $\llbracket S \rrbracket\ b \rightarrow Q$
else $\sqcap a{\in}c$ ‘ A. (*if* $a \in S$ *then STOP else* $a \rightarrow (P$ (*inv-into* $A\ c\ a$) $\llbracket S \rrbracket\ b \rightarrow Q$)) $\Box$
(*if* $b \in S$ *then STOP else* $b \rightarrow (a \rightarrow P$ (*inv-into* $A\ c\ a$) $\llbracket S \rrbracket\ Q$)) $\Box$
(*if* $a = b \wedge b \in S$ *then* $b \rightarrow (P$ (*inv-into* $A\ c\ a$) $\llbracket S \rrbracket\ Q$) *else STOP*)›
by (*simp add*: *ndet-write-Sync-write*[**where** $d = \langle \lambda x.\ x \rangle$, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-ndet-write-subset* :
‹$a \in S \Longrightarrow d$ ‘ $B \subseteq S \Longrightarrow$
$a \rightarrow P \llbracket S \rrbracket\ d!!b{\in}B \rightarrow Q\ b =$
(*if* $a \notin d$ ‘ B *then STOP else if* d ‘ $B = \{a\}$ *then* $a \rightarrow (P \llbracket S \rrbracket\ Q$ (*inv-into* $B\ d\ a$))

else $(a \rightarrow (P \llbracket S \rrbracket Q\ (\mathit{inv\text{-}into}\ B\ d\ a))) \sqcap \mathit{STOP})\rangle$
by (*simp add*: *write-Sync-ndet-write-subset*[**where** $c = \mathit{id}$, *unfolded write-is-write0*, *simplified*])

lemma *ndet-write-Sync-write0-subset* :
$\langle c\ `\ A \subseteq S \Longrightarrow b \in S \Longrightarrow$
$c!!a{\in}A \rightarrow P\ a \llbracket S \rrbracket\ b \rightarrow Q =$
$($ *if* $b \notin c\ `\ A$ *then STOP else if* $c\ `\ A = \{b\}$ *then* $b \rightarrow (P\ (\mathit{inv\text{-}into}\ A\ c\ b) \llbracket S \rrbracket Q)$
else $(b \rightarrow (P\ (\mathit{inv\text{-}into}\ A\ c\ b) \llbracket S \rrbracket\ Q)) \sqcap \mathit{STOP})\rangle$
by (*simp add*: *ndet-write-Sync-write-subset*[**where** $d = \mathit{id}$, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-ndet-write-indep* :
$\langle a \notin S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
$a \rightarrow P \llbracket S \rrbracket\ d!!b{\in}B \rightarrow Q\ b =$
$($ *if* $B = \{\}$ *then* $a \rightarrow (P \llbracket S \rrbracket \mathit{STOP})$
else $\sqcap b{\in}d\ `\ B.\ (a \rightarrow (P \llbracket S \rrbracket\ b \rightarrow Q\ (\mathit{inv\text{-}into}\ B\ d\ b))) \Box$
$(b \rightarrow (a \rightarrow P \llbracket S \rrbracket\ Q\ (\mathit{inv\text{-}into}\ B\ d\ b))))\rangle$
by (*simp add*: *write-Sync-ndet-write-indep*[**where** $c = \mathit{id}$, *unfolded write-is-write0*, *simplified*])

lemma *ndet-write-Sync-write0-indep* :
$\langle c\ `\ A \cap S = \{\} \Longrightarrow b \notin S \Longrightarrow$
$c!!a{\in}A \rightarrow P\ a \llbracket S \rrbracket\ b \rightarrow Q =$
$($ *if* $A = \{\}$ *then* $b \rightarrow (\mathit{STOP} \llbracket S \rrbracket\ Q)$
else $\sqcap a{\in}c\ `\ A.\ (a \rightarrow (P\ (\mathit{inv\text{-}into}\ A\ c\ a) \llbracket S \rrbracket\ b \rightarrow Q)) \Box$
$(b \rightarrow (a \rightarrow P\ (\mathit{inv\text{-}into}\ A\ c\ a) \llbracket S \rrbracket\ Q)))\rangle$
by (*simp add*: *ndet-write-Sync-write-indep*[**where** $d = \mathit{id}$, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-ndet-write-left* :
$\langle a \notin S \Longrightarrow d\ `\ B \subseteq S \Longrightarrow a \rightarrow P \llbracket S \rrbracket\ d!!b{\in}B \rightarrow Q\ b = a \rightarrow (P \llbracket S \rrbracket\ d!!b{\in}B \rightarrow Q\ b)\rangle$
by (*simp add*: *write-Sync-ndet-write-left*[**where** $c = \mathit{id}$, *unfolded write-is-write0*, *simplified*])

lemma *ndet-write-Sync-write0-left* :
$\langle c\ `\ A \cap S = \{\} \Longrightarrow b \in S \Longrightarrow c!!a{\in}A \rightarrow P\ a \llbracket S \rrbracket\ b \rightarrow Q = c!!a{\in}A \rightarrow (P\ a \llbracket S \rrbracket\ b \rightarrow Q)\rangle$
by (*simp add*: *ndet-write-Sync-write-left*[**where** $d = \mathit{id}$, *unfolded write-is-write0*, *simplified*])

lemma *write-Sync-ndet-write0-right* :
$\langle a \in S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow a \rightarrow P \llbracket S \rrbracket\ d!!b{\in}B \rightarrow Q\ b = d!!b{\in}B \rightarrow (a \rightarrow P \llbracket S \rrbracket\ Q\ b)\rangle$
by (*simp add*: *write-Sync-ndet-write-right*[**where** $c = \mathit{id}$, *unfolded write-is-write0*, *simplified*])

lemma *ndet-write-Sync-write0-right* :
‹c ‘ $A \subseteq S \Longrightarrow b \notin S \Longrightarrow c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ b \rightarrow Q = b \rightarrow (c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ Q)$›
by (*simp add*: *ndet-write-Sync-write-right*[**where** $d = id$, *unfolded write-is-write0*, *simplified*])

$(\rightarrow)$ **and** $(\rightarrow)$ **lemma** *write0-Sync-write0* :
‹$a \rightarrow P\ [\![S]\!]\ b \rightarrow Q =$
$(if\ b \in S\ then\ STOP\ else\ b \rightarrow (a \rightarrow P\ [\![S]\!]\ Q))\ \Box$
$(if\ a \in S\ then\ STOP\ else\ a \rightarrow (P\ [\![S]\!]\ b \rightarrow Q))\ \Box$
$(if\ a = b \wedge b \in S\ then\ a \rightarrow (P\ [\![S]\!]\ Q)\ else\ STOP)$›
by (*simp add*: *write-Sync-write*[**where** $c = id$ **and** $d = id$, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-write0-bis* :
‹$(a \rightarrow P)\ [\![S]\!]\ (b \rightarrow Q) =$
$($ *if* $a \in S$
then *if* $b \in S$
then *if* $a = b$
then $a \rightarrow (P\ [\![S]\!]\ Q)$
else $STOP$
else $(b \rightarrow ((a \rightarrow P)\ [\![S]\!]\ Q))$
else *if* $b \in S$
then $a \rightarrow (P\ [\![S]\!]\ (b \rightarrow Q))$
else $(a \rightarrow (P\ [\![S]\!]\ (b \rightarrow Q)))\ \Box\ (b \rightarrow ((a \rightarrow P)\ [\![S]\!]\ Q)))$›
by (*cases* ‹$a \in S$›; *cases* ‹$b \in S$›) (*auto simp add*: *write0-Sync-write0 Det-commute*)

lemma *write0-Inter-write0* :
‹$a \rightarrow P\ |||\ b \rightarrow Q = (a \rightarrow (P\ |||\ b \rightarrow Q))\ \Box\ (b \rightarrow (a \rightarrow P\ |||\ Q))$›
by (*simp add*: *write0-Sync-write0 Det-commute*)

lemma *write0-Par-write0* :
‹$a \rightarrow P\ ||\ b \rightarrow Q = (if\ a = b\ then\ a \rightarrow (P\ ||\ Q)\ else\ STOP)$›
by (*simp add*: *write0-Sync-write0*)

lemma *write0-Sync-write0-subset* :
‹$a \in S \Longrightarrow b \in S \Longrightarrow a \rightarrow P\ [\![S]\!]\ b \rightarrow Q = (if\ a = b\ then\ a \rightarrow (P\ [\![S]\!]\ Q)\ else\ STOP)$›
by (*simp add*: *write-Sync-write-subset*[**where** $c = id$ **and** $d = id$, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-write0-indep* :
‹$a \notin S \Longrightarrow b \notin S \Longrightarrow a \rightarrow P\ [\![S]\!]\ b \rightarrow Q = (a \rightarrow (P\ [\![S]\!]\ b \rightarrow Q))\ \Box\ (b \rightarrow (a \rightarrow P\ [\![S]\!]\ Q))$›
by (*simp add*: *write-Sync-write-indep*[**where** $c = id$ **and** $d = id$, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-write0-left* :
‹$a \notin S \Longrightarrow b \in S \Longrightarrow a \rightarrow P \llbracket S \rrbracket\ b \rightarrow Q = a \rightarrow (P \llbracket S \rrbracket\ b \rightarrow Q)$›
by (*simp add*: *write-Sync-write-left*[**where** $c = id$ **and** $d = id$, *unfolded write-is-write0*, *simplified*])

lemma *write0-Sync-write0-right* :
‹$a \in S \Longrightarrow b \notin S \Longrightarrow a \rightarrow P \llbracket S \rrbracket\ b \rightarrow Q = b \rightarrow (a \rightarrow P \llbracket S \rrbracket\ Q)$›
by (*simp add*: *write-Sync-write-right*[**where** $c = id$ **and** $d = id$, *unfolded write-is-write0*, *simplified*])

write **and** $(\rightarrow)$ **lemma** *write0-Sync-write* :
‹$a \rightarrow P \llbracket S \rrbracket\ d!b \rightarrow Q =$
($if\ d\ b \in S\ then\ STOP\ else\ d!b \rightarrow (a \rightarrow P \llbracket S \rrbracket\ Q)$) $\Box$
($if\ a \in S\ then\ STOP\ else\ a \rightarrow (P \llbracket S \rrbracket\ d!b \rightarrow Q)$) $\Box$
($if\ a = d\ b \wedge d\ b \in S\ then\ a \rightarrow (P \llbracket S \rrbracket\ Q)\ else\ STOP$)›
by (*simp add*: *write0-Sync-write0 write-is-write0*)

lemma *write-Sync-write0* :
‹$c!a \rightarrow P \llbracket S \rrbracket\ b \rightarrow Q =$
($if\ b \in S\ then\ STOP\ else\ b \rightarrow (c!a \rightarrow P \llbracket S \rrbracket\ Q)$) $\Box$
($if\ c\ a \in S\ then\ STOP\ else\ c!a \rightarrow (P \llbracket S \rrbracket\ b \rightarrow Q)$) $\Box$
($if\ c\ a = b \wedge b \in S\ then\ c!a \rightarrow (P \llbracket S \rrbracket\ Q)\ else\ STOP$)›
by (*simp add*: *write0-Sync-write0 write-is-write0*)

lemma *write0-Sync-write-subset* :
‹$a \in S \Longrightarrow d\ b \in S \Longrightarrow$
$a \rightarrow P \llbracket S \rrbracket\ d!b \rightarrow Q = (if\ a = d\ b\ then\ a \rightarrow (P \llbracket S \rrbracket\ Q)\ else\ STOP)$›
by (*simp add*: *write0-Sync-write*)

lemma *write-Sync-write0-subset* :
‹$c\ a \in S \Longrightarrow b \in S \Longrightarrow$
$c!a \rightarrow P \llbracket S \rrbracket\ b \rightarrow Q = (if\ c\ a = b\ then\ c!a \rightarrow (P \llbracket S \rrbracket\ Q)\ else\ STOP)$›
by (*simp add*: *write-Sync-write0*)

lemma *write0-Sync-write-indep* :
‹$a \notin S \Longrightarrow d\ b \notin S \Longrightarrow$
$a \rightarrow P \llbracket S \rrbracket\ d!b \rightarrow Q = (a \rightarrow (P \llbracket S \rrbracket\ d!b \rightarrow Q)) \Box (d!b \rightarrow (a \rightarrow P \llbracket S \rrbracket\ Q))$›
by (*simp add*: *Det-commute write0-Sync-write*)

lemma *write-Sync-write0-indep* :
‹$c\ a \notin S \Longrightarrow b \notin S \Longrightarrow$
$c!a \rightarrow P \llbracket S \rrbracket\ b \rightarrow Q = (c!a \rightarrow (P \llbracket S \rrbracket\ b \rightarrow Q)) \Box (b \rightarrow (c!a \rightarrow P \llbracket S \rrbracket\ Q))$›
by (*simp add*: *Det-commute write-Sync-write0*)

lemma *write0-Sync-write-left* :
‹$a \notin S \Longrightarrow d\ b \in S \Longrightarrow a \rightarrow P \llbracket S \rrbracket\ d!b \rightarrow Q = a \rightarrow (P \llbracket S \rrbracket\ d!b \rightarrow Q)$›

by (*simp add*: *write0-Sync-write0-left write-is-write0*)

lemma *write-Sync-write0-left* :
‹$c\ a \notin S \Longrightarrow b \in S \Longrightarrow c!a \rightarrow P \ [\![S]\!]\ b \rightarrow Q = c!a \rightarrow (P \ [\![S]\!]\ b \rightarrow Q)$›
by (*simp add*: *write0-Sync-write0-left write-is-write0*)

lemma *write0-Sync-write-right* :
‹$a \in S \Longrightarrow d\ b \notin S \Longrightarrow a \rightarrow P \ [\![S]\!]\ d!b \rightarrow Q = d!b \rightarrow (a \rightarrow P \ [\![S]\!]\ Q)$›
by (*simp add*: *write0-Sync-write0-right write-is-write0*)

lemma *write-Sync-write0-right* :
‹$c\ a \in S \Longrightarrow b \notin S \Longrightarrow c!a \rightarrow P \ [\![S]\!]\ b \rightarrow Q = b \rightarrow (c!a \rightarrow P \ [\![S]\!]\ Q)$›
by (*simp add*: *Sync-commute write0-Sync-write-left*)

12.5.4 *Sync*, *SKIP* and *STOP*

SKIP

Without injectivity, the result is a trivial corollary of *read ?c ?A ?P* ≡ *Mprefix* (*?c ‘ ?A*) (*?P* ∘ *inv-into ?A ?c*) and *Mprefix ?A ?P* ⟦*?S*⟧ *SKIP ?res* = □*a*∈(*?A − ?S*) → (*?P a* ⟦*?S*⟧ *SKIP ?res*).

lemma *read-Sync-SKIP* :
‹$c?a{\in}A \rightarrow P\ a \ [\![S]\!]\ SKIP\ r = c?a{\in}(A - c -` S) \rightarrow (P\ a \ [\![S]\!]\ SKIP\ r)$› **if** ‹*inj-on c A*›
proof −
have ‹*c ‘ (A − c −‘ S) = c ‘ A − S*› **by** *blast*
show ‹$c?a{\in}A \rightarrow P\ a \ [\![S]\!]\ SKIP\ r = c?a{\in}(A - c -` S) \rightarrow (P\ a \ [\![S]\!]\ SKIP\ r)$›
by (*auto simp add*: *read-def Mprefix-Sync-SKIP* ‹*?this*› *inj-on-diff* ‹*inj-on c A*› *intro*: *mono-Mprefix-eq*)
qed

lemma *SKIP-Sync-read* :
‹*inj-on d B* ⟹ $SKIP\ r \ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b = d?b{\in}(B - d -` S) \rightarrow (SKIP\ r \ [\![S]\!]\ Q\ b)$›
by (*subst* (*1 2*) *Sync-commute*) (*simp add*: *read-Sync-SKIP*)

corollary *write-Sync-SKIP* :
‹$c!a \rightarrow P \ [\![S]\!]\ SKIP\ r = (if\ c\ a \in S\ then\ STOP\ else\ c!a \rightarrow (P \ [\![S]\!]\ SKIP\ r))$›
and *SKIP-Sync-write* :
‹$SKIP\ r \ [\![S]\!]\ d!b \rightarrow Q = (if\ d\ b \in S\ then\ STOP\ else\ d!b \rightarrow (SKIP\ r \ [\![S]\!]\ Q))$›
by (*simp-all add*: *write-def Mprefix-Sync-SKIP SKIP-Sync-Mprefix Diff-triv*)

corollary *write0-Sync-SKIP* :
‹$a \rightarrow P \ [\![S]\!]\ SKIP\ r = (if\ a \in S\ then\ STOP\ else\ a \rightarrow (P \ [\![S]\!]\ SKIP\ r))$›
and *SKIP-Sync-write0* :
‹$SKIP\ r \ [\![S]\!]\ b \rightarrow Q = (if\ b \in S\ then\ STOP\ else\ b \rightarrow (SKIP\ r \ [\![S]\!]\ Q))$›
by (*simp-all add*: *write0-def Mprefix-Sync-SKIP SKIP-Sync-Mprefix Diff-triv*)

lemma *ndet-write-Sync-SKIP* :
‹*c*!!*a*∈*A* → *P a* ⟦*S*⟧ *SKIP r* =
(*if c ‘ A* ∩ *S* = {} *then c*!!*a*∈*A* → (*P a* ⟦*S*⟧ *SKIP r*)
else (*c*!!*a*∈(*A* − *c* −‘ *S*) → (*P a* ⟦*S*⟧ *SKIP r*)) ⊓ *STOP*)›
(**is** ‹*?lhs* = (*if c ‘ A* ∩ *S* = {} *then ?rhs1 else ?rhs2* ⊓ *STOP*)›) **if** ‹*inj-on c A*›
proof (*split if-split*, *intro conjI impI*)
assume ‹*c ‘ A* ∩ *S* = {}›
hence ‹*A* − *c* −‘ *S* = *A*› **by** *blast*
from ‹*c ‘ A* ∩ *S* = {}› **show** ‹*?lhs* = *?rhs1*›
by (*auto simp add*: ‹*?this*› *ndet-write-is-GlobalNdet-write0 disjoint-iff*
Sync-distrib-GlobalNdet-right write0-Sync-SKIP
intro!: *mono-GlobalNdet-eq split*: *if-split-asm*)
next
show ‹*?lhs* = *?rhs2* ⊓ *STOP*› **if** ‹*c ‘ A* ∩ *S* ≠ {}›
proof (*cases* ‹*c ‘ A* − *S* = {}›)
assume ‹*c ‘ A* − *S* = {}›
hence ‹*A* − *c* −‘ *S* = {}› **by** *blast*
from ‹*c ‘ A* − *S* = {}› **show** ‹*?lhs* = *?rhs2* ⊓ *STOP*›
by (*auto simp add*: *ndet-write-is-GlobalNdet-write0 GlobalNdet-is-STOP-iff*
‹*?this*› *Sync-distrib-GlobalNdet-right write0-Sync-SKIP*)
next
have ‹*c ‘* (*A* − *c* −‘ *S*) = *c ‘ A* − *S*› **by** *blast*
show ‹*?lhs* = *?rhs2* ⊓ *STOP*› **if** ‹*c ‘ A* − *S* ≠ {}›
by (*subst Ndet-commute*, *unfold ndet-write-is-GlobalNdet-write0 Sync-distrib-GlobalNdet-right*)
(*auto simp add*: *GlobalNdet-is-STOP-iff write0-Sync-SKIP*
‹*?this*› ‹*inj-on c A*› *inj-on-diff*
simp flip: *GlobalNdet-factorization-union*
[*OF* ‹*c ‘ A* ∩ *S* ≠ {}› ‹*c ‘ A* − *S* ≠ {}›, *unfolded Int-Diff-Un*]
intro!: *arg-cong2*[**where** *f* = ‹(⊓)›] *mono-GlobalNdet-eq*)
qed
qed

corollary *SKIP-Sync-ndet-write* :
‹*inj-on d B* ⟹
SKIP r ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* =
(*if d ‘ B* ∩ *S* = {} *then d*!!*b*∈*B* → (*SKIP r* ⟦*S*⟧ *Q b*)
else (*d*!!*b*∈(*B* − *d* −‘ *S*) → (*SKIP r* ⟦*S*⟧ *Q b*)) ⊓ *STOP*)›
by (*subst* (*1 2 3*) *Sync-commute*) (*simp add*: *ndet-write-Sync-SKIP*)

STOP

Without injectivity, the result is a trivial corollary of *read ?c ?A ?P* ≡ *Mprefix* (*?c ‘ ?A*) (*?P* ∘ *inv-into ?A ?c*) and *Mprefix ?A ?P* ⟦*?S*⟧ *SKIP ?res* = □*a*∈(*?A* − *?S*) → (*?P a* ⟦*?S*⟧ *SKIP ?res*).

lemma *read-Sync-STOP* :
‹*c*?*a*∈*A* → *P a* ⟦*S*⟧ *STOP* = *c*?*a*∈(*A* − *c* −‘ *S*) → (*P a* ⟦*S*⟧ *STOP*)› **if** ‹*inj-on c A*›

proof −
have ‹c ‘ $(A - c -‘ S) = c$ ‘ $A - S$› **by** *blast*
show ‹$c?a{\in}A \rightarrow P\ a\ [\![S]\!]\ STOP = c?a{\in}(A - c -‘ S) \rightarrow (P\ a\ [\![S]\!]\ STOP)$›
by (*auto simp add*: ‹*?this*› *read-def Mprefix-Sync-STOP inj-on-diff* ‹*inj-on c A*›
intro: *mono-Mprefix-eq*)
qed

lemma *STOP-Sync-read* :
‹*inj-on d B* $\Longrightarrow$ $STOP\ [\![S]\!]\ d?b{\in}B \rightarrow Q\ b = d?b{\in}(B - d -‘ S) \rightarrow (STOP\ [\![S]\!]\ Q\ b)$›
by (*subst* (*1 2*) *Sync-commute*) (*simp add*: *read-Sync-STOP*)

corollary *write-Sync-STOP* :
‹$c!a \rightarrow P\ [\![S]\!]\ STOP = (if\ c\ a \in S\ then\ STOP\ else\ c!a \rightarrow (P\ [\![S]\!]\ STOP))$›
and *STOP-Sync-write* :
‹$STOP\ [\![S]\!]\ d!b \rightarrow Q = (if\ d\ b \in S\ then\ STOP\ else\ d!b \rightarrow (STOP\ [\![S]\!]\ Q))$›
by (*simp-all add*: *write-def Mprefix-Sync-STOP STOP-Sync-Mprefix Diff-triv*)

corollary *write0-Sync-STOP* :
‹$a \rightarrow P\ [\![S]\!]\ STOP = (if\ a \in S\ then\ STOP\ else\ a \rightarrow (P\ [\![S]\!]\ STOP))$›
and *STOP-Sync-write0* :
‹$STOP\ [\![S]\!]\ b \rightarrow Q = (if\ b \in S\ then\ STOP\ else\ b \rightarrow (STOP\ [\![S]\!]\ Q))$›
by (*simp-all add*: *write0-def Mprefix-Sync-STOP STOP-Sync-Mprefix Diff-triv*)

lemma *ndet-write-Sync-STOP* :
‹$c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ STOP =$
$(\ if\ c$ ‘ $A \cap S = \{\}\ then\ c!!a{\in}A \rightarrow (P\ a\ [\![S]\!]\ STOP)$
$else\ (c!!a{\in}(A - c -‘ S) \rightarrow (P\ a\ [\![S]\!]\ STOP)) \sqcap STOP)$›
(**is** ‹*?lhs* = (*if c* ‘ $A \cap S = \{\}$ *then ?rhs1 else ?rhs2* $\sqcap$ *STOP*)›) **if** ‹*inj-on c A*›
proof (*split if-split*, *intro conjI impI*)
assume ‹c ‘ $A \cap S = \{\}$›
hence ‹$A - c -‘ S = A$› **by** *blast*
from ‹c ‘ $A \cap S = \{\}$› **show** ‹*?lhs* = *?rhs1*›
by (*auto simp add*: ‹*?this*› *ndet-write-is-GlobalNdet-write0 disjoint-iff*
Sync-distrib-GlobalNdet-right write0-Sync-STOP
intro!: *mono-GlobalNdet-eq split*: *if-split-asm*)
next
show ‹*?lhs* = *?rhs2* $\sqcap$ *STOP*› **if** ‹c ‘ $A \cap S \neq \{\}$›
proof (*cases* ‹c ‘ $A - S = \{\}$›)
assume ‹c ‘ $A - S = \{\}$›
hence ‹$A - c -‘ S = \{\}$› **by** *blast*
from ‹c ‘ $A - S = \{\}$› **show** ‹*?lhs* = *?rhs2* $\sqcap$ *STOP*›
by (*auto simp add*: *ndet-write-is-GlobalNdet-write0 GlobalNdet-is-STOP-iff*
‹*?this*› *Sync-distrib-GlobalNdet-right write0-Sync-STOP*)
next
have ‹c ‘ $(A - c -‘ S) = c$ ‘ $A - S$› **by** *blast*

show ‹*?lhs = ?rhs2 ⊓ STOP*› **if** ‹*c ‘ A − S ≠ {}*›
by (*subst Ndet-commute*, *unfold ndet-write-is-GlobalNdet-write0 Sync-distrib-GlobalNdet-right*)
(*auto simp add*: *GlobalNdet-is-STOP-iff write0-Sync-STOP*
‹*?this*› ‹*inj-on c A*› *inj-on-diff*
simp flip: *GlobalNdet-factorization-union*
[*OF* ‹*c ‘ A ∩ S ≠ {}*› ‹*c ‘ A − S ≠ {}*›, *unfolded Int-Diff-Un*]
intro!: *arg-cong2*[**where** *f* = ‹(⊓)›] *mono-GlobalNdet-eq*)
qed
qed

corollary *STOP-Sync-ndet-write* :
‹*inj-on d B* ⟹
STOP ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* =
(*if d ‘ B ∩ S = {} then d*!!*b*∈*B* → (*STOP* ⟦*S*⟧ *Q b*)
else (*d*!!*b*∈(*B − d −‘ S*) → (*STOP* ⟦*S*⟧ *Q b*)) ⊓ *STOP*)›
by (*subst* (*1 2 3*) *Sync-commute*) (*simp add*: *ndet-write-Sync-STOP*)

12.6 Powerful Laws of CSP

12.6.1 Laws for Mndetprefix and Sync

lemma *Mndetprefix-Sync-Det-distr-FD* :
‹(⊓ *a* ∈ *A* → (*P a* ⟦ *C* ⟧ (⊓ *b* ∈ *B* → *Q b*))) □
(⊓ *b* ∈ *B* → ((⊓ *a* ∈ *A* → *P a*) ⟦ *C* ⟧ *Q b*))
⊑*FD* (⊓ *a* ∈ *A* → *P a*) ⟦ *C* ⟧ (⊓ *b* ∈ *B* → *Q b*)›
(**is** ‹*?lhs1 □ ?lhs2* ⊑*FD ?rhs*›)
if ‹*A ≠ {}*› ‹*B ≠ {}*› ‹*A ∩ C = {}*› ‹*B ∩ C = {}*›
proof −
have ‹*?lhs1* = ⊓ *b*∈*B*. ⊓ *a*∈*A*. (*a* → (*P a* ⟦*C*⟧ (*b* → *Q b*)))› (**is** ‹- = *?lhs1′*›)
by (*simp add*: ‹*A ≠ {}*› ‹*B ≠ {}*› *Mndetprefix-GlobalNdet*
Sync-distrib-GlobalNdet-left Sync-distrib-GlobalNdet-right
write0-def GlobalNdet-Mprefix-distr[*OF* ‹*B ≠ {}*›, *symmetric*])
(*subst GlobalNdet-sets-commute*, *simp*)
moreover have ‹*?lhs2* = ⊓ *b*∈*B*. ⊓ *a*∈*A*. (*b* → (*a* → *P a* ⟦*C*⟧ *Q b*))› (**is** ‹- = *?lhs2′*›)
by (*simp add*: ‹*A ≠ {}*› ‹*B ≠ {}*› *Mndetprefix-GlobalNdet*
Sync-distrib-GlobalNdet-left Sync-distrib-GlobalNdet-right
write0-def GlobalNdet-Mprefix-distr[*OF* ‹*A ≠ {}*›, *symmetric*])
ultimately have ‹*?lhs1 □ ?lhs2 = ?lhs1′ □ ?lhs2′*› **by** *simp*
moreover have ‹*?lhs1′ □ ?lhs2′* ⊑*FD* ⊓ *b*∈*B*. ⊓ *a*∈*A*. (*a* → (*P a* ⟦*C*⟧ (*b* → *Q b*)))
□ (*b* → ((*a* → *P a*) ⟦*C*⟧ *Q b*))›
by (*auto simp add*: ‹*A ≠ {}*› ‹*B ≠ {}*› *refine-defs GlobalNdet-projs Det-projs write0-def*)
moreover have ‹*. . .* = ⊓ *b*∈*B*. ⊓ *a*∈*A*. ((*a* → *P a*) ⟦*C*⟧ (*b* → *Q b*))›

by (*rule mono-GlobalNdet-eq*, *rule mono-GlobalNdet-eq*,
simp add: *write0-def*, *subst Mprefix-Sync-Mprefix-indep*)
(*use* ‹$A \cap C = \{\}$› ‹$B \cap C = \{\}$› **in** *auto*)
moreover have ‹$\ldots$ = *?rhs*›
by (*simp add*: ‹$A \neq \{\}$› ‹$B \neq \{\}$› *Mndetprefix-GlobalNdet*
Sync-distrib-GlobalNdet-left Sync-distrib-GlobalNdet-right)
ultimately show ‹*?lhs1* $\Box$ *?lhs2* $\sqsubseteq_{FD}$ *?rhs*› **by** *argo*
qed

lemmas *Mndetprefix-Sync-Det-distr-F* = *Mndetprefix-Sync-Det-distr-FD*[*THEN leFD-imp-leF*]
and *Mndetprefix-Sync-Det-distr-D* = *Mndetprefix-Sync-Det-distr-FD*[*THEN leFD-imp-leD*]

lemmas *Mndetprefix-Sync-Det-distr-T* = *Mndetprefix-Sync-Det-distr-F*[*THEN leF-imp-leT*]

lemma *Mndetprefix-Sync-Det-distr-DT* :
‹$[\![A \neq \{\};\ B \neq \{\};\ A \cap C = \{\};\ B \cap C = \{\}]\!] \Longrightarrow$
$(\sqcap\ a \in A \rightarrow (P\ a\ [\![\ C\ ]\!]\ (\sqcap\ b \in B \rightarrow Q\ b))) \Box$
$(\sqcap\ b \in B \rightarrow ((\sqcap\ a \in A \rightarrow P\ a)\ [\![\ C\ ]\!]\ Q\ b))$
$\sqsubseteq_{DT}$ $(\sqcap\ a \in A \rightarrow P\ a)\ [\![\ C\ ]\!]\ (\sqcap\ b \in B \rightarrow Q\ b)$›
by (*simp add*: *Mndetprefix-Sync-Det-distr-D Mndetprefix-Sync-Det-distr-T leD-leT-imp-leDT*)

12.6.2 Hiding Operator Laws

theorem *Hiding-Hiding-less-eq-process-Hiding-Un* : ‹$P \setminus A \setminus B \sqsubseteq_{FD} P \setminus (A \cup B)$›
proof (*rule failure-divergence-refine-optimizedI*)
fix *s* **assume** ‹$s \in \mathcal{D}\ (P \setminus (A \cup B))$›
then obtain *t u*
where $*$: ‹*ftF u*› ‹*tF t*› ‹*s* = *trace-hide t* (*ev* ‘ (*A* ∪ *B*)) @ *u*›
‹*t* ∈ $\mathcal{D}$ *P* ∨ (∃*f*. *isInfHiddenRun f P* (*A* ∪ *B*) ∧ *t* ∈ *range f*)›
unfolding *D-Hiding* **by** *blast*
have $**$: ‹*trace-hide t* (*ev* ‘ (*A* ∪ *B*)) = *trace-hide* (*trace-hide t* (*ev* ‘ *A*)) (*ev* ‘ *B*)›
using *trace-hide-ev-union* **by** *blast*
from $*(4)$ **show** ‹$s \in \mathcal{D}\ (P \setminus A \setminus B)$›
proof (*elim disjE exE*)
assume ‹*t* ∈ $\mathcal{D}$ *P*›
with *mem-D-imp-mem-D-Hiding* **have** ‹*trace-hide t* (*ev* ‘ *A*) ∈ $\mathcal{D}$ (*P* \ *A*)› **by** *blast*
thus ‹$s \in \mathcal{D}\ (P \setminus A \setminus B)$›
by (*subst D-Hiding*) (*use*$*$(*1*, *2*, *3*) $**$ *Hiding-tickFree* **in** *blast*)
next
fix *f* **assume** $**$: ‹*isInfHiddenRun f P* (*A* ∪ *B*) ∧ *t* ∈ *range f*›
hence ‹*strict-mono f*› **by** *simp*
define *ff* **where** ‹*ff i* ≡ *take* (*i* + *length* (*f* 0)) (*f i*)› **for** *i*
have $***$: ‹*isInfHiddenRun ff P* (*A* ∪ *B*) ∧ *t* ∈ *range ff*›
proof (*intro conjI allI*)
show ‹*strict-mono ff*›

```
    proof (unfold strict-mono-Suc-iff, rule allI, unfold ff-def)
      fix i nat
      from length-strict-mono[of f ‹Suc i›, OF ‹strict-mono f›]
      have $ : ‹take (i + length (f 0)) (f (Suc i)) < take ((Suc i) + length (f 0))
(f (Suc i))›
        by (simp add: take-Suc-conv-app-nth)
      from ‹strict-mono f›[THEN strict-monoD, of i ‹Suc i›, simplified]
      obtain t where ‹f (Suc i) = f i @ t› by (meson strict-prefixE')
      with length-strict-mono[of f i, OF ‹strict-mono f›]
      have ‹take (i + length (f 0)) (f i) = take (i + length (f 0)) (f (Suc i))› by
simp
      with $ show ‹take (i + length (f 0)) (f i) < take (Suc i + length (f 0)) (f
(Suc i))› by simp
    qed
  next
    show ‹ff i ∈ 𝒯 P› for i
      by (unfold ff-def) (metis ** prefixI append-take-drop-id is-processT3-TR)
  next
    show ‹trace-hide (ff i) (ev ‘ (A ∪ B)) = trace-hide (ff 0) (ev ‘ (A ∪ B))› for
i
    proof (rule order-antisym)
      have ‹f 0 ≤ f i› by (simp add: ** strict-mono-less-eq)
      hence ‹f 0 ≤ take (i + length (f 0)) (f i)›
        by (metis prefixE Prefix-Order.prefixI le-add2 take-all-iff take-append)
      from mono-trace-hide[OF this]
      show ‹trace-hide (ff 0) (ev ‘ (A ∪ B)) ≤ trace-hide (ff i) (ev ‘ (A ∪ B))›
        by (unfold ff-def) (metis le-add2 take-all-iff)
    next
      have ‹take (i + length (f 0)) (f i) ≤ f i›
        by (metis prefixI append-take-drop-id)
      from mono-trace-hide[OF this]
      show ‹trace-hide (ff i) (ev ‘ (A ∪ B)) ≤ trace-hide (ff 0) (ev ‘ (A ∪ B))›
        by (unfold ff-def) (metis ** le-add2 take-all-iff)
    qed
  next
    from ** obtain i where ‹t = f i› by blast
    moreover have ‹f 0 ≤ f i› by (simp add: ** strict-mono-less-eq)
    ultimately have ‹t = ff (max i (length t − length (f 0)))›
      by (simp add: ff-def max-def le-length-mono)
        (metis ** prefixE append-eq-conv-conj strict-mono-less-eq)
    thus ‹t ∈ range ff› by blast
  qed

  show ‹s ∈ 𝒟 (P \ A \ B)›
  proof (cases ‹∃ m. ∀ i>m. last (ff i) ∈ ev ‘ A›)
    assume ‹∃ m. ∀ i>m. last (ff i) ∈ ev ‘ A›
    then obtain m where $ : ‹m < i ⟹ last (ff i) ∈ ev ‘ A› for i by blast
    hence ‹tF (ff m)›
    by (metis *** strict-prefixE' append-T-imp-tickFree list.distinct(1) strict-mono-Suc-iff)
```

```
    have $$ : ‹∃ t. ff (i + m) = ff m @ t ∧ set t ⊆ ev ‘ A› for i
    proof (induct i)
      show ‹∃ t. ff (0 + m) = ff m @ t ∧ set t ⊆ ev ‘ A› by simp
    next
      fix i assume ‹∃ t. ff (i + m) = ff m @ t ∧ set t ⊆ ev ‘ A›
      then obtain t where ‹ff (i + m) = ff m @ t› ‹set t ⊆ ev ‘ A› by blast
      obtain r where ‹ff (Suc i + m) = ff (i + m) @ r›
        by (metis *** strict-prefixE' add-Suc strict-mono-Suc-iff)
      moreover have ‹length (ff (Suc i + m)) = Suc (length (ff (i + m)))›
        by (simp add: ff-def) (metis ** add-Suc length-strict-mono min-absorb2)
       ultimately have ‹length r = 1› by (metis Suc-eq-plus1 add-left-cancel
length-append)
      with ‹ff (Suc i + m) = ff (i + m) @ r›
      have ‹ff (Suc i + m) = ff (i + m) @ [last (ff (Suc i + m))]› by (cases r)
simp-all
      with ‹ff (i + m) = ff m @ t›
      have ‹ff (Suc i + m) = ff m @ t @ [last (ff (Suc i + m))]› by simp
      moreover have ‹last (ff (Suc i + m)) ∈ ev ‘ A› by (simp add: $)
      ultimately show ‹∃ t. ff (Suc i + m) = ff m @ t ∧ set t ⊆ ev ‘ A›
        by (intro exI[of - ‹t @ [last (ff (Suc i + m))]›]) (simp add: ‹set t ⊆ ev ‘
A›)
    qed
    then obtain fff where $$$$ : ‹ff (i + m) = ff m @ fff i› ‹set (fff i) ⊆ ev ‘
A› for i by metis
    show ‹s ∈ 𝒟 (P \ A \ B)›
      apply (simp add: D-Hiding)
     apply (rule exI[of - ‹trace-hide (ff m) (ev ‘ A)›], rule exI[of - u], intro conjI)
      subgoal by (fact ‹ftF u›)
      subgoal using Hiding-tickFree ‹tF (ff m)› by blast
      subgoal by (metis (no-types) *(3) *** rangeE trace-hide-ev-union)
      apply (rule disjI1)
      apply (rule exI[of - ‹ff m›], rule exI[of - ‹[]›], simp add: ‹tF (ff m)›)
      apply (rule disjI2)
      apply (rule exI[of - ‹λi. ff (i + m)›], intro conjI)
      subgoal by (metis (mono-tags, lifting) *** add-Suc strict-mono-Suc-iff)
      subgoal using *** by blast
      subgoal using $$$$ by (simp add: subset-iff)
      by (metis (mono-tags) add-0 rangeI)
  next
    assume ‹∄ m. ∀ i>m. last (ff i) ∈ ev ‘ A›
    { fix i :: nat assume ‹0 < i›
      from *** obtain t where ‹ff i = ff 0 @ t› ‹set t ⊆ ev ‘ (A ∪ B)›
        unfolding isInfHiddenRun-1 by blast
      with ‹0 < i› have ‹last (ff i) ∈ ev ‘ (A ∪ B)›
        by (cases t) (auto simp add: *** strict-mono-eq)
    }
    with ‹∄ m. ∀ i>m. last (ff i) ∈ ev ‘ A› have $ : ‹∀ i. ∃ j>i. last(ff j) ∈ ev ‘
B − ev ‘ A›
      by (metis Un-Diff-cancel2 Un-iff gr0I image-Un not-less0)
```

```
    define fff where ‹fff = rec-nat t (λi t. ff (SOME j. t < ff j ∧ last (ff j) ∈
ev ‘ B − ev ‘ A))›
    hence ‹fff i ∈ range ff› for i
      unfolding fff-def
      by (metis (mono-tags, lifting) *** gr0-conv-Suc nat.rec(2)
          old.nat.simps(6) rangeI zero-less-iff-neq-zero)
    have $$$ : ‹strict-mono (λi. trace-hide (fff i) (ev ‘ A))›
    proof (unfold strict-mono-Suc-iff, rule allI)
      fix i
      have £ : ‹fff (Suc i) = ff (SOME j. fff i < ff j ∧ last (ff j) ∈ ev ‘ B − ev
‘ A)›
        by (simp add: fff-def)
      from ‹⋀i. fff i ∈ range ff› obtain j where ‹fff i = ff j› by blast
       hence ‹∃j. fff i < ff j ∧ last (ff j) ∈ ev ‘ B − ev ‘ A› by (metis $ ***
monotoneD)
      hence ££ : ‹fff i < fff (Suc i) ∧ last (fff (Suc i)) ∉ ev ‘ A›
        by (metis (no-types, lifting) Diff-iff £ someI-ex)
      then obtain r where ‹fff (Suc i) = fff i @ r› ‹last r ∉ ev ‘ A›
      by (metis append-self-conv last-append less-eq-list-def prefix-def less-list-def)
      thus ‹trace-hide (fff i) (ev ‘ A) < trace-hide (fff (Suc i)) (ev ‘ A)›
        by (metis (mono-tags, lifting) prefixI ££ empty-filter-conv
            filter-append last-in-set nless-le self-append-conv)
    qed
    show ‹s ∈ 𝒟 (P \ A \ B)›
      apply (simp add: D-Hiding)
      apply (rule exI[of - ‹trace-hide t (ev ‘ A)›], rule exI[of - u], intro conjI)
      subgoal by (fact ‹ftF u›)
      subgoal using Hiding-tickFree ‹tF t› by blast
      subgoal by (simp add: ∗(3))
      apply (rule disjI2)
      apply (rule exI[of - ‹λi. trace-hide (fff i) (ev ‘ A)›], intro conjI)
      subgoal by (fact $$$)
       subgoal by (metis *** ‹⋀i. fff i ∈ range ff› mem-T-imp-mem-T-Hiding
rangeE)
    subgoal by (metis (no-types) *** ‹⋀i. fff i ∈ range ff› rangeE trace-hide-ev-union)
      by (metis (mono-tags, lifting) fff-def old.nat.simps(6) rangeI)
  qed
 qed
next
 assume subset-div : ‹𝒟 (P \ (A ∪ B)) ⊆ 𝒟 (P \ A \ B)›
 fix s X assume ‹(s, X) ∈ ℱ (P \ (A ∪ B))›
 then consider ‹s ∈ 𝒟 (P \ (A ∪ B))›
  | t where ‹s = trace-hide t (ev ‘ (A ∪ B))› ‹(t, X ∪ ev ‘ (A ∪ B)) ∈ ℱ P›
  unfolding F-Hiding D-Hiding by blast
 thus ‹(s, X) ∈ ℱ (P \ A \ B)›
 proof cases
   from subset-div D-F show ‹s ∈ 𝒟 (P \ (A ∪ B)) ⟹ (s, X) ∈ ℱ (P \ A \
B)› by blast
 next
```

```
    fix t assume ‹s = trace-hide t (ev ‘ (A ∪ B))› ‹(t, X ∪ ev ‘ (A ∪ B)) ∈ ℱ P›
    hence ∗ : ‹s = trace-hide (trace-hide t (ev ‘ A)) (ev ‘ B)›
      ‹(t, X ∪ ev ‘ B ∪ ev ‘ A) ∈ ℱ P›
      by (simp-all add: image-Un sup-commute sup-left-commute)
    show ‹(s, X) ∈ ℱ (P \ A \ B)›
      by (simp add: F-Hiding) (use ∗ in blast)
  qed
qed


theorem Hiding-Un : ‹P \ (A ∪ B) = P \ A \ B›
  if ‹finite A› for P :: ‹('a, 'r) process_ptick›
proof (rule order-antisym)
  show ‹P \ (A ∪ B) ≤ P \ A \ B›
  proof (rule failure-divergence-refine-optimizedI)
    fix s assume ‹s ∈ 𝒟 (P \ A \ B)›
    then obtain t u
      where ∗ : ‹ftF u› ‹tF t› ‹s = trace-hide t (ev ‘ B) @ u›
        ‹t ∈ 𝒟 (P \ A) ∨ (∃ x. isInfHidden-seqRun-strong x (P \ A) B t)›
      by (elim D-Hiding-seqRunE)
    from ∗(4) show ‹s ∈ 𝒟 (P \ (A ∪ B))›
    proof (elim disjE exE)
      assume ‹t ∈ 𝒟 (P \ A)›
      then obtain t' u'
        where ∗∗ : ‹ftF u'› ‹tF t'› ‹t = trace-hide t' (ev ‘ A) @ u'›
          ‹t' ∈ 𝒟 P ∨ (∃ x. isInfHidden-seqRun-strong x P A t')›
        by (elim D-Hiding-seqRunE)
      from ∗(1, 2) ∗∗(3) have ∗∗∗ : ‹ftF (trace-hide u' (ev ‘ B) @ u)›
        using Hiding-tickFree front-tickFree-append tickFree-append-iff by blast
      show ‹s ∈ 𝒟 (P \ (A ∪ B))›
        apply (unfold D-Hiding-seqRun, clarify)
        apply (rule exI[of - t'], rule exI[of - ‹trace-hide u' (ev ‘ B) @ u›])
        apply (intro conjI)
        subgoal by (fact ∗∗∗)
        subgoal by (fact ∗∗(2))
        subgoal by (simp add: ∗(3) ∗∗(3))
        by (metis ∗∗(4) Un-iff image-Un)
    next
      fix x assume ∗∗ : ‹isInfHidden-seqRun-strong x (P \ A) B t›
      have trH-x : ‹trace-hide (seqRun t x i) (ev ‘ B) = trace-hide t (ev ‘ B)› for i
        using ∗∗ trace-hide-seqRun-eq-iff by blast
      from ∗∗ have ‹∀ i. seqRun t x i ∈ {trace-hide t (ev ‘ A) |t. (t, ev ‘ A) ∈ ℱ
P}›
        unfolding T-Hiding D-Hiding by blast
      with F-T have ‹∀ i. ∃ w. seqRun t x i = trace-hide w (ev ‘ A) ∧ w ∈ 𝒯 P›
by blast
      then obtain f where ∗∗∗ : ‹seqRun t x i = trace-hide (f i) (ev ‘ A)› ‹f i ∈
𝒯 P› for i by metis
      have ‹inj f› by (rule injI) (metis ∗∗∗(1) seqRun-eq-iff)
```

```
    with finite-imageD have ‹infinite (range f)› by blast
    have $ : ‹set (take i t') ⊆ set (seqRun t x i) ∪ ev ‘ A› if ‹t' ∈ range f› for i
t'
    proof −
      from ‹t' ∈ range f› obtain j where ‹t' = f j› ..
      hence £ : ‹seqRun t x j = trace-hide t' (ev ‘ A)› ‹t' ∈ 𝒯 P› by (simp-all
add: ∗∗∗)
      consider ‹i ≤ j› | ‹j ≤ i› by linarith
      thus ‹set (take i t') ⊆ set (seqRun t x i) ∪ ev ‘ A›
      proof cases
        assume ‹j ≤ i›
        hence ‹seqRun t x j ≤ seqRun t x i› by simp
        hence ‹set (seqRun t x j) ⊆ set (seqRun t x i)›
          by (metis prefixE Un-iff set-append subsetI)
        moreover have ‹set (take i t') ⊆ set (seqRun t x j) ∪ ev ‘ A›
          by (rule subsetI, drule in-set-takeD) (simp add: £(1))
        ultimately show ‹set (take i t') ⊆ set (seqRun t x i) ∪ ev ‘ A› by blast
      next
        assume ‹i ≤ j›
        hence ‹seqRun t x i ≤ seqRun t x j› by simp
        hence ‹take i (seqRun t x j) ≤ seqRun t x i› by simp
       have ‹seqRun t x j = trace-hide (take i t') (ev ‘ A) @ trace-hide (drop i t')
(ev ‘ A)›
          by (metis £(1) append-take-drop-id filter-append)
        moreover have ‹length (trace-hide (take i t') (ev ‘ A)) ≤ i›
          by (metis length-filter-le length-take min.bounded-iff)
       ultimately have ‹trace-hide (take i t') (ev ‘ A) ≤ take i (seqRun t x j)›
by simp
        with ‹take i (seqRun t x j) ≤ seqRun t x i› obtain t''
          where ‹seqRun t x i = trace-hide (take i t') (ev ‘ A) @ t''›
          by (meson prefixE dual-order.trans)
        thus ‹set (take i t') ⊆ set (seqRun t x i) ∪ ev ‘ A› by (simp add: subsetI)
      qed
    qed
    have ‹finite {take i w |w. w ∈ range f}› for i
    proof (induct i)
      show ‹finite {take 0 w |w. w ∈ range f}› by simp
    next
      fix i assume hyp : ‹finite {take i w |w. w ∈ range f}›
      show ‹finite {take (Suc i) w |w. w ∈ range f}›
      proof (rule finite-subset)
        show ‹  {take (Suc i) w |w. w ∈ range f}
              ⊆    {take i w |w. w ∈ range f}
                ∪ (⋃ e∈set (seqRun t x (Suc i)) ∪ ev ‘ A. {take i w @ [e] |w. w ∈
range f})›
          (is ‹?lhs ⊆ {take i w |w. w ∈ range f} ∪ (⋃ e∈?S1. ?S2 e)›)
        proof (intro subsetI)
          fix t' assume ‹t' ∈ ?lhs›
          then obtain w where £ : ‹t' = take (Suc i) w› ‹w ∈ range f› by blast
```

```
          show ‹t' ∈ {take i w |w. w ∈ range f} ∪ (⋃ e∈?S1. ?S2 e)›
          proof (cases ‹i < length t'›)
            assume ‹i < length t'›
            with £(1) take-Suc-conv-app-nth
            have ‹take (Suc i) t' = take i w @ [t' ! i]› by auto
             moreover from £ $ ‹i < length t'› nth-mem have ‹t' ! i ∈ ?S1› by
blast
            ultimately have ‹t' ∈ (⋃ e∈?S1. ?S2 e)›
              by (intro UN-I[of ‹t' ! i›], simp-all)
                (metis £ less-not-refl min.absorb2 not-le-imp-less take-take)
            thus ‹t' ∈ {take i w |w. w ∈ range f} ∪ (⋃ e∈?S1. ?S2 e)› by simp
          next
            assume ‹¬ i < length t'›
            hence ‹take (Suc i) t' = take i t'› by simp
             with £ show ‹t' ∈ {take i w |w. w ∈ range f} ∪ (⋃ e∈?S1. ?S2 e)›
by auto
          qed
        qed
      next
        show ‹finite ({take i w |w. w ∈ range f} ∪
                    (⋃ e∈set (seqRun t x (Suc i)) ∪ ev ‘ A.
                        {take i w @ [e] |w. w ∈ range f}))›
        proof (rule finite-UnI)
          show ‹finite {take i w |w. w ∈ range f}› by (fact hyp)
        next
          show ‹finite (⋃ e∈set (seqRun t x (Suc i)) ∪ ev ‘ A. {take i w @ [e] |w.
w ∈ range f})›
          proof (rule finite-UN-I)
            show ‹finite (set (seqRun t x (Suc i)) ∪ ev ‘ A)›
              by (simp add: ‹finite A›) — Here we use that A is finite.
          next
            fix e assume ‹e ∈ set (seqRun t x (Suc i)) ∪ ev ‘ A›
            have ‹  {take i w @ [e] |w. w ∈ range f}
                 = (λt'. t' @ [e]) ‘ {take i w |w. w ∈ range f}› by auto
            also have ‹finite …› by (rule finite-imageI) (fact hyp)
            finally show ‹finite {take i w @ [e] |w. w ∈ range f}› .
          qed
        qed
      qed
    qed
     also have ‹{take i w |w. w ∈ range f} = {w. ∃ t'∈range f. w = take i t'}›
for i by auto
    finally have ‹∀ i. finite {w. ∃ t'∈range f. w = take i t'}› ..
    from KoenigLemma[OF ‹infinite (range f)› this]
    obtain ff :: ‹nat ⇒ ('a, 'r) trace_ptick›
      where $$ : ‹strict-mono ff› ‹range ff ⊆ {t. ∃ t'∈range f. t ≤ t'}› by blast
    from $$(2) have $$$ : ‹∃ t'. ff (Suc j) ≤ t' ∧ t' ∈ range f› for j by blast
    hence ‹ftF (ff (Suc j))› for j
      by (metis ***(2) imageE is-processT2-TR is-processT3-TR)
```

```
    hence ‹tF (ff j)› for j
      using strict-monoD[OF $$(1), of j ‹Suc j›, simplified]
      by (metis strict-prefixE' front-tickFree-append-iff list.distinct(1))
    from $$(2) ***(2) have ‹ff (j + i) ∈ 𝒯 P› for i j
      by (simp add: subset-iff) (meson is-processT3-TR rangeI)
    have $$$$ : ‹∃ w. trace-hide (ff i) (ev ‘ A) ≤ t @ w› for i
    proof −
      from $$(2) obtain j where ‹ff i ≤ f j› by auto
      moreover from ** isInfHiddenRun-1 have ‹∃ w. seqRun t x j = t @ w›
        by (simp add: seqRun-def)
      ultimately show ‹∃ w. trace-hide (ff i) (ev ‘ A) ≤ t @ w›
        by (metis ***(1) mono-trace-hide)
    qed
    have ‹∃ i. t ≤ trace-hide (ff i) (ev ‘ A)›
    proof (rule ccontr)
      define fff where ‹fff i ≡ trace-hide (ff i) (ev ‘ A)› for i
      assume assm : ‹∄ i. t ≤ fff i›
      moreover from $$$$ have ‹∀ i. ∃ w. fff i ≤ t @ w› unfolding fff-def ..
      ultimately have assm-bis : ‹∀ i. fff i ≤ t›
        by (metis prefixI le-same-imp-eq-or-less order.order-iff-strict)
      have ‹mono fff›
        by (rule monoI, simp add: fff-def)
          (metis $$(1) prefixE prefixI filter-append strict-mono-less-eq)
      from mono-constant[OF this assm-bis]
      obtain j where ‹j ≤ i ⟹ fff i = fff j› for i by blast
      have ‹fff j ∈ 𝒟 (P \ A)›
        apply (unfold D-Hiding, clarify)
        apply (rule exI[of - ‹ff j›], rule exI[of - ‹[]›])
        apply (simp add: fff-def ‹tF (ff j)›)
        apply (rule disjI2)
        apply (rule exI[of - ‹λi. ff (j + i)›])
        apply (intro conjI)
        subgoal by (simp add: $$(1) strict-monoI strict-mono-less)
        subgoal by (simp add: ‹⋀i. ff (j + i) ∈ 𝒯 P›)
        subgoal by (metis ‹⋀i. j ≤ i ⟹ fff i = fff j› fff-def le-add1)
        subgoal by (metis Nat.add-0-right rangeI) .
      thus False
        by (metis (no-types) ** prefixE T-imp-front-tickFree append.right-neutral
assm-bis
            front-tickFree-append-iff is-processT3-TR is-processT7 t-le-seqRun)
    qed
    then obtain j where ‹t ≤ trace-hide (ff j) (ev ‘ A)› ..
    have £ : ‹s = trace-hide (ff j) (ev ‘ (A ∪ B)) @ u›
    proof (unfold *(3), rule arg-cong[where f = ‹λx. x @ -›])
      from $$(1) $$$ obtain l where ‹ff j ≤ f l›
        by (metis dual-order.trans order-less-imp-le rangeE strict-mono-Suc-iff)
      from mono-trace-hide[OF this] have ‹trace-hide (ff j) (ev ‘ A) ≤ seqRun t
x l›
        unfolding ***(1) by presburger
```

```
    with mono-trace-hide[OF this] mono-trace-hide[OF ‹t ≤ trace-hide (ff j) (ev ‘ A)›]
    show ‹trace-hide t (ev ‘ B) = trace-hide (ff j) (ev ‘ (A ∪ B))›
      by (metis trH-x dual-order.eq-iff trace-hide-ev-union)
  qed
  have ℒℒ : ‹trace-hide (ff (j + i)) (ev ‘ (A ∪ B)) = trace-hide (ff (j + 0)) (ev ‘ (A ∪ B))› for i
  proof −
    from $$(1) $$$ obtain l where ‹ff (j + i) ≤ f l›
      by (metis dual-order.trans order-less-imp-le rangeE strict-mono-Suc-iff)
    from mono-trace-hide[OF this] have ‹trace-hide (ff (j + i)) (ev ‘ A) ≤ seqRun t x l›
      unfolding ∗∗∗(1) by presburger
    from mono-trace-hide[OF this, of B]
      mono-trace-hide[THEN mono-trace-hide, of - - B A,
        OF ‹strict-mono ff›[THEN strict-mono-mono, THEN monoD, of j ‹j + i›, simplified]]
      mono-trace-hide[OF ‹t ≤ trace-hide (ff j) (ev ‘ A)›, of B]
    show ‹trace-hide (ff (j + i)) (ev ‘ (A ∪ B)) =
        trace-hide (ff (j + 0)) (ev ‘ (A ∪ B))› by (simp add: trH-x)
  qed
  show ‹s ∈ 𝒟 (P \ (A ∪ B))›
    apply (unfold D-Hiding, clarify)
    apply (rule exI[of - ‹ff j›], rule exI[of - u])
    apply (intro conjI)
    subgoal by (fact ‹ftF u›)
    subgoal by (fact ‹tF (ff j)›)
    subgoal by (fact ℒℒ)
    apply (rule disjI2)
    apply (rule exI[of - ‹λi. ff (j + i)›])
    apply (intro conjI)
    subgoal by (simp add: $$(1) strict-monoI strict-mono-less)
    subgoal by (simp add: ‹⋀i. ff (j + i) ∈ 𝒯 P›)
    subgoal by (use ℒℒ in blast)
    by (metis Nat.add-0-right rangeI)
 qed
next
 assume subset-div : ‹𝒟 (P \ A \ B) ⊆ 𝒟 (P \ (A ∪ B))›
 fix s X assume ‹(s, X) ∈ ℱ (P \ A \ B)›
 from this[simplified F-Hiding[of ‹P \ A› B]] D-Hiding[of ‹P \ A› B]
 consider ‹s ∈ 𝒟 (P \ A \ B)›
  | t where ‹s = trace-hide t (ev ‘ B)› ‹(t, X ∪ ev ‘ B) ∈ ℱ (P \ A)› by blast
 thus ‹(s, X) ∈ ℱ (P \ (A ∪ B))›
 proof cases
  from subset-div D-F show ‹s ∈ 𝒟 (P \ A \ B) ⟹ (s, X) ∈ ℱ (P \ (A ∪ B))› by blast
 next
  fix t assume ∗ : ‹s = trace-hide t (ev ‘ B)› ‹(t, X ∪ ev ‘ B) ∈ ℱ (P \ A)›
  from ∗(2) consider ‹t ∈ 𝒟 (P \ A)›
```

```
      | u where ‹t = trace-hide u (ev ‘ A)› ‹(u, X ∪ ev ‘ B ∪ ev ‘ A) ∈ ℱ P›
       unfolding F-Hiding D-Hiding by blast
     thus ‹(s, X) ∈ ℱ (P \ (A ∪ B))›
     proof cases
       assume ‹t ∈ 𝒟 (P \ A)›
       with ∗(1) mem-D-imp-mem-D-Hiding have ‹s ∈ 𝒟 (P \ A \ B)› by fast
       with subset-div D-F show ‹(s, X) ∈ ℱ (P \ (A ∪ B))› by blast
     next
       fix u assume ∗∗ : ‹t = trace-hide u (ev ‘ A)› ‹(u, X ∪ ev ‘ B ∪ ev ‘ A) ∈
ℱ P›
       from ∗(1) ∗∗(1) have ‹s = trace-hide u (ev ‘ (A ∪ B))› by simp
       moreover from ∗∗(2) have ‹(u, X ∪ ev ‘ (A ∪ B)) ∈ ℱ P›
         by (simp add: image-Un sup-commute sup-left-commute)
      ultimately show ‹(s, X) ∈ ℱ (P \ (A ∪ B))› unfolding F-Hiding by blast
     qed
   qed
 qed
next
 show ‹P \ A \ B ≤ P \ (A ∪ B)› by (fact Hiding-Hiding-less-eq-process-Hiding-Un)
qed
```

12.6.3 Sync Operator Laws

Preliminaries

```
lemma tickFree-isInfHiddenRun : ‹tF s›
 if ‹isInfHiddenRun f P A› and ‹s ∈ range f›
proof −
 from ‹s ∈ range f› obtain i where ‹s = f i› ..
 moreover have ‹f i < f (Suc i)› by (meson ‹isInfHiddenRun f P A› strict-mono-Suc-iff)
 ultimately obtain t where ‹t ≠ []› ‹f (Suc i) = s @ t› by (meson strict-prefixE′
list.discI)
 moreover from ‹isInfHiddenRun f P A› is-processT2-TR
 have ‹ftF (f (Suc i))› by blast
 ultimately show ‹tF s› by (simp add: front-tickFree-append-iff)
qed

lemma Hiding-interleave:
 ‹r setinterleaves ((t, u), C) ⟹
  (trace-hide r A) setinterleaves ((trace-hide t A, trace-hide u A), C)›

proof (induct ‹(t, C, u)› arbitrary: r t u rule: setinterleaving.induct)
 case 1 thus ?case by simp
next
 case (2 x t) thus ?case by auto
next
 case (3 y u) thus ?case by auto
next
 case (4 x t y u)
 thus ?case
```

```
    by (simp split: if-splits)
      (safe, simp-all, (use SyncSingleHeadAdd setinterleaving-sym in blast)+)
qed

lemma non-Sync-interleaving:
  ‹(set t ∪ set u) ∩ C = {} ⟹ setinterleaving (t, C, u) ≠ {}›
  by (induct ‹(t, C, u)› arbitrary: t u rule: setinterleaving.induct) simp-all


lemma interleave-Hiding:
  ‹s setinterleaves ((trace-hide t A, trace-hide u A), C)
   ⟹ ∃ r. s = trace-hide r A ∧ r setinterleaves ((t, u), C)› if ‹A ∩ C = {}›
proof (induct ‹(t, C, u)› arbitrary: t u s rule: setinterleaving.induct)
  case 1
  thus ?case by simp
next
  case (2 y u)
  from 2.prems ‹A ∩ C = {}› show ?case
    by (simp add: disjoint-iff split: if-split-asm)
      (metis 2.hyps 2.prems emptyLeftProperty filter.simps(1))+
next
  case (3 x t)
  from 3.prems ‹A ∩ C = {}› show ?case
    by (simp add: disjoint-iff split: if-split-asm)
      (metis 3.hyps 3.prems emptyRightProperty filter.simps(1))+
next
  case (4 x t y u)
  from 4.prems ‹A ∩ C = {}› show ?case
    apply (cases ‹x = y›, simp-all add: disjoint-iff split: if-split-asm)
    subgoal using 4.hyps(1) by force
    subgoal by (metis (no-types, lifting) 4.hyps(3) 4.hyps(4) filter.simps(2))
    subgoal by (metis (no-types, lifting) 4.hyps(3) filter.simps(2))
    subgoal using 4.hyps(2) by force
    subgoal by (metis (no-types, lifting) 4.hyps(3) 4.hyps(4) filter.simps(2))
    subgoal using 4.hyps(5) by force
    subgoal by (metis (no-types, lifting) 4.hyps(2) 4.hyps(4) filter.simps(2))
    subgoal by (metis (no-types, lifting) 4.hyps(3) 4.hyps(5) filter.simps(2))
    subgoal by (metis (no-types, lifting) 4.hyps(4) filter.simps(2))
    done
qed


lemma le-trace-hide : ‹u ≤ trace-hide t S ⟹ ∃ u'. u = trace-hide u' S ∧ u' ≤ t›
proof (induct t arbitrary: u)
  show ‹u ≤ trace-hide [] S ⟹ ∃ u'. u = trace-hide u' S ∧ u' ≤ []› for u by simp
next
  fix a t u assume prem: ‹u ≤ trace-hide (a # t) S›
  assume hyp : ‹u ≤ trace-hide t S ⟹ ∃ u'. u = trace-hide u' S ∧ u' ≤ t› for u
  from prem consider ‹u = []› | ‹a ∈ S› ‹u ≤ trace-hide t S›
```

```
    | v where ‹a ∉ S› ‹u = a # v› ‹v ≤ trace-hide t S›
    by (cases u) (simp-all split: if-split-asm)
  thus ‹∃ u'. u = trace-hide u' S ∧ u' ≤ a # t›
  proof cases
    show ‹u = [] ⟹ ∃ u'. u = trace-hide u' S ∧ u' ≤ a # t›
      by (metis filter.simps(1) nil-le)
  next
    assume ‹a ∈ S› ‹u ≤ trace-hide t S›
    from hyp[OF ‹u ≤ trace-hide t S›] obtain u' where ‹u = trace-hide u' S ∧
u' ≤ t› ..
    with ‹a ∈ S› have ‹u = trace-hide (a # u') S ∧ a # u' ≤ a # t› by simp
    thus ‹∃ u'. u = trace-hide u' S ∧ u' ≤ a # t› ..
  next
    fix v assume ‹a ∉ S› ‹u = a # v› ‹v ≤ trace-hide t S›
    from hyp[OF ‹v ≤ trace-hide t S›] obtain v' where ‹v = trace-hide v' S ∧ v'
≤ t› ..
    with ‹a ∉ S› ‹u = a # v› have ‹u = trace-hide (a # v') S ∧ a # v' ≤ a #
t› by simp
    thus ‹∃ u'. u = trace-hide u' S ∧ u' ≤ a # t› ..
  qed
qed
```

```
lemma append-interleave :
  ‹s1 setinterleaves ((t1, u1), S) ⟹ s2 setinterleaves ((t2, u2), S) ⟹
   (s1 @ s2) setinterleaves ((t1 @ t2, u1 @ u2), S)›
proof (induct ‹(t1, S, u1)› arbitrary: s1 t1 u1 rule: setinterleaving.induct)
  case 1 thus ?case by simp
next
  case (2 y u1)
  thus ?case by (auto split: if-split-asm)
      (use SyncSingleHeadAdd setinterleaving-sym in blast)
next
  case (3 x t1)
  thus ?case by (auto split: if-split-asm) (blast intro: SyncSingleHeadAdd)
next
  case (4 x t2 y u2)
  thus ?case by auto
qed
```

The Theorem: Sync and Hiding

```
theorem Hiding-Sync : ‹(P ⟦S⟧ Q) \ A = (P \ A) ⟦S⟧ (Q \ A)›
  if ‹finite A› and ‹A ∩ S = {}› for P Q :: ‹('a, 'r) process_ptick›
    — Monster theorem!
proof (subst Process-eq-spec-optimized, safe)
  let ?trH-A = ‹λt. trace-hide t (ev ‘ A)› and ?tick-S = ‹range tick ∪ ev ‘ S›
  fix s assume ‹s ∈ 𝒟 (P ⟦S⟧ Q \ A)›
```

```
  then obtain t u
    where ∗ : ‹ftF u› ‹tF t› ‹s = ?trH-A t @ u›
      ‹t ∈ 𝒟 (P ⟦S⟧ Q) ∨ (∃ x. isInfHidden-seqRun-strong x (P ⟦S⟧ Q) A t)›
    by (elim D-Hiding-seqRunE)
  from ∗(4) show ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
  proof (elim disjE exE)
    assume ‹t ∈ 𝒟 (P ⟦S⟧ Q)›
    then obtain t' u' r' v'
      where ∗∗ : ‹ftF v'› ‹tF r' ∨ v' = []›
        ‹t = r' @ v'› ‹r' setinterleaves ((t', u'), ?tick-S)›
        ‹t' ∈ 𝒟 P ∧ u' ∈ 𝒯 Q ∨ t' ∈ 𝒟 Q ∧ u' ∈ 𝒯 P›
      unfolding D-Sync by blast
    { fix t' u' and P Q
      assume ∗∗∗ : ‹r' setinterleaves ((t', u'), ?tick-S)›
        ‹t' ∈ 𝒟 P› ‹u' ∈ 𝒯 Q›
      have ‹?trH-A r' setinterleaves ((?trH-A t', ?trH-A u'), ?tick-S)›
        using ∗∗∗(1) Hiding-interleave by blast
      moreover from ∗∗∗(2) mem-D-imp-mem-D-Hiding have ‹?trH-A t' ∈ 𝒟 (P
\ A)› by blast
      moreover from ∗∗∗(3) mem-T-imp-mem-T-Hiding have ‹?trH-A u' ∈ 𝒯 (Q
\ A)› by blast
      ultimately have ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
        apply (simp add: ∗(3) D-Sync)
        apply (rule exI[of - ‹?trH-A t'›], rule exI[of - ‹?trH-A u'›],
            rule exI[of - ‹?trH-A r'›], rule exI[of - ‹?trH-A v' @ u›])
        apply (simp add: ∗(3) ∗∗(3) Hiding-tickFree)
       using ∗(1, 2) ∗∗(3) Hiding-tickFree front-tickFree-append tickFree-append-iff
by blast
    } note $ = this
    from ∗∗(5) show ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
    proof (elim disjE conjE)
      show ‹t' ∈ 𝒟 P ⟹ u' ∈ 𝒯 Q ⟹ s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
        by (metis $ ∗∗(4))
      show ‹t' ∈ 𝒟 Q ⟹ u' ∈ 𝒯 P ⟹ s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
        by (metis $ ∗∗(4) Sync-commute)
    qed
  next
    fix x assume ∗∗ : ‹isInfHidden-seqRun-strong x (P ⟦S⟧ Q) A t›
    from ∗∗ have ∗∗∗ : ‹∃ t' u'. t' ∈ 𝒯 P ∧ u' ∈ 𝒯 Q ∧
                              seqRun t x i setinterleaves ((t', u'), ?tick-S)› for i
      unfolding T-Sync D-Sync by blast
    define ftu where ‹ftu i ≡ SOME (t', u'). t' ∈ 𝒯 P ∧ u' ∈ 𝒯 Q ∧
                          seqRun t x i setinterleaves ((t', u'), ?tick-S)› for i
    define ft fu where ‹ft ≡ λi. fst (ftu i)› and ‹fu ≡ λi. snd (ftu i)›
    have ∗∗∗∗ : ‹ft i ∈ 𝒯 P› ‹fu i ∈ 𝒯 Q›
      ‹seqRun t x i setinterleaves ((ft i, fu i), ?tick-S)› for i
      by (use ∗∗∗[of i] in ‹simp add: ft-def fu-def,
                          cases ‹ftu i›, simp add: ftu-def,
                          metis (mono-tags, lifting) case-prod-conv someI-ex›)+
```

```
  from ** have ‹set (seqRun t x i) ⊆ set t ∪ ev ‘ A› for i
    by (auto simp add: seqRun-def)

  have ‹set (ft i) ∪ set (fu i) ⊆ set t ∪ ev ‘ A› for i
    by (rule subset-trans[OF interleave-set[OF ****(3)]])
      (fact ‹set (seqRun t x i) ⊆ set t ∪ ev ‘ A›)
  have ‹inj ftu›
  proof (rule injI)
    fix i j assume ‹ftu i = ftu j›
    obtain t' u' where ‹(t', u') = ftu i› by (metis surj-pair)
    with ‹ftu i = ftu j› have ‹seqRun t x i setinterleaves ((t', u'), ?tick-S)›
      ‹seqRun t x j setinterleaves ((t', u'), ?tick-S)›
      by (metis ****(3) fst-conv ft-def fu-def snd-conv)+
   from interleave-eq-size[OF this] have ‹length (seqRun t x i) = length (seqRun
t x j)› .
    thus ‹i = j› by simp
  qed
  hence ‹infinite (range ftu)› using finite-imageD by blast
  moreover have ‹range ftu ⊆ range ft × range fu›
    by (clarify, metis fst-conv ft-def fu-def rangeI snd-conv)
  ultimately have ‹infinite (range ft) ∨ infinite (range fu)›
    by (meson finite-SigmaI infinite-super)

  { fix ft fu P Q
    assume assms : ‹infinite (range ft)›
      ‹⋀i. set (ft i) ∪ set (fu i) ⊆ set t ∪ ev ‘ A›
      ‹⋀i. ft i ∈ T P› ‹⋀i. fu i ∈ T Q›
      ‹⋀i. seqRun t x i setinterleaves ((ft i, fu i), ?tick-S)›

    have ‹finite {t. ∃t'∈range ft. t = take i t'}› for i
    proof (rule finite-subset)
      show ‹ {w. ∃t'∈range ft. w = take i t'}
          ⊆ {w. set w ⊆ set t ∪ ev ‘ A ∧ length w ≤ i}›
        by auto (meson Un-iff assms(2) in-set-takeD subsetD)
    next
      show ‹finite {w. set w ⊆ set t ∪ ev ‘ A ∧ length w ≤ i}›
        by (rule finite-lists-length-le) (simp add: ‹finite A›)
          — Here we use the assumption finite A.
    qed
    with assms(1) obtain ft' :: ‹nat ⇒ -›
      where $ : ‹strict-mono ft'› ‹range ft' ⊆ {w. ∃t'∈range ft. w ≤ t'}›
      using KoenigLemma by blast
    from $(2) assms(3) is-processT3-TR have ‹range ft' ⊆ T P› by blast
    define ft'' where ‹ft'' i ≡ ft' (i + length t)› for i
    find-theorems name: unifo
    from ‹range ft' ⊆ T P› have ‹range ft'' ⊆ T P› and ‹strict-mono ft''›
      by (auto simp add: ft''-def $(1) strict-monoD strict-monoI)
    have $$ : ‹?trH-A (ft'' i) = ?trH-A (ft'' 0)› for i
    proof −
```

have ‹*length t* ≤ *length* (*ft'' 0*)›
by (*metis* $(*1*) *add-0 add-leD1 ft''-def length-strict-mono*)
obtain *t'* **where** ‹*ft'' i* = *ft'' 0* @ *t'*›
by (*meson prefixE* ‹*strict-mono ft''*› *strict-mono-less-eq zero-order*(*1*))
moreover from $(*2*) **obtain** *j* **where** ‹*ft'' i* ≤ *ft j*› **by** (*auto simp add*: *ft''-def*)
ultimately obtain *t''* **where** ‹*ft j* = *ft'' 0* @ *t'* @ *t''*› **by** (*metis prefixE append.assoc*)
have ‹*set* (*t'* @ *t''*) ⊆ *set* (*drop* (*length t*) (*seqRun t x j*))›
proof (*rule subset-trans*)
show ‹*set* (*t'* @ *t''*) ⊆ *set* (*drop* (*length* (*ft'' 0*)) (*seqRun t x j*))›
by (*rule interleave-order*, *fold* ‹*ft j* = *ft'' 0* @ *t'* @ *t''*›, *fact assms*(*5*))
next
show ‹*set* (*drop* (*length* (*ft'' 0*)) (*seqRun t x j*)) ⊆ *set* (*drop* (*length t*) (*seqRun t x j*))›
by (*simp add*: ‹*length t* ≤ *length* (*ft'' 0*)› *set-drop-subset-set-drop*)
qed
also from ∗∗ **have** ‹*set* (*drop* (*length t*) (*seqRun t x j*)) ⊆ *ev* ‘ *A*›
by (*auto simp add*: *seqRun-def*)
finally show ‹*?trH-A* (*ft'' i*) = *?trH-A* (*ft'' 0*)›
by (*simp add*: ‹*ft'' i* = *ft'' 0* @ *t'*› *subset-iff*)
qed
from $(*2*) **obtain** *i* **where** ‹*ft'' 0* ≤ *ft i*› **by** (*auto simp add*: *ft''-def*)
with *prefixE* **obtain** *v* **where** ‹*ft i* = *ft'' 0* @ *v*› **by** *blast*

have ‹*ftF u*› **by** (*fact* ∗(*1*))
moreover have ‹*tF* (*?trH-A* (*seqRun t x i*)) ∨ *u* = []›
by (*metis* ∗(*2*) ∗∗ *Hiding-tickFree trace-hide-seqRun-eq-iff*)
moreover have ‹*s* = *?trH-A* (*seqRun t x i*) @ *u*›
by (*metis* ∗(*3*) ∗∗ *trace-hide-seqRun-eq-iff*)
moreover have ‹*?trH-A* (*seqRun t x i*) *setinterleaves* ((*?trH-A* (*ft i*), *?trH-A* (*fu i*)), *?tick-S*)›
using *assms*(*5*) *Hiding-interleave* **by** *blast*
moreover have ‹*?trH-A* (*ft i*) ∈ $\mathcal{D}$ (*P* \ *A*)›
apply (*unfold D-Hiding*, *clarify*)
apply (*rule exI*[*of* - ‹*ft'' 0*›])
apply (*rule exI*[*of* - ‹*?trH-A v*›])
apply (*intro conjI*)
subgoal by (*metis assms*(*3*) *Hiding-front-tickFree* ‹*ft i* = *ft'' 0* @ *v*›
front-tickFree-Nil front-tickFree-nonempty-append-imp is-processT2-TR)

subgoal by (*metis strict-prefixE' T-imp-front-tickFree* ‹*range ft''* ⊆ $\mathcal{T}$ *P*› ‹*strict-mono ft''*›
front-tickFree-append-iff list.distinct(*1*) *range-subsetD strict-mono-Suc-iff*)
subgoal by (*simp add*: ‹*ft i* = *ft'' 0* @ *v*›)
subgoal using $$ ‹*range ft''* ⊆ $\mathcal{T}$ *P*› ‹*strict-mono ft''*› **by** *blast* **.**
moreover have ‹*?trH-A* (*fu i*) ∈ $\mathcal{T}$ (*Q* \ *A*)›
proof (*cases* ‹∃ *t'*. *?trH-A t'* = *?trH-A* (*fu i*) ∧ (*t'*, *ev* ‘ *A*) ∈ $\mathcal{F}$ *Q*›)
assume ‹∃ *t'*. *?trH-A t'* = *?trH-A* (*fu i*) ∧ (*t'*, *ev* ‘ *A*) ∈ $\mathcal{F}$ *Q*›

```
      then obtain t' where ‹?trH-A (fu i) = ?trH-A t'› ‹(t', ev ‘ A) ∈ ℱ Q› by
metis
       thus ‹?trH-A (fu i) ∈ 𝒯 (Q \ A)› unfolding T-Hiding by blast
      next
       assume ‹∄t'. ?trH-A t' = ?trH-A (fu i) ∧ (t', ev ‘ A) ∈ ℱ Q›
       with assms(4) inf-hidden obtain fu' j
         where $$$ : ‹isInfHiddenRun fu' Q A› ‹fu i = fu' j› by blast
       show ‹?trH-A (fu i) ∈ 𝒯 (Q \ A)›
         apply (unfold T-Hiding, simp)
         apply (rule disjI2)
         apply (rule exI[of - ‹fu' j›], rule exI[of - ‹[]›])
         apply (intro conjI)
         subgoal by simp
        subgoal by (metis $$$(1) strict-prefixE' T-imp-front-tickFree neq-Nil-conv
              front-tickFree-nonempty-append-imp strict-mono-Suc-iff)
         subgoal by (simp add: $$$(2))
         subgoal using $$$(1) by blast .
      qed
     ultimately have ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))› unfolding D-Sync by blast
    } note $ = this

    from ‹infinite (range ft) ∨ infinite (range fu)›
    show ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
    proof (elim disjE)
      from $ **** ‹⋀i. set (ft i) ∪ set (fu i) ⊆ set t ∪ ev ‘ A›
      show ‹infinite (range ft) ⟹ s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))› by blast
    next
      from $ **** ‹⋀i. set (ft i) ∪ set (fu i) ⊆ set t ∪ ev ‘ A›
      show ‹infinite (range fu) ⟹ s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
        by (metis Sync-commute setinterleaving-sym sup-commute)
    qed
  qed
next
  let ?trH-A = ‹λt. trace-hide t (ev ‘ A)› and ?tick-S = ‹range tick ∪ ev ‘ S›
  assume same-div : ‹𝒟 (P ⟦S⟧ Q \ A) = 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
  fix s X assume ‹(s, X) ∈ ℱ (P ⟦S⟧ Q \ A)›
  then consider ‹s ∈ 𝒟 (P ⟦S⟧ Q \ A)›
    | t where ‹s = trace-hide t (ev ‘ A)› ‹(t, X ∪ ev ‘ A) ∈ ℱ (P ⟦S⟧ Q)›
    unfolding F-Hiding D-Hiding by blast
  thus ‹(s, X) ∈ ℱ ((P \ A) ⟦S⟧ (Q \ A))›
  proof cases
    from same-div D-F show ‹s ∈ 𝒟 (P ⟦S⟧ Q \ A) ⟹ (s, X) ∈ ℱ ((P \ A) ⟦S⟧
(Q \ A))› by blast
  next
    fix t assume * : ‹s = ?trH-A t› ‹(t, X ∪ ev ‘ A) ∈ ℱ (P ⟦S⟧ Q)›
    then consider ‹t ∈ 𝒟 (P ⟦S⟧ Q)›
      | (F) t-P X-P t-Q X-Q
      where ‹(t-P, X-P) ∈ ℱ P› ‹(t-Q, X-Q) ∈ ℱ Q›
        ‹t setinterleaves ((t-P, t-Q), ?tick-S)›
```

```
    ‹X ∪ ev ‘ A = (X-P ∪ X-Q) ∩ ?tick-S ∪ X-P ∩ X-Q›
  by (auto simp add: F-Sync D-Sync)
 thus ‹(s, X) ∈ ℱ ((P \ A) ⟦S⟧ (Q \ A))›
 proof cases
  assume ‹t ∈ 𝒟 (P ⟦S⟧ Q)›
  with *(1) mem-D-imp-mem-D-Hiding have ‹s ∈ 𝒟 (P ⟦S⟧ Q \ A)› by blast
  with same-div D-F show ‹(s, X) ∈ ℱ ((P \ A) ⟦S⟧ (Q \ A))› by blast
 next
  case F
  from inf-hidden[OF - F(1)[THEN F-T], of A] inf-hidden[OF - F(2)[THEN
F-T], of A]
  consider (D-P) f-P where ‹isInfHiddenRun f-P P A› ‹t-P ∈ range f-P›
   | (D-Q) f-Q where ‹isInfHiddenRun f-Q Q A› ‹t-Q ∈ range f-Q›
   | (F-both) t-P' t-Q'
   where ‹?trH-A t-P' = ?trH-A t-P› ‹(t-P', ev ‘ A) ∈ ℱ P›
    ‹?trH-A t-Q' = ?trH-A t-Q› ‹(t-Q', ev ‘ A) ∈ ℱ Q›
   by blast
  thus ‹(s, X) ∈ ℱ ((P \ A) ⟦S⟧ (Q \ A))›
  proof cases
   case D-P
   have $ : ‹?trH-A t-P ∈ 𝒟 (P \ A)›
    apply (unfold D-Hiding, clarify)
    apply (rule exI[of - ‹t-P›], rule exI[of - ‹[]›])
    using tickFree-isInfHiddenRun D-P front-tickFree-Nil by blast
   from F(2) F-T mem-T-imp-mem-T-Hiding
   have $$ : ‹?trH-A t-Q ∈ 𝒯 (Q \ A)› by blast
   show ‹(s, X) ∈ ℱ ((P \ A) ⟦S⟧ (Q \ A))›
    apply (simp add: F-Sync)
    apply (rule disjI2)
    apply (rule exI[of - ‹?trH-A t-P›])
    apply (rule exI[of - ‹?trH-A t-Q›])
    apply (rule exI[of - ‹?trH-A t›], rule exI[of - ‹[]›])
    by (simp add: *(1) F(3) Hiding-interleave $ $$)
  next
   case D-Q
   from F(1) F-T mem-T-imp-mem-T-Hiding
   have $ : ‹?trH-A t-P∈ 𝒯 (P \ A)› by blast
   have $$ : ‹?trH-A t-Q ∈ 𝒟 (Q \ A)›
    apply (unfold D-Hiding, clarify)
    apply (rule exI[of - ‹t-Q›], rule exI[of - ‹[]›])
    using tickFree-isInfHiddenRun D-Q front-tickFree-Nil by blast
   show ‹(s, X) ∈ ℱ ((P \ A) ⟦S⟧ (Q \ A))›
    apply (simp add: F-Sync)
    apply (rule disjI2)
    apply (rule exI[of - ‹?trH-A t-Q›])
    apply (rule exI[of - ‹?trH-A t-P›])
    apply (rule exI[of - ‹?trH-A t›], rule exI[of - ‹[]›])
    by (simp add: *(1) F(3) Hiding-interleave setinterleaving-sym $ $$)
  next
```

```
      case F-both
       from F(4) ‹A ∩ S = {}› have ‹ev ‘ A ⊆ X-P› and ‹ev ‘ A ⊆ X-Q› by
auto
      have ‹(?trH-A t-P, X-P) ∈ ℱ (P \ A)›
      by (simp add: F-Hiding) (metis F(1) F-both(1) ‹ev ‘ A ⊆ X-P› sup.orderE)
      moreover have ‹(?trH-A t-Q, X-Q) ∈ ℱ (Q \ A)›
      by (simp add: F-Hiding) (metis F(2) F-both(3) ‹ev ‘ A ⊆ X-Q› sup.orderE)
      moreover from F(3) Hiding-interleave
      have ‹s setinterleaves ((?trH-A t-P, ?trH-A t-Q), ?tick-S)›
        unfolding ∗(1) by blast
       ultimately have ‹(s, (X-P ∪ X-Q) ∩ ?tick-S ∪ X-P ∩ X-Q) ∈ ℱ ((P \
A) ⟦S⟧ (Q \ A))›
        unfolding F-Sync by blast
      moreover from F(4) have ‹X ⊆ (X-P ∪ X-Q) ∩ ?tick-S ∪ X-P ∩ X-Q›
by blast
     ultimately show ‹(s, X) ∈ ℱ ((P \ A) ⟦S⟧ (Q \ A))› by (fact is-processT4)
    qed
   qed
  qed
next
  let ?trH-A = ‹λt. trace-hide t (ev ‘ A)› and ?tick-S = ‹range tick ∪ ev ‘ S›
  fix s assume ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
  then obtain t-P t-Q r v
    where ∗ : ‹ftF v› ‹tF r ∨ v = []› ‹s = r @ v› ‹r setinterleaves ((t-P, t-Q),
?tick-S)›
     ‹t-P ∈ 𝒟 (P \ A) ∧ t-Q ∈ 𝒯 (Q \ A) ∨ t-P ∈ 𝒟 (Q \ A) ∧ t-Q ∈ 𝒯 (P \
A)›
    unfolding D-Sync by blast
  from ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))› have ‹ftF s›
   by (simp add: D-imp-front-tickFree)
  { fix t-P t-Q and P Q
   assume ∗∗ : ‹r setinterleaves ((t-P, t-Q), ?tick-S)›
     ‹t-P ∈ 𝒟 (P \ A)› ‹t-Q ∈ 𝒯 (Q \ A)›
   from ∗∗(2) obtain t u
     where ∗∗∗ : ‹ftF u› ‹tF t› ‹t-P = ?trH-A t @ u›
       ‹t ∈ 𝒟 P ∨ (∃x. isInfHidden-seqRun-strong x P A t)›
     by (elim D-Hiding-seqRunE)
   from interleave-append-left[OF ∗∗(1)[unfolded ∗∗∗(3)]]
   obtain t-Q1 t-Q2 r1 r2
     where ∗∗∗∗ : ‹t-Q = t-Q1 @ t-Q2› ‹r = r1 @ r2›
       ‹r1 setinterleaves ((?trH-A t, t-Q1), ?tick-S)›
       ‹r2 setinterleaves ((u, t-Q2), ?tick-S)› by blast
   from ∗∗(3) consider t-Q' where ‹t-Q = ?trH-A t-Q'› ‹(t-Q', ev ‘ A) ∈ ℱ Q›
     | (D-Q) t' u' where ‹ftF u'› ‹tF t'› ‹t-Q = ?trH-A t' @ u'›
       ‹t' ∈ 𝒟 Q ∨ (∃y. isInfHidden-seqRun-strong y Q A t')›
     by (elim T-Hiding-seqRunE)
   hence ‹s ∈ 𝒟 (P ⟦S⟧ Q \ A)›
   proof cases
     fix t-Q' assume ‹t-Q = ?trH-A t-Q'› ‹(t-Q', ev ‘ A) ∈ ℱ Q›
```

```
    from trace-hide-append[OF this(1)[unfolded ****(1)]]
   obtain t-Q1' t-Q2' where $ : ‹t-Q' = t-Q1' @ t-Q2'› ‹t-Q1 = ?trH-A t-Q1'›
     ‹t-Q2 = ?trH-A t-Q2'› by blast
    from ‹A ∩ S = {}› have ‹ev ‘ A ∩ ?tick-S = {}› by blast
    from interleave-Hiding[OF this ****(3)[unfolded $(2)]]
    obtain r1' where $$ : ‹r1 = ?trH-A r1'›
     ‹r1' setinterleaves ((t, t-Q1'), ?tick-S)› by blast

    show ‹s ∈ 𝒟 (P ⟦S⟧ Q \ A)›
    proof (rule D-Hiding-seqRunI)
     show ‹ftF (r2 @ v)›
      by (metis *(1, 3) ****(2) ‹ftF s›
          front-tickFree-append-iff tickFree-append-iff)
    next
     show ‹tF r1'›
      by (metis $(1) $$(2) ***(2) F-imp-front-tickFree SyncWithTick-imp-NTF
       ‹(t-Q', ev ‘ A) ∈ ℱ Q› front-tickFree-dw-closed nonTickFree-n-frontTickFree
        non-tickFree-tick tickFree-append-iff ftf-Sync tickFree-imp-front-tickFree)
    next
     from $$(1) *(3) ****(2) append.assoc
     show ‹s = ?trH-A r1' @ r2 @ v› by blast
    next
     from ***(4) show ‹r1' ∈ 𝒟 (P ⟦S⟧ Q) ∨
                        (∃ x. ∀ i. seqRun r1' x i ∈ 𝒯 (P ⟦S⟧ Q) ∧ x i ∈ ev ‘ A)›
     proof (elim disjE exE)
      assume ‹t ∈ 𝒟 P›
      moreover have ‹t-Q1' ∈ 𝒯 Q›
       by (metis $(1) F-T prefixI ‹(t-Q', ev ‘ A) ∈ ℱ Q› is-processT3-TR)
      ultimately have ‹r1' ∈ 𝒟 (P ⟦S⟧ Q)›
       unfolding D-Sync using $$(2) front-tickFree-Nil by blast
      thus ‹r1' ∈ 𝒟 (P ⟦S⟧ Q) ∨ (∃ x. isInfHidden-seqRun x (P ⟦S⟧ Q) A r1')›
..
     next
      fix x assume £ : ‹isInfHidden-seqRun-strong x P A t›
      from £ ‹A ∩ S = {}› have ‹set (map x [0..<i]) ∩ ?tick-S = {}› for i
       by (simp add: disjoint-iff image-iff)
        (metis event_ptick.distinct(1) event_ptick.inject(1))
      from $$(2)[THEN interleave-append-tail-left, OF this, folded seqRun-def]
      have ££ : ‹seqRun r1' x i setinterleaves ((seqRun t x i, t-Q1'), ?tick-S)›
for i .
      have ‹isInfHidden-seqRun x (P ⟦S⟧ Q) A r1'›
      proof (intro allI conjI)
       have ‹t-Q1' ∈ 𝒯 Q›
        by (metis $(1) F-T prefixI ‹(t-Q', ev ‘ A) ∈ ℱ Q› is-processT3-TR)
       with £ ££ show ‹seqRun r1' x i ∈ 𝒯 (P ⟦S⟧ Q)› for i
        unfolding T-Sync by blast
      next
       show ‹x i ∈ ev ‘ A› for i by (simp add: £)
      qed
```

```
        thus ‹r1' ∈ 𝒟 (P ⟦S⟧ Q) ∨ (∃x. isInfHidden-seqRun x (P ⟦S⟧ Q) A r1')›
by blast
       qed
      qed
    next
      case D-Q
      have ‹t ∈ 𝒯 P› using ***(4) D-T by (metis seqRun-0)
      consider ‹?trH-A t' ≤ t-Q1› | ‹t-Q1 ≤ ?trH-A t'›
        by (metis ****(1) D-Q(3) prefixI dual-order.eq-iff le-same-imp-eq-or-less
nless-le)
      thus ‹s ∈ 𝒟 (P ⟦S⟧ Q \ A)›
      proof cases
        assume ‹?trH-A t' ≤ t-Q1›
        from interleave-le-right[OF ****(3) this]
        obtain r1-bis t-hidden-bis
          where ‹r1-bis ≤ r1› ‹t-hidden-bis ≤ ?trH-A t›
            ‹r1-bis setinterleaves ((t-hidden-bis, ?trH-A t'), ?tick-S)› by blast
        moreover from le-trace-hide[OF ‹t-hidden-bis ≤ ?trH-A t›]
        obtain t-bis where ‹t-hidden-bis = ?trH-A t-bis ∧ t-bis ≤ t› ..
        ultimately have $ : ‹r1-bis ≤ r1› ‹t-bis ≤ t›
          ‹r1-bis setinterleaves ((?trH-A t-bis, ?trH-A t'), ?tick-S)› by blast+
        from interleave-Hiding[OF - $(3)] ‹A ∩ S = {}›
        obtain r1-bis-unhidden
          where $$ : ‹r1-bis = ?trH-A r1-bis-unhidden›
            ‹r1-bis-unhidden setinterleaves ((t-bis, t'), ?tick-S)› by blast
        from $(2) ‹t ∈ 𝒯 P› is-processT3-TR have ‹t-bis ∈ 𝒯 P› by blast
        from D-Q(4) have ‹r1-bis ∈ 𝒟 (P ⟦S⟧ Q \ A)›
        proof (elim disjE exE)
          assume ‹t' ∈ 𝒟 Q›
          with $$(2) ‹t-bis ∈ 𝒯 P› have ‹r1-bis-unhidden ∈ 𝒟 (P ⟦S⟧ Q)›
            by (simp add: D-Sync)
              (use front-tickFree-Nil setinterleaving-sym in blast)
        with $$(1) mem-D-imp-mem-D-Hiding show ‹r1-bis ∈ 𝒟 (P ⟦S⟧ Q \ A)›
by blast
        next
          fix y assume £ : ‹isInfHidden-seqRun-strong y Q A t'›

          from £ ‹A ∩ S = {}› have ‹set (map y [0..<i]) ∩ ?tick-S = {}› for i
            by (simp add: disjoint-iff image-iff)
              (metis event_ptick.distinct(1) event_ptick.inject(1))
         from $$(2)[THEN interleave-append-tail-right, OF this, folded seqRun-def]
          have $$$ : ‹seqRun r1-bis-unhidden y i setinterleaves ((t-bis, seqRun t' y
i), ?tick-S)› for i .
          have $$$$ : ‹isInfHidden-seqRun y (P ⟦S⟧ Q) A r1-bis-unhidden›
          proof (intro allI conjI)
            from $$$ £ ‹t-bis ∈ 𝒯 P›
            show ‹seqRun r1-bis-unhidden y i ∈ 𝒯 (P ⟦S⟧ Q)› for i
              unfolding T-Sync by blast
          next
```

```
        show ‹y i ∈ ev ‘ A› for i by (simp add: £)
      qed
      show ‹r1-bis ∈ 𝒟 (P ⟦S⟧ Q \ A)›
      proof (rule D-Hiding-seqRunI)
        show ‹ftF []› by simp
      next
        show ‹tF r1-bis-unhidden›
          by (metis $$(2) D-Q(2) SyncWithTick-imp-NTF T-imp-front-tickFree
‹t-bis ∈ 𝒯 P›
              ftf-Sync nonTickFree-n-frontTickFree non-tickFree-tick
              tickFree-append-iff tickFree-imp-front-tickFree)
      next
        show ‹r1-bis = ?trH-A r1-bis-unhidden @ []› by (simp add: $$(1))
      next
        from $$$$ show ‹r1-bis-unhidden ∈ 𝒟 (P ⟦S⟧ Q) ∨
                        (∃x. isInfHidden-seqRun x (P ⟦S⟧ Q) A r1-bis-unhidden)›
by blast
      qed
    qed
    with $(1) show ‹s ∈ 𝒟 (P ⟦S⟧ Q \ A)›
      unfolding less-eq-list-def prefix-def
      by (metis (no-types, opaque-lifting) *(3) ****(2) ‹ftF s›
      append-Nil2 front-tickFree-append-iff front-tickFree-dw-closed is-processT7)
  next
    assume ‹t-Q1 ≤ ?trH-A t'›
    from le-trace-hide[OF this] obtain t-Q1-unhidden
      where $ : ‹t-Q1 = ?trH-A t-Q1-unhidden› ‹t-Q1-unhidden ≤ t'› by blast
    from ‹A ∩ S = {}› have ‹ev ‘ A ∩ ?tick-S = {}› by blast
    from ****(3)[unfolded $(1)]
    have ‹r1 setinterleaves ((?trH-A t, ?trH-A t-Q1-unhidden), ?tick-S)› .
    from interleave-Hiding[OF ‹ev ‘ A ∩ ?tick-S = {}› this]
    obtain r1-unhidden
      where $$ : ‹r1 = ?trH-A r1-unhidden›
        ‹r1-unhidden setinterleaves ((t, t-Q1-unhidden), ?tick-S)› by blast
    from ‹t-Q1-unhidden ≤ t'› have ‹t-Q1-unhidden ∈ 𝒯 Q›
      by (metis D-Q(4) D-T is-processT3-TR t-le-seqRun)

    from ***(4) have ‹r1 ∈ 𝒟 (P ⟦S⟧ Q \ A)›
    proof (elim disjE exE)
      assume ‹t ∈ 𝒟 P›
      have ‹r1-unhidden ∈ 𝒟 (P ⟦S⟧ Q)›
      proof (unfold D-Sync, clarify, intro exI conjI)
        show ‹ftF []› ‹tF r1-unhidden ∨ [] = []›
          ‹r1-unhidden = r1-unhidden @ []› by simp-all
      next
        show ‹r1-unhidden setinterleaves ((t, t-Q1-unhidden), ?tick-S)› by (fact
$$(2))
      next
        show ‹t ∈ 𝒟 P ∧ t-Q1-unhidden ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ t-Q1-unhidden ∈
```

```
⟨T P⟩
          by (simp add: ⟨t ∈ 𝒟 P⟩ ⟨t-Q1-unhidden ∈ T Q⟩)
        qed
      thus ⟨r1 ∈ 𝒟 (P ⟦S⟧ Q \ A)⟩ unfolding $$(1) by (fact mem-D-imp-mem-D-Hiding)
       next
        fix x assume £ : ⟨isInfHidden-seqRun-strong x P A t⟩
        from £ ⟨A ∩ S = {}⟩ have ⟨set (map x [0..<i]) ∩ ?tick-S = {}⟩ for i
          by (simp add: disjoint-iff image-iff)
            (metis event_ptick.distinct(1) event_ptick.inject(1))
        from $$(2)[THEN interleave-append-tail-left, OF this, folded seqRun-def]
           have ££ : ⟨seqRun r1-unhidden x i setinterleaves ((seqRun t x i,
t-Q1-unhidden), ?tick-S)⟩ for i .
        have £££ : ⟨isInfHidden-seqRun x (P ⟦S⟧ Q) A r1-unhidden⟩
        proof (intro allI conjI)
          from £ ££ ⟨t-Q1-unhidden ∈ T Q⟩
          show ⟨seqRun r1-unhidden x i ∈ T (P ⟦S⟧ Q)⟩ for i unfolding T-Sync
by blast
        next
          from £ show ⟨x i ∈ ev ‘ A⟩ for i by blast
        qed
        show ⟨r1 ∈ 𝒟 (P ⟦S⟧ Q \ A)⟩
        proof (unfold D-Hiding-seqRun, clarify, intro exI conjI)
          show ⟨ftF []⟩ by simp
        next
         from isInfHidden-seqRun-imp-tickFree[OF £££] show ⟨tF r1-unhidden⟩
.
        next
          show ⟨r1 = ?trH-A r1-unhidden @ []⟩ by (simp add: $$(1))
        next
          from £££ show ⟨r1-unhidden ∈ 𝒟 (P ⟦S⟧ Q) ∨
                        (∃x. isInfHidden-seqRun x (P ⟦S⟧ Q) A r1-unhidden)⟩ by
blast
        qed
       qed
       thus ⟨s ∈ 𝒟 (P ⟦S⟧ Q \ A)⟩
         by (metis ∗(3) ∗∗∗∗(2) ⟨ftF s⟩ append.right-neutral
            front-tickFree-append-iff front-tickFree-dw-closed is-processT7)
     qed
   qed
  } note $ = this
  from ∗(5) show ⟨s ∈ 𝒟 (P ⟦S⟧ Q \ A)⟩
    by (elim disjE conjE) (metis $ ∗(4), metis $ ∗(4) Sync-commute)
next
  let ?trH-A = ⟨λt. trace-hide t (ev ‘ A)⟩ and ?tick-S = ⟨range tick ∪ ev ‘ S⟩
  assume same-div : ⟨𝒟 (P ⟦S⟧ Q \ A) = 𝒟 ((P \ A) ⟦S⟧ (Q \ A))⟩
  fix s X assume ⟨(s, X) ∈ ℱ ((P \ A) ⟦S⟧ (Q \ A))⟩
  then consider ⟨s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))⟩
    | (F) s-P X-P s-Q X-Q
    where ⟨(s-P, X-P) ∈ ℱ (P \ A)⟩ ⟨(s-Q, X-Q) ∈ ℱ (Q \ A)⟩
```

```
    ‹s setinterleaves ((s-P, s-Q), ?tick-S)›
    ‹X = (X-P ∪ X-Q) ∩ ?tick-S ∪ X-P ∩ X-Q›
   unfolding F-Sync D-Sync by blast
 thus ‹(s, X) ∈ ℱ (P ⟦S⟧ Q \ A)›
 proof cases
   from same-div D-F show ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A)) ⟹ (s, X) ∈ ℱ (P
⟦S⟧ Q \ A)› by blast
 next
   case F
   from F(1, 2) consider ‹s-P ∈ 𝒟 (P \ A)› | ‹s-Q ∈ 𝒟 (Q \ A)›
     | (F-both) s-P' s-Q'
     where ‹s-P = ?trH-A s-P'› ‹(s-P', X-P ∪ ev ‘ A) ∈ ℱ P›
       ‹s-Q = ?trH-A s-Q'› ‹(s-Q', X-Q ∪ ev ‘ A) ∈ ℱ Q›
     unfolding F-Hiding D-Hiding by blast
   thus ‹(s, X) ∈ ℱ (P ⟦S⟧ Q \ A)›
   proof cases
     assume ‹s-P ∈ 𝒟 (P \ A)›
     moreover from F(2) F-T have ‹s-Q ∈ 𝒯 (Q \ A)› by blast
     ultimately have ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
       using F(3) front-tickFree-Nil unfolding D-Sync by blast
     with same-div D-F show ‹(s, X) ∈ ℱ (P ⟦S⟧ Q \ A)› by blast
   next
     from F(1) F-T have ‹s-P ∈ 𝒯 (P \ A)› by blast
     moreover assume ‹s-Q ∈ 𝒟 (Q \ A)›
     moreover have ‹s setinterleaves ((s-Q, s-P), range tick ∪ ev ‘ S)›
       by (simp add: F(3) setinterleaving-sym)
     ultimately have ‹s ∈ 𝒟 ((P \ A) ⟦S⟧ (Q \ A))›
       using front-tickFree-Nil unfolding D-Sync by blast
     with same-div D-F show ‹(s, X) ∈ ℱ (P ⟦S⟧ Q \ A)› by blast
   next
     case F-both
     from ‹A ∩ S = {}› have ‹ev ‘ A ∩ (range tick ∪ ev ‘ S) = {}› by blast
     from interleave-Hiding[OF this F(3)[unfolded F-both(1, 3)]]
     obtain s' where * : ‹s = ?trH-A s'›
       ‹s' setinterleaves ((s-P', s-Q'), range tick ∪ ev ‘ S)› by blast
     have ‹X ∪ ev ‘ A =   (X-P ∪ ev ‘ A ∪ (X-Q ∪ ev ‘ A)) ∩ (range tick ∪ ev ‘
S)
                        ∪ (X-P ∪ ev ‘ A) ∩ (X-Q ∪ ev ‘ A)›
       by (simp add: F(4) Un-Int-distrib2 Un-assoc sup-left-commute)
     thus ‹(s, X) ∈ ℱ (P ⟦S⟧ Q \ A)›
       by (simp add: *(1) F-Hiding F-Sync) (use *(2) F-both(2, 4) in blast)
   qed
 qed
qed
```

12.6.4 Renaming Operator Laws

lemma *Renaming-Ndet*: ‹*Renaming* (*P* ⊓ *Q*) *f g* = *Renaming P f g* ⊓ *Renaming Q f g*›

by (*subst Process-eq-spec*) (*auto simp add*: *F-Renaming D-Renaming F-Ndet D-Ndet*)

lemma *Renaming-Det*: ‹*Renaming* (*P* □ *Q*) *f g* = *Renaming P f g* □ *Renaming Q f g*›
proof (*subst Process-eq-spec*, *safe*)
show ‹(*s*, *X*) ∈ ℱ (*Renaming* (*P* □ *Q*) *f g*) ⟹
(*s*, *X*) ∈ ℱ (*Renaming P f g* □ *Renaming Q f g*)› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Renaming F-Det D-Renaming D-Det T-Renaming*)+
next
fix *s X* **assume** ‹(*s*, *X*) ∈ ℱ (*Renaming P f g* □ *Renaming Q f g*)›
then consider ‹*s* = []› ‹(*s*, *X*) ∈ ℱ (*Renaming P f g*)› ‹(*s*, *X*) ∈ ℱ (*Renaming Q f g*)›
| *e s'* **where** ‹*s* = *e* # *s'*› ‹(*s*, *X*) ∈ ℱ (*Renaming P f g*) ∨ (*s*, *X*) ∈ ℱ (*Renaming Q f g*)›
| ‹*s* = []› ‹*s* ∈ 𝒟 (*Renaming P f g*) ∨ *s* ∈ 𝒟 (*Renaming Q f g*)›
| *r* **where** ‹*s* = []› ‹✓(*r*) ∉ *X*› ‹([✓(*r*)] ∈ 𝒯 (*Renaming P f g*) ∨ [✓(*r*)] ∈ 𝒯 (*Renaming Q f g*))›
by (*cases s*) (*auto simp add*: *F-Det*)
thus ‹(*s*, *X*) ∈ ℱ (*Renaming* (*P* □ *Q*) *f g*)›
proof *cases*
show ‹⟦*s* = []; (*s*, *X*) ∈ ℱ (*Renaming P f g*); (*s*, *X*) ∈ ℱ (*Renaming Q f g*)⟧
⟹ (*s*, *X*) ∈ ℱ (*Renaming* (*P* □ *Q*) *f g*)›
by (*auto simp add*: *F-Renaming F-Det*)
next
show ‹*s* = *e* # *s'* ⟹ (*s*, *X*) ∈ ℱ (*Renaming P f g*) ∨ (*s*, *X*) ∈ ℱ (*Renaming Q f g*)
⟹ (*s*, *X*) ∈ ℱ (*Renaming* (*P* □ *Q*) *f g*)› **for** *s' e*
by (*simp add*: *F-Renaming F-Det D-Det*, *safe*)
(*metis list.distinct*(*1*) *list.simps*(*9*), *presburger*)+
next
show ‹*s* = [] ⟹ *s* ∈ 𝒟 (*Renaming P f g*) ∨ *s* ∈ 𝒟 (*Renaming Q f g*)
⟹ (*s*, *X*) ∈ ℱ (*Renaming* (*P* □ *Q*) *f g*)›
by (*simp add*: *D-Renaming F-Renaming D-Det*)
next
show ‹⟦*s* = []; ✓(*r*) ∉ *X*; [✓(*r*)] ∈ 𝒯 (*Renaming P f g*) ∨ [✓(*r*)] ∈ 𝒯 (*Renaming Q f g*)⟧
⟹ (*s*, *X*) ∈ ℱ (*Renaming* (*P* □ *Q*) *f g*)› **for** *r*
by (*auto simp add*: *T-Renaming F-Renaming D-Det F-Det*
dest: *map-event*$_{ptick}$*-eq-tick-iff*[*THEN iffD1*, *OF sym*, *of r*])
(*metis append-eq-Cons-conv list.map-disc-iff map-event*$_{ptick}$*-tickFree*
non-tickFree-tick tickFree-Nil tickFree-append-iff)+
qed
next
show ‹*s* ∈ 𝒟 (*Renaming* (*P* □ *Q*) *f g*) ⟹ *s* ∈ 𝒟 (*Renaming P f g* □ *Renaming Q f g*)›
and ‹*s* ∈ 𝒟 (*Renaming P f g* □ *Renaming Q f g*) ⟹ *s* ∈ 𝒟 (*Renaming* (*P* □ *Q*) *f g*)› **for** *s*

by (*auto simp add*: *D-Renaming D-Det*)
qed

lemma *Sliding-STOP-Det* [*simp*] : ‹$(P \rhd STOP) \Box Q = P \rhd Q$›
by (*simp add*: *Det-commute Det-distrib-Ndet-left Sliding-def*)

lemma *Sliding-Det*: ‹$(P \rhd P') \Box Q = P \rhd P' \Box Q$›
by (*metis Det-assoc Sliding-STOP-Det*)

lemma *Renaming-Sliding*:
‹*Renaming* $(P \rhd Q)$ *f g* = *Renaming P f g* $\rhd$ *Renaming Q f g*›
by (*simp add*: *Renaming-Det Renaming-Ndet Sliding-def*)

lemma *Renaming-Mprefix-image*:
‹*Renaming* $(\Box\ a \in A \to P\ (f\ a))$ *f g* =
$\Box\ b \in f\ `\ A \to$ *Renaming* $(P\ b)$ *f g*› (**is** ‹*?lhs* = *?rhs*›)
by (*subst Renaming-Mprefix*, *rule mono-Mprefix-eq*)
(*simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet F-Renaming D-Renaming*, *safe*, *auto*)

lemma *Renaming-Mprefix-image-inj-on*:
‹*Renaming* (*Mprefix A P*) *f g* = $\Box\ b \in f\ `\ A \to$ *Renaming* $(P\ (THE\ a.\ a \in A \wedge f\ a = b))$ *f g*›
if *inj-on-f*: ‹*inj-on f A*›
apply (*subst Renaming-Mprefix-image*[*symmetric*])
apply (*intro arg-cong*[**where** *f* = ‹$\lambda Q.$ *Renaming Q f g*›] *mono-Mprefix-eq*)
by (*metis that the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-Mprefix-image-inj*:
‹*Renaming* (*Mprefix A P*) *f g* = $\Box\ b \in f\ `\ A \to$ *Renaming* $(P\ (THE\ a.\ f\ a = b))$ *f g*› **if** *inj-f*: ‹*inj f*›
apply (*subst Renaming-Mprefix-image-inj-on*, *metis inj-eq inj-onI that*)
apply (*rule mono-Mprefix-eq*[*rule-format*])
by (*metis imageE inj-eq*[*OF inj-f*])

lemma *Renaming-Mndetprefix-image*: ‹*Renaming* $(\sqcap\ a \in A \to P\ (f\ a))$ *f g* = $\sqcap\ b \in f\ `\ A \to$ *Renaming* $(P\ b)$ *f g*›
by (*auto simp add*: *Mndetprefix-GlobalNdet Renaming-distrib-GlobalNdet write0-def Renaming-Mprefix*
intro!: *mono-Mprefix-eq mono-GlobalNdet-eq2 intro*: *sym*)

corollary *Renaming-Mndetprefix-inj-on*:
‹*Renaming* (*Mndetprefix A P*) *f g* = $\sqcap\ b \in f\ `\ A \to$ *Renaming* $(P\ (THE\ a.\ a \in$

$A \wedge f\ a = b))\ f\ g\rangle$
if *inj-on-f*: ‹*inj-on f A*›
apply (*subst Renaming-Mndetprefix-image*[*symmetric*])
apply (*intro arg-cong*[**where** $f = \langle\lambda Q.\ Renaming\ Q\ f\ g\rangle$] *mono-Mndetprefix-eq*)
by (*metis that the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-Mndetprefix-inj*:
‹*Renaming* (*Mndetprefix A P*) *f g* = ⊓ *b* ∈ *f* ‘ *A* → *Renaming* (*P* (*THE a. f a* = *b*)) *f g*›
if *inj-f*: ‹*inj f*›
apply (*subst Renaming-Mndetprefix-inj-on*, *metis inj-eq inj-onI that*)
apply (*rule mono-Mndetprefix-eq*[*rule-format*])
by (*metis imageE inj-eq*[*OF inj-f*])

lemma *Hiding-distrib-FD-GlobalNdet* :
‹(⊓*a* ∈ *A*. *P a*) \ *S* ⊑$_{FD}$ ⊓*a* ∈ *A*. (*P a* \ *S*)› (**is** ‹*?lhs* ⊑$_{FD}$ *?rhs*›)
proof (*cases* ‹*A* = {}›)
show ‹*A* = {} ⟹ *?lhs* ⊑$_{FD}$ *?rhs*› **by** *simp*
next
show ‹*?lhs* ⊑$_{FD}$ *?rhs*› **if** ‹*A* ≠ {}›
proof (*rule failure-divergence-refine-optimizedI*)
show ‹*s* ∈ $\mathcal{D}$ *?rhs* ⟹ *s* ∈ $\mathcal{D}$ *?lhs*› **for** *s*
by (*simp add*: *GlobalNdet-projs D-Hiding-seqRun*) *blast*
next
assume *subset-div* : ‹$\mathcal{D}$ *?rhs* ⊆ $\mathcal{D}$ *?lhs*›
fix *s X* **assume** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
then obtain *a* **where** ‹*a* ∈ *A*› ‹(*s*, *X*) ∈ $\mathcal{F}$ (*P a* \ *S*)›
by (*auto simp add*: *F-GlobalNdet* ‹*A* ≠ {}›)
from ‹(*s*, *X*) ∈ $\mathcal{F}$ (*P a* \ *S*)› **consider** ‹*s* ∈ $\mathcal{D}$ (*P a* \ *S*)›
| *t* **where** ‹*s* = *trace-hide t* (*ev* ‘ *S*)› ‹(*t*, *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ (*P a*)›
unfolding *D-Hiding F-Hiding* **by** *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*s* ∈ $\mathcal{D}$ (*P a* \ *S*)›
with ‹*a* ∈ *A*› **have** ‹*s* ∈ $\mathcal{D}$ *?rhs*› **by** (*auto simp add*: *D-GlobalNdet*)
with *subset-div D-F* **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
from ‹*a* ∈ *A*› **show** ‹*s* = *trace-hide t* (*ev* ‘ *S*) ⟹ (*t*, *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ (*P a*)
⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **for** *t*
by (*simp add*: *F-Hiding-seqRun F-GlobalNdet*) *blast*
qed
qed
qed

lemma *Renaming-Seq* :

‹*Renaming* (*P* ; *Q*) *f g* = *Renaming P f g* ; *Renaming Q f g*› (**is** ‹*?lhs* = *?rhs*›)
proof (*rule Process-eq-optimizedI*)
fix *t* **assume** ‹*t* ∈ $\mathcal{D}$ *?lhs*›
then obtain *u v* **where** ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u* @ *v*› ‹*tF u*› ‹*ftF v*› ‹*u* ∈ $\mathcal{D}$ (*P* ; *Q*)›
unfolding *D-Renaming* **by** *blast*
from ‹*u* ∈ $\mathcal{D}$ (*P* ; *Q*)› **consider** ‹*u* ∈ $\mathcal{D}$ *P*›
| *u1 u2 r* **where** ‹*u* = *u1* @ *u2*› ‹*u1* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*› ‹*u2* ∈ $\mathcal{D}$ *Q*›
unfolding *D-Seq* **by** *blast*
thus ‹*t* ∈ $\mathcal{D}$ *?rhs*›
proof *cases*
from ‹*ftF v*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u* @ *v*› ‹*tF u*›
show ‹*u* ∈ $\mathcal{D}$ *P* ⟹ *t* ∈ $\mathcal{D}$ *?rhs*› **by** (*auto simp add*: *D-Seq D-Renaming*)
next
fix *u1 u2 r* **assume** ‹*u* = *u1* @ *u2*› ‹*u1* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*› ‹*u2* ∈ $\mathcal{D}$ *Q*›
from ‹*u1* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*› **have** ‹*map* (*map-event*$_{ptick}$ *f g*) *u1* @ [✓(*g r*)] ∈ $\mathcal{T}$ (*Renaming P f g*)›
by (*auto simp add*: *T-Renaming*)
moreover from ‹*u2* ∈ $\mathcal{D}$ *Q*› ‹*tF u*› ‹*u* = *u1* @ *u2*›
have ‹*map* (*map-event*$_{ptick}$ *f g*) *u2* ∈ $\mathcal{D}$ (*Renaming Q f g*)›
by (*simp add*: *D-Renaming*) (*use front-tickFree-Nil* **in** *blast*)
ultimately have ‹*map* (*map-event*$_{ptick}$ *f g*) *u* ∈ $\mathcal{D}$ *?rhs*› **by** (*auto simp add*: ‹*u* = *u1* @ *u2*› *D-Seq*)
thus ‹*t* ∈ $\mathcal{D}$ *?rhs*› **by** (*simp add*: ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u* @ *v*› ‹*ftF v*› ‹*tF u*› *is-processT7 map-event*$_{ptick}$*-tickFree*)
qed
next
fix *t* **assume** ‹*t* ∈ $\mathcal{D}$ *?rhs*›
thm *this*[*simplified D-Seq*, *simplified*]
then consider ‹*t* ∈ $\mathcal{D}$ (*Renaming P f g*)›
| *t1 t2 s* **where** ‹*t* = *t1* @ *t2*› ‹*t1* @ [✓(*s*)] ∈ $\mathcal{T}$ (*Renaming P f g*)›
‹*t1* @ [✓(*s*)] ∉ $\mathcal{D}$ (*Renaming P f g*)› ‹*t2* ∈ $\mathcal{D}$ (*Renaming Q f g*)›
by (*simp add*: *D-Seq*) (*metis D-imp-front-tickFree append-T-imp-tickFree is-processT7 is-processT9 list.distinct*(*1*))
thus ‹*t* ∈ $\mathcal{D}$ *?lhs*›
proof *cases*
show ‹*t* ∈ $\mathcal{D}$ (*Renaming P f g*) ⟹ *t* ∈ $\mathcal{D}$ *?lhs*›
by (*auto simp add*: *D-Renaming D-Seq*)
next
fix *t1 t2 s* **assume** ‹*t* = *t1* @ *t2*› ‹*t1* @ [✓(*s*)] ∈ $\mathcal{T}$ (*Renaming P f g*)›
‹*t1* @ [✓(*s*)] ∉ $\mathcal{D}$ (*Renaming P f g*)› ‹*t2* ∈ $\mathcal{D}$ (*Renaming Q f g*)›
from *this*(*2*, *3*) **obtain** *u1′* **where** ‹*t1* @ [✓(*s*)] = *map* (*map-event*$_{ptick}$ *f g*) *u1′*› ‹*u1′* ∈ $\mathcal{T}$ *P*›
by (*auto simp add*: *Renaming-projs*)
then obtain *u1 r* **where** ‹*s* = *g r*› ‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1*› ‹*u1* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*›
by (*cases u1′ rule*: *rev-cases*) (*auto simp add*: *tick-eq-map-event*$_{ptick}$*-iff*)
from ‹*t2* ∈ $\mathcal{D}$ (*Renaming Q f g*)› **obtain** *u2 u3*
where ‹*t2* = *map* (*map-event*$_{ptick}$ *f g*) *u2* @ *u3*› ‹*tF u2*› ‹*ftF u3*› ‹*u2* ∈ $\mathcal{D}$

```
Q›
    unfolding D-Renaming by blast
  from ‹u1 @ [✓(r)] ∈ 𝒯 P› ‹u2 ∈ 𝒟 Q› have ‹u1 @ u2 ∈ 𝒟 (P ; Q)›
    by (auto simp add: D-Seq)
  with ‹ftF u3› ‹tF u2› ‹u1 @ [✓(r)] ∈ 𝒯 P› show ‹t ∈ 𝒟 ?lhs›
    by (simp add: ‹t = t1 @ t2› ‹t1 = map (map-event_ptick f g) u1›
        ‹t2 = map (map-event_ptick f g) u2 @ u3› D-Renaming)
      (metis append-T-imp-tickFree append-eq-appendI map-append not-Cons-self
tickFree-append-iff)
 qed
next
 fix t X assume ‹(t, X) ∈ ℱ ?lhs› ‹t ∉ 𝒟 ?lhs›
 from ‹(t, X) ∈ ℱ ?lhs› ‹t ∉ 𝒟 ?lhs› obtain u
   where ‹t = map (map-event_ptick f g) u› ‹(u, map-event_ptick f g -‘ X) ∈ ℱ
(P ; Q)›
   unfolding Renaming-projs by blast
 from ‹(u, map-event_ptick f g -‘ X) ∈ ℱ (P ; Q)› ‹t ∉ 𝒟 ?lhs›
 consider ‹(u, map-event_ptick f g -‘ X ∪ range tick) ∈ ℱ P› ‹tF u›
   | u1 r u2 where ‹u = u1 @ u2› ‹u1 @ [✓(r)] ∈ 𝒯 P› ‹(u2, map-event_ptick f
g -‘ X) ∈ ℱ Q›
   by (auto simp add: ‹t = map (map-event_ptick f g) u› Seq-projs D-Renaming)
   (metis D-imp-front-tickFree butlast-snoc front-tickFree-iff-tickFree-butlast front-tickFree-single
     is-processT8 is-processT9 map-append map-event_ptick-front-tickFree nonTick-
Free-n-frontTickFree)
 thus ‹(t, X) ∈ ℱ ?rhs›
 proof cases
   assume ‹(u, map-event_ptick f g -‘ X ∪ range tick) ∈ ℱ P› ‹tF u›
  have ‹map-event_ptick f g -‘ X ∪ map-event_ptick f g -‘ range tick ⊆ map-event_ptick
f g -‘ X ∪ range tick›
    by (auto simp add: map-event_ptick-eq-tick-iff)
   with ‹(u, map-event_ptick f g -‘ X ∪ range tick) ∈ ℱ P›
   have ‹(u, map-event_ptick f g -‘ X ∪ map-event_ptick f g -‘ range tick) ∈ ℱ P›
    by (meson is-processT4)
   with ‹tF u› show ‹(t, X) ∈ ℱ ?rhs›
     by (auto simp add: F-Seq ‹t = map (map-event_ptick f g) u› F-Renaming
map-event_ptick-tickFree)
 next
   fix u1 u2 r assume ‹u = u1 @ u2› ‹u1 @ [✓(r)] ∈ 𝒯 P› ‹(u2, map-event_ptick
f g -‘ X) ∈ ℱ Q›
   from ‹u1 @ [✓(r)] ∈ 𝒯 P› have ‹map (map-event_ptick f g) u1 @ [✓(g r)] ∈
𝒯 (Renaming P f g)›
    by (auto simp add: T-Renaming)
   moreover from ‹(u2, map-event_ptick f g -‘ X) ∈ ℱ Q›
   have ‹(map (map-event_ptick f g) u2, X) ∈ ℱ (Renaming Q f g)› by (auto simp
add: F-Renaming)
   ultimately show ‹(t, X) ∈ ℱ ?rhs›
    by (auto simp add: ‹t = map (map-event_ptick f g) u› ‹u = u1 @ u2› F-Seq)
 qed
next
```

fix t X **assume** ‹$(t, X) \in \mathcal{F}$ *?rhs*› ‹$t \notin \mathcal{D}$ *?rhs*› ‹$t \notin \mathcal{D}$ *?lhs*›
then consider ‹$(t, X \cup$ *range tick*$) \in \mathcal{F}$ (*Renaming P f g*)› ‹*tF t*›
| *t1 t2 s* **where** ‹$t =$ *t1* @ *t2*› ‹*t1* @ $[\checkmark(s)] \in \mathcal{T}$ (*Renaming P f g*)› ‹$(t2, X) \in \mathcal{F}$ (*Renaming Q f g*)›
unfolding *Seq-projs* **by** *blast*
thus ‹$(t, X) \in \mathcal{F}$ *?lhs*›
proof *cases*
assume ‹$(t, X \cup$ *range tick*$) \in \mathcal{F}$ (*Renaming P f g*)› ‹*tF t*›
with ‹$t \notin \mathcal{D}$ *?rhs*› **obtain** *u*
where ‹$t =$ *map* (*map-event*$_{ptick}$ *f g*) *u*›
‹$(u,$ *map-event*$_{ptick}$ *f g* $-‘$ $X \cup$ *map-event*$_{ptick}$ *f g* $-‘$ *range tick*$) \in \mathcal{F}$ *P*›
by (*auto simp add*: *D-Seq Renaming-projs*)
from *this*(*2*) **have** ‹$(u,$ *map-event*$_{ptick}$ *f g* $-‘$ $X \cup$ *range tick*$) \in \mathcal{F}$ *P*›
by (*rule is-processT4*) *auto*
with ‹*tF t*› **show** ‹$(t, X) \in \mathcal{F}$ *?lhs*›
by (*auto simp add*: *F-Renaming F-Seq* ‹$t =$ *map* (*map-event*$_{ptick}$ *f g*) *u*› *map-event*$_{ptick}$*-tickFree*)
next
fix *t1 t2 s* **assume** ‹$t =$ *t1* @ *t2*› ‹*t1* @ $[\checkmark(s)] \in \mathcal{T}$ (*Renaming P f g*)› ‹$(t2, X) \in \mathcal{F}$ (*Renaming Q f g*)›
from ‹$t \notin \mathcal{D}$ *?rhs*› **have** ‹*t1* @ $[\checkmark(s)] \notin \mathcal{D}$ (*Renaming P f g*)›
by (*simp add*: ‹$t =$ *t1* @ *t2*› *D-Seq*)
(*metis D-imp-front-tickFree F-imp-front-tickFree* ‹$(t2, X) \in \mathcal{F}$ (*Renaming Q f g*)›
front-tickFree-append-iff is-processT7 is-processT9 not-Cons-self)
with ‹*t1* @ $[\checkmark(s)] \in \mathcal{T}$ (*Renaming P f g*)› **obtain** *u1′*
where ‹*t1* @ $[\checkmark(s)] =$ *map* (*map-event*$_{ptick}$ *f g*) *u1′*› ‹*u1′* $\in \mathcal{T}$ *P*›
unfolding *Renaming-projs* **by** *blast*
then obtain *u1 r* **where** ‹$s = g\ r$› ‹*t1* $=$ *map* (*map-event*$_{ptick}$ *f g*) *u1*› ‹*u1* @ $[\checkmark(r)] \in \mathcal{T}$ *P*›
by (*cases u1′ rule*: *rev-cases*) (*auto simp add*: *tick-eq-map-event*$_{ptick}$*-iff*)
from ‹$t \notin \mathcal{D}$ *?rhs*› ‹*t1* @ $[\checkmark(s)] \in \mathcal{T}$ (*Renaming P f g*)› **have** ‹*t2* $\notin \mathcal{D}$ (*Renaming Q f g*)›
by (*auto simp add*: ‹$t =$ *t1* @ *t2*› *D-Seq*)
with ‹$(t2, X) \in \mathcal{F}$ (*Renaming Q f g*)› **obtain** *u2*
where ‹*t2* $=$ *map* (*map-event*$_{ptick}$ *f g*) *u2*› ‹$(u2,$ *map-event*$_{ptick}$ *f g* $-‘$ $X) \in \mathcal{F}$ *Q*›
unfolding *Renaming-projs* **by** *blast*
from ‹$(u2,$ *map-event*$_{ptick}$ *f g* $-‘$ $X) \in \mathcal{F}$ *Q*› ‹*u1* @ $[\checkmark(r)] \in \mathcal{T}$ *P*›
have ‹$(u1$ @ $u2,$ *map-event*$_{ptick}$ *f g* $-‘$ $X) \in \mathcal{F}$ (P **;** Q)› **by** (*auto simp add*: *F-Seq*)
thus ‹$(t, X) \in \mathcal{F}$ *?lhs*›
by (*auto simp add*: ‹$t =$ *t1* @ *t2*› ‹*t1* $=$ *map* (*map-event*$_{ptick}$ *f g*) *u1*›
‹*t2* $=$ *map* (*map-event*$_{ptick}$ *f g*) *u2*› *F-Renaming*)
qed
qed

lemma *Renaming-fix* :
‹*Renaming* (μ *X*. φ *X*) *f g* = (μ *X*. ((λ*P*. *Renaming P f g*) ∘ φ ∘ (λ*P*. *Renaming P* (*inv f*) (*inv g*))) *X*)›
(**is** ‹*Renaming* (μ *X*. φ *X*) *f g* = (μ *X*. ?φ′ *X*)›) **if** ‹*cont* φ›
and ‹*bij f*› **and** ‹*bij g*› **for** φ :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *process*$_{ptick}$›
proof −
— Some facts first.
have ∗ : ‹φ = (λ*P*. *Renaming P* (*inv f*) (*inv g*)) ∘ ?φ′ ∘ (λ*P*. *Renaming P f g*)›
by (*rule ext*) (*simp add*: *Renaming-inv bij-is-inj* ‹*bij f*› ‹*bij g*›)

have *mono-φ* : ‹*P* ⊑ *Q* ⟹ φ *P* ⊑ φ *Q*› **for** *P Q*
by (*simp add*: *cont2monofunE* ‹*cont* φ›)
have *mono-φ′* : ‹*P* ⊑ *Q* ⟹ ?φ′ *P* ⊑ ?φ′ *Q*› **for** *P Q*
by (*simp add*: *mono-Renaming mono-φ*)

have *finitary-props* : ‹*finitary f*› ‹*finitary g*› ‹*finitary* (*inv f*)› ‹*finitary* (*inv g*)›
by (*simp-all add*: *bij-is-inj bij-is-surj surj-imp-inj-inv* ‹*bij f*› ‹*bij g*›)
have ‹*cont* (λ*X*. *Renaming* (φ *X*) *f g*)› **by** (*simp add*: *finitary-props*(*1*, *2*) ‹*cont* φ›)
moreover have ‹*cont* (λ*X*. *Renaming X* (*inv f*) (*inv g*))› **by** (*simp add*: *finitary-props*(*3*, *4*))
ultimately have ‹*cont* ?φ′› **unfolding** *comp-def* **by** (*rule cont-compose*)

from ‹*cont* φ› ‹*cont* ?φ′› *cont-process-rec*
have ∗∗ : ‹(μ *X*. φ *X*) = φ (μ *X*. φ *X*)› ‹(μ *X*. ?φ′ *X*) = ?φ′ (μ *X*. ?φ′ *X*)› **by** *blast*+

show ‹*Renaming* (μ *X*. φ *X*) *f g* = (μ *X*. ?φ′ *X*)›
proof (*rule below-antisym*)
show ‹*Renaming* (μ *X*. φ *X*) *f g* ⊑ (μ *X*. ?φ′ *X*)›
proof (*induct rule*: *fix-ind*[**where** *F* = ‹Λ *X*. φ *X*›])
show ‹*adm* (λ*a*. *Renaming a f g* ⊑ (μ *X*. ?φ′ *X*))›
by (*simp add*: ‹*finitary f*› ‹*finitary g*› *monofunI*)
next
show ‹*Renaming* ⊥ *f g* ⊑ (μ *X*. ?φ′ *X*)› **by** *simp*
next
fix *X* **assume** ‹*Renaming X f g* ⊑ (μ *X*. ?φ′ *X*)›
hence ‹?φ′ (*Renaming X f g*) ⊑ ?φ′ (μ *X*. ?φ′ *X*)› **by** (*fact mono-φ′*)
hence ‹*Renaming* (?φ′ (*Renaming X f g*)) (*inv f*) (*inv g*) ⊑
Renaming (?φ′ (μ *X*. ?φ′ *X*)) (*inv f*) (*inv g*)› **by** (*fact mono-Renaming*)
also from ∗ **have** ‹*Renaming* (?φ′ (*Renaming X f g*)) (*inv f*) (*inv g*) = φ *X*›
unfolding *comp-def* **by** *metis*
also from ∗∗(*2*) **have** ‹?φ′ (μ *X*. ?φ′ *X*) = (μ *X*. ?φ′ *X*)› **by** *argo*
finally have ‹*Renaming* (φ *X*) *f g* ⊑ *Renaming* (*Renaming* (μ *X*. ?φ′ *X*) (*inv f*) (*inv g*)) *f g*›
by (*fact mono-Renaming*)
also have ‹. . . = (μ *X*. ?φ′ *X*)› **by** (*simp add*: *inv-Renaming* ‹*bij f*› ‹*bij g*›)
finally show ‹*Renaming* ((Λ *X*. φ *X*)·*X*) *f g* ⊑ (μ *X*. ?φ′ *X*)›
by (*subst beta-cfun*[*OF* ‹*cont* φ›]) (*simp add*: *comp-def*)

```
    qed
  next
    show ‹(μ X. ?φ' X) ⊑ Renaming (μ X. φ X) f g›
    proof (induct rule: fix-ind[where F = ‹Λ X. ?φ' X›])
      show ‹adm (λa. a ⊑ Renaming (μ x. φ x) f g)› by (simp add: monofunI)
    next
      show ‹⊥ ⊑ Renaming (μ x. φ x) f g› by simp
    next
      fix X assume hyp : ‹X ⊑ Renaming (μ X. φ X) f g›
     hence ‹Renaming X (inv f) (inv g) ⊑ Renaming (Renaming (μ X. φ X) f g)
(inv f) (inv g)›
       by (fact mono-Renaming)
     also have ‹... = (μ X. φ X)› by (simp add: Renaming-inv bij-is-inj ‹bij f›
‹bij g›)
      finally have ‹φ (Renaming X (inv f) (inv g)) ⊑ φ (μ X. φ X)› by (fact
mono-φ)
     hence ‹Renaming (φ (Renaming X (inv f) (inv g))) f g ⊑
           Renaming (φ (μ x. φ x)) f g› by (fact mono-Renaming)
     also from **(1) have ‹φ (μ x. φ x) = (μ X. φ X)› by presburger
     finally show ‹(Λ X. ?φ' X)·X ⊑ Renaming (μ X. φ X) f g›
       by (subst beta-cfun[OF ‹cont ?φ'›]) (simp add: comp-def)
    qed
  qed
qed
```

Chapter 13

Events and Ticks of a Process

13.1 Definitions

13.1.1 Events of a Process

definition *events-of* :: ‹('a, 'r) $process_{ptick}$ ⇒ 'a *set*› (‹α'(-')›)
— $\alpha(P)$ for "alphabet of P"
where ‹$\alpha(P) \equiv \bigcup t{\in}\mathcal{T}\ P.\ \{a.\ ev\ a \in set\ t\}$›

lemma *events-of-memI* : ‹$t \in \mathcal{T}\ P \Longrightarrow ev\ a \in set\ t \Longrightarrow a \in \alpha(P)$›
unfolding *events-of-def* **by** *blast*

lemma *events-of-memD* : ‹$a \in \alpha(P) \Longrightarrow \exists\, t \in \mathcal{T}\ P.\ ev\ a \in set\ t$›
unfolding *events-of-def* **by** *blast*

lemma *events-of-memE* :
‹$a \in \alpha(P) \Longrightarrow (\bigwedge t.\ t \in \mathcal{T}\ P \Longrightarrow ev\ a \in set\ t \Longrightarrow thesis) \Longrightarrow thesis$›
by (*meson events-of-memD*)

definition *strict-events-of* :: ‹('a, 'r) $process_{ptick}$ ⇒ 'a *set*› (‹$\boldsymbol{\alpha}$'(-')›)
where ‹$\boldsymbol{\alpha}(P) \equiv \bigcup t{\in}\mathcal{T}\ P - \mathcal{D}\ P.\ \{a.\ ev\ a \in set\ t\}$›

lemma *strict-events-of-memI* :
‹$t \in \mathcal{T}\ P \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow ev\ a \in set\ t \Longrightarrow a \in \boldsymbol{\alpha}(P)$›
unfolding *strict-events-of-def* **by** *blast*

lemma *strict-events-of-memD* : ‹$a \in \boldsymbol{\alpha}(P) \Longrightarrow \exists\, t \in \mathcal{T}\ P.\ t \notin \mathcal{D}\ P \wedge ev\ a \in set\ t$›
unfolding *strict-events-of-def* **by** *blast*

lemma *strict-events-of-memE* :
‹$a \in \boldsymbol{\alpha}(P) \Longrightarrow (\bigwedge t.\ t \in \mathcal{T}\ P \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow ev\ a \in set\ t \Longrightarrow thesis) \Longrightarrow$

thesis›
by (*meson strict-events-of-memD*)

lemma *events-of-is-strict-events-of-or-UNIV* :
‹$\alpha(P) = (\mathit{if}\ \mathcal{D}\ P = \{\}\ \mathit{then}\ \boldsymbol{\alpha}(P)\ \mathit{else}\ \mathit{UNIV})$›
proof (*split if-split*, *intro conjI impI*)
show ‹$\mathcal{D}\ P = \{\} \Longrightarrow \alpha(P) = \boldsymbol{\alpha}(P)$› **by** (*simp add*: *events-of-def strict-events-of-def*)
next
assume ‹$\mathcal{D}\ P \neq \{\}$›
with *nonempty-divE* **obtain** t **where** ‹$tF\ t$› ‹$t \in \mathcal{D}\ P$› **by** *blast*
hence ‹$t\ @\ [ev\ a] \in \mathcal{D}\ P$› **for** a **by** (*simp add*: *is-processT7*)
thus ‹$\alpha(P) = \mathit{UNIV}$› **by** (*metis D-T UNIV-eq-I events-of-memI in-set-conv-decomp*)
qed

lemma *strict-events-of-subset-events-of* : ‹$\boldsymbol{\alpha}(P) \subseteq \alpha(P)$›
by (*simp add*: *events-of-is-strict-events-of-or-UNIV*)

13.1.2 Ticks of a Process

definition *ticks-of* :: ‹('*a*, '*r*) $\mathit{process}_{ptick}$ ⇒ '*r set*› (‹$\checkmark s'(-')$›)
where ‹$\checkmark s(P) \equiv \{r.\ \exists t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P\}$›

lemma *ticks-of-memI* : ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow r \in \checkmark s(P)$›
unfolding *ticks-of-def* **by** *blast*

lemma *ticks-of-memD* : ‹$r \in \checkmark s(P) \Longrightarrow \exists t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P$›
unfolding *ticks-of-def* **by** *blast*

lemma *ticks-of-memE* :
‹$r \in \checkmark s(P) \Longrightarrow (\bigwedge t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$›
by (*meson ticks-of-memD*)

definition *strict-ticks-of* :: ‹('*a*, '*r*) $\mathit{process}_{ptick}$ ⇒ '*r set*› (‹$\checkmark\mathbf{s}'(-')$›)
where ‹$\checkmark\mathbf{s}(P) \equiv \{r.\ \exists s.\ s\ @\ [\checkmark(r)] \in \mathcal{T}\ P - \mathcal{D}\ P\}$›

lemma *strict-ticks-of-memI* :
‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow t\ @\ [\checkmark(r)] \notin \mathcal{D}\ P \Longrightarrow r \in \checkmark\mathbf{s}(P)$›
unfolding *strict-ticks-of-def* **by** *blast*

lemma *strict-ticks-of-memD* :
‹$r \in \checkmark\mathbf{s}(P) \Longrightarrow \exists t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \wedge t \notin \mathcal{D}\ P$›
by (*simp add*: *strict-ticks-of-def*)
(*metis T-imp-front-tickFree butlast-snoc*
front-tickFree-iff-tickFree-butlast front-tickFree-single is-processT7)

lemma *strict-ticks-of-memE* :
‹$r \in \checkmark\mathbf{s}(P) \Longrightarrow (\bigwedge t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$›

by (*meson strict-ticks-of-memD*)

lemma *ticks-of-is-strict-ticks-of-or-UNIV* :
‹✓$s(P)$ = (*if* $\mathcal{D}$ P = {} *then* ✓**s**(P) *else UNIV*)›
proof (*split if-split*, *intro conjI impI*)
show ‹$\mathcal{D}$ P = {} ⟹ ✓$s(P)$ = ✓**s**(P)› **by** (*simp add*: *ticks-of-def strict-ticks-of-def*)
next
assume ‹$\mathcal{D}$ P ≠ {}›
with *nonempty-divE* **obtain** t **where** ‹*tF t*› ‹t ∈ $\mathcal{D}$ P› **by** *blast*
hence ‹t @ [✓(r)] ∈ $\mathcal{D}$ P› **for** r **by** (*simp add*: *is-processT7*)
thus ‹✓$s(P)$ = *UNIV*› **by** (*metis D-T UNIV-eq-I ticks-of-memI*)
qed

lemma *strict-ticks-of-subset-ticks-of* : ‹✓**s**(P) ⊆ ✓$s(P)$›
by (*simp add*: *ticks-of-is-strict-ticks-of-or-UNIV*)

13.2 Laws

13.2.1 Preliminaries

lemma *inj-on-map-map-event*$_{ptick}$*-T-tickFree* :
‹*inj-on* (*map* (*map-event*$_{ptick}$ f g)) {t ∈ $\mathcal{T}$ P. *tF t*}› **if** ‹*inj-on* f $\alpha(P)$›
proof (*rule inj-onI*)
have ∗ : ‹*map-event*$_{ptick}$ f g ∘ *ev* = *ev* ∘ f› **by** (*rule ext*) *simp*
have ∗∗ : ‹*map ev v* ∈ $\mathcal{T}$ P ⟹ *map ev v'* ∈ $\mathcal{T}$ P ⟹
map (*ev* ∘ f) v = *map* (*ev* ∘ f) v' ⟹ v = v'› **for** v v'
proof (*induct v arbitrary*: v' *rule*: *rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc a v*)
from *snoc.prems*(*3*) **obtain** a' v'' **where** ‹v' = v'' @ [a']›
by (*metis list.map-disc-iff rev-exhaust*)
from *snoc.prems*(*1−3*) **have** ‹a = a'›
by (*auto simp add*: ‹v' = v'' @ [a']›
intro!: *inj-onD*[*OF* ‹*inj-on* f $\alpha(P)$›] *events-of-memI*)
from *is-processT3-TR-append snoc.prems*(*1*, *2*)
have ‹*map ev v* ∈ $\mathcal{T}$ P› ‹*map ev v''* ∈ $\mathcal{T}$ P› **by** (*simp-all add*: ‹v' = v'' @ [a']›)
with ‹a = a'› ‹v' = v'' @ [a']› *snoc.hyps snoc.prems*(*3*) **show** *?case*
by (*metis append-same-eq map-append*)
qed

fix t t' **assume** $: ‹t ∈ {t ∈ $\mathcal{T}$ P. *tF t*}› ‹t' ∈ {t ∈ $\mathcal{T}$ P. *tF t*}›
‹*map* (*map-event*$_{ptick}$ f g) t = *map* (*map-event*$_{ptick}$ f g) t'›
from $(*1*, *2*) **obtain** v v' **where** ‹t = *map ev v*› ‹t' = *map ev v'*›
by (*auto simp add*: *tickFree-iff-is-map-ev*)
with $ ∗ ∗∗ **show** ‹t = t'› **by** *auto*
qed

lemma *inj-on-map-map-event$_{ptick}$-T* :
‹*inj-on* (*map* (*map-event$_{ptick}$ f g*)) (*𝒯 P*)› **if** ‹*inj-on f α*(*P*)› ‹*inj-on g ✓s*(*P*)›
proof (*rule inj-onI*)
fix *t t′* **assume** $: ‹*t* ∈ *𝒯 P*› ‹*t′* ∈ *𝒯 P*› ‹*map* (*map-event$_{ptick}$ f g*) *t* = *map* (*map-event$_{ptick}$ f g*) *t′*›
then consider ‹*tF t*› ‹*tF t′*›
| *r r′ u u′* **where** ‹*t* = *u* @ [✓(*r*)]› ‹*t′* = *u′* @ [✓(*r′*)]› ‹*tF u*› ‹*tF u′*›
‹*map* (*map-event$_{ptick}$ f g*) *u* = *map* (*map-event$_{ptick}$ f g*) *u′*›
by (*cases t rule*: *rev-cases*; *cases t′ rule*: *rev-cases*, *simp-all*)
(*metis append-T-imp-tickFree event$_{ptick}$.disc*(1) *event$_{ptick}$.exhaust event$_{ptick}$.map-disc-iff*(1) *neq-Nil-conv*)
thus ‹*t* = *t′*›
proof *cases*
show ‹*t* = *t′*› **if** ‹*tF t*› ‹*tF t′*›
by (*rule inj-on-map-map-event$_{ptick}$-T-tickFree*
[*OF* ‹*inj-on f α*(*P*)›, *THEN inj-onD*, *OF* $(3)])
(*simp-all add*: $(1, 2) ‹*tF t*› ‹*tF t′*›)
next
fix *r r′ u u′* **assume** $$: ‹*t* = *u* @ [✓(*r*)]› ‹*t′* = *u′* @ [✓(*r′*)]› ‹*tF u*› ‹*tF u′*›
‹*map* (*map-event$_{ptick}$ f g*) *u* = *map* (*map-event$_{ptick}$ f g*) *u′*›
from $$(1, 2) $(1, 2) **have** ‹*u* ∈ *𝒯 P*› ‹*u′* ∈ *𝒯 P*›
by (*meson is-processT3-TR-append*)+
have ‹*u* = *u′*›
by (*rule inj-on-map-map-event$_{ptick}$-T-tickFree*
[*OF* ‹*inj-on f α*(*P*)›, *THEN inj-onD*, *OF* $$(5)])
(*simp-all add*: ‹*u* ∈ *𝒯 P*› ‹*u′* ∈ *𝒯 P*› ‹*tF u*› ‹*tF u′*›)
moreover from $(1−3) $$(1, 2) **have** ‹*r* = *r′*›
by (*auto intro*: *inj-onD*[*OF* ‹*inj-on g ✓s*(*P*)›] *ticks-of-memI*)
ultimately show ‹*t* = *t′*› **by** (*simp add*: $$)
qed
qed

lemma *inj-on-map-map-event$_{ptick}$-T-diff-D-tickFree* :
‹*inj-on* (*map* (*map-event$_{ptick}$ f g*)) {*t* ∈ *𝒯 P* − *𝒟 P*. *tF t*}› **if** ‹*inj-on f α*(*P*)›
proof (*rule inj-onI*)
have ∗ : ‹*map-event$_{ptick}$ f g* ∘ *ev* = *ev* ∘ *f*› **by** (*rule ext*) *simp*
have ∗∗ : ‹⟦*map ev v* ∈ *𝒯 P*; *map ev v* ∉ *𝒟 P*;
map ev v′ ∈ *𝒯 P*; *map ev v′* ∉ *𝒟 P*;
map (*ev* ∘ *f*) *v* = *map* (*ev* ∘ *f*) *v′*⟧ ⟹ *v* = *v′*› **for** *v v′*
proof (*induct v arbitrary*: *v′ rule*: *rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc a v*)
from *snoc.prems*(5) **obtain** *a′ v″* **where** ‹*v′* = *v″* @ [*a′*]›
by (*metis list.map-disc-iff rev-exhaust*)
from *snoc.prems*(1−5) **have** ‹*a* = *a′*›
by (*auto simp add*: ‹*v′* = *v″* @ [*a′*]›

```
        intro!: inj-onD[OF ‹inj-on f α(P)›] strict-events-of-memI)
    from is-processT3-TR-append is-processT7 snoc.prems(1−4)
    have ‹map ev v ∈ 𝒯 P› ‹map ev v ∉ 𝒟 P›
      ‹map ev v'' ∈ 𝒯 P› ‹map ev v'' ∉ 𝒟 P›
      by (fastforce simp add: ‹v' = v'' @ [a']›)+
    with ‹a = a'› ‹v' = v'' @ [a']› snoc.hyps snoc.prems(5) show ?case
      by (metis append-same-eq map-append)
  qed

  fix t t' assume $ : ‹t ∈ {t ∈ 𝒯 P − 𝒟 P. tF t}› ‹t' ∈ {t ∈ 𝒯 P − 𝒟 P. tF t}›
    ‹map (map-event_ptick f g) t = map (map-event_ptick f g) t'›
  from $(1, 2) obtain v v' where ‹t = map ev v› ‹t' = map ev v'›
    by (auto simp add: tickFree-iff-is-map-ev)
  with $ ∗ ∗∗ show ‹t = t'› by auto
qed


lemma inj-on-map-map-event_ptick-T-diff-D :
  ‹inj-on (map (map-event_ptick f g)) (𝒯 P − 𝒟 P)› if ‹inj-on f α(P)› and ‹inj-on
g ✓s(P)›
proof (rule inj-onI)
  fix t t' assume $ : ‹t ∈ 𝒯 P − 𝒟 P› ‹t' ∈ 𝒯 P − 𝒟 P›
    ‹map (map-event_ptick f g) t = map (map-event_ptick f g) t'›
  then consider ‹tF t› ‹tF t'›
    | r r' u u' where ‹t = u @ [✓(r)]› ‹t' = u' @ [✓(r')]› ‹tF u› ‹tF u'›
      ‹map (map-event_ptick f g) u = map (map-event_ptick f g) u'›
    by (cases t rule: rev-cases; cases t' rule: rev-cases, simp-all)
    (metis append-T-imp-tickFree event_ptick.disc(1) event_ptick.exhaust event_ptick.map-disc-iff(1)
neq-Nil-conv)
  thus ‹t = t'›
  proof cases
    show ‹t = t'› if ‹tF t› ‹tF t'›
      by (rule inj-on-map-map-event_ptick-T-diff-D-tickFree
          [OF ‹inj-on f α(P)›, THEN inj-onD, OF $(3)])
        (use $(1, 2) in ‹simp-all add: ‹tF t› ‹tF t'››)
  next
    fix r r' u u' assume $$ : ‹t = u @ [✓(r)]› ‹t' = u' @ [✓(r')]› ‹tF u› ‹tF u'›
      ‹map (map-event_ptick f g) u = map (map-event_ptick f g) u'›
    from $$(1−4) $(1, 2) have ‹u ∈ 𝒯 P› ‹u ∉ 𝒟 P› ‹u' ∈ 𝒯 P› ‹u' ∉ 𝒟 P›
      by (auto intro: is-processT3-TR-append is-processT7)
    have ‹u = u'›
      by (rule inj-on-map-map-event_ptick-T-diff-D-tickFree
          [OF ‹inj-on f α(P)›, THEN inj-onD, OF $$(5)])
        (simp-all add: ‹u ∈ 𝒯 P› ‹u ∉ 𝒟 P› ‹u' ∈ 𝒯 P› ‹u' ∉ 𝒟 P› ‹tF u› ‹tF u'›)
    moreover from $(1−3) $$(1, 2) have ‹r = r'›
      by (auto intro: inj-onD[OF ‹inj-on g ✓s(P)›] strict-ticks-of-memI)
    ultimately show ‹t = t'› by (simp add: $$)
  qed
qed
```

13.2.2 Events of a Process

lemma *events-of-BOT* [*simp*] : ‹$\alpha(\bot) = UNIV$›
and *events-of-SKIP* [*simp*] : ‹$\alpha(SKIP\ r) = \{\}$›
and *events-of-STOP* [*simp*] : ‹$\alpha(STOP) = \{\}$›
by (*auto simp add*: *events-of-def T-BOT T-SKIP T-STOP*)
(*meson front-tickFree-single list.set-intros*(*1*))

lemma *anti-mono-events-of-T*: ‹$P \sqsubseteq_T Q \Longrightarrow \alpha(Q) \subseteq \alpha(P)$›
unfolding *trace-refine-def events-of-def* **by** *blast*

lemma *anti-mono-events-of-F*: ‹$P \sqsubseteq_F Q \Longrightarrow \alpha(Q) \subseteq \alpha(P)$›
by (*intro anti-mono-events-of-T leF-imp-leT*)

lemma *anti-mono-events-of-FD*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow \alpha(Q) \subseteq \alpha(P)$›
by (*intro anti-mono-events-of-F leFD-imp-leF*)

lemma *anti-mono-events-of-DT*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow \alpha(Q) \subseteq \alpha(P)$›
by (*intro anti-mono-events-of-T leDT-imp-leT*)

lemma *anti-mono-events-of* : ‹$P \sqsubseteq Q \Longrightarrow \alpha(Q) \subseteq \alpha(P)$›
by (*intro anti-mono-events-of-FD le-approx-imp-le-ref*)

lemma *events-of-GlobalNdet*: ‹$\alpha(\sqcap a \in A.\ P\ a) = (\bigcup a{\in}A.\ \alpha(P\ a))$›
by (*simp add*: *events-of-def T-GlobalNdet*)

lemma *events-of-write0* : ‹$\alpha(a \rightarrow P) = insert\ a\ \alpha(P)$›
by (*fastforce simp add*: *events-of-def write0-def T-Mprefix*)

lemma *events-of-Mndetprefix*: ‹$\alpha(\sqcap a{\in}A \rightarrow P\ a) = A \cup (\bigcup a{\in}A.\ \alpha(P\ a))$›
by (*auto simp add*: *Mndetprefix-GlobalNdet events-of-GlobalNdet events-of-write0*)

lemma *events-of-Mprefix*: ‹$\alpha(\Box a{\in}A \rightarrow P\ a) = A \cup (\bigcup a{\in}A.\ \alpha(P\ a))$›
by (*simp add*: *events-of-def write0-def T-Mprefix set-eq-iff*)
(*metis* $event_{ptick}$*.inject*(*1*) *is-processT1-TR list.set-intros set-ConsD*)

lemma *events-of-read* :
‹$\alpha(c?a{\in}A \rightarrow P\ a) = c\ `\ A \cup (\bigcup a{\in}c\ `\ A.\ \alpha(P\ (inv\text{-}into\ A\ c\ a)))$›
by (*simp add*: *read-def events-of-Mprefix*)

lemma *events-of-inj-on-read* :
‹$inj\text{-}on\ c\ A \Longrightarrow \alpha(c?a{\in}A \rightarrow P\ a) = c\ `\ A \cup (\bigcup a{\in}A.\ \alpha(P\ a))$›
by (*simp add*: *events-of-read*)

lemma *events-of-ndet-write* :
‹$\alpha(c!!a{\in}A \rightarrow P\ a) = c\ `\ A \cup (\bigcup a{\in}c\ `\ A.\ \alpha(P\ (inv\text{-}into\ A\ c\ a)))$›

by (*simp add*: *ndet-write-def events-of-Mndetprefix*)

lemma *events-of-inj-on-ndet-write* :
‹*inj-on c A* ⟹ $\alpha(c!!a{\in}A \rightarrow P\ a) = c\ `\ A \cup (\bigcup a{\in}A.\ \alpha(P\ a))$›
by (*simp add*: *events-of-ndet-write*)

lemma *events-of-write* : ‹$\alpha(c!a \rightarrow P) = insert\ (c\ a)\ \alpha(P)$›
by (*simp add*: *write-def events-of-Mprefix*)

lemma *events-of-Ndet*: ‹$\alpha(P \sqcap Q) = \alpha(P) \cup \alpha(Q)$›
unfolding *events-of-def* **by** (*simp add*: *T-Ndet*)

lemma *events-of-Det*: ‹$\alpha(P \ \square\ Q) = \alpha(P) \cup \alpha(Q)$›
unfolding *events-of-def* **by** (*simp add*: *T-Det*)

lemma *events-of-Sliding*: ‹$\alpha(P \rhd Q) = \alpha(P) \cup \alpha(Q)$›
unfolding *Sliding-def* **by** (*simp add*: *events-of-Ndet events-of-Det*)

lemma *events-of-Renaming*:
‹$\alpha(Renaming\ P\ f\ g) = (if\ \mathcal{D}\ P = \{\}\ then\ f\ `\ \alpha(P)\ else\ UNIV)$›
proof (*simp*, *intro conjI impI*)
show ‹$\mathcal{D}\ P \neq \{\} \Longrightarrow \alpha(Renaming\ P\ f\ g) = UNIV$›
by (*simp add*: *events-of-is-strict-events-of-or-UNIV D-Renaming*)
(*metis front-tickFree-Nil nonempty-divE*)
next
show ‹$\mathcal{D}\ P = \{\} \Longrightarrow \alpha(Renaming\ P\ f\ g) = f\ `\ \alpha(P)$›
by (*auto simp add*: *events-of-def T-Renaming image-UN image-iff* $event_{ptick}$.*case-eq-if*)
(*metis* $event_{ptick}$.*inject*(*1*) $event_{ptick}$.*map-disc-iff*(*1*) $event_{ptick}$.*simps*(*9*)
is-ev-def,
metis (*mono-tags*, *lifting*) $event_{ptick}$.*simps*(*9*) *image-iff list.set-map*)
qed

lemma *events-of-Seq* : ‹$\alpha(P\ ;\ Q) = \alpha(P) \cup (if\ \checkmark\mathbf{s}(P) = \{\}\ then\ \{\}\ else\ \alpha(Q))$›
(**is** ‹- = *?A*›)
proof (*intro subset-antisym subsetI*)
show ‹$a \in \alpha(P\ ;\ Q) \Longrightarrow a \in ?A$› **for** *a*
proof (*elim events-of-memE*, *unfold T-Seq*, *elim UnE disjE conjE exE*)
show ‹*ev a* ∈ *set t* ⟹ $t \in \{t \in \mathcal{T}\ P.\ tF\ t\} \Longrightarrow a \in ?A$› **for** *t*
by (*auto intro*: *events-of-memI*)
next
show ‹*ev a* ∈ *set t* ⟹ $t \in \{t\ @\ u\ |t\ u\ r.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q\} \Longrightarrow$
$a \in ?A$› **for** *t*
by *simp* (*metis UNIV-I UnE UnI1 empty-iff events-of-is-strict-events-of-or-UNIV*
events-of-memI set-append strict-ticks-of-memI)
next

```
    show ‹ev a ∈ set t ⟹ t ∈ 𝒟 P ⟹ a ∈ ?A› for t
      by simp (metis UNIV-I empty-iff events-of-is-strict-events-of-or-UNIV)
  qed
next
  show ‹a ∈ ?A ⟹ a ∈ α(P ; Q)› for a
   by (elim UnE events-of-memE, simp-all add: events-of-def T-Seq split: if-split-asm)
    (metis T-nonTickFree-imp-decomp Un-iff event_ptick.distinct(1) is-processT1-TR
set-ConsD set-append,
       metis Un-iff ex-in-conv set-append strict-ticks-of-memD)
qed



lemma events-of-Sync-subset : ‹α(P ⟦S⟧ Q) ⊆ α(P) ∪ α(Q)›
  by (subst events-of-def, simp add: T-Sync subset-iff)
    (metis UNIV-I empty-iff events-of-is-strict-events-of-or-UNIV events-of-memI
ftf-Sync1)


lemma events-of-Inter: ‹α((P :: ('a, 'r) process_ptick) ||| Q) = α(P) ∪ α(Q)›
proof (rule subset-antisym[OF events-of-Sync-subset])
  have ‹α(P :: ('a, 'r) process_ptick) ⊆ α(P ||| Q)› for P Q
  proof (auto simp add: events-of-def T-Sync, goal-cases)
    case (1 e t-P)
    show ?case
    proof (cases ‹tF t-P›)
      case True
      thus ?thesis
        by (metis 1(1) 1(3) emptyLeftSelf insert-absorb insert-disjoint(2)
            is-processT1-TR setinterleaving-sym tickFree-def)
    next
      case False
      then obtain t-P' r where ‹t-P = t-P' @ [✓(r)]› ‹tF t-P'› ‹t-P' ∈ 𝒯 P›
        by (metis 1(1) prefixI T-nonTickFree-imp-decomp
            append-T-imp-tickFree is-processT3-TR not-Cons-self)
      moreover have ‹ev e ∈ set t-P'›
        using 1(3) calculation(1) by auto
      ultimately show ?thesis
        apply (rule-tac x = t-P' in exI, simp)
        apply (rule-tac x = t-P' in exI, simp)
        apply (rule-tac x = ‹[]› in exI, simp)
        by (metis disjoint-iff emptyLeftSelf setinterleaving-sym tickFree-def)
    qed
  qed
  thus ‹α(P) ∪ α(Q) ⊆ α(P ||| Q)› by (metis Sync-commute Un-least)
qed
```

lemma *events-of-Par-div* :
‹$\mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P \neq \{\} \Longrightarrow \alpha(P \parallel Q) = UNIV$›
and *events-of-Par-subset* :
‹$\mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P = \{\} \Longrightarrow \alpha(P \parallel Q) \subseteq \alpha(P) \cap \alpha(Q)$›
proof −
assume ‹$\mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P \neq \{\}$›
hence ‹$\mathcal{D}\ (P \parallel Q) \neq \{\}$› **by** (*simp add*: *D-Sync setinterleaving-UNIV-iff*)
(*use front-tickFree-Nil* **in** *blast*)
thus ‹$\alpha(P \parallel Q) = UNIV$› **by** (*simp add*: *events-of-is-strict-events-of-or-UNIV*)
next
show ‹$\mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P = \{\} \Longrightarrow \alpha(P \parallel Q) \subseteq \alpha(P) \cap \alpha(Q)$›
by (*auto simp add*: *events-of-def T-Par*)
qed

lemma *events-of-Hiding*:
‹$\alpha(P \setminus B) = (if\ \mathcal{D}\ (P \setminus B) = \{\}\ then\ \alpha(P) - B\ else\ UNIV)$›
proof (*split if-split*, *intro conjI impI*)
show ‹$\mathcal{D}\ (P \setminus B) \neq \{\} \Longrightarrow \alpha(P \setminus B) = UNIV$›
by (*simp add*: *events-of-is-strict-events-of-or-UNIV*)
next
show ‹$\alpha(P \setminus B) = \alpha(P) - B$› **if** ‹$\mathcal{D}\ (P \setminus B) = \{\}$›
proof (*intro subset-antisym subsetI*)
from ‹$\mathcal{D}\ (P \setminus B) = \{\}$› **have** ‹*div-hide P B* = $\{\}$› **unfolding** *D-Hiding* **by** *blast*
fix *a* **assume** ‹$a \in \alpha(P \setminus B)$›
then obtain *t* **where** ‹*ev a* ∈ *set* (*trace-hide t* (*ev* ‘ *B*))› ‹$(t,$ *ev* ‘ *B*$) \in \mathcal{F}\ P$›
by (*elim events-of-memE*, *unfold T-Hiding* ‹*div-hide P B* = $\{\}$›, *blast*)
thus ‹$a \in \alpha(P) - B$› **by** (*metis DiffI F-T events-of-memI*
filter-set image-eqI member-filter)
next
fix *a* **assume** ‹$a \in \alpha(P) - B$›
then obtain *t* **where** ‹*ev a* ∈ *set t*› ‹$t \in \mathcal{T}\ P$› ‹$a \notin B$›
by (*metis DiffE events-of-memD*)
hence ‹*ev a* ∈ *set* (*trace-hide t* (*ev* ‘ *B*))› ‹*trace-hide t* (*ev* ‘ *B*) ∈ $\mathcal{T}\ (P \setminus B)$›
by (*auto intro*: *mem-T-imp-mem-T-Hiding*)
thus ‹$a \in events\text{-}of\ (P \setminus B)$› **by** (*simp add*: *events-of-memI*)
qed
qed

13.2.3 Strict Events of a Process

lemma *strict-events-of-BOT* [*simp*] : ‹$\boldsymbol{\alpha}(\bot) = \{\}$›
and *strict-events-of-SKIP* [*simp*] : ‹$\boldsymbol{\alpha}(SKIP\ r) = \{\}$›
and *strict-events-of-STOP* [*simp*] : ‹$\boldsymbol{\alpha}(STOP) = \{\}$›
by (*auto simp add*: *strict-events-of-def T-BOT T-SKIP T-STOP D-BOT*)

lemma *strict-events-of-GlobalNdet-subset* : ‹$\boldsymbol{\alpha}(\sqcap a \in A.\ P\ a) \subseteq (\bigcup a \in A.\ \boldsymbol{\alpha}(P\ a))$›

by (*auto simp add*: *strict-events-of-def GlobalNdet-projs*)

lemma *strict-events-of-Mprefix*:
‹$\boldsymbol{\alpha}(\Box a{\in}A \to P\ a) = \{a{\in}A.\ P\ a \neq \bot\} \cup (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
proof −
have ‹$(\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a)) = (\bigcup a{\in}A.\ \boldsymbol{\alpha}(P\ a))$› **by** *auto*
show ‹$\boldsymbol{\alpha}(\Box a{\in}A \to P\ a) = \{a{\in}A.\ P\ a \neq \bot\} \cup (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
proof (*unfold* ‹*?this*›, *safe*)
show ‹$a \in \boldsymbol{\alpha}(\Box a{\in}A \to P\ a) \Longrightarrow a \notin (\bigcup a{\in}A.\ \boldsymbol{\alpha}(P\ a)) \Longrightarrow a \in A$› **for** *a*
by (*auto simp add*: *strict-events-of-def Mprefix-projs*) *blast*
next
show ‹$a \in \boldsymbol{\alpha}(\Box a{\in}A \to P\ a) \Longrightarrow a \notin (\bigcup a{\in}A.\ \boldsymbol{\alpha}(P\ a)) \Longrightarrow P\ a = \bot \Longrightarrow$ *False*›
for *a*
by (*auto simp add*: *strict-events-of-def Mprefix-projs BOT-projs*) *blast*
next
fix *a* **assume** ‹$a \in A$› ‹$P\ a \neq \bot$›
hence ∗ : ‹$\exists\, t{\in}$*insert* [] $\{ev\ a\ \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{T}\ (P\ a)\} -$
$\{ev\ a\ \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{D}\ (P\ a)\}.\ ev\ a \in$ *set t*›
by (*intro bexI*[**where** *x* = ‹[*ev a*]›]) (*simp-all add*: *BOT-iff-Nil-D*)
show ‹$a \in \boldsymbol{\alpha}(\Box a{\in}A \to P\ a)$›
by (*auto simp add*: *strict-events-of-def Mprefix-projs intro*: ∗)
next
fix *a b* **assume** ‹$a \in A$› ‹$b \in \boldsymbol{\alpha}(P\ a)$›
then obtain *t* **where** ‹$t \in \mathcal{T}\ (P\ a)$› ‹$t \notin \mathcal{D}\ (P\ a)$› ‹*ev b* ∈ *set t*›
by (*meson strict-events-of-memE*)
hence ∗ : ‹$\exists\, u{\in}$*insert* [] $\{ev\ a\ \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{T}\ (P\ a)\} -$
$\{ev\ a\ \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{D}\ (P\ a)\}.\ ev\ b \in$ *set u*›
by (*intro bexI*[**where** *x* = ‹*ev a # t*›]) (*simp-all add*: ‹$a \in A$›)
show ‹$b \in \boldsymbol{\alpha}(\Box a{\in}A \to P\ a)$›
by (*auto simp add*: *strict-events-of-def Mprefix-projs intro*: ∗)
qed
qed

lemma *strict-events-of-Mndetprefix*:
‹$\boldsymbol{\alpha}(\sqcap a{\in}A \to P\ a) = \{a{\in}A.\ P\ a \neq \bot\} \cup (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
proof −
have ‹$\mathcal{T}\ (\sqcap a{\in}A \to P\ a) = \mathcal{T}\ (\Box a{\in}A \to P\ a)$› **by** (*simp add*: *T-Mndetprefix′ T-Mprefix*)
moreover have ‹$\mathcal{D}\ (\sqcap a{\in}A \to P\ a) = \mathcal{D}\ (\Box a{\in}A \to P\ a)$› **by** (*simp add*: *D-Mndetprefix′ D-Mprefix*)
ultimately have ‹$\boldsymbol{\alpha}(\sqcap a{\in}A \to P\ a) = \boldsymbol{\alpha}(\Box a{\in}A \to P\ a)$› **by** (*simp add*: *strict-events-of-def*)
thus ‹$\boldsymbol{\alpha}(\sqcap a{\in}A \to P\ a) = \{a{\in}A.\ P\ a \neq \bot\} \cup (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
by (*simp add*: *strict-events-of-Mprefix*)
qed

lemma *strict-events-of-write0* : ‹$\boldsymbol{\alpha}(a \to P) =$ (*if* $P = \bot$ *then* {} *else insert a* $\boldsymbol{\alpha}(P)$)›
by (*simp add*: *write0-def strict-events-of-Mprefix*)

lemma *strict-events-of-read* :
‹$\boldsymbol{\alpha}(c?a{\in}A \to P\ a) = \{c\ a\ |a.\ a \in A \wedge P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\} \cup (\bigcup a{\in}\{a \in A.\ P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\}.\ \boldsymbol{\alpha}(P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a))))$›
by (*auto simp add*: *read-def strict-events-of-Mprefix*)

lemma *strict-events-of-inj-on-read* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \boldsymbol{\alpha}(c?a{\in}A \to P\ a) = \{c\ a\ |a.\ a \in A \wedge P\ a \neq \bot\} \cup (\bigcup a{\in}\{a \in A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
by (*auto simp add*: *strict-events-of-read*)

lemma *strict-events-of-ndet-write* :
‹$\boldsymbol{\alpha}(c!!a{\in}A \to P\ a) = \{c\ a\ |a.\ a \in A \wedge P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\} \cup (\bigcup a{\in}\{a \in A.\ P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\}.\ \boldsymbol{\alpha}(P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a))))$›
by (*auto simp add*: *ndet-write-def strict-events-of-Mndetprefix*)

lemma *strict-events-of-inj-on-ndet-write* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \boldsymbol{\alpha}(c!!a{\in}A \to P\ a) = \{c\ a\ |a.\ a \in A \wedge P\ a \neq \bot\} \cup (\bigcup a{\in}\{a \in A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
by (*auto simp add*: *strict-events-of-ndet-write*)

lemma *strict-events-of-write* : ‹$\boldsymbol{\alpha}(c!a \to P) = (\mathit{if}\ P = \bot\ \mathit{then}\ \{\}\ \mathit{else}\ \mathit{insert}\ (c\ a)\ \boldsymbol{\alpha}(P))$›
by (*simp add*: *write-def strict-events-of-Mprefix*)

lemma *strict-events-of-Ndet-subset* : ‹$\boldsymbol{\alpha}(P \sqcap Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
unfolding *strict-events-of-def* **by** (*auto simp add*: *Ndet-projs*)

lemma *strict-events-of-Det-subset* : ‹$\boldsymbol{\alpha}(P \,\square\, Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
unfolding *strict-events-of-def* **by** (*auto simp add*: *Det-projs*)

lemma *strict-events-of-Sliding-subset* : ‹$\boldsymbol{\alpha}(P \rhd Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
unfolding *strict-events-of-def* **by** (*auto simp add*: *Sliding-projs*)

lemma *strict-events-of-Renaming-subset* : ‹$\boldsymbol{\alpha}(\mathit{Renaming}\ P\ f\ g) \subseteq f\ `\ \boldsymbol{\alpha}(P)$›
proof (*intro subsetI*)
show ‹$b \in \boldsymbol{\alpha}(\mathit{Renaming}\ P\ f\ g) \Longrightarrow b \in f\ `\ \boldsymbol{\alpha}(P)$› **for** *b*
proof (*elim strict-events-of-memE*)
fix *u* **assume** ‹$u \in \mathcal{T}\ (\mathit{Renaming}\ P\ f\ g)$› ‹$u \notin \mathcal{D}\ (\mathit{Renaming}\ P\ f\ g)$› ‹$\mathit{ev}\ b \in \mathit{set}\ u$›
then obtain u' **where** ‹$\mathit{tF}\ u'$› ‹$u' \in \mathcal{T}\ (\mathit{Renaming}\ P\ f\ g)$› ‹$u' \notin \mathcal{D}\ (\mathit{Renaming}\ P\ f\ g)$› ‹$\mathit{ev}\ b \in \mathit{set}\ u'$›
by (*cases u rule*: *rev-cases*, *simp-all*)
(*metis prefixI* ‹$\mathit{ev}\ b \in \mathit{set}\ u$› *append-T-imp-tickFree* $\mathit{event}_{ptick}.\mathit{disc}(1)$

front-tickFree-single
is-processT3-TR is-processT7 not-Cons-self2 tickFree-Cons-iff tickFree-Nil tickFree-append-iff)
from *this*(*1*−*3*) *front-tickFree-Nil map-event*$_{ptick}$*-tickFree* **obtain** *t*
where ‹$u' = map\ (map\text{-}event_{ptick}\ f\ g)\ t$› ‹$t \in \mathcal{T}\ P$› ‹$t \notin \mathcal{D}\ P$› **unfolding** *Renaming-projs* **by** *blast*
from *this*(*1*) ‹*ev b* ∈ *set u'*› **obtain** *a* **where** ‹*b* = *f a*› ‹*ev a* ∈ *set t*›
by (*auto simp add*: *in-set-conv-decomp*)
(*metis event*$_{ptick}$*.map-disc-iff*(*1*) *event*$_{ptick}$*.map-sel*(*1*) *event*$_{ptick}$*.sel*(*1*) *is-ev-def*)
with ‹$t \in \mathcal{T}\ P$› ‹$t \notin \mathcal{D}\ P$› **show** ‹$b \in f\ `\ \boldsymbol{\alpha}(P)$› **by** (*auto intro*: *strict-events-of-memI*)
qed
qed

lemma *strict-events-of-inj-on-Renaming* :
‹$\boldsymbol{\alpha}(Renaming\ P\ f\ g) = f\ `\ \boldsymbol{\alpha}(P)$› **if** ‹*inj-on f* $\alpha(P)$›
proof (*rule subset-antisym*)
show ‹$\boldsymbol{\alpha}(Renaming\ P\ f\ g) \subseteq f\ `\ \boldsymbol{\alpha}(P)$› **by** (*fact strict-events-of-Renaming-subset*)
next
show ‹$f\ `\ \boldsymbol{\alpha}(P) \subseteq \boldsymbol{\alpha}(Renaming\ P\ f\ g)$›
proof (*rule subsetI*, *elim imageE strict-events-of-memE*)
fix *b a t* **assume** ‹*b* = *f a*› ‹$t \in \mathcal{T}\ P$› ‹$t \notin \mathcal{D}\ P$› ‹*ev a* ∈ *set t*›
from ‹$t \in \mathcal{T}\ P$› ‹$t \notin \mathcal{D}\ P$› ‹*ev a* ∈ *set t*› **obtain** *t'*
where ‹*tF t'*› ‹$t' \in \mathcal{T}\ P$› ‹$t' \notin \mathcal{D}\ P$› ‹*ev a* ∈ *set t'*›
by (*cases t rule*: *rev-cases*, *simp-all*)
(*metis prefixI* ‹*ev a* ∈ *set t*› *append-T-imp-tickFree event*$_{ptick}$*.disc*(*1*) *front-tickFree-single*
is-processT3-TR is-processT7 not-Cons-self2 tickFree-Cons-iff tickFree-Nil tickFree-append-iff)
from ‹$t' \in \mathcal{T}\ P$› **have** ‹$map\ (map\text{-}event_{ptick}\ f\ g)\ t' \in \mathcal{T}\ (Renaming\ P\ f\ g)$›
by (*auto simp add*: *T-Renaming*)
have ‹$map\ (map\text{-}event_{ptick}\ f\ g)\ t' \notin \mathcal{D}\ (Renaming\ P\ f\ g)$›
proof (*rule ccontr*)
assume ‹$\neg\ map\ (map\text{-}event_{ptick}\ f\ g)\ t' \notin \mathcal{D}\ (Renaming\ P\ f\ g)$›
then obtain *u1 u2* **where** ∗ : ‹*ftF u2*› ‹*tF u1*› ‹$u1 \in \mathcal{D}\ P$›
‹$map\ (map\text{-}event_{ptick}\ f\ g)\ t' = map\ (map\text{-}event_{ptick}\ f\ g)\ u1\ @\ u2$›
unfolding *D-Renaming* **by** *blast*
from *this*(*4*) **obtain** *t1 t2*
where ∗∗ : ‹$t' = t1\ @\ t2$› ‹$map\ (map\text{-}event_{ptick}\ f\ g)\ t2 = u2$›
‹$map\ (map\text{-}event_{ptick}\ f\ g)\ t1 = map\ (map\text{-}event_{ptick}\ f\ g)\ u1$›
by (*metis* (*no-types*) *append-eq-map-conv*)
from ∗∗(*1*) ‹$t' \in \mathcal{T}\ P$› ‹*tF t'*› *is-processT3-TR-append tickFree-append-iff*
have ‹$t1 \in \{t \in \mathcal{T}\ P.\ tF\ t\}$› **by** *auto*
moreover have ‹$u1 \in \{t \in \mathcal{T}\ P.\ tF\ t\}$› **by** (*simp add*: *D-T* ‹*tF u1*› ‹$u1 \in \mathcal{D}\ P$›)
ultimately have ‹*t1* = *u1*› **by** (*intro inj-on-map-map-event*$_{ptick}$*-T-tickFree* [*THEN inj-onD*, *OF* ‹*inj-on f* $\alpha(P)$› ∗∗(*3*)])
with ∗(*1*−*3*) ∗∗(*1*, *2*) ‹$t' \notin \mathcal{D}\ P$› *is-processT7*

map-event$_{ptick}$-front-tickFree **show** *False* **by** *blast*
qed
moreover from ‹*ev a ∈ set t'*› ‹*b = f a*› **have** ‹*ev b ∈ set* (*map* (*map-event$_{ptick}$ f g*) *t'*)› **by** *force*
ultimately show ‹*b ∈ α*(*Renaming P f g*)›
using ‹*map* (*map-event$_{ptick}$ f g*) *t' ∈ 𝒯* (*Renaming P f g*)› **by** (*auto intro*: *strict-events-of-memI*)
qed
qed

lemma *strict-events-of-Seq-subseteq* :
‹*α*(*P* ; *Q*) ⊆ *α*(*P*) ∪ (*if* ✓**s**(*P*) = {} *then* {} *else α*(*Q*))›
by (*rule subsetI*, *elim strict-events-of-memE*, *simp add*: *Seq-projs*)
(*metis T-imp-front-tickFree Un-iff append-T-imp-tickFree empty-iff is-processT7 is-processT9 not-Cons-self2 set-append strict-events-of-memI strict-ticks-of-memI*)

lemma *strict-events-of-Sync-subset* : ‹*α*(*P* ⟦*S*⟧ *Q*) ⊆ *α*(*P*) ∪ *α*(*Q*)›
by (*subst strict-events-of-def*, *auto simp add*: *Sync-projs subset-iff*)
(*metis* (*full-types*) *append-Nil2 front-tickFree-Nil ftf-Sync1 setinterleaving-sym strict-events-of-memI*)

13.3 Ticks of a Process

lemma *ticks-of-BOT* [*simp*] : ‹✓*s*(⊥) = *UNIV*›
and *ticks-of-SKIP* [*simp*] : ‹✓*s*(*SKIP r*) = {*r*}›
and *ticks-of-STOP* [*simp*] : ‹✓*s*(*STOP*) = {}›
by (*simp-all add*: *set-eq-iff ticks-of-def T-BOT T-SKIP T-STOP*)
(*metis append-Nil front-tickFree-single*)

lemma *anti-mono-ticks-of-T*: ‹*P* ⊑$_T$ *Q* ⟹ ✓*s*(*Q*) ⊆ ✓*s*(*P*)›
unfolding *trace-refine-def ticks-of-def* **by** *blast*

lemma *anti-mono-ticks-of-F*: ‹*P* ⊑$_F$ *Q* ⟹ ✓*s*(*Q*) ⊆ ✓*s*(*P*)›
by (*intro anti-mono-ticks-of-T leF-imp-leT*)

lemma *anti-mono-ticks-of-FD*: ‹*P* ⊑$_{FD}$ *Q* ⟹ ✓*s*(*Q*) ⊆ ✓*s*(*P*)›
by (*intro anti-mono-ticks-of-F leFD-imp-leF*)

lemma *anti-mono-ticks-of-DT*: ‹*P* ⊑$_{DT}$ *Q* ⟹ ✓*s*(*Q*) ⊆ ✓*s*(*P*)›
by (*intro anti-mono-ticks-of-T leDT-imp-leT*)

lemma *anti-mono-ticks-of* : ‹*P* ⊑ *Q* ⟹ ✓*s*(*Q*) ⊆ ✓*s*(*P*)›
by (*intro anti-mono-ticks-of-FD le-approx-imp-le-ref*)

lemma *ticks-of-GlobalNdet*: ‹✓s(⊓a ∈ A. P a) = (⋃ a∈A. ✓s(P a))›
by (*auto simp add*: *ticks-of-def T-GlobalNdet*)

lemma *ticks-of-Mprefix*: ‹✓s(□a∈A → P a) = (⋃ a∈A. ✓s(P a))›
by (*auto simp add*: *set-eq-iff ticks-of-def T-Mprefix*)
(*metis append-butlast-last-id event*$_{ptick}$*.distinct*(*1*) *last.simps last-snoc*, *meson append-Cons*)

lemma *ticks-of-write0* : ‹✓s(a → P) = ✓s(P)›
by (*simp add*: *write0-def ticks-of-Mprefix*)

lemma *ticks-of-Mndetprefix*: ‹✓s(⊓a∈A → P a) = (⋃ a∈A. ✓s(P a))›
by (*simp add*: *Mndetprefix-GlobalNdet ticks-of-GlobalNdet ticks-of-write0*)

lemma *ticks-of-read* :
‹✓s(c?a∈A → P a) = (⋃ a∈c ‘ A. ✓s(P (*inv-into A c a*)))›
by (*simp add*: *read-def ticks-of-Mprefix*)

lemma *ticks-of-inj-on-read* :
‹*inj-on c A* ⟹ ✓s(c?a∈A → P a) = (⋃ a∈A. ✓s(P a))›
by (*simp add*: *ticks-of-read*)

lemma *ticks-of-ndet-write* :
‹✓s(c!!a∈A → P a) = (⋃ a∈c ‘ A. ✓s(P (*inv-into A c a*)))›
by (*simp add*: *ndet-write-def ticks-of-Mndetprefix*)

lemma *ticks-of-inj-on-ndet-write* :
‹*inj-on c A* ⟹ ✓s(c!!a∈A → P a) = (⋃ a∈A. ✓s(P a))›
by (*simp add*: *ticks-of-ndet-write*)

lemma *ticks-of-write* : ‹✓s(c!a → P) = ✓s(P)›
by (*simp add*: *write-def ticks-of-Mprefix*)

lemma *ticks-of-Ndet*: ‹✓s(P ⊓ Q) = ✓s(P) ∪ ✓s(Q)›
by (*auto simp add*: *ticks-of-def T-Ndet*)

lemma *ticks-of-Det*: ‹✓s(P □ Q) = ✓s(P) ∪ ✓s(Q)›
by (*auto simp add*: *ticks-of-def T-Det*)

lemma *ticks-of-Sliding*: ‹✓s(P ▷ Q) = ✓s(P) ∪ ✓s(Q)›
by (*auto simp add*: *ticks-of-def T-Sliding*)

lemma *ticks-of-Renaming*:

```
  ‹✓s(Renaming P f g) = (if 𝒟 P = {} then g ` ✓s(P) else UNIV)›
proof (simp, intro conjI impI)
  show ‹𝒟 P ≠ {} ⟹ ✓s(Renaming P f g) = UNIV›
    by (simp add: ticks-of-is-strict-ticks-of-or-UNIV D-Renaming)
      (meson front-tickFree-Nil nonempty-divE)
next
  show ‹✓s(Renaming P f g) = g ` ✓s(P)› if ‹𝒟 P = {}›
  proof (intro subset-antisym subsetI)
    show ‹r ∈ ✓s(Renaming P f g) ⟹ r ∈ g ` ✓s(P)› for r
    by (auto simp add: T-Renaming ‹𝒟 P = {}› append-eq-map-conv tick-eq-map-map-event_ptick-iff
        intro!: ticks-of-memI elim!: ticks-of-memE)
  next
    show ‹r ∈ g ` ✓s(P) ⟹ r ∈ ✓s(Renaming P f g)› for r
      by (simp add: ticks-of-def T-Renaming ‹𝒟 P = {}›
          append-eq-map-conv tick-eq-map-map-event_ptick-iff) blast
  qed
qed


lemma ticks-of-Seq :
  ‹✓s(P ; Q) = (if 𝒟 P = {} then if ✓s(P) = {} then {} else ✓s(Q) else UNIV)›
(is ‹?lhs = ?rhs›)
proof (intro subset-antisym subsetI)
  show ‹a ∈ ?lhs ⟹ a ∈ ?rhs› for a
    by (elim ticks-of-memE, auto simp add: T-Seq ticks-of-def)
      (metis T-nonTickFree-imp-decomp append-T-imp-tickFree last-appendR
        last-snoc non-tickFree-tick tickFree-Nil tickFree-append-iff)
next
  show ‹a ∈ ?rhs ⟹ a ∈ ?lhs› for a
    by (auto simp add: ticks-of-def T-Seq split: if-split-asm)
      (meson append-assoc, metis all-not-in-conv front-tickFree-single is-processT7
nonempty-divE)
qed


lemma ticks-of-Sync-subset : ‹✓s(P ⟦S⟧ Q) ⊆ ✓s(P) ∪ ✓s(Q)›
  by (auto simp add: T-Sync elim!: ticks-of-memE)
    (metis SyncWithTick-imp-NTF T-imp-front-tickFree ticks-of-memI,
      (metis UNIV-I empty-iff ticks-of-is-strict-ticks-of-or-UNIV)+)

lemma ticks-of-no-div-Sync-subset :
  ‹✓s(P ⟦S⟧ Q) ⊆ ✓s(P) ∩ ✓s(Q)› if ‹𝒟 (P ⟦S⟧ Q) = {}›
proof (rule subsetI)
  fix r assume ‹r ∈ ✓s(P ⟦S⟧ Q)›
  with ‹𝒟 (P ⟦S⟧ Q) = {}› obtain t t-P t-Q
    where * : ‹t-P ∈ 𝒯 P› ‹t-Q ∈ 𝒯 Q›
      ‹(t @ [✓(r)]) setinterleaves ((t-P, t-Q), range tick ∪ ev ` S)›
    by (elim ticks-of-memE, unfold Sync-projs, blast)
  from SyncWithTick-imp-NTF[OF *(3) *(1, 2)[THEN is-processT2-TR]]
```

obtain *t-P′ t-Q′* **where** ‹*t-P* = *t-P′* @ [✓(*r*)]› ‹*t-Q* = *t-Q′* @ [✓(*r*)]› **by** *blast*
with ∗(*1*, *2*) **show** ‹*r* ∈ ✓*s*(*P*) ∩ ✓*s*(*Q*)› **by** (*auto intro*: *ticks-of-memI*)
qed

lemma *ticks-of-Par-div* :
‹$\mathcal{D}$ *P* ∩ $\mathcal{T}$ *Q* ∪ $\mathcal{D}$ *Q* ∩ $\mathcal{T}$ *P* ≠ {} ⟹ ✓*s*(*P* || *Q*) = *UNIV*›
and *ticks-of-no-div-Par-subset* :
‹$\mathcal{D}$ *P* ∩ $\mathcal{T}$ *Q* ∪ $\mathcal{D}$ *Q* ∩ $\mathcal{T}$ *P* = {} ⟹ ✓*s*(*P* || *Q*) ⊆ ✓*s*(*P*) ∩ ✓*s*(*Q*)›
proof −
assume ‹$\mathcal{D}$ *P* ∩ $\mathcal{T}$ *Q* ∪ $\mathcal{D}$ *Q* ∩ $\mathcal{T}$ *P* ≠ {}›
hence ‹$\mathcal{D}$ (*P* || *Q*) ≠ {}› **by** (*simp add*: *D-Sync setinterleaving-UNIV-iff*)
(*use front-tickFree-Nil* **in** *blast*)
thus ‹✓*s*(*P* || *Q*) = *UNIV*› **by** (*simp add*: *ticks-of-is-strict-ticks-of-or-UNIV*)
next
show ‹$\mathcal{D}$ *P* ∩ $\mathcal{T}$ *Q* ∪ $\mathcal{D}$ *Q* ∩ $\mathcal{T}$ *P* = {} ⟹ ✓*s*(*P* || *Q*) ⊆ ✓*s*(*P*) ∩ ✓*s*(*Q*)›
by (*rule ticks-of-no-div-Sync-subset*) (*auto simp add*: *D-Par*)
qed

lemma *ticks-of-Hiding*:
‹✓*s*(*P* \ *B*) = (*if* $\mathcal{D}$ (*P* \ *B*) = {} *then* ✓*s*(*P*) *else UNIV*)›
proof (*split if-split*, *intro conjI impI*)
show ‹$\mathcal{D}$ (*P* \ *B*) ≠ {} ⟹ ✓*s*(*P* \ *B*) = *UNIV*›
by (*simp add*: *ticks-of-is-strict-ticks-of-or-UNIV*)
next
show ‹✓*s*(*P* \ *B*) = ✓*s*(*P*)› **if** ‹$\mathcal{D}$ (*P* \ *B*) = {}›
proof (*intro subset-antisym subsetI*)
fix *r* **assume** ‹*r* ∈ ✓*s*(*P* \ *B*)›
then obtain *u* **where** ‹*u* @ [✓(*r*)] ∈ $\mathcal{T}$ (*P* \ *B*)› **by** (*meson ticks-of-memE*)
with ‹$\mathcal{D}$ (*P* \ *B*) = {}› **obtain** *t*
where ‹*u* @ [✓(*r*)] = *trace-hide t* (*ev* ‘ *B*)› ‹(*t*, *ev* ‘ *B*) ∈ $\mathcal{F}$ *P*›
unfolding *T-Hiding D-Hiding* **by** *blast*
then obtain *t′* **where** ‹*t′* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*›
by (*cases t rule*: *rev-cases*, *auto split*: *if-split-asm intro*: *F-T*)
(*metis F-T Hiding-tickFree append-T-imp-tickFree list.distinct*(*1*)
non-tickFree-tick tickFree-append-iff)
thus ‹*r* ∈ ✓*s*(*P*)› **by** (*simp add*: *ticks-of-memI*)
next
fix *r* **assume** ‹*r* ∈ ✓*s*(*P*)›
then obtain *t* **where** ‹*t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*› **by** (*metis ticks-of-memD*)
hence ‹*trace-hide* (*t* @ [✓(*r*)]) (*ev* ‘ *B*) ∈ $\mathcal{T}$ (*P* \ *B*)›
by (*fact mem-T-imp-mem-T-Hiding*)
thus ‹*r* ∈ ✓*s*(*P* \ *B*)› **by** (*auto intro*: *ticks-of-memI split*: *if-split-asm*)
qed
qed

lemma *tickFree-traces-iff-empty-ticks-of* : ‹$(\forall\, t \in \mathcal{T}\ P.\ tF\ t) \longleftrightarrow \checkmark s(P) = \{\}$›
using *T-nonTickFree-imp-decomp* **unfolding** *ticks-of-def* **by** *auto*

13.3.1 Strict Events of a Process

lemma *strict-ticks-of-BOT* [*simp*] : ‹$\checkmark\mathbf{s}(\bot) = \{\}$›
and *strict-ticks-of-SKIP* [*simp*] : ‹$\checkmark\mathbf{s}(SKIP\ r) = \{r\}$›
and *strict-ticks-of-STOP* [*simp*] : ‹$\checkmark\mathbf{s}(STOP) = \{\}$›
by (*auto simp add*: *strict-ticks-of-def BOT-projs SKIP-projs T-STOP*)

lemma *strict-ticks-of-GlobalNdet-subset* : ‹$\checkmark\mathbf{s}(\sqcap a \in A.\ P\ a) \subseteq (\bigcup a{\in}A.\ \checkmark\mathbf{s}(P\ a))$›
by (*auto simp add*: *strict-ticks-of-def GlobalNdet-projs*)

lemma *strict-ticks-of-Mprefix*:
‹$\checkmark\mathbf{s}(\Box a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$›
proof −
have ‹$(\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a)) = (\bigcup a{\in}A.\ \checkmark\mathbf{s}(P\ a))$›
by (*auto intro*: *strict-ticks-of-memI is-processT9 elim*!: *strict-ticks-of-memE*)
(*metis D-BOT T-BOT is-processT9 strict-ticks-of-memI*)
show ‹$\checkmark\mathbf{s}(\Box a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$›
proof (*unfold* ‹*?this*›, *safe elim*!: *strict-ticks-of-memE*)
show ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ (\Box a{\in}A \rightarrow P\ a) \Longrightarrow t \notin \mathcal{D}\ (\Box a{\in}A \rightarrow P\ a)$
$\Longrightarrow r \in (\bigcup a{\in}A.\ \checkmark\mathbf{s}(P\ a))$› **for** *t r*
by (*auto simp add*: *strict-ticks-of-def Mprefix-projs*)
(*metis append-butlast-last-id butlast.simps*(*2*) *butlast-snoc*
event$_{ptick}$*.distinct*(*1*) *is-processT9 last.simps last-snoc*)
next
fix *a t r* **assume** ‹$a \in A$› ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ (P\ a)$› ‹$t \notin \mathcal{D}\ (P\ a)$›
hence ‹$(ev\ a\ \#\ t)\ @\ [\checkmark(r)] \in \mathcal{T}\ (\Box a{\in}A \rightarrow P\ a)$›
‹$(ev\ a\ \#\ t) \notin \mathcal{D}\ (\Box a{\in}A \rightarrow P\ a)$›
by (*auto simp add*: *strict-ticks-of-def Mprefix-projs*)
thus ‹$r \in \checkmark\mathbf{s}(\Box a{\in}A \rightarrow P\ a)$› **by** (*meson strict-ticks-of-memI is-processT9*)
qed
qed

lemma *strict-ticks-of-Mndetprefix*:
‹$\checkmark\mathbf{s}(\sqcap a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$›
proof −
have ‹$\mathcal{T}\ (\sqcap a{\in}A \rightarrow P\ a) = \mathcal{T}\ (\Box a{\in}A \rightarrow P\ a)$› **by** (*simp add*: *T-Mndetprefix′ T-Mprefix*)
moreover have ‹$\mathcal{D}\ (\sqcap a{\in}A \rightarrow P\ a) = \mathcal{D}\ (\Box a{\in}A \rightarrow P\ a)$› **by** (*simp add*: *D-Mndetprefix′ D-Mprefix*)
ultimately have ‹$\checkmark\mathbf{s}(\sqcap a{\in}A \rightarrow P\ a) = \checkmark\mathbf{s}(\Box a{\in}A \rightarrow P\ a)$› **by** (*simp add*: *strict-ticks-of-def*)
thus ‹$\checkmark\mathbf{s}(\sqcap a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$› **by** (*simp add*:

strict-ticks-of-Mprefix)
qed

lemma *strict-ticks-of-write0* : ‹$\checkmark\mathbf{s}(a \rightarrow P) = (\mathit{if}\ P = \bot\ \mathit{then}\ \{\}\ \mathit{else}\ \checkmark\mathbf{s}(P))$›
by (*simp add*: *write0-def strict-ticks-of-Mprefix*)

lemma *strict-ticks-of-read* :
‹$\checkmark\mathbf{s}(c?a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a \in A.\ P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\}.\ \checkmark\mathbf{s}(P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a))))$›
by (*auto simp add*: *read-def strict-ticks-of-Mprefix*)

lemma *strict-ticks-of-inj-on-read* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \checkmark\mathbf{s}(c?a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a \in A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$›
by (*auto simp add*: *strict-ticks-of-read*)

lemma *strict-ticks-of-ndet-write* :
‹$\checkmark\mathbf{s}(c!!a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a \in A.\ P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\}.\ \checkmark\mathbf{s}(P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a))))$›
by (*auto simp add*: *ndet-write-def strict-ticks-of-Mndetprefix*)

lemma *strict-ticks-of-inj-on-ndet-write* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \checkmark\mathbf{s}(c!!a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a \in A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$›
by (*auto simp add*: *strict-ticks-of-ndet-write*)

lemma *strict-ticks-of-write* : ‹$\checkmark\mathbf{s}(c!a \rightarrow P) = (\mathit{if}\ P = \bot\ \mathit{then}\ \{\}\ \mathit{else}\ \checkmark\mathbf{s}(P))$›
unfolding *write-def* **by** (*simp add*: *strict-ticks-of-Mprefix*)

lemma *strict-ticks-of-Ndet-subset* : ‹$\checkmark\mathbf{s}(P \sqcap Q) \subseteq \checkmark\mathbf{s}(P) \cup \checkmark\mathbf{s}(Q)$›
unfolding *strict-ticks-of-def* **by** (*auto simp add*: *Ndet-projs*)

lemma *strict-ticks-of-Det-subset* : ‹$\checkmark\mathbf{s}(P \mathbin{\square} Q) \subseteq \checkmark\mathbf{s}(P) \cup \checkmark\mathbf{s}(Q)$›
unfolding *strict-ticks-of-def* **by** (*auto simp add*: *Det-projs*)

lemma *strict-ticks-of-Sliding-subset* : ‹$\checkmark\mathbf{s}(P \rhd Q) \subseteq \checkmark\mathbf{s}(P) \cup \checkmark\mathbf{s}(Q)$›
unfolding *strict-ticks-of-def* **by** (*auto simp add*: *Sliding-projs*)

lemma *strict-ticks-of-Renaming-subset* : ‹$\checkmark\mathbf{s}(\mathit{Renaming}\ P\ f\ g) \subseteq g\ `\ \checkmark\mathbf{s}(P)$›
proof (*intro subsetI*)
fix *s* **assume** ‹$s \in \checkmark\mathbf{s}(\mathit{Renaming}\ P\ f\ g)$›
then obtain *u* **where** ‹$u\ @\ [\checkmark(s)] \in \mathcal{T}\ (\mathit{Renaming}\ P\ f\ g)$›
‹$u \notin \mathcal{D}\ (\mathit{Renaming}\ P\ f\ g)$› **by** (*meson strict-ticks-of-memD*)
then obtain *t r* **where** ‹$s = g\ r$› ‹$u = \mathit{map}\ (\mathit{map\text{-}event}_{ptick}\ f\ g)\ t$› ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P$› ‹$t \notin \mathcal{D}\ P$›
by (*auto simp add*: *Renaming-projs append-eq-map-conv tick-eq-map-event*$_{ptick}$*-iff*)
(*use append-T-imp-tickFree front-tickFree-Nil* **in** *blast*,
metis append-assoc butlast-snoc front-tickFree-iff-tickFree-butlast map-event$_{ptick}$*-tickFree*

nonTickFree-n-frontTickFree non-tickFree-tick tickFree-append-iff tickFree-imp-front-tickFree)
thus ‹$s \in g$ ‘ ✓**s**(P)› **by** (*auto intro*: *strict-ticks-of-memI is-processT9*)
qed

lemma *strict-ticks-of-inj-on-Renaming* :
‹✓**s**(*Renaming* $P\ f\ g$) = g ‘ ✓**s**(P)› **if** ‹*inj-on* $f\ \alpha(P)$›
proof (*rule subset-antisym*)
show ‹✓**s**(*Renaming* $P\ f\ g$) ⊆ g ‘ ✓**s**(P)› **by** (*fact strict-ticks-of-Renaming-subset*)
next
show ‹g ‘ ✓**s**(P) ⊆ ✓**s**(*Renaming* $P\ f\ g$)›
proof (*rule subsetI*, *elim imageE strict-ticks-of-memE*)
fix $s\ r\ t$ **assume** ‹$s = g\ r$› ‹t @ [✓(r)] ∈ $\mathcal{T}\ P$› ‹$t \notin \mathcal{D}\ P$›
from ‹$s = g\ r$› ‹t @ [✓(r)] ∈ $\mathcal{T}\ P$›
have ‹*map* (*map-event*$_{ptick}$ $f\ g$) t @ [✓(s)] ∈ $\mathcal{T}$ (*Renaming* $P\ f\ g$)›
by (*auto simp add*: *T-Renaming*)
moreover have ‹*map* (*map-event*$_{ptick}$ $f\ g$) t @ [✓(s)] ∉ $\mathcal{D}$ (*Renaming* $P\ f\ g$)›
proof (*rule ccontr*)
assume ‹¬ *map* (*map-event*$_{ptick}$ $f\ g$) t @ [✓(s)] ∉ $\mathcal{D}$ (*Renaming* $P\ f\ g$)›
then obtain *u1 u2* **where** ∗ : ‹*ftF u2*› ‹*tF u1*› ‹*u1* ∈ $\mathcal{D}\ P$›
‹*map* (*map-event*$_{ptick}$ $f\ g$) t @ [✓(s)] = *map* (*map-event*$_{ptick}$ $f\ g$) *u1* @ *u2*›
by (*auto simp add*: *D-Renaming*)
from ∗(*1*, *2*, *4*) **obtain** *u2′* **where** ‹*u2* = *u2′* @ [✓(s)]›
by (*metis last-appendR map-event*$_{ptick}$*-tickFree nonTickFree-n-frontTickFree non-tickFree-tick snoc-eq-iff-butlast tickFree-append-iff*)
obtain *t1 t2* **where** ∗∗ : ‹t = *t1* @ *t2*› ‹*map* (*map-event*$_{ptick}$ $f\ g$) *t1* = *map* (*map-event*$_{ptick}$ $f\ g$) *u1*›
by (*metis* ∗(*4*) ‹*u2* = *u2′* @ [✓(s)]› *append.assoc append-eq-map-conv butlast-snoc*)
moreover from ∗(*2*) ‹t @ [✓(r)] ∈ $\mathcal{T}\ P$› *calculation* **have** ‹*t1* ∈ {t ∈ $\mathcal{T}\ P$. *tF* t}›
by *simp* (*metis is-processT3-TR-append map-event*$_{ptick}$*-tickFree*)
moreover have ‹*u1* ∈ {t ∈ $\mathcal{T}\ P$. *tF* t}› **by** (*simp add*: ∗(*2*) ∗(*3*) *D-T*)
ultimately have ‹*t1* = *u1*›
by (*intro inj-on-map-map-event*$_{ptick}$*-T-tickFree*[*THEN inj-onD*, *OF* ‹*inj-on* $f\ \alpha(P)$›])
with ∗(*2*, *3*) ∗∗(*1*) ‹t @ [✓(r)] ∈ $\mathcal{T}\ P$› ‹$t \notin \mathcal{D}\ P$› **show** *False*
using *T-imp-front-tickFree front-tickFree-dw-closed*
front-tickFree-nonempty-append-imp is-processT7 **by** *simp blast*
qed
ultimately show ‹s ∈ ✓**s**(*Renaming* $P\ f\ g$)› **by** (*simp add*: *strict-ticks-of-memI*)
qed
qed

lemma *strict-ticks-of-Seq-subset* : ‹✓**s**(P **;** Q) ⊆ (*if* ✓**s**(P) = {} *then* {} *else* ✓**s**(Q))›
proof (*rule subsetI*, *elim strict-ticks-of-memE*)

show ‹t @ [✓(r)] ∈ $\mathcal{T}$ (P ; Q) ⟹ $t \notin \mathcal{D}$ (P ; Q) ⟹
$r \in$ (*if* ✓**s**(P) = {} *then* {} *else* ✓**s**(Q))› **for** r t
by (*simp add*: *Seq-projs strict-ticks-of-def*)
(*metis* (*no-types*, *lifting*) *T-imp-front-tickFree T-nonTickFree-imp-decomp append-T-imp-tickFree butlast-append butlast-snoc is-processT7 is-processT9 last-appendR last-snoc non-tickFree-tick tickFree-Nil tickFree-append-iff*)
qed

lemma *strict-ticks-of-Sync-subset* : ‹✓**s**(P ⟦S⟧ Q) ⊆ ✓**s**(P) ∩ ✓**s**(Q)›
proof (*rule subsetI*)
fix r **assume** ‹$r \in$ ✓**s**(P ⟦S⟧ Q)›
then obtain t **where** ‹t @ [✓(r)] ∈ $\mathcal{T}$ (P ⟦S⟧ Q)› ‹t @ [✓(r)] ∉ $\mathcal{D}$ (P ⟦S⟧ Q)› ‹$t \notin \mathcal{D}$ (P ⟦S⟧ Q)›
by (*meson strict-ticks-of-memE is-processT9*)
from *this*(*1*, *2*) **obtain** *t-P t-Q*
where ‹*t-P* ∈ $\mathcal{T}$ P› ‹*t-Q* ∈ $\mathcal{T}$ Q›
‹(t @ [✓(r)]) *setinterleaves* ((*t-P*, *t-Q*), *range tick* ∪ *ev* ‘ S)›
unfolding *Sync-projs* **by** *blast*
then obtain *t-P′ t-Q′*
where ∗ : ‹*t-P* = *t-P′* @ [✓(r)]› ‹*t-Q* = *t-Q′* @ [✓(r)]› ‹*t-P′* ∈ $\mathcal{T}$ P› ‹*t-Q′* ∈ $\mathcal{T}$ Q›
‹t *setinterleaves* ((*t-P′*, *t-Q′*), *range tick* ∪ *ev* ‘ S)›
by (*metis SyncWithTick-imp-NTF T-imp-front-tickFree is-processT3-TR-append*)
have ‹*t-P′* ∉ $\mathcal{D}$ P›
proof (*rule ccontr*)
assume ‹¬ *t-P′* ∉ $\mathcal{D}$ P›
with ∗(*4*, *5*) **have** ‹$t \in \mathcal{D}$ (P ⟦S⟧ Q)›
by (*simp add*: *D-Sync*) (*use front-tickFree-Nil* **in** *blast*)
with ‹$t \notin \mathcal{D}$ (P ⟦S⟧ Q)› **show** *False* **..**
qed
moreover have ‹*t-Q′* ∉ $\mathcal{D}$ Q›
proof (*rule ccontr*)
assume ‹¬ *t-Q′* ∉ $\mathcal{D}$ Q›
with ∗(*3*, *5*) **have** ‹$t \in \mathcal{D}$ (P ⟦S⟧ Q)›
by (*simp add*: *D-Sync*) (*use front-tickFree-Nil setinterleaving-sym* **in** *blast*)
with ‹$t \notin \mathcal{D}$ (P ⟦S⟧ Q)› **show** *False* **..**
qed
ultimately show ‹$r \in$ ✓**s**(P) ∩ ✓**s**(Q)›
using ∗(*1*, *2*) ‹*t-P* ∈ $\mathcal{T}$ P› ‹*t-Q* ∈ $\mathcal{T}$ Q›
by (*meson IntI is-processT9 strict-ticks-of-memI*)
qed

Chapter 14

CSP Assertions

14.1 Reference Processes

definition *DF* :: ‹*'a set* ⇒ (*'a*, *'r*) *process*$_{ptick}$›
where ‹*DF A* ≡ μ *X*. □*a*∈*A* → *X*›

lemma *DF-unfold* : ‹*DF A* = □*a*∈*A* → *DF A*›
by (*simp add*: *DF-def*, *rule trans*, *rule fix-eq*, *simp*)

definition *DF*$_{SKIPS}$ :: ‹*'a set* ⇒ *'r set* ⇒ (*'a*,*'r*) *process*$_{ptick}$›
where ‹*DF*$_{SKIPS}$ *A R* ≡ μ *X*. (□*a* ∈ *A* → *X*) □ *SKIPS R*›

lemma *DF*$_{SKIPS}$*-unfold* : ‹*DF*$_{SKIPS}$ *A R* = (□*a* ∈ *A* → *DF*$_{SKIPS}$ *A R*) □ *SKIPS R*›
by (*simp add*: *DF*$_{SKIPS}$*-def*, *rule trans*, *rule fix-eq*, *simp*)

definition *RUN* :: ‹*'a set* ⇒ (*'a*,*'r*) *process*$_{ptick}$›
where ‹*RUN A* ≡ μ *X*. □*x*∈*A* → *X*›

lemma *RUN-unfold* : ‹*RUN A* = □*a*∈*A* → *RUN A*›
by (*simp add*: *RUN-def*, *rule trans*, *rule fix-eq*, *simp*)

definition *CHAOS* :: ‹*'a set* ⇒ (*'a*,*'r*) *process*$_{ptick}$›
where ‹*CHAOS A* ≡ μ *X*. *STOP* □ (□*a* ∈ *A* → *X*)›

lemma *CHAOS-unfold* : ‹*CHAOS A* = *STOP* □ (□*a*∈*A* → *CHAOS A*)›
by (*simp add*: *CHAOS-def*, *rule trans*, *rule fix-eq*, *simp*)

definition *CHAOS*$_{SKIPS}$ :: ‹*'a set* ⇒ *'r set* ⇒ (*'a*,*'r*) *process*$_{ptick}$›
where ‹*CHAOS*$_{SKIPS}$ *A R* ≡ μ *X*. *SKIPS R* □ *STOP* □ (□*a*∈*A* → *X*)›

lemma *CHAOS$_{SKIPS}$-unfold*: ‹*CHAOS$_{SKIPS}$ A R = SKIPS R* ⊓ *STOP* ⊓ (□*a* ∈ *A* → *CHAOS$_{SKIPS}$ A R*)›
by (*simp add*: *CHAOS$_{SKIPS}$-def*, *rule trans*, *rule fix-eq*, *simp*)

14.2 Assertions

definition *deadlock-free* :: ‹('*a*,'*r*) *process$_{ptick}$* ⇒ *bool*›
where ‹*deadlock-free P* ≡ *DF UNIV* ⊑$_{FD}$ *P*›

definition *deadlock-free$_{SKIPS}$* :: ‹('*a*, '*r*) *process$_{ptick}$* ⇒ *bool*›
where ‹*deadlock-free$_{SKIPS}$ P* ≡ *DF$_{SKIPS}$ UNIV UNIV* ⊑$_{F}$ *P*›

definition *lifelock-free* :: ‹('*a*,'*r*) *process$_{ptick}$* ⇒ *bool*›
where ‹*lifelock-free P* ≡ *CHAOS UNIV* ⊑$_{FD}$ *P*›

definition *lifelock-free$_{SKIPS}$* :: ‹('*a*,'*r*) *process$_{ptick}$* ⇒ *bool*›
where ‹*lifelock-free$_{SKIPS}$ P* ≡ *CHAOS$_{SKIPS}$ UNIV UNIV* ⊑$_{FD}$ *P*›

definition *non-terminating* :: ‹('*a*,'*r*) *process$_{ptick}$* ⇒ *bool*›
where ‹*non-terminating P* ≡ *RUN UNIV* ⊑$_{T}$ *P*›

14.3 Properties

lemma *DF$_{SKIPS}$-FD-DF* : ‹*DF$_{SKIPS}$ A R* ⊑$_{FD}$ *DF A*›
proof (*subst DF$_{SKIPS}$-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* ⊑$_{FD}$ *DF A*)› **by** *simp*
next
show ‹⊥ ⊑$_{FD}$ *DF A*› **by** *simp*
next
show ‹*X* ⊑$_{FD}$ *DF A* ⟹ (Λ *X*. (⊓*a*∈*A* → *X*) ⊓ *SKIPS R*)·*X* ⊑$_{FD}$ *DF A*› **for** *X*
by (*subst DF-unfold*, *simp*)
(*meson Ndet-FD-self-left mono-Mndetprefix-FD trans-FD*)
qed

lemma *SKIPS-FD-SKIPS-iff* :
‹*SKIPS S* ⊑$_{FD}$ *SKIPS R* ⟷ (*if R* = {} *then S* = {} *else R* ⊆ *S*)›
by (*auto simp add*: *failure-divergence-refine-def failure-refine-def divergence-refine-def F-SKIPS D-SKIPS*)

lemma *SKIPS-F-SKIPS-iff* :
‹*SKIPS S* ⊑$_{F}$ *SKIPS R* ⟷ (*if R* = {} *then S* = {} *else R* ⊆ *S*)›

by (*auto simp add*: *failure-refine-def F-SKIPS*)

lemma *SKIPS-T-SKIPS-iff* : ‹*SKIPS S* $\sqsubseteq_T$ *SKIPS R* $\longleftrightarrow$ (*R* $\subseteq$ *S*)›
by (*auto simp add*: *trace-refine-def T-SKIPS subset-iff*)

lemma *DF-subset* : ‹*DF B* $\sqsubseteq_{FD}$ *DF A*› **if** ‹*A* $\neq$ {}› ‹*A* $\subseteq$ *B*› **for** *A B* :: ‹'*a set*›
proof (*subst DF-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *DF A*)› **by** *simp*
next
show ‹⊥ $\sqsubseteq_{FD}$ *DF A*› **by** *simp*
next
show ‹*X* $\sqsubseteq_{FD}$ *DF A* $\Longrightarrow$ (Λ *X*. ⊓*b*∈*B* → *X*)·*X* $\sqsubseteq_{FD}$ *DF A*› **for** *X* :: ‹('*a*, '*r*) *process*$_{ptick}$›
by (*subst DF-unfold*, *simp*)
(*meson* ‹*A* $\neq$ {}› ‹*A* $\subseteq$ *B*› *Mndetprefix-FD-subset mono-Mndetprefix-FD trans-FD*)
qed

lemma *DF-Univ-freeness* : ‹*A* $\neq$ {} $\Longrightarrow$ *DF A* $\sqsubseteq_{FD}$ *P* $\Longrightarrow$ *deadlock-free P*›
by (*meson DF-subset deadlock-free-def subset-UNIV trans-FD*)

lemma *deadlock-free-Ndet-iff* :
‹*deadlock-free* (*P* ⊓ *Q*) $\longleftrightarrow$ *deadlock-free P* ∧ *deadlock-free Q*›
by (*auto simp add*: *F-Ndet D-Ndet deadlock-free-def refine-defs*)

lemma *DF*$_{SKIPS}$*-subset* : ‹*DF*$_{SKIPS}$ *B S* $\sqsubseteq_{FD}$ *DF*$_{SKIPS}$ *A R*›
if ‹*A* $\neq$ {}› ‹*A* $\subseteq$ *B*› ‹*R* $\neq$ {}› ‹*R* $\subseteq$ *S*›
proof (*subst DF*$_{SKIPS}$*-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *DF*$_{SKIPS}$ *A R*)› **by** *simp*
next
show ‹⊥ $\sqsubseteq_{FD}$ *DF*$_{SKIPS}$ *A R*› **by** *simp*
next
show ‹(Λ *X*. (⊓*b*∈*B* → *X*) ⊓ *SKIPS S*)·*X* $\sqsubseteq_{FD}$ *DF*$_{SKIPS}$ *A R*› **if** ‹*X* $\sqsubseteq_{FD}$ *DF*$_{SKIPS}$ *A R*› **for** *X*
proof (*subst DF*$_{SKIPS}$*-unfold*, *subst beta-cfun*)
show ‹*cont* (λ*X*. (⊓*b*∈*B* → *X*) ⊓ *SKIPS S*)› **by** *simp*
next
show ‹(⊓*b*∈*B* → *X*) ⊓ *SKIPS S* $\sqsubseteq_{FD}$ (⊓*a*∈*A* → *DF*$_{SKIPS}$ *A R*) ⊓ *SKIPS R*›
proof (*rule mono-Ndet-FD*)
show ‹⊓*b*∈*B* → *X* $\sqsubseteq_{FD}$ ⊓*a*∈*A* → *DF*$_{SKIPS}$ *A R*›
by (*meson Mndetprefix-FD-subset* ‹*A* $\neq$ {}› ‹*A* $\subseteq$ *B*›
mono-Mndetprefix-FD ‹*X* $\sqsubseteq_{FD}$ *DF*$_{SKIPS}$ *A R*› *trans-FD*)
next
show ‹*SKIPS S* $\sqsubseteq_{FD}$ *SKIPS R*›
by (*simp add*: *SKIPS-FD-SKIPS-iff* ‹*R* $\neq$ {}› ‹*R* $\subseteq$ *S*›)

qed
qed
qed

lemma *DF_{SKIPS}-Univ-freeness* : ‹$[\![A \neq \{\}; R \neq \{\}; DF_{SKIPS}\ A\ R \sqsubseteq_{FD} P]\!] \Longrightarrow$ *deadlock-free$_{SKIPS}$* P›
by (*meson DF_{SKIPS}-subset deadlock-free$_{SKIPS}$-def leFD-imp-leF subset-UNIV trans-F*)

lemma *deadlock-free$_{SKIPS}$-Ndet-iff* :
‹*deadlock-free$_{SKIPS}$* $(P \sqcap Q) \longleftrightarrow$ *deadlock-free$_{SKIPS}$* $P \wedge$ *deadlock-free$_{SKIPS}$* Q›
by (*simp add*: *F-Ndet deadlock-free$_{SKIPS}$-def failure-refine-def*)

lemma *div-free-DF_{SKIPS}*: ‹$\mathcal{D}\ (DF_{SKIPS}\ A\ R) = \{\}$›
proof (*rule equals0I*)
show ‹$s \in \mathcal{D}\ (DF_{SKIPS}\ A\ R) \Longrightarrow$ *False*› **for** s
by (*induct s*; *subst* (*asm*) *DF_{SKIPS}-unfold*)
(*auto simp add*: *D-Ndet D-SKIPS D-Mndetprefix write0-def D-Mprefix split*: *if-split-asm*)
qed

lemma *div-free-DF*: ‹$\mathcal{D}\ (DF\ A) = \{\}$›
by (*metis DF_{SKIPS}-FD-DF div-free-DF_{SKIPS} empty-subsetI subset-antisym le-ref1*)

lemma *deadlock-free-implies-div-free*: ‹*deadlock-free* $P \Longrightarrow \mathcal{D}\ P = \{\}$›
by (*simp add*: *deadlock-free-def div-free-DF refine-defs*)

14.4 Events and Ticks of Reference Processes

lemma *events-of-SKIPS* : ‹$\alpha(SKIPS\ R) = \{\}$›
and *ticks-of-SKIPS* : ‹$\checkmark s(SKIPS\ R) = R$›
by (*auto simp add*: *events-of-def ticks-of-def T-SKIPS*)

lemma *no-ticks-imp-tickFree-T* : ‹$\checkmark s(P) = \{\} \Longrightarrow s \in \mathcal{T}\ P \Longrightarrow tF\ s$›
by (*simp add*: *ticks-of-def tickFree-def disjoint-iff image-iff*)
(*metis T-nonTickFree-imp-decomp event$_{ptick}$.disc*(2) *split-list tickFree-Cons-iff tickFree-append-iff*)

lemma *events-of-DF* : ‹$\alpha(DF\ A) = A$›
proof (*intro subset-antisym subsetI*)
fix a **assume** ‹$a \in A$›

hence ‹$[ev\ a] \in \mathcal{T}\ (DF\ A)$›
by (*subst DF-unfold*) (*auto simp add*: *T-Mndetprefix write0-def T-Mprefix*)
thus ‹$a \in \alpha(DF\ A)$› **by** (*force simp add*: *events-of-def*)
next
fix a **assume** ‹$a \in \alpha((DF\ A) :: ('a, 'r)\ process_{ptick})$›
then obtain $t :: ('a, 'r)\ trace_{ptick}$ **where** ‹$ev\ a \in set\ t$› ‹$t \in \mathcal{T}\ (DF\ A)$›
by (*auto simp add*: *events-of-def*)
thus ‹$a \in A$›
by (*induct t*, *simp*, *subst* (*asm*) *DF-unfold*)
(*auto simp add*: *T-Mndetprefix write0-def T-Mprefix split*: *if-split-asm*)
qed

lemma *ticks-DF* : ‹$\checkmark s(DF\ A) = \{\}$›
proof (*rule equals0I*)
fix $r :: 'r$ **assume** ‹$r \in \checkmark s(DF\ A)$›
then obtain s **where** ‹$s\ @\ [\checkmark(r)] \in \mathcal{T}\ (DF\ A)$› **unfolding** *ticks-of-def* **by** *blast*
thus *False*
by (*induct s*; *subst* (*asm*) *DF-unfold*)
(*simp-all add*: *T-Mndetprefix write0-def T-Mprefix split*: *if-split-asm*)
qed

lemma *events-of-DF$_{SKIPS}$* : ‹$\alpha(DF_{SKIPS}\ A\ R) = A$›
proof (*intro subset-antisym subsetI*)
fix a **assume** ‹$a \in A$›
hence ‹$[ev\ a] \in \mathcal{T}\ (DF_{SKIPS}\ A\ R)$›
by (*subst DF$_{SKIPS}$-unfold*) (*auto simp add*: *T-Ndet T-Mndetprefix write0-def T-Mprefix*)
thus ‹$a \in \alpha(DF_{SKIPS}\ A\ R)$› **by** (*force simp add*: *events-of-def*)
next
fix a **assume** ‹$a \in \alpha(DF_{SKIPS}\ A\ R)$›
then obtain t **where** ‹$ev\ a \in set\ t$› ‹$t \in \mathcal{T}\ (DF_{SKIPS}\ A\ R)$›
by (*auto simp add*: *events-of-def*)
thus ‹$a \in A$›
by (*induct t*, *simp*, *subst* (*asm*) *DF$_{SKIPS}$-unfold*)
(*auto simp add*: *T-Ndet T-SKIPS T-Mndetprefix write0-def T-Mprefix split*: *if-split-asm*)
qed

lemma *ticks-DF$_{SKIPS}$* : ‹$\checkmark s(DF_{SKIPS}\ A\ R) = R$›
proof (*intro subset-antisym subsetI*)
fix r **assume** ‹$r \in R$›
hence ‹$[\checkmark(r)] \in \mathcal{T}\ (DF_{SKIPS}\ A\ R)$› **by** (*subst DF$_{SKIPS}$-unfold*) (*simp add*: *T-Ndet T-SKIPS*)
thus ‹$r \in \checkmark s(DF_{SKIPS}\ A\ R)$›
unfolding *ticks-of-def* **by** (*metis* (*no-types*, *lifting*) *CollectI append-self-conv2*)
next
fix r **assume** ‹$r \in \checkmark s(DF_{SKIPS}\ A\ R)$›
then obtain s **where** ‹$s\ @\ [\checkmark(r)] \in \mathcal{T}\ (DF_{SKIPS}\ A\ R)$› **unfolding** *ticks-of-def*

by *blast*
thus ‹$r \in R$›
by (*induct s*; *subst* (*asm*) DF_{SKIPS}-*unfold*)
(*simp-all add*: *T-Ndet T-SKIPS T-Mndetprefix write0-def T-Mprefix split*: *if-split-asm*)
qed

lemma *events-of-RUN* : ‹$\alpha(RUN\ A) = A$›
proof (*intro subset-antisym subsetI*)
fix *a* **assume** ‹$a \in A$›
hence ‹$[ev\ a] \in \mathcal{T}\ (RUN\ A)$›
by (*subst RUN-unfold*) (*auto simp add*: *T-Mprefix*)
thus ‹$a \in \alpha(RUN\ A)$› **by** (*force simp add*: *events-of-def*)
next
fix *a* **assume** ‹$a \in \alpha((RUN\ A) :: ('a, 'r)\ process_{ptick})$›
then obtain $t :: ('a, 'r)\ trace_{ptick}$ **where** ‹$ev\ a \in set\ t$› ‹$t \in \mathcal{T}\ (RUN\ A)$›
by (*auto simp add*: *events-of-def*)
thus ‹$a \in A$›
by (*induct t*, *simp*, *subst* (*asm*) *RUN-unfold*) (*auto simp add*: *T-Mprefix*)
qed

lemma *ticks-RUN* : ‹$\checkmark s(RUN\ A) = \{\}$›
proof (*rule equals0I*)
fix $r :: 'r$ **assume** ‹$r \in \checkmark s(RUN\ A)$›
then obtain *s* **where** ‹$s\ @\ [\checkmark(r)] \in \mathcal{T}\ (RUN\ A)$› **unfolding** *ticks-of-def* **by** *blast*
thus *False*
by (*induct s*; *subst* (*asm*) *RUN-unfold*)
(*auto simp add*: *T-Mprefix split*: *if-split-asm*)
qed

lemma *events-of-CHAOS* : ‹$\alpha(CHAOS\ A) = A$›
proof (*intro subset-antisym subsetI*)
fix *a* **assume** ‹$a \in A$›
hence ‹$[ev\ a] \in \mathcal{T}\ (CHAOS\ A)$›
by (*subst CHAOS-unfold*) (*auto simp add*: *T-Ndet T-Mprefix*)
thus ‹$a \in \alpha(CHAOS\ A)$› **by** (*force simp add*: *events-of-def*)
next
fix *a* **assume** ‹$a \in \alpha((CHAOS\ A) :: ('a, 'r)\ process_{ptick})$›
then obtain $t :: ('a, 'r)\ trace_{ptick}$ **where** ‹$ev\ a \in set\ t$› ‹$t \in \mathcal{T}\ (CHAOS\ A)$›
by (*auto simp add*: *events-of-def*)
thus ‹$a \in A$›
by (*induct t*, *simp*, *subst* (*asm*) *CHAOS-unfold*) (*auto simp add*: *T-Ndet T-STOP T-Mprefix*)

qed

lemma *ticks-CHAOS* : ‹✓*s*(*CHAOS A*) = {}›
proof (*rule equals0I*)
fix *r* :: *'r* **assume** ‹*r* ∈ ✓*s*(*CHAOS A*)›
then obtain *s* **where** ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ (*CHAOS A*)› **unfolding** *ticks-of-def* **by** *blast*
thus *False*
by (*induct s*; *subst* (*asm*) *CHAOS-unfold*)
(*auto simp add*: *T-Ndet T-STOP T-Mprefix split*: *if-split-asm*)
qed

lemma *events-of-CHAOS*$_{SKIPS}$: ‹*α*(*CHAOS*$_{SKIPS}$ *A R*) = *A*›
proof (*intro subset-antisym subsetI*)
fix *a* **assume** ‹*a* ∈ *A*›
hence ‹[*ev a*] ∈ $\mathcal{T}$ (*CHAOS*$_{SKIPS}$ *A R*)›
by (*subst CHAOS*$_{SKIPS}$*-unfold*) (*auto simp add*: *T-Ndet T-Mprefix*)
thus ‹*a* ∈ *α*(*CHAOS*$_{SKIPS}$ *A R*)› **by** (*force simp add*: *events-of-def*)
next
fix *a* **assume** ‹*a* ∈ *α*(*CHAOS*$_{SKIPS}$ *A R*)›
then obtain *t* **where** ‹*ev a* ∈ *set t*› ‹*t* ∈ $\mathcal{T}$ (*CHAOS*$_{SKIPS}$ *A R*)›
by (*auto simp add*: *events-of-def*)
thus ‹*a* ∈ *A*›
by (*induct t*, *simp*, *subst* (*asm*) *CHAOS*$_{SKIPS}$*-unfold*)
(*auto simp add*: *T-Ndet T-STOP T-SKIPS T-Mprefix*)
qed

lemma *ticks-CHAOS*$_{SKIPS}$: ‹✓*s*(*CHAOS*$_{SKIPS}$ *A R*) = *R*›
proof (*intro subset-antisym subsetI*)
fix *r* **assume** ‹*r* ∈ *R*›
hence ‹[✓(*r*)] ∈ $\mathcal{T}$ (*CHAOS*$_{SKIPS}$ *A R*)› **by** (*subst CHAOS*$_{SKIPS}$*-unfold*) (*simp add*: *T-Ndet T-SKIPS*)
thus ‹*r* ∈ ✓*s*(*CHAOS*$_{SKIPS}$ *A R*)› **by** (*metis append-Nil ticks-of-memI*)
next
fix *r* **assume** ‹*r* ∈ ✓*s*(*CHAOS*$_{SKIPS}$ *A R*)›
then obtain *s* **where** ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ (*CHAOS*$_{SKIPS}$ *A R*)› **unfolding** *ticks-of-def* **by** *blast*
thus ‹*r* ∈ *R*›
by (*induct s*; *subst* (*asm*) *CHAOS*$_{SKIPS}$*-unfold*)
(*auto simp add*: *T-Ndet T-STOP T-SKIPS T-Mprefix*)
qed

lemma *RUN-subset-DT*: ‹*RUN B* ⊑$_{DT}$ *RUN A*›
if ‹*A* ⊆ *B*› **for** *A B* :: ‹*'a set*›
proof (*subst RUN-def*, *induct rule*: *fix-ind*)
show ‹*adm* (*λa*. *a* ⊑$_{DT}$ *RUN A*)› **by** *simp*
next

show ‹$\bot \sqsubseteq_{DT} RUN\ A$› **by** *simp*
next
show ‹$X \sqsubseteq_{DT} RUN\ A \Longrightarrow (\Lambda\ X.\ \Box b{\in}B \rightarrow X){\cdot}X \sqsubseteq_{DT} RUN\ A$› **for** X :: ‹('a, 'r) $process_{ptick}$›
by (*subst RUN-unfold*, *simp*)
(*meson Mprefix-DT-subset mono-Mprefix-DT* ‹$A \subseteq B$› *trans-DT*)
qed

lemma *CHAOS-subset-FD* : ‹$CHAOS\ B \sqsubseteq_{FD} CHAOS\ A$›
if ‹$A \subseteq B$› **for** $A\ B$ :: ‹'a *set*›
proof (*subst CHAOS-def*, *induct rule*: *fix-ind*)
show ‹*adm* ($\lambda a.\ a \sqsubseteq_{FD} CHAOS\ A$)› **by** *simp*
next
show ‹$\bot \sqsubseteq_{FD} CHAOS\ A$› **by** *simp*
next
from ‹$A \subseteq B$› **show** ‹$X \sqsubseteq_{FD} CHAOS\ A \Longrightarrow (\Lambda\ X.\ STOP \sqcap (\Box b{\in}B \rightarrow X)){\cdot}X \sqsubseteq_{FD} CHAOS\ A$›
for X :: ‹('a, 'r) $process_{ptick}$›
by (*subst CHAOS-unfold*)
(*auto simp add*: *refine-defs Mprefix-projs Ndet-projs F-STOP D-STOP*)
qed

lemma $CHAOS_{SKIPS}$*-subset-FD* : ‹$CHAOS_{SKIPS}\ B\ S \sqsubseteq_{FD} CHAOS_{SKIPS}\ A\ R$›
if ‹$A \subseteq B$› ‹$R \neq \{\}$› ‹$R \subseteq S$›
proof (*subst* $CHAOS_{SKIPS}$*-def*, *induct rule*: *fix-ind*)
show ‹*adm* ($\lambda a.\ a \sqsubseteq_{FD} CHAOS_{SKIPS}\ A\ R$)› **by** *simp*
next
show ‹$\bot \sqsubseteq_{FD} CHAOS_{SKIPS}\ A\ R$› **by** *simp*
next
from ‹$A \subseteq B$› ‹$R \neq \{\}$› ‹$R \subseteq S$›
show ‹$X \sqsubseteq_{FD} CHAOS_{SKIPS}\ A\ R \Longrightarrow (\Lambda\ X.\ SKIPS\ S \sqcap STOP \sqcap (\Box a{\in}B \rightarrow X)){\cdot}X \sqsubseteq_{FD} CHAOS_{SKIPS}\ A\ R$› **for** X
by (*subst* $CHAOS_{SKIPS}$*-unfold*)
(*auto simp add*: *refine-defs Mprefix-projs Ndet-projs F-STOP D-STOP F-SKIPS D-SKIPS*)
qed

14.5 Relations between refinements on reference processes

lemma *CHAOS-has-all-tickFree-failures* :
‹$tF\ s \Longrightarrow \{a.\ ev\ a \in set\ s\} \subseteq A \Longrightarrow (s, X) \in \mathcal{F}\ (CHAOS\ A)$›
proof (*induct s*)
show ‹$([], X) \in \mathcal{F}\ (CHAOS\ A)$›
by (*subst CHAOS-unfold*) (*simp add*: *F-Ndet F-STOP*)

next
fix *e s*
assume *hyp* : ‹*tF s* ⟹ {*a*. *ev a* ∈ *set s*} ⊆ *A* ⟹ (*s*, *X*) ∈ ℱ (*CHAOS A*)›
assume *prems* : ‹*tF* (*e* # *s*)› ‹{*a*. *ev a* ∈ *set* (*e* # *s*)} ⊆ *A*›
from *prems* **have** ‹*tF s*› ‹{*a*. *ev a* ∈ *set s*} ⊆ *A*› **by** *auto*
hence ‹(*s*, *X*) ∈ ℱ (*CHAOS A*)› **by** (*fact hyp*)
with *prems* **show** ‹(*e* # *s*, *X*) ∈ ℱ (*CHAOS A*)›
by (*subst CHAOS-unfold*, *cases e*) (*auto simp add*: *F-Ndet F-Mprefix*)
qed

lemma *CHAOS*$_{SKIPS}$*-superset-events-of-ticks-of-leF* :
‹*CHAOS*$_{SKIPS}$ *A R* ⊑$_F$ *P*› **if** ‹*α*(*P*) ⊆ *A*› **and** ‹✓*s*(*P*) ⊆ *R*›
proof (*unfold failure-refine-def*)
have ‹*ftF s* ⟹ *set s* ⊆ (⋃ *t*∈*𝒯 P*. *set t*) ⟹ (*s*, *X*) ∈ ℱ (*CHAOS*$_{SKIPS}$ *A R*)›
for *s X*
proof (*induct s*)
show ‹([], *X*) ∈ ℱ (*CHAOS*$_{SKIPS}$ *A R*)›
by (*subst CHAOS*$_{SKIPS}$*-unfold*) (*simp add*: *F-Ndet F-STOP*)
next
fix *e s*
assume *prems* : ‹*ftF* (*e* # *s*)› ‹*set* (*e* # *s*) ⊆ ⋃ (*set* ‘ *𝒯 P*)›
assume *hyp* : ‹*ftF s* ⟹ *set s* ⊆ ⋃ (*set* ‘ *𝒯 P*) ⟹ (*s*, *X*) ∈ ℱ (*CHAOS*$_{SKIPS}$ *A R*)›
show ‹(*e* # *s*, *X*) ∈ ℱ (*CHAOS*$_{SKIPS}$ *A R*)›
proof (*cases e*)
fix *a* **assume** ‹*e* = *ev a*›
with *prems*(*2*) **have** ‹*a* ∈ *α*(*P*)› **by** (*auto simp add*: *events-of-def*)
with ‹*α*(*P*) ⊆ *A*› **have** ‹*a* ∈ *A*› **by** *fast*
from *prems* **have** ‹*ftF s*› ‹*set s* ⊆ ⋃ (*set* ‘ *𝒯 P*)›
by *auto* (*metis front-tickFree-Cons-iff front-tickFree-Nil*)
hence ‹(*s*, *X*) ∈ ℱ (*CHAOS*$_{SKIPS}$ *A R*)› **by** (*fact hyp*)
thus ‹(*e* # *s*, *X*) ∈ ℱ (*CHAOS*$_{SKIPS}$ *A R*)›
by (*subst CHAOS*$_{SKIPS}$*-unfold*) (*simp add*: *F-Ndet F-Mprefix* ‹*e* = *ev a*› ‹*a* ∈ *A*›)
next
fix *r* **assume** ‹*e* = ✓(*r*)›
with *prems*(*2*) **have** ‹*r* ∈ ✓*s*(*P*)›
by (*simp add*: *ticks-of-def*)
(*metis T-imp-front-tickFree front-tickFree-Cons-iff front-tickFree-append-iff in-set-conv-decomp non-tickFree-tick tickFree-Cons-iff tickFree-Nil*)
with ‹✓*s*(*P*) ⊆ *R*› **have** ‹*r* ∈ *R*› **by** *fast*
moreover from ‹*e* = ✓(*r*)› *prems*(*1*) **have** ‹*s* = []› **by** (*simp add*: *front-tickFree-Cons-iff*)
ultimately show ‹(*e* # *s*, *X*) ∈ ℱ (*CHAOS*$_{SKIPS}$ *A R*)›
by (*subst CHAOS*$_{SKIPS}$*-unfold*) (*auto simp add*: *F-Ndet F-SKIPS* ‹*e* = ✓(*r*)›)
qed
qed

thus ‹$\mathcal{F}\ P \subseteq \mathcal{F}\ (CHAOS_{SKIPS}\ A\ R)$› **by** (*meson F-T F-imp-front-tickFree SUP-upper subrelI*)
qed

corollary $CHAOS_{SKIPS}$*-events-of-ticks-of-leF*: ‹$CHAOS_{SKIPS}\ \alpha(P)\ \checkmark s(P) \sqsubseteq_F P$›
and $CHAOS_{SKIPS}$*-UNIV-UNIV-leF*: ‹$CHAOS_{SKIPS}\ UNIV\ UNIV \sqsubseteq_F P$›
by (*simp-all add*: $CHAOS_{SKIPS}$*-superset-events-of-ticks-of-leF*)

lemma DF_{SKIPS}*-F-DF* : ‹$DF_{SKIPS}\ A\ R \sqsubseteq_F DF\ A$›
by (*simp add*: DF_{SKIPS}*-FD-DF leFD-imp-leF*)

lemma *DF-F-RUN* : ‹$DF\ A \sqsubseteq_F RUN\ A$› **for** A :: ‹$'a\ set$›
proof (*unfold DF-def*, *induct rule*: *fix-ind*)
show ‹$adm\ (\lambda a.\ a \sqsubseteq_F RUN\ A)$› **by** *simp*
next
show ‹$\bot \sqsubseteq_F RUN\ A$› **by** *simp*
next
show ‹$X \sqsubseteq_F RUN\ A \Longrightarrow (\Lambda\ X.\ \sqcap a{\in}A \rightarrow X){\cdot}X \sqsubseteq_F RUN\ A$› **for** X :: ‹$('a, 'r)\ process_{ptick}$›
by (*subst RUN-unfold*, *simp*)
(*meson Mndetprefix-F-Mprefix mono-Mprefix-F trans-F*)
qed

lemma *CHAOS-F-DF* : ‹$CHAOS\ A \sqsubseteq_F DF\ A$›
proof (*unfold failure-refine-def*, *safe*, *rule CHAOS-has-all-tickFree-failures*)
show ‹$(s, X) \in \mathcal{F}\ (DF\ A) \Longrightarrow tF\ s$› **for** s :: ‹$('a, 'r)\ trace_{ptick}$› **and** X
by (*metis F-T no-ticks-imp-tickFree-T ticks-DF*)
next
show ‹$(s, X) \in \mathcal{F}\ (DF\ A) \Longrightarrow \{a.\ ev\ a \in set\ s\} \subseteq A$› **for** s :: ‹$('a, 'r)\ trace_{ptick}$›
and X
by (*drule F-T*) (*use events-of-DF*[*of A*] **in** ‹*auto simp add*: *events-of-def*›)
qed

corollary $CHAOS_{SKIPS}$*-F-CHAOS* : ‹$CHAOS_{SKIPS}\ A\ R \sqsubseteq_F CHAOS\ A$›
and $CHAOS_{SKIPS}$*-F-*DF_{SKIPS} : ‹$CHAOS_{SKIPS}\ A\ R \sqsubseteq_F DF_{SKIPS}\ A\ R$›
by (*rule* $CHAOS_{SKIPS}$*-superset-events-of-ticks-of-leF*;
simp add: *events-of-CHAOS ticks-CHAOS events-of-*DF_{SKIPS} *ticks-*DF_{SKIPS})+

lemma *div-free-*$CHAOS_{SKIPS}$: ‹$\mathcal{D}\ (CHAOS_{SKIPS}\ A\ R) = \{\}$›
proof (*rule equals0I*)
show ‹$s \in \mathcal{D}\ (CHAOS_{SKIPS}\ A\ R) \Longrightarrow False$› **for** s
by (*induct s*; *subst* (*asm*) $CHAOS_{SKIPS}$*-unfold*)
(*auto simp add*: *D-Ndet D-STOP D-SKIPS D-Mprefix*)

qed

lemma *div-free-CHAOS*: ‹$\mathcal{D}$ (*CHAOS A*) = {}›
proof (*rule equals0I*)
show ‹$s \in \mathcal{D}$ (*CHAOS A*) $\Longrightarrow$ *False*› **for** s :: ‹('a, 'r) $trace_{ptick}$›
by (*induct s*; *subst* (*asm*) *CHAOS-unfold*)
(*auto simp add*: *D-Ndet D-STOP D-Mprefix*)
qed

lemma *div-free-RUN*: ‹$\mathcal{D}$ (*RUN A*) = {}›
proof (*rule equals0I*)
show ‹$s \in \mathcal{D}$ (*RUN A*) $\Longrightarrow$ *False*› **for** s :: ‹('a, 'r) $trace_{ptick}$›
by (*induct s*; *subst* (*asm*) *RUN-unfold*) (*auto simp add*: *D-Mprefix*)
qed

corollary *DF-FD-RUN* : ‹*DF A* $\sqsubseteq_{FD}$ *RUN A*›
and *CHAOS-FD-DF* : ‹*CHAOS A* $\sqsubseteq_{FD}$ *DF A*›
and $CHAOS_{SKIPS}$*-FD-CHAOS* : ‹$CHAOS_{SKIPS}$ *A R* $\sqsubseteq_{FD}$ *CHAOS A*›
and $CHAOS_{SKIPS}$*-FD-*DF_{SKIPS} : ‹$CHAOS_{SKIPS}$ *A R* $\sqsubseteq_{FD}$ DF_{SKIPS} *A R*›
by (*simp-all add*: *DF-F-RUN div-free-RUN CHAOS-F-DF div-free-DF* $CHAOS_{SKIPS}$*-F-CHAOS div-free-CHAOS*
$CHAOS_{SKIPS}$*-F-*DF_{SKIPS} DF_{SKIPS}*-div-free divergence-refine-def leF-leD-imp-leFD*)

lemma *traces-CHAOS-subset* : ‹$\mathcal{T}$ (*CHAOS A*) $\subseteq$ {*s*. *set s* $\subseteq$ *ev* ' *A*}›
proof (*rule subsetI*)
show ‹$s \in \mathcal{T}$ (*CHAOS A*) $\Longrightarrow$ $s \in$ {*s*. *set s* $\subseteq$ *ev* ' *A*}› **for** s :: ‹('a, 'r) $trace_{ptick}$›
by (*induct s*; *subst* (*asm*) *CHAOS-unfold*) (*auto simp add*: *T-Ndet T-STOP T-Mprefix*)
qed

lemma *traces-RUN-superset* : ‹{*s*. *set s* $\subseteq$ *ev* ' *A*} $\subseteq$ $\mathcal{T}$ (*RUN A*)›
proof (*rule subsetI*)
show ‹$s \in$ {*s*. *set s* $\subseteq$ *ev* ' *A*} $\Longrightarrow$ $s \in \mathcal{T}$ (*RUN A*)› **for** s :: ‹('a, 'r) $trace_{ptick}$›
by (*induct s*, *simp*) (*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-F-RUN*[*THEN leF-imp-leT*, *simplified trace-refine-def*]
CHAOS-F-DF[*THEN leF-imp-leT*,*simplified trace-refine-def*]
traces-CHAOS-subset traces-RUN-superset **by** *blast*+

corollary *RUN-all-tickfree-traces2* : ‹$s \in \mathcal{T}\ (RUN\quad UNIV)$›
and *DF-all-tickfree-traces2* : ‹$s \in \mathcal{T}\ (DF\quad UNIV)$›
and *CHAOS-all-tickfree-trace2* : ‹$s \in \mathcal{T}\ (CHAOS\ UNIV)$› **if** ‹*tF s*›
using ‹*tF s*›
by (*simp add*: *tickFree-def RUN-all-tickfree-traces1 DF-all-tickfree-traces1 CHAOS-all-tickfree-traces1 disjoint-iff image-iff subset-iff*; *metis event*$_{ptick}$*.exhaust*)+

lemma *traces-CHAOS*$_{SKIPS}$*-subset* :
‹$\mathcal{T}\ (CHAOS_{SKIPS}\ A\ R) \subseteq \{s.\ ftF\ s \wedge set\ s \subseteq ev\ `\ A \cup tick\ `\ R\}$›
proof (*rule subsetI*)
show ‹$s \in \mathcal{T}\ (CHAOS_{SKIPS}\ A\ R) \Longrightarrow s \in \{s.\ ftF\ s \wedge set\ s \subseteq ev\ `\ A \cup tick\ `\ R\}$› **for** *s*
by (*induct s*; *subst* (*asm*) *CHAOS*$_{SKIPS}$*-unfold*)
(*auto simp add*: *T-Ndet T-STOP T-SKIPS T-Mprefix front-tickFree-Cons-iff*)
qed

lemma *traces-DF*$_{SKIPS}$*-superset* :
‹$\{s.\ ftF\ s \wedge set\ s \subseteq ev\ `\ A \cup tick\ `\ R\} \subseteq \mathcal{T}\ (DF_{SKIPS}\ A\ R)$›
proof (*rule subsetI*)
show ‹$s \in \{s.\ ftF\ s \wedge set\ s \subseteq ev\ `\ A \cup tick\ `\ R\} \Longrightarrow s \in \mathcal{T}\ (DF_{SKIPS}\ A\ R)$ ›
for *s*
by (*induct s*; *subst DF*$_{SKIPS}$*-unfold*)
(*auto simp add*: *T-Ndet T-SKIPS T-Mndetprefix write0-def T-Mprefix front-tickFree-Cons-iff*)
qed

corollary *DF*$_{SKIPS}$*-all-front-tickfree-traces1*:
‹$\mathcal{T}\ (DF_{SKIPS}\ A\ R) = \{s.\ ftF\ s \wedge set\ s \subseteq ev\ `\ A \cup tick\ `\ R\}$›
and *CHAOS*$_{SKIPS}$*-all-front-tickfree-traces1*:
‹$\mathcal{T}\ (CHAOS_{SKIPS}\ A\ R) = \{s.\ ftF\ s \wedge set\ s \subseteq ev\ `\ A \cup tick\ `\ R\}$›
using *CHAOS*$_{SKIPS}$*-F-DF*$_{SKIPS}$[*THEN leF-imp-leT*, *simplified trace-refine-def*]
traces-CHAOS$_{SKIPS}$*-subset traces-DF*$_{SKIPS}$*-superset* **by** *blast*+

corollary *DF*$_{SKIPS}$*-all-front-tickfree-traces2* : ‹$s \in \mathcal{T}\ (DF_{SKIPS}\ UNIV\ UNIV)$›
and *CHAOS*$_{SKIPS}$*-all-front-tickfree-traces2*: ‹$s \in \mathcal{T}\ (CHAOS_{SKIPS}\ UNIV\ UNIV)$›
if ‹*ftF s*›
using ‹*ftF s*›
by (*simp add*: *tickFree-def DF*$_{SKIPS}$*-all-front-tickfree-traces1 CHAOS*$_{SKIPS}$*-all-front-tickfree-traces1*; *metis UnCI event*$_{ptick}$*.exhaust rangeI subsetI*)+

corollary *DF*$_{SKIPS}$*-UNIV-UNIV-leT* : ‹$DF_{SKIPS}\ UNIV\ UNIV \sqsubseteq_T P$›
and *CHAOS*$_{SKIPS}$*-UNIV-UNIV-leT* : ‹$CHAOS_{SKIPS}\ UNIV\ UNIV \sqsubseteq_T P$›
by (*simp add*:*trace-refine-def DF*$_{SKIPS}$*-all-front-tickfree-traces2 is-processT2-TR subsetI*)

(*simp add*:*trace-refine-def CHAOS$_{SKIPS}$-all-front-tickfree-traces2 is-processT2-TR subsetI*)

lemma *deadlock-free-implies-lifelock-free*: ‹*deadlock-free P* ⟹ *lifelock-free P*›
unfolding *deadlock-free-def lifelock-free-def*
using *CHAOS-FD-DF trans-FD* **by** *blast*

lemma *deadlock-free-implies-non-terminating*:
‹*tF s*› **if** ‹*deadlock-free P*› ‹*s* ∈ *T P*›
proof −
from ‹*deadlock-free P*› **have** ‹*T P* ⊆ *T* (*DF UNIV*)›
by (*simp add*: *deadlock-free-def le-ref2T*)
with ‹*s* ∈ *T P*› **show** ‹*tF s*›
by (*meson no-ticks-imp-tickFree-T subsetD ticks-DF*)
qed

lemma *deadlock-free$_{SKIPS}$-is-right*:
‹*deadlock-free$_{SKIPS}$* (*P* :: ('*a*, '*r*) *process$_{ptick}$*) ⟷
(∀ *s*∈*T P*. *tF s* ⟶ (*s*, *UNIV* :: ('*a*, '*r*) *event$_{ptick}$ set*) ∉ *F P*)›
proof (*intro iffI ballI impI*)
have ‹*tF s* ⟹ (*s*, *UNIV*) ∉ *F* (*DF$_{SKIPS}$ UNIV UNIV*)› **for** *s* :: ‹('*a*, '*r*) *event$_{ptick}$ list*›
by (*induct s*; *subst DF$_{SKIPS}$-unfold*)
(*auto simp add*: *F-Ndet F-SKIPS F-Mndetprefix write0-def F-Mprefix*)
thus ‹*deadlock-free$_{SKIPS}$ P* ⟹ *s* ∈ *T P* ⟹ *tF s* ⟹ (*s*, *UNIV*) ∉ *F P*› **for** *s*
using *deadlock-free$_{SKIPS}$-def failure-refine-def* **by** *blast*
next
assume *as1* : ‹∀ *s*∈*T P*. *tF s* ⟶ (*s*, *UNIV*) ∉ *F P*›
have *as2* : ‹*ftF s* ⟹ (∃ *a* ∈ *UNIV*. *ev a* ∉ *X*) ⟹
(*s*, *X*) ∈ *F* (*DF$_{SKIPS}$ UNIV UNIV*)› **for** *s* :: ‹('*a*, '*r*) *trace$_{ptick}$*› **and** *X*
proof(*induct s*)
case *Nil*
then show *?case*
by (*subst DF$_{SKIPS}$-unfold*, *auto simp add*:*F-Mndetprefix write0-def F-Mprefix F-Ndet F-SKIP*)
next
case (*Cons hds tls*)
then show *?case*
proof (*simp add*: *DF$_{SKIPS}$-def fix-def*)
define *Y* **where** *Y* ≡ λ*i*. *iterate i*·(Λ *x*. (⊓*xa*∈(*UNIV* :: '*a set*) → *x*) ⊓ *SKIPS* (*UNIV* :: '*r set*))·⊥
assume *a*:*ftF* (*hds* # *tls*) **and** *b*:*ftF tls* ⟹ (*tls*, *X*) ∈ *F* (⨆ *i*. *Y i*)
and *c*:∃ *a*. *ev a* ∉ *X*
from *Y-def* **have** *cc*:*chain Y* **by** *simp*
from *b* **have** *d*:*ftF tls* ⟹ ∃ *a*∈*UNIV*. *ev a* ∉ *X* ⟹(*tls*, *X*) ∈ *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* ($\lambda x.$ *Y i*) $\sqcap$ *SKIPS UNIV*) $\subseteq \mathcal{F}$ (*Y* (*Suc i*)) **for** *i* **by**(*simp*)
from *a* **have** *f*:*tls* $\neq$ [] $\Longrightarrow$ *hds* $\notin$ *range tick ftF tls*
by (*metis event*$_{ptick}$.*disc*(*2*) *front-tickFree-Cons-iff imageE*)
(*metis a front-tickFree-Cons-iff front-tickFree-Nil*)
have *g*:(*hds*#*tls*, *X*) $\in \mathcal{F}$ (*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-SKIPS image-iff*)
(*meson event*$_{ptick}$.*exhaust*)+
have *h*:(*hds*#*tls*, *X*) $\in \mathcal{F}$ (*Y 0*)
using *D-F cc g po-class.chainE proc-ord2a* **by** *blast*
from *a b c* **show** (*hds*#*tls*, *X*) $\in \mathcal{F}$ ($\bigsqcup i.$ *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 process-charn*)
qed
qed
show *deadlock-free*$_{SKIPS}$ *P*
proof(*unfold deadlock-free*$_{SKIPS}$*-def failure-refine-def*, *safe*)
fix *s X*
assume *as3*:(*s*, *X*) $\in \mathcal{F}$ *P*
hence *a1*:*s* $\in \mathcal{T}$ *P ftF s*
using *F-T as3 is-processT2* **by** *blast*+
show (*s*, *X*) $\in \mathcal{F}$ (*DF*$_{SKIPS}$ *UNIV UNIV*)
proof(*cases tF s*)
case *FT-True*:*True*
hence *a2*:(*s*, *UNIV*) $\notin \mathcal{F}$ *P*
using *a1 as1* **by** *blast*
then show *?thesis*
by (*metis DF*$_{SKIPS}$*-all-front-tickfree-traces2 FT-True UNIV-I UNIV-eq-I a1*(*2*) *as2 as3*
event$_{ptick}$.*exhaust is-processT6-TR-notin tickFree-imp-front-tickFree-snoc*)
next
case *FT-False*: *False*
then show *?thesis*
by (*metis DF*$_{SKIPS}$*-all-front-tickfree-traces2 a1*(*2*) *front-tickFree-append-iff*
is-processT2-TR is-processT5-S7 list.distinct(*1*))
qed
qed
qed

lemma *deadlock-free*$_{SKIPS}$*-implies-div-free*: ‹*deadlock-free*$_{SKIPS}$ *P* $\Longrightarrow$ $\mathcal{D}$ *P* = {}›
by (*metis D-T D-imp-front-tickFree T-nonTickFree-imp-decomp all-not-in-conv butlast-snoc*
deadlock-free$_{SKIPS}$*-is-right front-tickFree-iff-tickFree-butlast is-processT8 is-processT9*)

corollary *deadlock-free*$_{SKIPS}$*-FD*: ‹*deadlock-free*$_{SKIPS}$ *P* $\longleftrightarrow$ *DF*$_{SKIPS}$ *UNIV*

UNIV $\sqsubseteq_{FD}$ *P*›
by (*metis deadlock-free*$_{SKIPS}$*-def deadlock-free*$_{SKIPS}$*-implies-div-free less-eq-process*$_{ptick}$*-def order-refl refine-defs*(3))

lemma *all-events-ticks-refusal*:
‹(*s*, *tick* ‘ ✓*s*(*P*) ∪ *ev* ‘ *α*(*P*)) ∈ *F P* ⟹ (*s*, *UNIV*) ∈ *F P*›
proof (*rule ccontr*)
assume ‹(*s*, *tick* ‘ ✓*s*(*P*) ∪ *ev* ‘ *α*(*P*)) ∈ *F P*› **and** ‹(*s*, *UNIV*) ∉ *F P*›
then obtain *c* **where** ‹*c* ∉ *tick* ‘ ✓*s*(*P*) ∪ *ev* ‘ *α*(*P*) ∧ *s* @ [*c*] ∈ *T P*›
using *is-processT5*[*of s* ‹*tick* ‘ ✓*s*(*P*) ∪ *ev* ‘ *α*(*P*)› *P*
‹*UNIV* − (*tick* ‘ ✓*s*(*P*) ∪ *ev* ‘ *α*(*P*))›, *simplified*] *F-T* **by** *blast*
thus *False* **by** (*simp add*: *events-of-def ticks-of-def*, *cases c*) *fastforce*+
qed

corollary *deadlock-free*$_{SKIPS}$*-is-right-wrt-events*:
‹*deadlock-free*$_{SKIPS}$ *P* ⟷
(∀ *s*∈*T P*. *tF s* ⟶ (*s*, *tick* ‘ ✓*s*(*P*) ∪ *ev* ‘ *α*(*P*)) ∉ *F P*)›
unfolding *deadlock-free*$_{SKIPS}$*-is-right* **using** *all-events-ticks-refusal is-processT4*
by *blast*

lemma *deadlock-free-imp-deadlock-free*$_{SKIPS}$: ‹*deadlock-free P* ⟹ *deadlock-free*$_{SKIPS}$ *P*›
using *DF*$_{SKIPS}$*-FD-DF deadlock-free*$_{SKIPS}$*-FD deadlock-free-def trans-FD* **by** *blast*

14.6 *deadlock-free* and *deadlock-free*$_{SKIPS}$ with *SKIP* and *STOP*

lemma *non-deadlock-free*$_{SKIPS}$*-STOP*: ‹¬ *deadlock-free*$_{SKIPS}$ *STOP*›
by (*unfold deadlock-free*$_{SKIPS}$*-def*, *subst DF*$_{SKIPS}$*-unfold*, *unfold failure-refine-def*)
(*auto simp add*: *F-Ndet F-STOP F-SKIPS F-Mndetprefix write0-def F-Mprefix*)

lemma *non-deadlock-free-STOP*: ‹¬ *deadlock-free STOP*›
using *deadlock-free-imp-deadlock-free*$_{SKIPS}$ *non-deadlock-free*$_{SKIPS}$*-STOP* **by** *blast*

lemma *deadlock-free*$_{SKIPS}$*-SKIPS* : ‹*deadlock-free*$_{SKIPS}$ (*SKIPS R*) ⟷ *R* ≠ {}›
proof (*rule iffI*)
show ‹*deadlock-free*$_{SKIPS}$ (*SKIPS R*) ⟹ *R* ≠ {}›
by (*rule ccontr*, *simp add*: *SKIPS-def*)
(*use non-deadlock-free*$_{SKIPS}$*-STOP* **in** *blast*)
next
show ‹*R* ≠ {} ⟹ *deadlock-free*$_{SKIPS}$ (*SKIPS R*)›
by (*unfold deadlock-free*$_{SKIPS}$*-def*, *subst DF*$_{SKIPS}$*-unfold*, *unfold failure-refine-def*)
(*auto simp add*: *F-Ndet F-STOP F-SKIPS F-Mndetprefix write0-def F-Mprefix*

subset-iff)
qed

lemma *deadlock-free$_{SKIPS}$-SKIP*: ‹*deadlock-free$_{SKIPS}$* (*SKIP r*)›
by (*metis SKIPS-singl-is-SKIP deadlock-free$_{SKIPS}$-SKIPS empty-not-insert*)

lemma *non-deadlock-free-SKIPS* : ‹¬ *deadlock-free* (*SKIPS R*)›
proof (*cases* ‹*R* = {}›)
show ‹*R* = {} ⟹ ¬ *deadlock-free* (*SKIPS R*)›
by (*simp add*: *non-deadlock-free-STOP*)
next
assume ‹*R* ≠ {}›
then obtain *r* **where** ‹*r* ∈ *R*› **by** *blast*
hence ‹[✓(*r*)] ∈ *T* (*SKIPS R*)› **by** (*simp add*: *T-SKIPS*)
moreover have ‹¬ *tF* ([✓(*r*)])› **by** *simp*
ultimately show ‹¬ *deadlock-free* (*SKIPS R*)›
using *deadlock-free-implies-non-terminating* **by** *blast*
qed

lemma *non-deadlock-free-SKIP*: ‹¬ *deadlock-free* (*SKIP r*)›
by (*metis SKIPS-singl-is-SKIP non-deadlock-free-SKIPS*)

corollary *non-terminating-refine-DF* : ‹*non-terminating P* ⟷ *DF UNIV* ⊑$_T$ *P*›

and *non-terminating-refine-CHAOS* : ‹*non-terminating P* ⟷ *CHAOS UNIV* ⊑$_T$ *P*›
by (*simp-all add*: *DF-all-tickfree-traces1 RUN-all-tickfree-traces1 CHAOS-all-tickfree-traces1*

non-terminating-def trace-refine-def)

lemma *non-terminating-is-right* : ‹*non-terminating P* ⟷ (∀ *s*∈*T* *P*. *tF s*)›
by (*meson RUN-all-tickfree-traces2 no-ticks-imp-tickFree-T*
non-terminating-def subset-iff ticks-RUN trace-refine-def)

lemma *nonterminating-implies-div-free* : ‹*non-terminating P* ⟹ *D P* = {}›
by (*metis D-T ex-in-conv front-tickFree-single is-processT7*
non-terminating-is-right non-tickFree-tick tickFree-append-iff)

lemma *non-terminating-implies-F* : ‹*non-terminating P* ⟹ *CHAOS UNIV* ⊑$_F$ *P*›
by (*meson CHAOS-has-all-tickFree-failures F-T failure-refine-def in-mono no-ticks-imp-tickFree-T*
non-terminating-refine-CHAOS subrelI ticks-CHAOS top-greatest trace-refine-def)

corollary *non-terminating-F* : ‹*non-terminating P* ⟷ *CHAOS UNIV* ⊑$_F$ *P*›

by (*auto simp add*:*non-terminating-implies-F non-terminating-refine-CHAOS leF-imp-leT*)

corollary *non-terminating-FD* : ‹*non-terminating P* ⟷ *CHAOS UNIV* ⊑$_{FD}$ *P*›
by (*metis failure-refine-def less-eq-process*$_{ptick}$*-def non-terminating-F nonterminating-implies-div-free order-refl*)

corollary *lifelock-free-is-non-terminating*: ‹*lifelock-free P* = *non-terminating P*›
unfolding *lifelock-free-def non-terminating-FD* **by** (*fact refl*)

lemma *divergence-refine-div-free* :
‹*CHAOS*$_{SKIPS}$ *UNIV UNIV* ⊑$_D$ *P* ⟷ $\mathcal{D}$ *P* = {}›
‹*CHAOS UNIV* ⊑$_D$ *P* ⟷ $\mathcal{D}$ *P* = {}›
‹*RUN UNIV* ⊑$_D$ *P* ⟷ $\mathcal{D}$ *P* = {}›
‹*DF*$_{SKIPS}$ *UNIV UNIV* ⊑$_D$ *P* ⟷ $\mathcal{D}$ *P* = {}›
‹*DF UNIV* ⊑$_D$ *P* ⟷ $\mathcal{D}$ *P* = {}›
by (*simp-all add*: *div-free-CHAOS div-free-RUN div-free-DF div-free-DF*$_{SKIPS}$ *div-free-CHAOS*$_{SKIPS}$ *divergence-refine-def*)

lemma *lifelock-free*$_{SKIPS}$*-iff-div-free* : ‹*lifelock-free*$_{SKIPS}$ *P* ⟷ $\mathcal{D}$ *P* = {}›
by (*simp add*: *CHAOS*$_{SKIPS}$*-UNIV-UNIV-leF divergence-refine-div-free*(*1*) *failure-divergence-refine-def lifelock-free*$_{SKIPS}$*-def*)

lemma *lifelock-free-imp-lifelock-free*$_{SKIPS}$: ‹*lifelock-free P* ⟹ *lifelock-free*$_{SKIPS}$ *P*›
by (*simp add*: *lifelock-free*$_{SKIPS}$*-iff-div-free lifelock-free-is-non-terminating nonterminating-implies-div-free*)

corollary *deadlock-free*$_{SKIPS}$*-imp-lifelock-free*$_{SKIPS}$: ‹*deadlock-free*$_{SKIPS}$ *P* ⟹ *lifelock-free*$_{SKIPS}$ *P*›
by (*simp add*: *deadlock-free*$_{SKIPS}$*-implies-div-free lifelock-free*$_{SKIPS}$*-iff-div-free*)

lemma *non-terminating-Seq* : ‹*P* ; *Q* = *P*› **if** ‹*non-terminating P*›
proof −
from ‹*non-terminating P*› **have** ∗ : ‹*s* ∈ $\mathcal{T}$ *P* ⟹ *tF s*› **for** *s*
unfolding *non-terminating-is-right* **..**
show ‹*P* ; *Q* = *P*›
proof (*subst Process-eq-spec-optimized*, *safe*)
from ∗ **show** ‹*s* ∈ $\mathcal{D}$ (*P* ; *Q*) ⟹ *s* ∈ $\mathcal{D}$ *P*› **for** *s* **by** (*force simp add*: *D-Seq*)
next
show ‹*s* ∈ $\mathcal{D}$ *P* ⟹ *s* ∈ $\mathcal{D}$ (*P* ; *Q*)› **for** *s* **by** (*simp add*: *D-Seq*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ (*P* ; *Q*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *P*› **for** *s X*
by (*simp add*: *F-Seq*)

```
    (meson ∗ is-processT4 is-processT8 non-tickFree-tick sup-ge1 tickFree-append-iff)
  next
    show ‹(s, X) ∈ ℱ P ⟹ (s, X) ∈ ℱ (P ; Q)› for s X
      by (simp add: F-Seq)
        (metis (mono-tags, lifting) ∗ F-T f-inv-into-f is-processT5-S7′
          non-tickFree-tick tickFree-append-iff)
  qed
qed


lemma non-terminating-Sync :
  ‹non-terminating P ⟹ lifelock-free_SKIPS Q ⟹ non-terminating (P ⟦A⟧ Q)›
  by (simp add: lifelock-free_SKIPS-iff-div-free T-Sync
      non-terminating-is-right nonterminating-implies-div-free)
    (metis SyncWithTick-imp-NTF T-imp-front-tickFree ftf-Sync
      nonTickFree-n-frontTickFree non-tickFree-tick tickFree-append-iff)


lemmas non-terminating-Par = non-terminating-Sync[where A = ‹UNIV›]
  and non-terminating-Inter = non-terminating-Sync[where A = ‹{}›]
```

Chapter 15

Advanced Induction Schemata

15.1 k-fixpoint-induction

lemma *nat-k-induct* [*case-names base step*]:
 ‹*P n*› **if** ‹∀ *i*<*k*. *P i*› **and** ‹∀ n_0. (∀ *i*<*k*. *P* (n_0 + *i*)) ⟶ *P* (n_0 + *k*)› **for** *k n* :: *nat*
proof (*induct rule*: *nat-less-induct*)
 fix *n* **assume** ‹∀ *m*<*n*. *P m*›
 show ‹*P n*›
 proof (*cases* ‹*n* < *k*›)
  from *that*(*1*) **show** ‹*n* < *k* ⟹ *P n*› **by** *blast*
 next
  from ‹∀ *m*<*n*. *P m*› *that*(*2*)[*rule-format*, *of* ‹*n* − *k*›]
  show ‹¬ *n* < *k* ⟹ *P n*› **by** *auto*
 qed
qed

thm *fix-ind fix-ind2*

lemma *fix-ind-k* [*case-names admissibility base-k-steps step*]:
 assumes *adm* : ‹*adm P*›
  and *base-k-steps* : ‹∀ *i*<*k*. *P* (*iterate i*·*f*·⊥)›
  and *step* : ‹⋀*X*. (∀ *i*<*k*. *P* (*iterate i*·*f*·*X*)) ⟹ *P* (*iterate k*·*f*·*X*)›
 shows ‹*P* (*fix*·*f*)›
proof (*unfold fix-def2*, *rule admD* [*OF adm chain-iterate*])
 show ‹*P* (*iterate i*·*f*·⊥)› **for** *i*
 proof (*induct i rule* : *nat-k-induct*[*of k*])
  show ‹∀ *i*<*k*. *P* (*iterate i*·*f*·⊥)› **by** (*fact base-k-steps*)
 next
  show ‹∀ n_0. (∀ *i*<*k*. *P* (*iterate* (n_0 + *i*)·*f*·⊥)) ⟶ *P* (*iterate* (n_0 + *k*)·*f*·⊥)›
   by (*metis add.commute iterate-iterate step*)
 qed
qed

lemma *nat-k-skip-induct* [*case-names lower-bound base-k step*]:
‹*P n*› **if** ‹*1* ≤ *k*› **and** ‹∀ *i*<*k*. *P i*› **and** ‹∀ n_0. *P* n_0 ⟶ *P* (n_0 + *k*)› **for** *k n* :: *nat*
proof (*induct n rule*: *nat-less-induct*)
fix *n* **assume** ‹∀ *m*<*n*. *P m*›
show ‹*P n*›
proof (*cases* ‹*n* < *k*›)
from *that*(*2*) **show** ‹*n* < *k* ⟹ *P n*› **by** *blast*
next
from ‹∀ *m*<*n*. *P m*› ‹*1* ≤ *k*› *that*(*3*)[*rule-format*, *of* ‹*n* − *k*›]
show ‹¬ *n* < *k* ⟹ *P n*› **by** *auto*
qed
qed

lemma *fix-ind-k-skip* [*case-names lower-bound admissibility base-k-steps step*]:
assumes *k-1* : ‹*1* ≤ *k*›
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*)›
proof (*unfold fix-def2*, *rule admD* [*OF adm chain-iterate*])
show ‹*P* (*iterate i*·*f*·⊥)› **for** *i*
proof (*induct i rule* : *nat-k-skip-induct*[*of k*])
show ‹*1* ≤ *k*› **by** (*fact* ‹*1* ≤ *k*›)
next
show ‹∀ *i*<*k*. *P* (*iterate i*·*f*·⊥)› **by** (*fact base-k-steps*)
next
show ‹∀ n_0. *P* (*iterate* n_0·*f*·⊥) ⟶ *P* (*iterate* (n_0 + *k*)·*f*·⊥)›
by (*metis add.commute iterate-iterate step*)
qed
qed

15.2 Parallel fixpoint-induction

lemma *parallel-fix-ind-inc*[*case-names admissibility base-fst base-snd step*]:
assumes *adm*: ‹*adm* (λ*X*. *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·⊥)›
      using 1.hyps add-strict-mono i j by blast
    moreover have ‹P (iterate i'·G·⊥) (iterate j·H·⊥)›
      using 1.hyps i by auto
    moreover have ‹P (iterate i·G·⊥) (iterate j'·H·⊥)›
      using 1.hyps j by auto
    ultimately have ?case by (simp add: i j step)
   }
   thus ?case by (cases i; cases j; simp add: base-fst base-snd)
  qed
  hence ‹case-prod P (iterate i·G·⊥, iterate i·H·⊥)› for i by simp
  hence ‹case-prod P (⨆i. (iterate i·G·⊥, iterate i·H·⊥))›
    by − (rule admD [OF adm'], simp, assumption)
  hence ‹P (⨆i. iterate i·G·⊥) (⨆i. iterate i·H·⊥)›
    by (simp add: lub-Pair)
  thus ‹P (fix·G) (fix·H)› by (simp add: fix-def2)
qed
```

Chapter 16

The Main Entry Point

This is the theory `HOL-CSP` should be imported from.

Chapter 17

Conclusion

17.1 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.

17.2 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.

It is also worth noting that the restructuring of the theory, as well as the proofs (declarative Isar style), has substantially increased the possibility to parallelize the proof checking process and makes the entire theory more maintainable.

Finally, having the entire theory formalized makes extensions such as parametized ticks possible since the effect of changes of basic definitions can be traced consequently. This is an important aspect that extensions of this kind are not ad hoc and do not endanger global consistency.

17.3 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 $(\setminus)$ over *Sync* $[\![$*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. distribution of *Mprefix* over *Sync Mprefix ?A ?P ⟦?S⟧ Mprefix ?B ?Q* = $(\Box a{\in}(?A - ?S) \rightarrow (?P\ a \llbracket ?S \rrbracket Mprefix\ ?B\ ?Q)) \Box (\Box b{\in}(?B - ?S) \rightarrow (Mprefix\ ?A\ ?P \llbracket ?S \rrbracket ?Q\ b)) \Box (\Box x{\in}(?A \cap ?B \cap ?S) \rightarrow (?P\ x \llbracket ?S \rrbracket ?Q\ x))$ in the most generalized case. Also a true monster proof, but reworked using symmetries and abstractions to be more reasonable (and faster)

5. the synchronization operator is associative $?P \llbracket ?S \rrbracket (?Q \llbracket ?S \rrbracket ?R) = ?P \llbracket ?S \rrbracket ?Q \llbracket ?S \rrbracket ?R$. (In HOL-CSP 1.0, this had only be shown for special cases like $?P \parallel (?Q \parallel ?R) = ?P \parallel ?Q \parallel ?R$).

6. the generalized non deterministic prefix choice operator — relevant for proofs of deadlock-freeness — has been added to the theory *Mndet-prefix* ≡ *map-fun id* (*map-fun id process-of-process*$_0$) ($\lambda A\ P$. *if* $A = \{\}$ *then process*$_0$*-of-process STOP else* ($\bigcup_{a\in A} \mathcal{F}\ (a \rightarrow P\ a)$, $\bigcup_{a\in A} \mathcal{D}\ (a \rightarrow P\ a)$)); it is proven monotone and continuous ($\bigwedge a.\ a \in ?A \Longrightarrow cont\ (?f\ a)) \Longrightarrow cont\ (\lambda b.\ \sqcap a{\in}?A \rightarrow ?f\ a\ b)$ in the general case (in contrast to the global choice without prefix, see $\llbracket finite\ ?A; \bigwedge a.\ a \in ?A \Longrightarrow cont\ (?P\ a) \rrbracket \Longrightarrow cont\ (\lambda y.\ \sqcap z{\in}?A.\ ?P\ z\ y)$). This is relevant for the definition of the deadlock reference processes $DF\ ?A \equiv \mu\ x.\ \sqcap a{\in}?A \rightarrow x$ and $DF_{SKIPS}\ ?A\ ?R \equiv \mu\ x.\ (\sqcap a{\in}?A \rightarrow x) \sqcap SKIPS\ ?R$.

7. since Isabelle-2025, new support for read *read c A P* and non deterministic write *ndet-write c A P* has been added. Also, sliding choice has been added and new algebraic laws involving this operator (see $?A \cap ?S \neq \{\} \Longrightarrow Mprefix\ ?A\ ?P \setminus ?S = (\Box a{\in}(?A - ?S) \rightarrow (?P\ a \setminus ?S)) \triangleright (\sqcap a{\in}(?A \cap ?S).\ (?P\ a \setminus ?S))$ have been proven.

Chapter 18

Annex: Refinement Example with Buffer over infinite Alphabet

18.1 Defining the Copy-Buffer Example

datatype *'a channel = left 'a | right 'a | mid 'a | ack*

definition *SYN :: 'a channel set*
where *SYN ≡ range mid ∪ {ack}*

definition *COPY :: 'a channel process*
where *COPY ≡ μ COPY. left?x → right!x → COPY*

definition *SEND :: 'a channel process*
where *SEND ≡ μ SEND. left?x → mid!x → ack → SEND*

definition *REC :: 'a channel process*
where *REC ≡ μ REC. mid?x → right!x → ack → REC*

definition *SYSTEM :: 'a channel process*
where *‹SYSTEM ≡ SEND ⟦ SYN ⟧ REC \ SYN›*

thm *SYSTEM-def*

18.2 The Standard Proof

18.2.1 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 *simplification-lemmas* [*simp*] :
‹*range left* ∩ *SYN* = {}›
‹*range right* ∩ *SYN* = {}›
‹*ack* ∈ *SYN*›
‹*range mid* ⊆ *SYN*›
‹*mid x* ∈ *SYN*›
‹*right x* ∉ *SYN*›
‹*left x* ∉ *SYN*›
‹*inj mid*›
by (*auto simp*: *SYN-def inj-on-def*)

lemma *finite* (*SYN*:: *'a channel set*) ⟹ *finite* {(*t*::*'a*). *True*}
by (*metis* (*no-types*) *SYN-def UNIV-def channel.inject*(*3*) *finite-Un finite-imageD inj-on-def*)

18.2.2 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* → *right*!*x* → *COPY*
by(*simp add*: *COPY-def*,*rule trans*, *rule fix-eq*, *simp*)

lemma *SEND-rec*:
SEND = *left*?*x* → *mid*!*x* → *ack* → *SEND*
by(*simp add*: *SEND-def*,*rule trans*, *rule fix-eq*, *simp*)

lemma *REC-rec*:
REC = *mid*?*x* → *right*!*x* → *ack* → *REC*
by(*simp add*: *REC-def*,*rule trans*, *rule fix-eq*, *simp*)

18.2.3 Various Samples of Refinement Proofs

lemmas *Sync-rules* = *read-Sync-read-subset-forced-read-same-chan*
read-Sync-read-left read-Sync-read-right
write-Sync-read-left write-Sync-read-right
read-Sync-write-left read-Sync-write-right
write-Sync-write-subset
write-Sync-read-subset read-Sync-write-subset

write0-Sync-write-right write0-Sync-write0

lemmas *Hiding-rules* = *Hiding-read-disjoint Hiding-write-subset Hiding-write-disjoint*
Hiding-write0-non-disjoint Hiding-write0-disjoint

lemmas *mono-rules* = *mono-read-FD mono-write-FD mono-write0-FD*

An example for a very explicit structured proof. Slow-motion for presenta-

tions. Note that the proof makes no assumption over the structure of the content of the channel; it is truly polymorphic wrt. the channel type $'\alpha$.

lemma *impl-refines-spec'* : (*COPY* :: *'a channel process*) $\sqsubseteq_{FD}$ *SYSTEM*
unfolding *SYSTEM-def COPY-def*
proof (*rule fix-ind*)
show *adm* ($\lambda a.$ $a \sqsubseteq_{FD}$ (*SEND* ⟦*SYN*⟧ *REC* \ *SYN*))
by (*simp add*: *cont2mono*)
next
show $\bot \sqsubseteq_{FD}$ (*SEND* ⟦*SYN*⟧ *REC* \ *SYN*)
by *simp*
next
fix *x* :: *'a channel process*
assume *hyp* : $x \sqsubseteq_{FD}$ (*SEND* ⟦*SYN*⟧ *REC* \ *SYN*)
show (Λ *x*. *left*?*xa* → *right*!*xa* → *x*)·*x* $\sqsubseteq_{FD}$ (*SEND* ⟦*SYN*⟧ *REC* \ *SYN*)
apply (*subst SEND-rec*, *subst REC-rec*)
apply (*simp add*: *cont-fun*)
proof −
have *left*?*x* → *mid*!*x* → *ack* → *SEND* ⟦*SYN*⟧ *mid*?*x* → *right*!*x* → *ack* → *REC* \ *SYN*
= *left*?*x* → (*mid*!*x* → *ack* → *SEND* ⟦*SYN*⟧ *mid*?*x* → *right*!*x* → *ack* → *REC* \ *SYN*)
(**is** *?lhs* = -)
by (*simp add*: *Sync-rules Hiding-rules*)
also have ... = *left*?*x* → (*mid*!*x* → (*ack* → *SEND* ⟦*SYN*⟧ *right*!*x* → *ack* → *REC*) \ *SYN*)
by (*simp add*: *Sync-rules Hiding-rules*)
also have ... = *left*?*x* → (*ack* → *SEND* ⟦*SYN*⟧ *right*!*x* → *ack* → *REC*) \ *SYN*
by (*simp add*: *Sync-rules Hiding-rules*)
also have ... = *left*?*x* → *right*!*x* → (*ack* → *SEND* ⟦*SYN*⟧ *ack* → *REC*) \ *SYN*
by (*simp add*: *Sync-rules Hiding-rules*)
also have ... = *left*?*x* → *right*!*x* → (*SEND* ⟦*SYN*⟧ *REC* \ *SYN*) (**is** - = *?rhs*)
by (*simp add*: *Sync-rules Hiding-rules*)
finally have ∗ : *?lhs* = *?rhs* **.**
show *left*?*xa* → *right*!*xa* → *x* $\sqsubseteq_{FD}$ *?lhs*
by (*simp only* : ∗ *mono-read-FD mono-write-FD hyp*)
qed
qed

An example for a highly automated proof.

Not too bad in automation considering what is inferred, but wouldn't scale for large examples.

lemma *impl-refines-spec* : *COPY* $\sqsubseteq_{FD}$ *SYSTEM*
unfolding *SYSTEM-def COPY-def*
apply(*rule fix-ind*, *auto intro*: *le-FD-adm simp*: *cont-fun monofunI*)
apply(*subst SEND-rec*, *subst REC-rec*)
apply (*simp add*: *Sync-rules Hiding-rules mono-read-FD mono-write-FD*)

oops

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*: *Sync-commute*)
apply (*subst COPY-rec*, *subst REC-rec*)
apply (*simp add*: *read-def write-def comp-def*)
apply (*subst Mprefix-Sync-Mprefix-right*) **apply** *auto*[*2*]
apply (*simp add*: *Sync-rules Hiding-rules comp-def*)
apply (*subst Hiding-Mprefix-disjoint*) **apply** *auto*[*1*]
apply (*intro mono-Mprefix-FD ballI*)
apply (*subst Mprefix-Sync-Mprefix-subset*) **apply** *auto*[*2*]
apply (*subst Hiding-Mprefix-non-disjoint*) **apply** *auto*[*1*]
apply *simp*
apply (*fold write0-def*)
by (*simp add*: *Hiding-write0-disjoint Hiding-write0-non-disjoint Sync-commute mono-write0-FD write0-Sync-write0*)

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*)

18.2.4 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-COPY* : (*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 (*simp add*: *Mndetprefix-FD-subset*)
have *2*: (⊓*xa*∈ *range left* → ⊥) $\sqsubseteq_{FD}$ (*left*?*x* → ⊥)

unfolding *read-def*
by (*meson Mndetprefix-FD-Mprefix BOT-leFD mono-Mndetprefix-FD trans-FD*)
show (Λ *x*. $\sqcap$*xa*$\in$(*range left* $\cup$ *range right*) $\rightarrow$ *x*)$\cdot\bot \sqsubseteq_{FD}$ *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 (*simp add*: *Mndetprefix-FD-subset*)
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 Mndetprefix-FD-subset*[*of* ‹{*right x*}› ‹*range right*›]])
by (*simp-all add*: *Mndetprefix-FD-Mprefix Mprefix-singl*)
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 add*: *mono-read-FD*)
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 Mndetprefix-FD-subset*, *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 Mndetprefix-FD-Mprefix*)
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 add*: *mono-read-FD mono-write-FD*)

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

18.3 An Alternative Approach: Using the fixrec-Package

18.3.1 Channels and Synchronisation Sets

As before.

18.3.2 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)›
```

18.3.3 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) ⊑FD SYSTEM'
  apply (unfold SYSTEM'-def)
 apply (rule-tac P=‹λ a b c. a ⊑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 mono-read-FD mono-write-FD)

lemma spec-refines-impl' :
  assumes fin:  finite (SYN::'a channel set)
  shows         SYSTEM' ⊑FD (COPY'::'a channel process)
proof(unfold SYSTEM'-def, rule-tac P=‹λ a b c. ((b ⟦SYN⟧ REC') \ SYN) ⊑FD
COPY'›
    in  COPY'-SEND'-REC'.induct, goal-cases)
  case 1
 have aa:‹adm (λ(a::'a channel process). ((a ⟦SYN⟧ REC') \ SYN) ⊑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: Sync-commute)
next
  case (3 a aa b)
  then show ?case
    by (subst COPY'-rec, subst REC'-rec)
      (simp add: Sync-rules Hiding-rules mono-read-FD mono-write-FD)
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

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.