A Definitional Encoding of TLA in Isabelle/HOL

Gudmund Grov & Stephan Merz

March 17, 2025

Abstract

We mechanise the logic TLA* [8], an extension of Lamport's Temporal Logic of Actions (TLA) [5] for specifying and reasoning about concurrent and reactive systems. Aiming at a framework for mechanising the verification of TLA (or TLA*) specifications, this contribution reuses some elements from a previous axiomatic encoding of TLA in Isabelle/HOL by the second author [7], which has been part of the Isabelle distribution. In contrast to that previous work, we give here a shallow, definitional embedding, with the following highlights:

- a theory of infinite sequences, including a formalisation of the concepts of stuttering invariance central to TLA and TLA*;
- a definition of the semantics of TLA*, which extends TLA by a mutually-recursive definition of formulas and pre-formulas, generalising TLA action formulas;
- a substantial set of derived proof rules, including the TLA* axioms and Lamport's proof rules for system verification;
- a set of examples illustrating the usage of Isabelle/TLA* for reasoning about systems.

Note that this work is unrelated to the ongoing development of a proof system for the specification language TLA+, which includes an encoding of TLA+ as a new Isabelle object logic [1].

A previous version of this embedding has been used heavily in the work described in [4].

Contents

1 (Infinite) Sequences

theory *Sequence*
imports *Main*
begin

Lamport's Temporal Logic of Actions (TLA) is a linear-time temporal logic, and its semantics is defined over infinite sequence of states, which we simply represent by the type $'a\ seq$, defined as an abbreviation for the type $nat \Rightarrow\ 'a$, where $'a$ is the type of sequence elements.

This theory defines some useful notions about such sequences, and in particular concepts related to stuttering (finite repetitions of states), which are important for the semantics of TLA. We identify a finite sequence with an infinite sequence that ends in infinite stuttering. In this way, we avoid the complications of having to handle both finite and infinite sequences of states: see e.g. Devillers et al [2] who discuss several variants of representing possibly infinite sequences in HOL, Isabelle and PVS.

type-synonym $'a\ seq = nat \Rightarrow\ 'a$

1.1 Some operators on sequences

Some general functions on sequences are provided

definition *first* :: $'a\ seq \Rightarrow\ 'a$
where *first* $s \equiv s\ 0$

definition *second* :: $('a\ seq) \Rightarrow\ 'a$
where *second* $s \equiv s\ 1$

definition *suffix* :: $'a$ *seq* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *seq* (**infixl** ‹$|_s$› *60*)
where $s\ |_s\ i \equiv \lambda\ n.\ s\ (n+i)$

definition *tail* :: $'a$ *seq* $\Rightarrow$ $'a$ *seq*
where *tail* $s \equiv s\ |_s\ 1$

definition
app :: $'a \Rightarrow ('a\ seq) \Rightarrow ('a\ seq)$ (**infixl** ‹##› *60*)
where
s ## $\sigma \equiv \lambda\ n.$ *if* $n=0$ *then* s *else* $\sigma\ (n - 1)$

$s\ |_s\ i$ returns the suffix of sequence s from index i. *first* returns the first element of a sequence while *second* returns the second element. *tail* returns the sequence starting at the second element. s ## σ prefixes the sequence σ by element s.

1.1.1 Properties of *first* and *second*

lemma *first-tail-second*: *first*(*tail* s) = *second* s
by (*simp add*: *first-def second-def tail-def suffix-def*)

1.1.2 Properties of $(|_s)$

lemma *suffix-first*: *first* $(s\ |_s\ n) = s\ n$
by (*auto simp add*: *suffix-def first-def*)

lemma *suffix-second*: *second* $(s\ |_s\ n) = s\ (Suc\ n)$
by (*auto simp add*: *suffix-def second-def*)

lemma *suffix-plus*: $s\ |_s\ n\ |_s\ m = s\ |_s\ (m + n)$
by (*simp add*: *suffix-def add.assoc*)

lemma *suffix-commute*: $((s\ |_s\ n)\ |_s\ m) = ((s\ |_s\ m)\ |_s\ n)$
by (*simp add*: *suffix-plus add.commute*)

lemma *suffix-plus-com*: $s\ |_s\ m\ |_s\ n = s\ |_s\ (m + n)$
proof −
have $s\ |_s\ n\ |_s\ m = s\ |_s\ (m + n)$ **by** (*rule suffix-plus*)
thus $s\ |_s\ m\ |_s\ n = s\ |_s\ (m + n)$ **by** (*simp add*: *suffix-commute*)
qed

lemma *suffix-zero*[*simp*]: $s\ |_s\ 0 = s$
by (*simp add*: *suffix-def*)

lemma *suffix-tail*: $s\ |_s\ 1$ = *tail* s
by (*simp add*: *tail-def*)

lemma *tail-suffix-suc*: $s\ |_s\ (Suc\ n)$ = *tail* $(s\ |_s\ n)$
by (*simp add*: *suffix-def tail-def*)

1.1.3 Properties of $(\#\#)$

lemma *seq-app-second*: $(s \; \#\# \; \sigma) \; 1 = \sigma \; 0$
  by (*simp add*: *app-def*)

lemma *seq-app-first*: $(s \; \#\# \; \sigma) \; 0 = s$
  by (*simp add*: *app-def*)

lemma *seq-app-first-tail*: $(\mathit{first} \; s) \; \#\# \; (\mathit{tail} \; s) = s$
proof (*rule ext*)
  fix x
  show $(\mathit{first} \; s \; \#\# \; \mathit{tail} \; s) \; x = s \; x$
    by (*simp add*: *first-def app-def suffix-def tail-def*)
qed

lemma *seq-app-tail*: $\mathit{tail} \; (x \; \#\# \; s) = s$
  by (*simp add*: *app-def tail-def suffix-def*)

lemma *seq-app-greater-than-zero*: $n > 0 \Longrightarrow (s \; \#\# \; \sigma) \; n = \sigma \; (n - 1)$
  by (*simp add*: *app-def*)

1.2 Finite and Empty Sequences

We identify finite and empty sequences and prove lemmas about them.

definition *fin* :: $('a \; seq) \Rightarrow bool$
where $\mathit{fin} \; s \equiv \exists \; i. \; \forall \; j \geq i. \; s \; j = s \; i$

abbreviation *inf* :: $('a \; seq) \Rightarrow bool$
where $\mathit{inf} \; s \equiv \neg(\mathit{fin} \; s)$

definition *last* :: $('a \; seq) \Rightarrow nat$
where $\mathit{last} \; s \equiv \mathit{LEAST} \; i. \; (\forall \; j \geq i. \; s \; j = s \; i)$

definition *laststate* :: $('a \; seq) \Rightarrow 'a$
where $\mathit{laststate} \; s \equiv s \; (\mathit{last} \; s)$

definition *emptyseq* :: $('a \; seq) \Rightarrow bool$
where $\mathit{emptyseq} \equiv \lambda \; s. \; \forall \; i. \; s \; i = s \; 0$

abbreviation *notemptyseq* :: $('a \; seq) \Rightarrow bool$
where $\mathit{notemptyseq} \; s \equiv \neg(\mathit{emptyseq} \; s)$

Predicate *fin* holds if there is an element in the sequence such that all subsequent elements are identical, i.e. the sequence is finite. *Sequence.last s* returns the smallest index from which on all elements of a finite sequence s are identical. Note that if s is not finite then an arbitrary number is returned. *laststate* returns the last element of a finite sequence. We assume that the sequence is finite when using *Sequence.last* and *laststate*. Predicate *emptyseq* identifies empty sequences – i.e. all states in the sequence are identical to the initial one, while *notemptyseq* holds if the given sequence is not empty.

1.2.1 Properties of *emptyseq*

lemma *empty-is-finite*: **assumes** *emptyseq s* **shows** *fin s*
 using *assms* **by** (*auto simp*: *fin-def emptyseq-def*)

lemma *empty-suffix-is-empty*: **assumes** *H*: *emptyseq s* **shows** *emptyseq* $(s \mid_s n)$
proof (*clarsimp simp*: *emptyseq-def*)
 fix *i*
 from *H* **have** $(s \mid_s n)\ i = s\ 0$ **by** (*simp add*: *emptyseq-def suffix-def*)
 moreover
 from *H* **have** $(s \mid_s n)\ 0 = s\ 0$ **by** (*simp add*: *emptyseq-def suffix-def*)
 ultimately
 show $(s \mid_s n)\ i = (s \mid_s n)\ 0$ **by** *simp*
qed

lemma *suc-empty*: **assumes** *H1*: *emptyseq* $(s \mid_s m)$ **shows** *emptyseq* $(s \mid_s (Suc\ m))$
proof −
 from *H1* **have** *emptyseq* $((s \mid_s m) \mid_s 1)$ **by** (*rule empty-suffix-is-empty*)
 thus *?thesis* **by** (*simp add*: *suffix-plus*)
qed

lemma *empty-suffix-exteq*: **assumes** *H*:*emptyseq s* **shows** $(s \mid_s n)\ m = s\ m$
proof (*unfold suffix-def*)
 from *H* **have** $s\ (m+n) = s\ 0$ **by** (*simp add*: *emptyseq-def*)
 moreover
 from *H* **have** $s\ m = s\ 0$ **by** (*simp add*: *emptyseq-def*)
 ultimately show $s\ (m + n) = s\ m$ **by** *simp*
qed

lemma *empty-suffix-eq*: **assumes** *H*: *emptyseq s* **shows** $(s \mid_s n) = s$
proof (*rule ext*)
 fix *m*
 from *H* **show** $(s \mid_s n)\ m = s\ m$ **by** (*rule empty-suffix-exteq*)
qed

lemma *seq-empty-all*: **assumes** *H*: *emptyseq s* **shows** $s\ i = s\ j$
proof −
 from *H* **have** $s\ i = s\ 0$ **by** (*simp add*: *emptyseq-def*)
 moreover
 from *H* **have** $s\ j = s\ 0$ **by** (*simp add*: *emptyseq-def*)
 ultimately
 show *?thesis* **by** *simp*
qed

1.2.2 Properties of *Sequence.last* and *laststate*

lemma *fin-stut-after-last*: **assumes** H: *fin s* **shows** $\forall j \geq last\ s.\ s\ j = s\ (last\ s)$
proof (*clarify*)
  fix j
  assume j: $j \geq last\ s$
  from H **obtain** i **where** $\forall j \geq i.\ s\ j = s\ i$ (**is** *?P i*) **by** (*auto simp*: *fin-def*)
  hence *?P* (*last s*) **unfolding** *last-def* **by** (*rule LeastI*)
  with j **show** $s\ j = s\ (last\ s)$ **by** *blast*
qed

1.3 Stuttering Invariance

This subsection provides functions for removing stuttering steps of sequences, i.e. we formalise Lamports $\natural$ operator. Our formal definition is close to that of Wahab in the PVS prover.

The key novelty with the *Sequence* theory, is the treatment of stuttering invariance, which enables verification of stuttering invariance of the operators derived using it. Such proofs require comparing sequences up to stuttering. Here, Lamport's [5] method is used to mechanise the equality of sequences up to stuttering: he defines the $\natural$ operator, which collapses a sequence by removing all stuttering steps, except possibly infinite stuttering at the end of the sequence. These are left unchanged.

definition *nonstutseq* :: $('a\ seq) \Rightarrow bool$
where $nonstutseq\ s \equiv \forall\ i.\ s\ i = s\ (Suc\ i) \longrightarrow (\forall\ j > i.\ s\ i = s\ j)$

definition *stutstep* :: $('a\ seq) \Rightarrow nat \Rightarrow bool$
where $stutstep\ s\ n \equiv (s\ n = s\ (Suc\ n))$

definition *nextnat* :: $('a\ seq) \Rightarrow nat$
where $nextnat\ s \equiv if\ emptyseq\ s\ then\ 0\ else\ LEAST\ i.\ s\ i \neq s\ 0$

definition *nextsuffix* :: $('a\ seq) \Rightarrow ('a\ seq)$
where $nextsuffix\ s \equiv s\ |_s\ (nextnat\ s)$

fun *next* :: $nat \Rightarrow ('a\ seq) \Rightarrow ('a\ seq)$ **where**
  $next\ 0 = id$
| $next\ (Suc\ n) = nextsuffix\ o\ (next\ n)$

definition *collapse* :: $('a\ seq) \Rightarrow ('a\ seq)$ (‹$\natural$›)
where $\natural\ s \equiv \lambda\ n.\ (next\ n\ s)\ 0$

Predicate *nonstutseq* identifies sequences without any stuttering steps – except possibly for infinite stuttering at the end. Further, *stutstep s n* is a predicate which holds if the element after $s\ n$ is equal to $s\ n$, i.e. $Suc\ n$ is a stuttering step. $\natural\ s$ formalises Lamports $\natural$ operator. It returns the first state of the result of *next n s*. *next n s* finds suffix of the n^{th} change. Hence the first element, which $\natural$ *s* returns, is the state after the n^{th} change. *next n s* is defined by primitive recursion on *n* using function composition of function *nextsuffix*. E.g. *next 3 s* equals *nextsuffix* (*nextsuffix* (*nextsuffix s*)). *nextsuffix s* returns the suffix of the sequence starting at the next changing state. It uses *nextnat* to obtain this. All the real computation is done in this function. Firstly, an empty sequence will obviously not contain any changes, and *0* is therefore returned. In this case *nextsuffix* behaves like the identify function. If the sequence is not empty then the smallest number *i* such that *s i* is different from the initial state is returned. This is achieved by *Least*.

1.3.1 Properties of *nonstutseq*

lemma *seq-empty-is-nonstut*:
  assumes *H*: *emptyseq s* **shows** *nonstutseq s*
  using *H* **by** (*auto simp*: *nonstutseq-def seq-empty-all*)

lemma *notempty-exist-nonstut*:
  assumes *H*: $\neg$ *emptyseq* $(s \mid_s m)$ **shows** $\exists\ i.\ s\ i \neq s\ m \wedge i > m$
using *H* **proof** (*auto simp*: *emptyseq-def suffix-def*)
  fix *i*
  assume *i*: $s\ (i + m) \neq s\ m$
  hence $i \neq 0$ **by** (*intro notI*, *simp*)
  with *i* **show** *?thesis* **by** *auto*
qed

1.3.2 Properties of *nextnat*

lemma *nextnat-le-unch*: **assumes** *H*: $n <$ *nextnat s* **shows** $s\ n = s\ 0$
proof (*cases emptyseq s*)
  assume *emptyseq s*
  hence *nextnat s* $= 0$ **by** (*simp add*: *nextnat-def*)
  with *H* **show** *?thesis* **by** *auto*
next
  assume $\neg$ *emptyseq s*
  hence *a1*: *nextnat s* $=$ (*LEAST* $i.\ s\ i \neq s\ 0$) **by** (*simp add*: *nextnat-def*)
  show *?thesis*
  proof (*rule ccontr*)
    assume *a2*: $s\ n \neq s\ 0$ (**is** *?P n*)
    hence (*LEAST* $i.\ s\ i \neq s\ 0$) $\leq n$ **by** (*rule Least-le*)
    hence $\neg(n <$ (*LEAST* $i.\ s\ i \neq s\ 0$)) **by** *auto*
    also from *H a1* **have** $n <$ (*LEAST* $i.\ s\ i \neq s\ 0$) **by** *simp*
    ultimately show *False* **by** *auto*
  qed
qed

lemma *stutnempty*:
  assumes *H*: $\neg$ *stutstep s n* **shows** $\neg$ *emptyseq* $(s \mid_s n)$
proof (*unfold emptyseq-def suffix-def*)

from H **have** s (*Suc n*) $\neq$ s n **by** (*auto simp add*: *stutstep-def*)
hence s $(1+n) \neq s$ $(0+n)$ **by** *simp*
thus $\neg(\forall\ i.\ s\ (i+n) = s\ (0+n))$ **by** *blast*
qed

lemma *notstutstep-nexnat1*:
assumes H: $\neg$ *stutstep s n* **shows** *nextnat* $(s\ |_s\ n) = 1$
proof $-$
from H **have** h': *nextnat* $(s\ |_s\ n) = (LEAST\ i.\ (s\ |_s\ n)\ i \neq (s\ |_s\ n)\ 0)$
by (*auto simp add*: *nextnat-def stutnempty*)
from H **have** s (*Suc n*) $\neq$ s n **by** (*auto simp add*: *stutstep-def*)
hence $(s\ |_s\ n)\ 1 \neq (s\ |_s\ n)\ 0$ (**is** *?P 1*) **by** (*auto simp add*: *suffix-def*)
hence *Least ?P* ≤ 1 **by** (*rule Least-le*)
hence *g1*: *Least ?P* $= 0 \vee$ *Least ?P* $= 1$ **by** *auto*
with h' **have** *g1'*: *nextnat* $(s\ |_s\ n) = 0 \vee$ *nextnat* $(s\ |_s\ n) = 1$ **by** *auto*
also have *nextnat* $(s\ |_s\ n) \neq 0$
proof $-$
from H **have** $\neg$ *emptyseq* $(s\ |_s\ n)$ **by** (*rule stutnempty*)
then obtain i **where** $(s\ |_s\ n)\ i \neq (s\ |_s\ n)\ 0$ **by** (*auto simp add*: *emptyseq-def*)
hence $(s\ |_s\ n)\ (LEAST\ i.\ (s\ |_s\ n)\ i \neq (s\ |_s\ n)\ 0) \neq (s\ |_s\ n)\ 0$ **by** (*rule LeastI*)
with h' **have** *g2*: $(s\ |_s\ n)$ (*nextnat* $(s\ |_s\ n)) \neq (s\ |_s\ n)\ 0$ **by** *auto*
show (*nextnat* $(s\ |_s\ n)) \neq 0$
proof
assume (*nextnat* $(s\ |_s\ n)) = 0$
with *g2* **show** *False* **by** *simp*
qed
qed
ultimately show *nextnat* $(s\ |_s\ n) = 1$ **by** *auto*
qed

lemma *stutstep-notempty-notempty*:
assumes *h1*: *emptyseq* $(s\ |_s$ *Suc n*$)$ (**is** *emptyseq ?sn*)
and *h2*: *stutstep s n*
shows *emptyseq* $(s\ |_s\ n)$ (**is** *emptyseq ?s*)
proof (*auto simp*: *emptyseq-def*)
fix k
show *?s k* = *?s 0*
proof (*cases k*)
assume $k = 0$ **thus** *?thesis* **by** *simp*
next
fix m
assume k: $k = Suc\ m$
hence *?s k* = *?sn m* **by** (*simp add*: *suffix-def*)
also from *h1* **have** ... = *?sn 0* **by** (*simp add*: *emptyseq-def*)
also from *h2* **have** ... = s n **by** (*simp add*: *suffix-def stutstep-def*)
finally show *?thesis* **by** (*simp add*: *suffix-def*)
qed
qed

lemma *stutstep-empty-suc*:
  assumes *stutstep s n*
  shows *emptyseq* $(s \mid_s Suc\ n) = emptyseq\ (s \mid_s n)$
using *assms* **by** (*auto elim*: *stutstep-notempty-notempty suc-empty*)

lemma *stutstep-notempty-sucnextnat*:
  assumes *h1*: $\neg$ *emptyseq* $(s \mid_s n)$ **and** *h2*: *stutstep s n*
  shows $(nextnat\ (s \mid_s n)) = Suc\ (nextnat\ (s \mid_s (Suc\ n)))$
proof −
  from *h2* **have** *g1*: $\neg(s\ (0+n) \neq s\ (Suc\ n))$ (**is** $\neg$ *?P 0*) **by** (*auto simp add*: *stutstep-def*)
  from *h1* **obtain** *i* **where** $s\ (i+n) \neq s\ n$ **by** (*auto simp*: *emptyseq-def suffix-def*)
  with *h2* **have** *g2*: $s\ (i+n) \neq s\ (Suc\ n)$ (**is** *?P i*) **by** (*simp add*: *stutstep-def*)
  from *g2 g1* **have** (*LEAST n. ?P n*) = *Suc* (*LEAST n. ?P* (*Suc n*)) **by** (*rule Least-Suc*)
  from *g2 g1* **have** (*LEAST i.* $s\ (i+n) \neq s\ (Suc\ n)$) = *Suc* (*LEAST i.* $s\ ((Suc\ i)+n) \neq s\ (Suc\ n)$)
    by (*rule Least-Suc*)
  hence *G1*: (*LEAST i.* $s\ (i+n) \neq s\ (Suc\ n)$) = *Suc* (*LEAST i.* $s\ (i+Suc\ n) \neq s\ (Suc\ n)$) **by** *auto*
  from *h1 h2* **have** $\neg$ *emptyseq* $(s \mid_s Suc\ n)$ **by** (*simp add*: *stutstep-empty-suc*)
  hence $nextnat\ (s \mid_s Suc\ n) = (LEAST\ i.\ (s \mid_s Suc\ n)\ i \neq (s \mid_s Suc\ n)\ 0)$
    by (*auto simp add*: *nextnat-def*)
  hence *g1*: $nextnat\ (s \mid_s Suc\ n) = (LEAST\ i.\ s\ (i+(Suc\ n)) \neq s\ (Suc\ n))$
    by (*simp add*: *suffix-def*)
  from *h1* **have** $nextnat\ (s \mid_s n) = (LEAST\ i.\ (s \mid_s n)\ i \neq (s \mid_s n)\ 0)$
    by (*auto simp add*: *nextnat-def*)
  hence *g2*: $nextnat\ (s \mid_s n) = (LEAST\ i.\ s\ (i+n) \neq s\ n)$ **by** (*simp add*: *suffix-def*)
  with *h2* **have** *g2'*: $nextnat\ (s \mid_s n) = (LEAST\ i.\ s\ (i+n) \neq s\ (Suc\ n))$
    by (*auto simp add*: *stutstep-def*)
  from *G1 g1 g2'* **show** *?thesis* **by** *auto*
qed

lemma *nextnat-empty-neq*: **assumes** *H*: $\neg$ *emptyseq s* **shows** $s\ (nextnat\ s) \neq s\ 0$
proof −
  from *H* **have** *a1*: $nextnat\ s = (LEAST\ i.\ s\ i \neq s\ 0)$ **by** (*simp add*: *nextnat-def*)
  from *H* **obtain** *i* **where** $s\ i \neq s\ 0$ **by** (*auto simp*: *emptyseq-def*)
  hence $s\ (LEAST\ i.\ s\ i \neq s\ 0) \neq s\ 0$ **by** (*rule LeastI*)
  with *a1* **show** *?thesis* **by** *auto*
qed

lemma *nextnat-empty-gzero*: **assumes** *H*: $\neg$ *emptyseq s* **shows** $nextnat\ s > 0$
proof −
  from *H* **have** *a1*: $s\ (nextnat\ s) \neq s\ 0$ **by** (*rule nextnat-empty-neq*)
  have $nextnat\ s \neq 0$
  proof
    assume $nextnat\ s = 0$
    with *a1* **show** *False* **by** *simp*
  qed

thus *nextnat s* $>$ *0* **by** *simp*
qed

1.3.3 Properties of *nextsuffix*

lemma *empty-nextsuffix*:
assumes *H*: *emptyseq s* **shows** *nextsuffix s* = *s*
using *H* **by** (*simp add*: *nextsuffix-def nextnat-def*)

lemma *empty-nextsuffix-id*:
assumes *H*: *emptyseq s* **shows** *nextsuffix s* = *id s*
using *H* **by** (*simp add*: *empty-nextsuffix*)

lemma *notstutstep-nextsuffix1*:
assumes *H*: $\neg$ *stutstep s n* **shows** *nextsuffix* $(s \mid_s n) = s \mid_s (Suc\ n)$
proof (*unfold nextsuffix-def*)
show $(s \mid_s n \mid_s (nextnat\ (s \mid_s n))) = s \mid_s (Suc\ n)$
proof $-$
from *H* **have** $nextnat\ (s \mid_s n) = 1$ **by** (*rule notstutstep-nexnat1*)
hence $(s \mid_s n \mid_s (nextnat\ (s \mid_s n))) = s \mid_s n \mid_s 1$ **by** *auto*
thus *?thesis* **by** (*simp add*: *suffix-def*)
qed
qed

1.3.4 Properties of *next*

lemma *next-suc-suffix*: *next* (*Suc n*) *s* = *nextsuffix* (*next n s*)
by *simp*

lemma *next-suffix-com*: *nextsuffix* (*next n s*) = (*next n* (*nextsuffix s*))
by (*induct n*, *auto*)

lemma *next-plus*: *next* $(m+n)$ *s* = *next m* (*next n s*)
by (*induct m*, *auto*)

lemma *next-empty*: **assumes** *H*: *emptyseq s* **shows** *next n s* = *s*
proof (*induct n*)
from *H* **show** *next 0 s* = *s* **by** *auto*
next
fix *n*
assume *a1*: *next n s* = *s*
have *next* (*Suc n*) *s* = *nextsuffix* (*next n s*) **by** *auto*
with *a1* **have** *next* (*Suc n*) *s* = *nextsuffix s* **by** *simp*
with *H* **show** *next* (*Suc n*) *s* = *s*
by (*simp add*: *nextsuffix-def nextnat-def*)
qed

lemma *notempty-nextnotzero*:
assumes *H*: $\neg$*emptyseq s* **shows** (*next* (*Suc 0*) *s*) *0* $\neq$ *s 0*
proof $-$

from H **have** $g1$: s ($nextnat$ s) $\neq$ s 0 **by** ($rule$ $nextnat$-$empty$-neq)
have $next$ (Suc 0) s = $nextsuffix$ s **by** $auto$
hence ($next$ (Suc 0) s) 0 = s ($nextnat$ s) **by** ($simp$ add: $nextsuffix$-def $suffix$-def)
with $g1$ **show** $?thesis$ **by** $simp$
qed

lemma *next-ex-id*: $\exists$ i. s i = ($next$ m s) 0
proof $-$
have $\exists$ i. (s $|_s$ i) = ($next$ m s)
proof (*induct m*)
have s $|_s$ 0 = $next$ 0 s **by** *simp*
thus $\exists$ i. (s $|_s$ i) = ($next$ 0 s) **..**
next
fix m
assume $a1$: $\exists$ i. (s $|_s$ i) = ($next$ m s)
then obtain i **where** $a1'$: (s $|_s$ i) = ($next$ m s) **..**
have $next$ (Suc m) s = $nextsuffix$ ($next$ m s) **by** *auto*
hence $next$ (Suc m) s = ($next$ m s) $|_s$ ($nextnat$ ($next$ m s)) **by** (*simp add*: *nextsuffix-def*)
hence $\exists$ i. $next$ (Suc m) s = ($next$ m s) $|_s$ i **..**
then obtain j **where** $next$ (Suc m) s = ($next$ m s) $|_s$ j **..**
with $a1'$ **have** $next$ (Suc m) s = (s $|_s$ i) $|_s$ j **by** *simp*
hence $next$ (Suc m) s = (s $|_s$ (j+i)) **by** (*simp add*: *suffix-plus*)
hence (s $|_s$ (j+i)) = $next$ (Suc m) s **by** *simp*
thus $\exists$ i. (s $|_s$ i) = ($next$ (Suc m) s) **..**
qed
then obtain i **where** (s $|_s$ i) = ($next$ m s) **..**
hence (s $|_s$ i) 0 = ($next$ m s) 0 **by** *auto*
hence s i = ($next$ m s) 0 **by** (*auto simp add*: *suffix-def*)
thus $?thesis$ **..**
qed

1.3.5 Properties of $\natural$

lemma *emptyseq-collapse-eq*: **assumes** $A1$: *emptyseq* s **shows** $\natural$ s = s
proof (*unfold collapse-def*, *rule ext*)
fix n
from $A1$ **have** $next$ n s = s **by** (*rule next-empty*)
moreover
from $A1$ **have** s n = s 0 **by** (*simp add*: *emptyseq-def*)
ultimately
show ($next$ n s) 0 = s n **by** *simp*
qed

lemma *empty-collapse-empty*:
assumes H: *emptyseq* s **shows** *emptyseq* ($\natural$ s)
using H **by** (*simp add*: *emptyseq-collapse-eq*)

lemma *collapse-empty-empty*:

assumes H: *emptyseq* ($\natural$ s) **shows** *emptyseq s*
proof (*rule ccontr*)
assume *a1*: $\neg$*emptyseq s*
from H **have** $\forall$ i. (*next i s*) *0* = *s 0* **by** (*simp add*: *collapse-def emptyseq-def*)
moreover
from *a1* **have** (*next* (*Suc 0*) *s*) *0* $\neq$ *s 0* **by** (*rule notempty-nextnotzero*)
ultimately show *False* **by** *blast*
qed

lemma *collapse-empty-iff-empty* [*simp*]: *emptyseq* ($\natural$ s) = *emptyseq s*
by (*auto elim*: *empty-collapse-empty collapse-empty-empty*)

1.4 Similarity of Sequences

Since adding or removing stuttering steps does not change the validity of a stuttering-invarant formula, equality is often too strong, and the weaker equality *up to stuttering* is sufficient. This is often called *similarity* ($\approx$) of sequences in the literature, and is required to show that logical operators are stuttering invariant. This is mechanised as:

definition *seqsimilar* :: ($'a$ *seq*) $\Rightarrow$ ($'a$ *seq*) $\Rightarrow$ *bool* (**infixl** ‹$\approx$› *50*)
where $\sigma \approx \tau \equiv (\natural\ \sigma) = (\natural\ \tau)$

1.4.1 Properties of ($\approx$)

lemma *seqsim-refl* [*iff*]: $s \approx s$
by (*simp add*: *seqsimilar-def*)

lemma *seqsim-sym*: **assumes** H: $s \approx t$ **shows** $t \approx s$
using H **by** (*simp add*: *seqsimilar-def*)

lemma *seqeq-imp-sim*: **assumes** H: $s = t$ **shows** $s \approx t$
using H **by** *simp*

lemma *seqsim-trans* [*trans*]: **assumes** *h1*: $s \approx t$ **and** *h2*: $t \approx z$ **shows** $s \approx z$
using *assms* **by** (*simp add*: *seqsimilar-def*)

theorem *sim-first*: **assumes** H: $s \approx t$ **shows** *first s* = *first t*
proof −
from H **have** ($\natural$ s) *0* = ($\natural$ t) *0* **by** (*simp add*: *seqsimilar-def*)
thus *?thesis* **by** (*simp add*: *collapse-def first-def*)
qed

lemmas *sim-first2* = *sim-first*[*unfolded first-def*]

lemma *tail-sim-second*: **assumes** H: *tail s* $\approx$ *tail t* **shows** *second s* = *second t*
proof −
from H **have** *first* (*tail s*) = *first* (*tail t*) **by** (*simp add*: *sim-first*)
thus *second s* = *second t* **by** (*simp add*: *first-tail-second*)

qed

lemma *seqsimilarI*:
  assumes *1*: *first s* = *first t* **and** *2*: *nextsuffix s* ≈ *nextsuffix t*
  shows *s* ≈ *t*
  unfolding *seqsimilar-def collapse-def*
proof
  fix *n*
  show *next n s 0* = *next n t 0*
  proof (*cases n*)
    assume *n* = *0*
    with *1* **show** *?thesis* **by** (*simp add*: *first-def*)
  next
    fix *m*
    assume *m*: *n* = *Suc m*
    from *2* **have** *next m* (*nextsuffix s*) *0* = *next m* (*nextsuffix t*) *0*
      unfolding *seqsimilar-def collapse-def* **by** (*rule fun-cong*)
    with *m* **show** *?thesis* **by** (*simp add*: *next-suffix-com*)
  qed
qed

lemma *seqsim-empty-empty*:
  assumes *H1*: *s* ≈ *t* **and** *H2*: *emptyseq s* **shows** *emptyseq t*
proof −
  from *H2* **have** *emptyseq* (♮ *s*) **by** *simp*
  with *H1* **have** *emptyseq* (♮ *t*) **by** (*simp add*: *seqsimilar-def*)
  thus *?thesis* **by** *simp*
qed

lemma *seqsim-empty-iff-empty*:
  assumes *H*: *s* ≈ *t* **shows** *emptyseq s* = *emptyseq t*
proof
  assume *emptyseq s* **with** *H* **show** *emptyseq t* **by** (*rule seqsim-empty-empty*)
next
  assume *t*: *emptyseq t*
  from *H* **have** *t* ≈ *s* **by** (*rule seqsim-sym*)
  from *this t* **show** *emptyseq s* **by** (*rule seqsim-empty-empty*)
qed

lemma *seq-empty-eq*:
  assumes *H1*: *s 0* = *t 0* **and** *H2*: *emptyseq s* **and** *H3*: *emptyseq t*
  shows *s* = *t*
proof (*rule ext*)
  fix *n*
  from *assms* **have** *t n* = *s n* **by** (*auto simp*: *emptyseq-def*)
  thus *s n* = *t n* **by** *simp*
qed

lemma *seqsim-notstutstep*:

```
  assumes H: ¬ (stutstep s n) shows (s |_s (Suc n)) ≈ nextsuffix (s |_s n)
  using H by (simp add: notstutstep-nextsuffix1)

lemma stut-nextsuf-suc:
  assumes H: stutstep s n shows nextsuffix (s |_s n) = nextsuffix (s |_s (Suc n))
proof (cases emptyseq (s |_s n))
  case True
  hence g1: nextsuffix (s |_s n) = (s |_s n) by (simp add: nextsuffix-def nextnat-def)
  from True have g2: nextsuffix (s |_s Suc n) = (s |_s Suc n)
    by (simp add: suc-empty nextsuffix-def nextnat-def)
  have (s |_s n) = (s |_s Suc n)
  proof
    fix x
    from True have s (x + n) = s (0 + n) s (Suc x + n) = s (0 + n)
      unfolding emptyseq-def suffix-def by (blast+)
    thus (s |_s n) x = (s |_s Suc n) x by (simp add: suffix-def)
  qed
  with g1 g2 show ?thesis by auto
next
  case False
  with H have (nextnat (s |_s n)) = Suc (nextnat (s |_s Suc n))
    by (simp add: stutstep-notempty-sucnextnat)
  thus ?thesis
    by (simp add: nextsuffix-def suffix-plus)
qed

lemma seqsim-suffix-seqsim:
  assumes H: s ≈ t shows nextsuffix s ≈ nextsuffix t
  unfolding seqsimilar-def collapse-def
proof
  fix n
  from H have (next (Suc n) s) 0 = (next (Suc n) t) 0
    unfolding seqsimilar-def collapse-def by (rule fun-cong)
  thus next n (nextsuffix s) 0 = next n (nextsuffix t) 0
    by (simp add: next-suffix-com)
qed

lemma seqsim-stutstep:
  assumes H: stutstep s n shows (s |_s (Suc n)) ≈ (s |_s n) (is ?sn ≈ ?s)
  unfolding seqsimilar-def collapse-def
proof
  fix m
  show next m (s |_s Suc n) 0 = next m (s |_s n) 0
  proof (cases m)
    assume m=0
    with H show ?thesis by (simp add: suffix-def stutstep-def)
  next
    fix k
    assume m: m = Suc k
```

with H **have** *next m* $(s \mid_s$ *Suc n*$)$ = *next k* (*nextsuffix* $(s \mid_s n))$
by (*simp add*: *stut-nextsuf-suc next-suffix-com*)
moreover from m **have** *next m* $(s \mid_s n)$ = *next k* (*nextsuffix* $(s \mid_s n))$
by (*simp add*: *next-suffix-com*)
ultimately show *next m* $(s \mid_s$ *Suc n*$)$ *0* = *next m* $(s \mid_s n)$ *0* **by** *simp*
qed
qed

lemma *addfeqstut*: *stutstep* ((*first t*) ## *t*) *0*
by (*simp add*: *first-def stutstep-def app-def suffix-def*)

lemma *addfeqsim*: ((*first t*) ## *t*) ≈ *t*
proof −
have *stutstep* ((*first t*) ## *t*) *0* **by** (*rule addfeqstut*)
hence (((*first t*) ## *t*) $\mid_s$ (*Suc 0*)) ≈ (((*first t*) ## *t*) $\mid_s$ *0*) **by** (*rule seqsim-stutstep*)
hence *tail* ((*first t*) ## *t*) ≈ ((*first t*) ## *t*) **by** (*simp add*: *suffix-def tail-def*)
hence *t* ≈ ((*first t*) ## *t*) **by** (*simp add*: *tail-def app-def suffix-def*)
thus *?thesis* **by** (*rule seqsim-sym*)
qed

lemma *addfirststut*:
assumes *H*: *first s* = *second s* **shows** *s* ≈ *tail s*
proof −
have *g1*: (*first s*) ## (*tail s*) = *s* **by** (*rule seq-app-first-tail*)
from *H* **have** (*first s*) = *first* (*tail s*)
by (*simp add*: *first-def second-def tail-def suffix-def*)
hence (*first s*) ## (*tail s*) ≈ (*tail s*) **by** (*simp add*: *addfeqsim*)
with *g1* **show** *?thesis* **by** *simp*
qed

lemma *app-seqsimilar*:
assumes *h1*: *s* ≈ *t* **shows** (*x* ## *s*) ≈ (*x* ## *t*)
proof (*cases stutstep* (*x* ## *s*) *0*)
case *True*
from *h1* **have** *first s* = *first t* **by** (*rule sim-first*)
with *True* **have** *a2*: *stutstep* (*x* ## *t*) *0*
by (*simp add*: *stutstep-def first-def app-def*)
from *True* **have** ((*x* ## *s*) $\mid_s$ (*Suc 0*)) ≈ ((*x* ## *s*) $\mid_s$ *0*) **by** (*rule seqsim-stutstep*)
hence *tail* (*x* ## *s*) ≈ (*x* ## *s*) **by** (*simp add*: *tail-def suffix-def*)
hence *g1*: *s* ≈ (*x* ## *s*) **by** (*simp add*: *app-def tail-def suffix-def*)
from *a2* **have** ((*x* ## *t*) $\mid_s$ (*Suc 0*)) ≈ ((*x* ## *t*) $\mid_s$ *0*) **by** (*rule seqsim-stutstep*)
hence *tail* (*x* ## *t*) ≈ (*x* ## *t*) **by** (*simp add*: *tail-def suffix-def*)
hence *g2*: *t* ≈ (*x* ## *t*) **by** (*simp add*: *app-def tail-def suffix-def*)
from *h1 g2* **have** *s* ≈ (*x* ## *t*) **by** (*rule seqsim-trans*)
from *this*[*THEN seqsim-sym*] *g1* **show** (*x* ## *s*) ≈ (*x* ## *t*)
by (*rule seqsim-sym*[*OF seqsim-trans*])
next
case *False*

from *h1* **have** *first* s = *first* t **by** (*rule sim-first*)
with *False* **have** *a2*: $\neg$ *stutstep* (x ## t) *0*
  by (*simp add*: *stutstep-def first-def app-def*)
from *False* **have** ((x ## s) $|_s$ (*Suc 0*)) $\approx$ *nextsuffix* ((x ## s) $|_s$ *0*)
  by (*rule seqsim-notstutstep*)
hence (*tail* (x ## s)) $\approx$ *nextsuffix* (x ## s)
  by (*simp add*: *tail-def*)
hence *g1*: $s \approx$ *nextsuffix* (x ## s) **by** (*simp add*: *seq-app-tail*)
from *a2* **have** ((x ## t) $|_s$ (*Suc 0*)) $\approx$ *nextsuffix* ((x ## t) $|_s$ *0*)
  by (*rule seqsim-notstutstep*)
hence (*tail* (x ## t)) $\approx$ *nextsuffix* (x ## t) **by** (*simp add*: *tail-def*)
hence *g2*: $t \approx$ *nextsuffix* (x ## t) **by** (*simp add*: *seq-app-tail*)
with *h1* **have** $s \approx$ *nextsuffix* (x ## t) **by** (*rule seqsim-trans*)
from *this*[*THEN seqsim-sym*] *g1* **have** *g3*: *nextsuffix* (x ## s) $\approx$ *nextsuffix* (x ## t)
  by (*rule seqsim-sym*[*OF seqsim-trans*])
have *first* (x ## s) = *first* (x ## t) **by** (*simp add*: *first-def app-def*)
from *this g3* **show** *?thesis* **by** (*rule seqsimilarI*)
qed

If two sequences are similar then for any suffix of one of them there exists a similar suffix of the other one. We will prove a stronger result below.

lemma *simstep-disj1*: **assumes** *H*: $s \approx t$ **shows** $\exists$ *m*. ((s $|_s$ n) $\approx$ (t $|_s$ m))
proof (*induct n*)
  from *H* **have** ((s $|_s$ *0*) $\approx$ (t $|_s$ *0*)) **by** *auto*
  thus $\exists$ *m*. ((s $|_s$ *0*) $\approx$ (t $|_s$ m)) **..**
next
  fix n
  assume $\exists$ *m*. ((s $|_s$ n) $\approx$ (t $|_s$ m))
  then obtain m **where** *a1'*: (s $|_s$ n) $\approx$ (t $|_s$ m) **..**
  show $\exists$ *m*. ((s $|_s$ (*Suc n*)) $\approx$ (t $|_s$ m))
  proof (*cases stutstep s n*)
    case *True*
    hence (s $|_s$ (*Suc n*)) $\approx$ (s $|_s$ n) **by** (*rule seqsim-stutstep*)
    from *this a1'* **have** ((s $|_s$ (*Suc n*)) $\approx$ (t $|_s$ m)) **by** (*rule seqsim-trans*)
    thus *?thesis* **..**
  next
    case *False*
    hence (s $|_s$ (*Suc n*)) $\approx$ *nextsuffix* (s $|_s$ n) **by** (*rule seqsim-notstutstep*)
    moreover
    from *a1'* **have** *nextsuffix* (s $|_s$ n) $\approx$ *nextsuffix* (t $|_s$ m)
      by (*simp add*: *seqsim-suffix-seqsim*)
    ultimately have (s $|_s$ (*Suc n*)) $\approx$ *nextsuffix* (t $|_s$ m) **by** (*rule seqsim-trans*)
    hence (s $|_s$ (*Suc n*)) $\approx$ t $|_s$ (m + (*nextnat* (t $|_s$ m)))
      by (*simp add*: *nextsuffix-def suffix-plus-com*)
    thus $\exists$ *m*. (s $|_s$ (*Suc n*)) $\approx$ t $|_s$ m **..**
  qed
qed

lemma *nextnat-le-seqsim*:
  assumes *n*: $n < nextnat\ s$ **shows** $s \approx (s\ |_s\ n)$
proof (*cases emptyseq s*)
  case *True* — case impossible
  with *n* **show** *?thesis* **by** (*simp add*: *nextnat-def*)
next
  case *False*
  from *n* **show** *?thesis*
  proof (*induct n*)
    show $s \approx (s\ |_s\ 0)$ **by** *simp*
  next
    fix *n*
    assume *a2*: $n < nextnat\ s \Longrightarrow s \approx (s\ |_s\ n)$ **and** *a3*: $Suc\ n < nextnat\ s$
    from *a3* **have** *g1*: $s\ (Suc\ n) = s\ 0$ **by** (*rule nextnat-le-unch*)
    from *a3* **have** *a3'*: $n < nextnat\ s$ **by** *simp*
    hence $s\ n = s\ 0$ **by** (*rule nextnat-le-unch*)
    with *g1* **have** *stutstep s n* **by** (*simp add*: *stutstep-def*)
    hence *g2*: $(s\ |_s\ n) \approx (s\ |_s\ (Suc\ n))$ **by** (*rule seqsim-stutstep*[*THEN seqsim-sym*])
    with *a3' a2* **show** $s \approx (s\ |_s\ (Suc\ n))$ **by** (*auto elim*: *seqsim-trans*)
  qed
qed

lemma *seqsim-prev-nextnat*: $s \approx s\ |_s\ ((nextnat\ s) - 1)$
proof (*cases emptyseq s*)
  case *True*
  hence $s\ |_s\ ((nextnat\ s) - (1{::}nat)) = s\ |_s\ 0$ **by** (*simp add*: *nextnat-def*)
  thus *?thesis* **by** *simp*
next
  case *False*
  hence $nextnat\ s > 0$ **by** (*rule nextnat-empty-gzero*)
  thus *?thesis* **by** (*simp add*: *nextnat-le-seqsim*)
qed

Given a suffix $s\ |_s\ n$ of some sequence s that is similar to some suffix $t\ |_s\ m$ of sequence t, there exists some suffix $t\ |_s\ m'$ of t such that $s\ |_s\ n$ and $t\ |_s\ m'$ are similar and also $s\ |_s\ (n+1)$ is similar to either $t\ |_s\ m'$ or to $t\ |_s\ (m'+1)$.

lemma *seqsim-suffix-suc*:
  assumes *H*: $s\ |_s\ n \approx t\ |_s\ m$
  shows $\exists\, m'.\ s\ |_s\ n \approx t\ |_s\ m' \wedge ((s\ |_s\ Suc\ n \approx t\ |_s\ Suc\ m') \vee (s\ |_s\ Suc\ n \approx t\ |_s\ m'))$
proof (*cases stutstep s n*)
  case *True*
  hence $s\ |_s\ Suc\ n \approx s\ |_s\ n$ **by** (*rule seqsim-stutstep*)
  from *this H* **have** $s\ |_s\ Suc\ n \approx t\ |_s\ m$ **by** (*rule seqsim-trans*)
  with *H* **show** *?thesis* **by** *blast*
next
  case *False*
  hence $\neg$ *emptyseq* $(s\ |_s\ n)$ **by** (*rule stutnempty*)

with *H* **have** *a2*: $\neg$ *emptyseq* $(t \mid_s m)$ **by** (*simp add*: *seqsim-empty-iff-empty*)
hence *g4*: *nextsuffix* $(t \mid_s m) = (t \mid_s m) \mid_s$ *Suc* (*nextnat* $(t \mid_s m) - 1$)
by (*simp add*: *nextnat-empty-gzero nextsuffix-def*)
have *g3*: $(t \mid_s m) \approx (t \mid_s m) \mid_s$ (*nextnat* $(t \mid_s m) - 1$)
by (*rule seqsim-prev-nextnat*)
with *H* **have** *G1*: $s \mid_s n \approx (t \mid_s m) \mid_s$ (*nextnat* $(t \mid_s m) - 1$)
by (*rule seqsim-trans*)
from *False* **have** *G1'*: $(s \mid_s$ *Suc* $n) =$ *nextsuffix* $(s \mid_s n)$
by (*rule notstutstep-nextsuffix1*[*THEN sym*])
from *H* **have** *nextsuffix* $(s \mid_s n) \approx$ *nextsuffix* $(t \mid_s m)$
by (*rule seqsim-suffix-seqsim*)
with *G1 G1' g4*
have $s \mid_s n \approx t \mid_s (m + ($*nextnat* $(t \mid_s m) - 1))$
$\wedge\ s \mid_s ($*Suc* $n) \approx t \mid_s$ *Suc* $(m + ($*nextnat* $(t \mid_s m) - 1))$
by (*simp add*: *suffix-plus-com*)
thus *?thesis* **by** *blast*
qed

The following main result about similar sequences shows that if $s \approx t$ holds then for any suffix $s \mid_s n$ of s there exists a suffix $t \mid_s m$ such that

- $s \mid_s n$ and $t \mid_s m$ are similar, and
- $s \mid_s (n+1)$ is similar to either $t \mid_s (m+1)$ or $t \mid_s m$.

The idea is to pick the largest m such that $s \mid_s n \approx t \mid_s m$ (or some such m if $s \mid_s n$ is empty).

theorem *sim-step*:
assumes *H*: $s \approx t$
shows $\exists\ m.\ s \mid_s n \approx t \mid_s m \wedge$
$((s \mid_s$ *Suc* $n \approx t \mid_s$ *Suc* $m) \vee (s \mid_s$ *Suc* $n \approx t \mid_s m))$
(**is** $\exists m.$ *?Sim n m*)
proof (*induct n*)
from *H* **have** $s \mid_s 0 \approx t \mid_s 0$ **by** *simp*
thus $\exists\ m.$ *?Sim 0 m* **by** (*rule seqsim-suffix-suc*)
next
fix *n*
assume $\exists\ m.$ *?Sim n m*
hence $\exists k.\ s \mid_s$ *Suc* $n \approx t \mid_s k$ **by** *blast*
thus $\exists\ m.$ *?Sim* (*Suc n*) *m* **by** (*blast dest*: *seqsim-suffix-suc*)
qed

end

2 Representing Intensional Logic

theory *Intensional*
imports *Main*

begin

In higher-order logic, every proof rule has a corresponding tautology, i.e. the *deduction theorem* holds. Isabelle/HOL implements this since object-level implication ($\longrightarrow$) and meta-level entailment ($\Longrightarrow$) commute, viz. the proof rule *impI*: $(?P \Longrightarrow ?Q) \Longrightarrow ?P \longrightarrow ?Q$. However, the deduction theorem does not hold for most modal and temporal logics [6, page 95][7]. For example $A \vdash \Box A$ holds, meaning that if A holds in any world, then it always holds. However, $\vdash A \longrightarrow \Box A$, stating that A always holds if it initially holds, is not valid.

Merz [7] overcame this problem by creating an *Intensional* logic. It exploits Isabelle's axiomatic type class feature [9] by creating a type class *world*, which provides Skolem constants to associate formulas with the world they hold in. The class is trivial, not requiring any axioms.

class *world*

world is a type class of possible worlds. It is a subclass of all HOL types *type*. No axioms are provided, since its only purpose is to avoid silly use of the *Intensional* syntax.

2.1 Abstract Syntax and Definitions

type-synonym $('w,'a)\ expr = 'w \Rightarrow 'a$
type-synonym $'w\ form = ('w,\ bool)\ expr$

The intention is that $'a$ will be used for unlifted types (class *type*), while $'w$ is lifted (class *world*).

definition $Valid :: ('w::world)\ form \Rightarrow bool$
where $Valid\ A \equiv \forall\, w.\ A\ w$

definition $const :: 'a \Rightarrow ('w::world,\ 'a)\ expr$
where $unl\text{-}con$: $const\ c\ w \equiv c$

definition $lift :: ['a \Rightarrow 'b,\ ('w::world,\ 'a)\ expr] \Rightarrow ('w,'b)\ expr$
where $unl\text{-}lift$: $lift\ f\ x\ w \equiv f\ (x\ w)$

definition $lift2 :: ['a \Rightarrow 'b \Rightarrow 'c,\ ('w::world,'a)\ expr,\ ('w,'b)\ expr] \Rightarrow ('w,'c)\ expr$
where $unl\text{-}lift2$: $lift2\ f\ x\ y\ w \equiv f\ (x\ w)\ (y\ w)$

definition $lift3 :: ['a \Rightarrow 'b => 'c \Rightarrow 'd,\ ('w::world,'a)\ expr,\ ('w,'b)\ expr,\ ('w,'c)\ expr] \Rightarrow ('w,'d)\ expr$
where $unl\text{-}lift3$: $lift3\ f\ x\ y\ z\ w \equiv f\ (x\ w)\ (y\ w)\ (z\ w)$

definition $lift4 :: ['a \Rightarrow 'b => 'c \Rightarrow 'd \Rightarrow 'e,\ ('w::world,'a)\ expr,\ ('w,'b)\ expr,\ ('w,'c)\ expr,('w,'d)\ expr] \Rightarrow ('w,'e)\ expr$
where $unl\text{-}lift4$: $lift4\ f\ x\ y\ z\ zz\ w \equiv f\ (x\ w)\ (y\ w)\ (z\ w)\ (zz\ w)$

Valid F asserts that the lifted formula *F* holds everywhere. *const* allows lifting of a constant, while *lift* through *lift4* allow functions with arity 1–4 to be lifted. (Note that there is no way to define a generic lifting operator for functions of arbitrary arity.)

definition *RAll* :: ('*a* ⇒ ('*w*::*world*) *form*) ⇒ '*w form* (**binder** ‹*Rall* › *10*)
  where *unl-Rall*: (*Rall x. A x*) *w* ≡ ∀ *x. A x w*

definition *REx* :: ('*a* ⇒ ('*w*::*world*) *form*) ⇒ '*w form* (**binder** ‹*Rex* › *10*)
  where *unl-Rex*: (*Rex x. A x*) *w* ≡ ∃ *x. A x w*

definition *REx1* :: ('*a* ⇒ ('*w*::*world*) *form*) ⇒ '*w form* (**binder** ‹*Rex*! › *10*)
  where *unl-Rex1*: (*Rex*! *x. A x*) *w* ≡ ∃!*x. A x w*

RAll, *REx* and *REx1* introduces "rigid" quantification over values (of non-world types) within "intensional" formulas. *RAll* is universal quantification, *REx* is existential quantifcation. *REx1* requires unique existence.

We declare the "unlifting rules" as rewrite rules that will be applied automatically.

lemmas *intensional-rews*[*simp*] =
  unl-con unl-lift unl-lift2 unl-lift3 unl-lift4
  unl-Rall unl-Rex unl-Rex1

2.2 Concrete Syntax

nonterminal
  lift **and** *liftargs*

The non-terminal *lift* represents lifted expressions. The idea is to use Isabelle's macro mechanism to convert between the concrete and abstract syntax.

syntax

	:: *id* ⇒ *lift*	(‹-›)
	:: *longid* ⇒ *lift*	(‹-›)
	:: *var* ⇒ *lift*	(‹-›)
-applC	:: [*lift*, *cargs*] ⇒ *lift*	(‹(*1*-/ -)› [*1000*, *1000*] *999*)
	:: *lift* ⇒ *lift*	(‹'(-')›)
-lambda	:: [*idts*, '*a*] ⇒ *lift*	(‹(*3*%-./ -)› [*0*, *3*] *3*)
-constrain	:: [*lift*, *type*] ⇒ *lift*	(‹(-::-)› [*4*, *0*] *3*)
	:: *lift* ⇒ *liftargs*	(‹-›)
-liftargs	:: [*lift*, *liftargs*] ⇒ *liftargs*	(‹-,/ -›)
-Valid	:: *lift* ⇒ *bool*	(‹(⊢ -)› *5*)
-holdsAt	:: ['*a*, *lift*] ⇒ *bool*	(‹(- ⊨ -)› [*100*,*10*] *10*)

LIFT	:: *lift* ⇒ '*a*	(‹*LIFT* -›)

-const :: *'a* ⇒ *lift* (‹(#-)› [*1000*] *999*)
-lift :: [*'a*, *lift*] ⇒ *lift* (‹(-<->)› [*1000*] *999*)
-lift2 :: [*'a*, *lift*, *lift*] ⇒ *lift* (‹(-<-,/ ->)› [*1000*] *999*)
-lift3 :: [*'a*, *lift*, *lift*, *lift*] ⇒ *lift* (‹(-<-,/ -,/ ->)› [*1000*] *999*)
-lift4 :: [*'a*, *lift*, *lift*, *lift*,*lift*] ⇒ *lift* (‹(-<-,/ -,/ -,/ ->)› [*1000*] *999*)

-liftEqu :: [*lift*, *lift*] ⇒ *lift* (‹(- =/ -)› [*50*,*51*] *50*)
-liftNeq :: [*lift*, *lift*] ⇒ *lift* (**infixl** ‹≠› *50*)
-liftNot :: *lift* ⇒ *lift* (‹¬ -› [*90*] *90*)
-liftAnd :: [*lift*, *lift*] ⇒ *lift* (**infixr** ‹∧› *35*)
-liftOr :: [*lift*, *lift*] ⇒ *lift* (**infixr** ‹∨› *30*)
-liftImp :: [*lift*, *lift*] ⇒ *lift* (**infixr** ‹⟶› *25*)
-liftIf :: [*lift*, *lift*, *lift*] ⇒ *lift* (‹(*if* (-)/ *then* (-)/ *else* (-))› *10*)
-liftPlus :: [*lift*, *lift*] ⇒ *lift* (‹(- +/ -)› [*66*,*65*] *65*)
-liftMinus :: [*lift*, *lift*] ⇒ *lift* (‹(- −/ -)› [*66*,*65*] *65*)
-liftTimes :: [*lift*, *lift*] ⇒ *lift* (‹(- ∗/ -)› [*71*,*70*] *70*)
-liftDiv :: [*lift*, *lift*] ⇒ *lift* (‹(- *div* -)› [*71*,*70*] *70*)
-liftMod :: [*lift*, *lift*] ⇒ *lift* (‹(- *mod* -)› [*71*,*70*] *70*)
-liftLess :: [*lift*, *lift*] ⇒ *lift* (‹(-/ < -)› [*50*, *51*] *50*)
-liftLeq :: [*lift*, *lift*] ⇒ *lift* (‹(-/ ≤ -)› [*50*, *51*] *50*)
-liftMem :: [*lift*, *lift*] ⇒ *lift* (‹(-/ ∈ -)› [*50*, *51*] *50*)
-liftNotMem :: [*lift*, *lift*] ⇒ *lift* (‹(-/ ∉ -)› [*50*, *51*] *50*)
-liftFinset :: *liftargs* => *lift* (‹{(-)}›)

-liftPair :: [*lift*,*liftargs*] ⇒ *lift* (‹(*1*'(-,/ -'))›)

-liftCons :: [*lift*, *lift*] ⇒ *lift* (‹(- #/ -)› [*65*,*66*] *65*)
-liftApp :: [*lift*, *lift*] ⇒ *lift* (‹(- @/ -)› [*65*,*66*] *65*)
-liftList :: *liftargs* ⇒ *lift* (‹[(-)]›)

-ARAll :: [*idts*, *lift*] ⇒ *lift* (‹(*3*! -./ -)› [*0*, *10*] *10*)
-AREx :: [*idts*, *lift*] ⇒ *lift* (‹(*3?* -./ -)› [*0*, *10*] *10*)
-AREx1 :: [*idts*, *lift*] ⇒ *lift* (‹(*3?*! -./ -)› [*0*, *10*] *10*)
-RAll :: [*idts*, *lift*] ⇒ *lift* (‹(*3*∀ -./ -)› [*0*, *10*] *10*)
-REx :: [*idts*, *lift*] ⇒ *lift* (‹(*3*∃ -./ -)› [*0*, *10*] *10*)
-REx1 :: [*idts*, *lift*] ⇒ *lift* (‹(*3*∃!-./ -)› [*0*, *10*] *10*)

translations
-const ⇌ *CONST const*

translations
-lift ⇌ *CONST lift*
-lift2 ⇌ *CONST lift2*
-lift3 ⇌ *CONST lift3*
-lift4 ⇌ *CONST lift4*
-Valid ⇌ *CONST Valid*

translations

$\textit{-RAll } x\ A \rightleftharpoons \textit{Rall } x.\ A$
$\textit{-REx } x\ A \rightleftharpoons \textit{Rex } x.\ A$
$\textit{-REx1 } x\ A \rightleftharpoons \textit{Rex}!\ x.\ A$

translations

$\textit{-ARAll} \rightharpoonup \textit{-RAll}$
$\textit{-AREx} \rightharpoonup \textit{-REx}$
$\textit{-AREx1} \rightharpoonup \textit{-REx1}$

$w \models A \rightharpoonup A\ w$
$\textit{LIFT } A \rightharpoonup A\text{::-}\Rightarrow\text{-}$

translations

$\textit{-liftEqu} \rightleftharpoons \textit{-lift2 } (=)$
$\textit{-liftNeq } u\ v \rightleftharpoons \textit{-liftNot } (\textit{-liftEqu } u\ v)$
$\textit{-liftNot} \rightleftharpoons \textit{-lift } (\textit{CONST Not})$
$\textit{-liftAnd} \rightleftharpoons \textit{-lift2 } (\&)$
$\textit{-liftOr} \rightleftharpoons \textit{-lift2 } ((|)\ )$
$\textit{-liftImp} \rightleftharpoons \textit{-lift2 } (-->)$
$\textit{-liftIf} \rightleftharpoons \textit{-lift3 } (\textit{CONST If})$
$\textit{-liftPlus} \rightleftharpoons \textit{-lift2 } (+)$
$\textit{-liftMinus} \rightleftharpoons \textit{-lift2 } (-)$
$\textit{-liftTimes} \rightleftharpoons \textit{-lift2 } (*)$
$\textit{-liftDiv} \rightleftharpoons \textit{-lift2 } (\textit{div})$
$\textit{-liftMod} \rightleftharpoons \textit{-lift2 } (\textit{mod})$
$\textit{-liftLess} \rightleftharpoons \textit{-lift2 } (<)$
$\textit{-liftLeq} \rightleftharpoons \textit{-lift2 } (<=)$
$\textit{-liftMem} \rightleftharpoons \textit{-lift2 } (:)$
$\textit{-liftNotMem } x\ xs \rightleftharpoons \textit{-liftNot } (\textit{-liftMem } x\ xs)$

translations

$\textit{-liftFinset } (\textit{-liftargs } x\ xs) \rightleftharpoons \textit{-lift2 } (\textit{CONST insert})\ x\ (\textit{-liftFinset } xs)$
$\textit{-liftFinset } x \rightleftharpoons \textit{-lift2 } (\textit{CONST insert})\ x\ (\textit{-const } (\textit{CONST Set.empty}))$
$\textit{-liftPair } x\ (\textit{-liftargs } y\ z) \rightleftharpoons \textit{-liftPair } x\ (\textit{-liftPair } y\ z)$
$\textit{-liftPair} \rightleftharpoons \textit{-lift2 } (\textit{CONST Pair})$
$\textit{-liftCons} \rightleftharpoons \textit{-lift2 } (\textit{CONST Cons})$
$\textit{-liftApp} \rightleftharpoons \textit{-lift2 } (@)$
$\textit{-liftList } (\textit{-liftargs } x\ xs) \rightleftharpoons \textit{-liftCons } x\ (\textit{-liftList } xs)$
$\textit{-liftList } x \rightleftharpoons \textit{-liftCons } x\ (\textit{-const } [])$

$w \models \neg\ A \leftharpoondown \textit{-liftNot } A\ w$
$w \models A \wedge B \leftharpoondown \textit{-liftAnd } A\ B\ w$
$w \models A \vee B \leftharpoondown \textit{-liftOr } A\ B\ w$
$w \models A \longrightarrow B \leftharpoondown \textit{-liftImp } A\ B\ w$
$w \models u = v \leftharpoondown \textit{-liftEqu } u\ v\ w$
$w \models \forall x.\ A \leftharpoondown \textit{-RAll } x\ A\ w$
$w \models \exists x.\ A \leftharpoondown \textit{-REx } x\ A\ w$
$w \models \exists!x.\ A \leftharpoondown \textit{-REx1 } x\ A\ w$

syntax (*ASCII*)
-Valid :: *lift* ⇒ *bool* (‹(|− -)› *5*)
-holdsAt :: [*'a*, *lift*] ⇒ *bool* (‹(- |= -)› [*100*,*10*] *10*)
-liftNeq :: [*lift*, *lift*] ⇒ *lift* (‹(- ~=/ -)› [*50*,*51*] *50*)
-liftNot :: *lift* ⇒ *lift* (‹(~ -)› [*90*] *90*)
-liftAnd :: [*lift*, *lift*] ⇒ *lift* (‹(- &/ -)› [*36*,*35*] *35*)
-liftOr :: [*lift*, *lift*] ⇒ *lift* (‹(- |/ -)› [*31*,*30*] *30*)
-liftImp :: [*lift*, *lift*] ⇒ *lift* (‹(- −−>/ -)› [*26*,*25*] *25*)
-liftLeq :: [*lift*, *lift*] ⇒ *lift* (‹(-/ <= -)› [*50*, *51*] *50*)
-liftMem :: [*lift*, *lift*] ⇒ *lift* (‹(-/ : -)› [*50*, *51*] *50*)
-liftNotMem :: [*lift*, *lift*] ⇒ *lift* (‹(-/ ~: -)› [*50*, *51*] *50*)
-RAll :: [*idts*, *lift*] ⇒ *lift* (‹(*3ALL* -./ -)› [*0*, *10*] *10*)
-REx :: [*idts*, *lift*] ⇒ *lift* (‹(*3EX* -./ -)› [*0*, *10*] *10*)
-REx1 :: [*idts*, *lift*] ⇒ *lift* (‹(*3EX*! -./ -)› [*0*, *10*] *10*)

2.3 Lemmas and Tactics

lemma *intD*[*dest*]: ⊢ *A* ⟹ *w* ⊨ *A*
proof −
assume *a*:⊢ *A*
from *a* **have** ∀ *w*. *w* ⊨ *A* **by** (*auto simp add*: *Valid-def*)
thus *?thesis* **..**
qed

lemma *intI* [*intro*!]: **assumes** *P1*:(⋀ *w*. *w* ⊨ *A*) **shows** ⊢ *A*
using *assms* **by** (*auto simp*: *Valid-def*)

Basic unlifting introduces a parameter *w* and applies basic rewrites, e.g ⊢ *F* = *G* becomes *F w* = *G w* and ⊢ *F* ⟶ *G* becomes *F w* ⟶ *G w*.

method-setup *int-unlift* = ‹
Scan.succeed (*fn ctxt* => *SIMPLE-METHOD'*
(*resolve-tac ctxt* @{*thms intI*} *THEN'* *rewrite-goal-tac ctxt* @{*thms intensional-rews*}))
› *method to unlift and followed by intensional rewrites*

lemma *inteq-reflection*: **assumes** *P1*: ⊢ *x*=*y* **shows** (*x* ≡ *y*)
proof −
from *P1* **have** *P2*: ∀ *w*. *x w* = *y w* **by** (*unfold Valid-def unl-lift2*)
hence *P3*:*x*=*y* **by** *blast*
thus *x* ≡ *y* **by** (*rule eq-reflection*)
qed

lemma *int-simps*:
⊢ (*x*=*x*) = #*True*
⊢ (¬ #*True*) = #*False*
⊢ (¬ #*False*) = #*True*
⊢ (¬¬ *P*) = *P*
⊢ ((¬ *P*) = *P*) = #*False*

$\vdash (P = (\neg P)) = \#False$
$\vdash (P \neq Q) = (P = (\neg\ Q))$
$\vdash (\#True{=}P) = P$
$\vdash (P{=}\#True) = P$
$\vdash (\#True \longrightarrow P) = P$
$\vdash (\#False \longrightarrow P) = \#True$
$\vdash (P \longrightarrow \#True) = \#True$
$\vdash (P \longrightarrow P) = \#True$
$\vdash (P \longrightarrow \#False) = (\neg P)$
$\vdash (P \longrightarrow {\sim}P) = (\neg P)$
$\vdash (P \wedge \#True) = P$
$\vdash (\#True \wedge P) = P$
$\vdash (P \wedge \#False) = \#False$
$\vdash (\#False \wedge P) = \#False$
$\vdash (P \wedge P) = P$
$\vdash (P \wedge {\sim}P) = \#False$
$\vdash (\neg P \wedge P) = \#False$
$\vdash (P \vee \#True) = \#True$
$\vdash (\#True \vee P) = \#True$
$\vdash (P \vee \#False) = P$
$\vdash (\#False \vee P) = P$
$\vdash (P \vee P) = P$
$\vdash (P \vee \neg P) = \#True$
$\vdash (\neg P \vee P) = \#True$
$\vdash (\forall\ x.\ P) = P$
$\vdash (\exists\ x.\ P) = P$
by *auto*

lemmas *intensional-simps*[*simp*] = *int-simps*[*THEN inteq-reflection*]

method-setup *int-rewrite* = ‹
Scan.succeed (*fn ctxt* => *SIMPLE-METHOD′* (*rewrite-goal-tac ctxt* @{*thms intensional-simps*}))
› *rewrite method at intensional level*

lemma *Not-Rall*: $\vdash (\neg(\forall\ x.\ F\ x)) = (\exists\ x.\ \neg F\ x)$
by *auto*

lemma *Not-Rex*: $\vdash (\neg(\exists\ x.\ F\ x)) = (\forall\ x.\ \neg F\ x)$
by *auto*

lemma *TrueW* [*simp*]: $\vdash \#True$
by *auto*

lemma *int-eq*: $\vdash X = Y \Longrightarrow X = Y$
by (*auto simp*: *inteq-reflection*)

lemma *int-iffI*:
assumes $\vdash F \longrightarrow G$ **and** $\vdash G \longrightarrow F$

shows $\vdash F = G$
using *assms* **by** *force*

lemma *int-iffD1*: **assumes** h: $\vdash F = G$ **shows** $\vdash F \longrightarrow G$
using h **by** *auto*

lemma *int-iffD2*: **assumes** h: $\vdash F = G$ **shows** $\vdash G \longrightarrow F$
using h **by** *auto*

lemma *lift-imp-trans*:
assumes $\vdash A \longrightarrow B$ **and** $\vdash B \longrightarrow C$
shows $\vdash A \longrightarrow C$
using *assms* **by** *force*

lemma *lift-imp-neg*: **assumes** $\vdash A \longrightarrow B$ **shows** $\vdash \neg B \longrightarrow \neg A$
using *assms* **by** *auto*

lemma *lift-and-com*: $\vdash (A \wedge B) = (B \wedge A)$
by *auto*

end

3 Semantics

theory *Semantics*
imports *Sequence Intensional*
begin

This theory mechanises a *shallow* embedding of TLA* using the *Sequence* and *Intensional* theories. A shallow embedding represents TLA* using Isabelle/HOL predicates, while a *deep* embedding would represent TLA* formulas and pre-formulas as mutually inductive datatypes[1]. The choice of a shallow over a deep embedding is motivated by the following factors: a shallow embedding is usually less involved, and existing Isabelle theories and tools can be applied more directly to enhance automation; due to the lifting in the *Intensional* theory, a shallow embedding can reuse standard logical operators, whilst a deep embedding requires a different set of operators for both formulas and pre-formulas. Finally, since our target is system verification rather than proving meta-properties of TLA*, which requires a deep embedding, a shallow embedding is more fit for purpose.

3.1 Types of Formulas

To mechanise the TLA* semantics, the following type abbreviations are used:

type-synonym $('a,'b)$ *formfun* $=$ $'a$ *seq* $\Rightarrow$ $'b$

[1]See e.g. [10] for a discussion about deep vs. shallow embeddings in Isabelle/HOL.

type-synonym *'a formula* = (*'a*,*bool*) *formfun*
type-synonym (*'a*,*'b*) *stfun* = *'a* ⇒ *'b*
type-synonym *'a stpred* = (*'a*,*bool*) *stfun*

instance
fun :: (*type*,*type*) *world* **..**

instance
prod :: (*type*,*type*) *world* **..**

Pair and function are instantiated to be of type class world. This allows use of the lifted intensional logic for formulas, and standard logical connectives can therefore be used.

3.2 Semantics of TLA*

The semantics of TLA* is defined.

definition *always* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *always F* ≡ λ *s*. ∀ *n*. (*s* $|_s$ *n*) ⊨ *F*

definition *nexts* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *nexts F* ≡ λ *s*. (*tail s*) ⊨ *F*

definition *before* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun*
where *before f* ≡ λ *s*. (*first s*) ⊨ *f*

definition *after* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun*
where *after f* ≡ λ *s*. (*second s*) ⊨ *f*

definition *unch* :: (*'a*::*world*,*'b*) *stfun* ⇒ *'a formula*
where *unch v* ≡ λ *s*. *s* ⊨ (*after v*) = (*before v*)

definition *action* :: (*'a*::*world*) *formula* ⇒ (*'a*,*'b*) *stfun* ⇒ *'a formula*
where *action P v* ≡ λ *s*. ∀ *i*. ((*s* $|_s$ *i*) ⊨ *P*) ∨ ((*s* $|_s$ *i*) ⊨ *unch v*)

3.2.1 Concrete Syntax

This is the concrete syntax for the (abstract) operators above.

syntax
-always :: *lift* ⇒ *lift* (‹(□-)› [*90*] *90*)
-nexts :: *lift* ⇒ *lift* (‹(○-)› [*90*] *90*)
-action :: [*lift*,*lift*] ⇒ *lift* (‹(□[-]'-(-))› [*20*,*1000*] *90*)
-before :: *lift* ⇒ *lift* (‹($-)› [*100*] *99*)
-after :: *lift* ⇒ *lift* (‹(-$)› [*100*] *99*)
-prime :: *lift* ⇒ *lift* (‹(-`)› [*100*] *99*)
-unch :: *lift* ⇒ *lift* (‹(*Unchanged* -)› [*100*] *99*)
TEMP :: *lift* ⇒ *'b* (‹(*TEMP* -)›)

syntax (*ASCII*)
-always :: *lift* ⇒ *lift* (‹([]-)› [*90*] *90*)
-nexts :: *lift* ⇒ *lift* (‹(*Next* -)› [*90*] *90*)
-action :: [*lift*,*lift*] ⇒ *lift* (‹([][-]′-(-))› [*20*,*1000*] *90*)

translations
-always ⇌ *CONST always*
-nexts ⇌ *CONST nexts*
-action ⇌ *CONST action*
-before ⇌ *CONST before*
-after ⇌ *CONST after*
-prime ⇀ *CONST after*
-unch ⇌ *CONST unch*
TEMP F ⇀ (*F*:: (*nat* ⇒ -) ⇒ -)

3.3 Abbreviations

Some standard temporal abbreviations, with their concrete syntax.

definition *actrans* :: (′*a*::*world*) *formula* ⇒ (′*a*,′*b*) *stfun* ⇒ ′*a formula*
where *actrans P v* ≡ *TEMP*(*P* ∨ *unch v*)

definition *eventually* :: (′*a*::*world*) *formula* ⇒ ′*a formula*
where *eventually F* ≡ *LIFT*(¬□(¬*F*))

definition *angle-action* :: (′*a*::*world*) *formula* ⇒ (′*a*,′*b*) *stfun* ⇒ ′*a formula*
where *angle-action P v* ≡ *LIFT*(¬□[¬*P*]-*v*)

definition *angle-actrans* :: (′*a*::*world*) *formula* ⇒ (′*a*,′*b*) *stfun* ⇒ ′*a formula*
where *angle-actrans P v* ≡ *TEMP* (¬ *actrans* (*LIFT*(¬*P*)) *v*)

definition *leadsto* :: (′*a*::*world*) *formula* ⇒ ′*a formula* ⇒ ′*a formula*
where *leadsto P Q* ≡ *LIFT* □(*P* ⟶ *eventually Q*)

3.3.1 Concrete Syntax

syntax (*ASCII*)
-actrans :: [*lift*,*lift*] ⇒ *lift* (‹([-]′-(-))› [*20*,*1000*] *90*)
-eventually :: *lift* ⇒ *lift* (‹(<>-)› [*90*] *90*)
-angle-action :: [*lift*,*lift*] ⇒ *lift* (‹(<><->′-(-))› [*20*,*1000*] *90*)
-angle-actrans :: [*lift*,*lift*] ⇒ *lift* (‹(<->′-(-))› [*20*,*1000*] *90*)
-leadsto :: [*lift*,*lift*] ⇒ *lift* (‹(- ~> -)› [*26*,*25*] *25*)

syntax
-eventually :: *lift* ⇒ *lift* (‹(◇-)› [*90*] *90*)
-angle-action :: [*lift*,*lift*] ⇒ *lift* (‹(◇⟨-⟩′-(-))› [*20*,*1000*] *90*)
-angle-actrans :: [*lift*,*lift*] ⇒ *lift* (‹(⟨-⟩′-(-))› [*20*,*1000*] *90*)
-leadsto :: [*lift*,*lift*] ⇒ *lift* (‹(- ⇝ -)› [*26*,*25*] *25*)

translations

-actrans $\rightleftharpoons$ *CONST actrans*
-eventually $\rightleftharpoons$ *CONST eventually*
-angle-action $\rightleftharpoons$ *CONST angle-action*
-angle-actrans $\rightleftharpoons$ *CONST angle-actrans*
-leadsto $\rightleftharpoons$ *CONST leadsto*

3.4 Properties of Operators

The following lemmas show that these operators have the expected semantics.

lemma *eventually-defs*: $(w \models \Diamond\ F) = (\exists\ n.\ (w\ |_s\ n) \models F)$
by (*simp add*: *eventually-def always-def*)

lemma *angle-action-defs*: $(w \models \Diamond\langle P\rangle\text{-}v) = (\exists\ i.\ ((w\ |_s\ i) \models P) \wedge ((w\ |_s\ i) \models v\$ \neq \$v))$
by (*simp add*: *angle-action-def action-def unch-def*)

lemma *unch-defs*: $(w \models \mathit{Unchanged}\ v) = (((\mathit{second}\ w) \models v) = ((\mathit{first}\ w) \models v))$
by (*simp add*: *unch-def before-def nexts-def after-def tail-def suffix-def first-def second-def*)

lemma *linalw*:
assumes *h1*: $a \leq b$ **and** *h2*: $(w\ |_s\ a) \models \Box A$
shows $(w\ |_s\ b) \models \Box A$
proof (*clarsimp simp*: *always-def*)
fix *n*
from *h1* **obtain** *k* **where** *g1*: $b = a + k$ **by** (*auto simp*: *le-iff-add*)
with *h2* **show** $(w\ |_s\ b\ |_s\ n) \models A$ **by** (*auto simp*: *always-def suffix-plus ac-simps*)
qed

3.5 Invariance Under Stuttering

A key feature of TLA* is that specification at different abstraction levels can be compared. The soundness of this relies on the stuttering invariance of formulas. Since the embedding is shallow, it cannot be shown that a generic TLA* formula is stuttering invariant. However, this section will show that each operator is stuttering invariant or preserves stuttering invariance in an appropriate sense, which can be used to show stuttering invariance for given specifications.

Formula F is stuttering invariant if for any two similar behaviours (i.e., sequences of states), F holds in one iff it holds in the other. The definition is generalised to arbitrary expressions, and not just predicates.

definition *stutinv* :: $('a,'b)$ *formfun* $\Rightarrow$ *bool*
where *stutinv* $F \equiv \forall\ \sigma\ \tau.\ \sigma \approx \tau \longrightarrow (\sigma \models F) = (\tau \models F)$

The requirement for stuttering invariance is too strong for pre-formulas. For example, an action formula specifies a relation between the first two states of a behaviour, and will rarely be satisfied by a stuttering step. This is why pre-formulas are "protected" by (square or angle) brackets in TLA*: the only place a pre-formula P can be used is inside an action: $\Box[P]$-v. To show that $\Box[P]$-v is stuttering invariant, is must be shown that a slightly weaker predicate holds for P. For example, if P contains a term of the form $\bigcirc\bigcirc Q$, then it is not a well-formed pre-formula, thus $\Box[P]$-v is not stuttering invariant. This weaker version of stuttering invariance has been named *near stuttering invariance*.

definition *nstutinv* :: $('a,'b)$ *formfun* $\Rightarrow$ *bool*
where *nstutinv* $P \equiv \forall\ \sigma\ \tau.\ (\mathit{first}\ \sigma = \mathit{first}\ \tau) \wedge (\mathit{tail}\ \sigma) \approx (\mathit{tail}\ \tau) \longrightarrow (\sigma \models P) = (\tau \models P)$

syntax
  -stutinv :: *lift* $\Rightarrow$ *bool* (‹(*STUTINV* -)› [*40*] *40*)
  -nstutinv :: *lift* $\Rightarrow$ *bool* (‹(*NSTUTINV* -)› [*40*] *40*)

translations
  -stutinv $\rightleftharpoons$ *CONST stutinv*
  -nstutinv $\rightleftharpoons$ *CONST nstutinv*

Predicate *STUTINV F* formalises stuttering invariance for formula F. That is if two sequences are similar $s \approx t$ (equal up to stuttering) then the validity of F under both s and t are equivalent. Predicate *NSTUTINV P* should be read as *nearly stuttering invariant* – and is required for some stuttering invariance proofs.

lemma *stutinv-strictly-stronger*:
  assumes *h*: *STUTINV F* **shows** *NSTUTINV F*
  unfolding *nstutinv-def*
proof (*clarify*)
  fix $s\ t$ :: *nat* $\Rightarrow$ *'a*
  assume *a1*: *first* s = *first* t **and** *a2*: (*tail* s) $\approx$ (*tail* t)
  have $s \approx t$
  proof –
    have *tg1*: (*first* s) ## (*tail* s) = s **by** (*rule seq-app-first-tail*)
    have *tg2*: (*first* t) ## (*tail* t) = t **by** (*rule seq-app-first-tail*)
    with *a1* **have** *tg2'*: (*first* s) ## (*tail* t) = t **by** *simp*
   from *a2* **have** (*first* s) ## (*tail* s) $\approx$ (*first* s) ## (*tail* t) **by** (*rule app-seqsimilar*)
    with *tg1 tg2'* **show** *?thesis* **by** *simp*
  qed
  with *h* **show** $(s \models F) = (t \models F)$ **by** (*simp add*: *stutinv-def*)
qed

3.5.1 Properties of *-stutinv*

This subsection proves stuttering invariance, preservation of stuttering invariance and introduction of stuttering invariance for different formulas. First, state predicates are stuttering invariant.

theorem *stut-before*: *STUTINV* $\$F$
proof (*clarsimp simp*: *stutinv-def*)
  fix *s t* :: *'a seq*
  assume *a1*: $s \approx t$
  hence (*first s*) = (*first t*) **by** (*rule sim-first*)
  thus $(s \models \$F) = (t \models \$F)$ **by** (*simp add*: *before-def*)
qed

lemma *nstut-after*: *NSTUTINV* $F\$$
proof (*clarsimp simp*: *nstutinv-def*)
  fix *s t* :: *'a seq*
  assume *a1*: *tail s* $\approx$ *tail t*
  thus $(s \models F\$) = (t \models F\$)$ **by** (*simp add*: *after-def tail-sim-second*)
qed

The always operator preserves stuttering invariance.

theorem *stut-always*: **assumes** *H*:*STUTINV F* **shows** *STUTINV* $\Box F$
proof (*clarsimp simp*: *stutinv-def*)
  fix *s t* :: *'a seq*
  assume *a2*: $s \approx t$
  show $(s \models (\Box\ F)) = (t \models (\Box\ F))$
  proof
    assume *a1*: $t \models \Box\ F$
    show $s \models \Box\ F$
    proof (*clarsimp simp*: *always-def*)
      fix *n*
      from *a2*[*THEN sim-step*] **obtain** *m* **where** *m*: $s\ |_s\ n \approx t\ |_s\ m$ **by** *blast*
      from *a1* **have** $(t\ |_s\ m) \models F$ **by** (*simp add*: *always-def*)
      with *H m* **show** $(s\ |_s\ n) \models F$ **by** (*simp add*: *stutinv-def*)
    qed
  next
    assume *a1*: $s \models (\Box\ F)$
    show $t \models (\Box\ F)$
    proof (*clarsimp simp*: *always-def*)
      fix *n*
      from *a2*[*THEN seqsim-sym*, *THEN sim-step*] **obtain** *m* **where** *m*: $t\ |_s\ n \approx s\ |_s\ m$ **by** *blast*
      from *a1* **have** $(s\ |_s\ m) \models F$ **by** (*simp add*: *always-def*)
      with *H m* **show** $(t\ |_s\ n) \models F$ **by** (*simp add*: *stutinv-def*)
    qed
  qed
qed

Assuming that formula P is nearly suttering invariant then $\Box[P]\text{-}v$ will be stuttering invariant.

lemma *stut-action-lemma*:
  assumes *H*: *NSTUTINV P* **and** *st*: $s \approx t$ **and** *P*: $t \models \Box[P]\text{-}v$
  shows $s \models \Box[P]\text{-}v$
proof (*clarsimp simp*: *action-def*)

fix n
assume $\neg\ ((s\ |_s\ n) \models \mathit{Unchanged}\ v)$
hence v: $v\ (s\ (\mathit{Suc}\ n)) \neq v\ (s\ n)$
  by (*simp add*: *unch-defs first-def second-def suffix-def*)
from *st*[*THEN sim-step*] **obtain** m **where**
  a2′: $s\ |_s\ n \approx t\ |_s\ m$
    $\wedge\ (s\ |_s\ \mathit{Suc}\ n \approx t\ |_s\ \mathit{Suc}\ m \vee s\ |_s\ \mathit{Suc}\ n \approx t\ |_s\ m)$ **..**
hence *g1*: $(s\ |_s\ n \approx t\ |_s\ m)$ **by** *simp*
hence *g1″*: $\mathit{first}\ (s\ |_s\ n) = \mathit{first}\ (t\ |_s\ m)$ **by** (*simp add*: *sim-first*)
hence *g1′*: $s\ n = t\ m$ **by** (*simp add*: *suffix-def first-def*)
from *a2′* **have** *g2*: $s\ |_s\ \mathit{Suc}\ n \approx t\ |_s\ \mathit{Suc}\ m \vee s\ |_s\ \mathit{Suc}\ n \approx t\ |_s\ m$ **by** *simp*
from P **have** *a1′*: $((t\ |_s\ m) \models P) \vee ((t\ |_s\ m) \models \mathit{Unchanged}\ v)$ **by** (*simp add*: *action-def*)
from *g2* **show** $(s\ |_s\ n) \models P$
proof
  assume $s\ |_s\ \mathit{Suc}\ n \approx t\ |_s\ m$
  hence $\mathit{first}\ (s\ |_s\ \mathit{Suc}\ n) = \mathit{first}\ (t\ |_s\ m)$ **by** (*simp add*: *sim-first*)
  hence $s\ (\mathit{Suc}\ n) = t\ m$ **by** (*simp add*: *suffix-def first-def*)
  with *g1′* v **show** *?thesis* **by** *simp* — by contradiction
next
  assume *a3*: $s\ |_s\ \mathit{Suc}\ n \approx t\ |_s\ \mathit{Suc}\ m$
  hence $\mathit{first}\ (s\ |_s\ \mathit{Suc}\ n) = \mathit{first}\ (t\ |_s\ \mathit{Suc}\ m)$ **by** (*simp add*: *sim-first*)
  hence *a3′*: $s\ (\mathit{Suc}\ n) = t\ (\mathit{Suc}\ m)$ **by** (*simp add*: *suffix-def first-def*)
  from *a1′* **show** *?thesis*
  proof
    assume $(t\ |_s\ m) \models \mathit{Unchanged}\ v$
    hence $v\ (t\ (\mathit{Suc}\ m)) = v\ (t\ m)$
      by (*simp add*: *unch-defs first-def second-def suffix-def*)
    with *g1′ a3′* v **show** *?thesis* **by** *simp* — again, by contradiction
  next
    assume *a4*: $(t\ |_s\ m) \models P$
    from *a3* **have** $\mathit{tail}\ (s\ |_s\ n) \approx \mathit{tail}\ (t\ |_s\ m)$ **by** (*simp add*: *tail-def suffix-plus*)
    with H *g1″ a4* **show** *?thesis* **by** (*auto simp*: *nstutinv-def*)
  qed
qed
qed

theorem *stut-action*: **assumes** H: $\mathit{NSTUTINV}\ P$ **shows** $\mathit{STUTINV}\ \Box[P]\text{-}v$
proof (*clarsimp simp*: *stutinv-def*)
fix $s\ t :: {'a}\ \mathit{seq}$
assume *st*: $s \approx t$
show $(s \models \Box[P]\text{-}v) = (t \models \Box[P]\text{-}v)$
proof
  assume $t \models \Box[P]\text{-}v$
  with H *st* **show** $s \models \Box[P]\text{-}v$ **by** (*rule stut-action-lemma*)
next
  assume $s \models \Box[P]\text{-}v$
  with H *st*[*THEN seqsim-sym*] **show** $t \models \Box[P]\text{-}v$ **by** (*rule stut-action-lemma*)
qed

qed

The lemmas below shows that propositional and predicate operators preserve stuttering invariance.

lemma *stut-and*: $[\![STUTINV\ F;STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \wedge G)$
  by (*simp add*: *stutinv-def*)

lemma *stut-or*: $[\![STUTINV\ F;STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \vee G)$
  by (*simp add*: *stutinv-def*)

lemma *stut-imp*: $[\![STUTINV\ F;STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \longrightarrow G)$
  by (*simp add*: *stutinv-def*)

lemma *stut-eq*: $[\![STUTINV\ F;STUTINV\ G]\!] \Longrightarrow STUTINV\ (F = G)$
  by (*simp add*: *stutinv-def*)

lemma *stut-noteq*: $[\![STUTINV\ F;STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \neq G)$
  by (*simp add*: *stutinv-def*)

lemma *stut-not*: $STUTINV\ F \Longrightarrow STUTINV\ (\neg\ F)$
  by (*simp add*: *stutinv-def*)

lemma *stut-all*: $(\bigwedge x.\ STUTINV\ (F\ x)) \Longrightarrow STUTINV\ (\forall\ x.\ F\ x)$
  by (*simp add*: *stutinv-def*)

lemma *stut-ex*: $(\bigwedge x.\ STUTINV\ (F\ x)) \Longrightarrow STUTINV\ (\exists\ x.\ F\ x)$
  by (*simp add*: *stutinv-def*)

lemma *stut-const*: $STUTINV\ \#c$
  by (*simp add*: *stutinv-def*)

lemma *stut-fun1*: $STUTINV\ X \Longrightarrow STUTINV\ (f\ <X>)$
  by (*simp add*: *stutinv-def*)

lemma *stut-fun2*: $[\![STUTINV\ X;STUTINV\ Y]\!] \Longrightarrow STUTINV\ (f\ <X,Y>)$
  by (*simp add*: *stutinv-def*)

lemma *stut-fun3*: $[\![STUTINV\ X;STUTINV\ Y;STUTINV\ Z]\!] \Longrightarrow STUTINV\ (f\ <X,Y,Z>)$
  by (*simp add*: *stutinv-def*)

lemma *stut-fun4*: $[\![STUTINV\ X;STUTINV\ Y;STUTINV\ Z;\ STUTINV\ W]\!] \Longrightarrow STUTINV\ (f\ <X,Y,Z,W>)$
  by (*simp add*: *stutinv-def*)

lemma *stut-plus*: $[\![STUTINV\ x;STUTINV\ y]\!] \Longrightarrow STUTINV\ (x+y)$
  by (*simp add*: *stutinv-def*)

3.5.2 Properties of *-nstutinv*

This subsection shows analogous properties about near stuttering invariance. If a formula F is stuttering invariant then $\bigcirc F$ is nearly stuttering invariant.

lemma *nstut-nexts*: **assumes** *H*: *STUTINV F* **shows** *NSTUTINV* $\bigcirc$*F*
using *H* **by** (*simp add*: *stutinv-def nstutinv-def nexts-def*)

The lemmas below shows that propositional and predicate operators preserves near stuttering invariance.

lemma *nstut-and*: $[\![$*NSTUTINV F*;*NSTUTINV G*$]\!] \Longrightarrow$ *NSTUTINV* $(F \wedge G)$
by (*auto simp*: *nstutinv-def*)

lemma *nstut-or*: $[\![$*NSTUTINV F*;*NSTUTINV G*$]\!] \Longrightarrow$ *NSTUTINV* $(F \vee G)$
by (*auto simp*: *nstutinv-def*)

lemma *nstut-imp*: $[\![$*NSTUTINV F*;*NSTUTINV G*$]\!] \Longrightarrow$ *NSTUTINV* $(F \longrightarrow G)$
by (*auto simp*: *nstutinv-def*)

lemma *nstut-eq*: $[\![$*NSTUTINV F*; *NSTUTINV G*$]\!] \Longrightarrow$ *NSTUTINV* $(F = G)$
by (*force simp*: *nstutinv-def*)

lemma *nstut-not*: *NSTUTINV F* $\Longrightarrow$ *NSTUTINV* $(\neg\ F)$
by (*auto simp*: *nstutinv-def*)

lemma *nstut-noteq*: $[\![$*NSTUTINV F*; *NSTUTINV G*$]\!] \Longrightarrow$ *NSTUTINV* $(F \neq G)$
by (*simp add*: *nstut-eq nstut-not*)

lemma *nstut-all*: $(\bigwedge x.$ *NSTUTINV* $(F\ x)) \Longrightarrow$ *NSTUTINV* $(\forall\ x.\ F\ x)$
by (*auto simp*: *nstutinv-def*)

lemma *nstut-ex*: $(\bigwedge x.$ *NSTUTINV* $(F\ x)) \Longrightarrow$ *NSTUTINV* $(\exists\ x.\ F\ x)$
by (*auto simp*: *nstutinv-def*)

lemma *nstut-const*: *NSTUTINV* $\#c$
by (*auto simp*: *nstutinv-def*)

lemma *nstut-fun1*: *NSTUTINV X* $\Longrightarrow$ *NSTUTINV* $(f\ {<}X{>})$
by (*force simp*: *nstutinv-def*)

lemma *nstut-fun2*: $[\![$*NSTUTINV X*; *NSTUTINV Y*$]\!] \Longrightarrow$ *NSTUTINV* $(f\ {<}X,Y{>})$
by (*force simp*: *nstutinv-def*)

lemma *nstut-fun3*: $[\![$*NSTUTINV X*; *NSTUTINV Y*; *NSTUTINV Z*$]\!] \Longrightarrow$ *NSTUTINV* $(f\ {<}X,Y,Z{>})$
by (*force simp*: *nstutinv-def*)

lemma *nstut-fun4*: $[\![$*NSTUTINV X*; *NSTUTINV Y*; *NSTUTINV Z*; *NSTUTINV W*$]\!] \Longrightarrow$ *NSTUTINV* $(f\ {<}X,Y,Z,W{>})$
by (*force simp*: *nstutinv-def*)

lemma *nstut-plus*: $[\![NSTUTINV\ x;NSTUTINV\ y]\!] \Longrightarrow NSTUTINV\ (x{+}y)$
by (*simp add*: *nstut-fun2*)

3.5.3 Abbreviations

We show the obvious fact that the same properties holds for abbreviated operators.

lemmas *nstut-before* = *stut-before*[*THEN stutinv-strictly-stronger*]

lemma *nstut-unch*: *NSTUTINV* (*Unchanged v*)
proof (*unfold unch-def*)
have *g1*: *NSTUTINV v$* **by** (*rule nstut-after*)
have *NSTUTINV $v* **by** (*rule stut-before*[*THEN stutinv-strictly-stronger*])
with *g1* **show** *NSTUTINV* (*v$* = *$v*) **by** (*rule nstut-eq*)
qed

Formulas $[P]$-v are not TLA* formulas by themselves, but we need to reason about them when they appear wrapped inside $\Box[-]$-v. We only require that it preserves nearly stuttering invariance. Observe that $[P]$-v trivially holds for a stuttering step, so it cannot be stuttering invariant.

lemma *nstut-actrans*: $NSTUTINV\ P \Longrightarrow NSTUTINV\ [P]\text{-}v$
by (*simp add*: *actrans-def nstut-unch nstut-or*)

lemma *stut-eventually*: $STUTINV\ F \Longrightarrow STUTINV\ \Diamond F$
by (*simp add*: *eventually-def stut-not stut-always*)

lemma *stut-leadsto*: $[\![STUTINV\ F;\ STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \leadsto G)$
by (*simp add*: *leadsto-def stut-always stut-eventually stut-imp*)

lemma *stut-angle-action*: $NSTUTINV\ P \Longrightarrow STUTINV\ \Diamond\langle P\rangle\text{-}v$
by (*simp add*: *angle-action-def nstut-not stut-action stut-not*)

lemma *nstut-angle-acttrans*: $NSTUTINV\ P \Longrightarrow NSTUTINV\ \langle P\rangle\text{-}v$
by (*simp add*: *angle-actrans-def nstut-not nstut-actrans*)

lemmas *stutinvs* = *stut-before stut-always stut-action*
stut-and stut-or stut-imp stut-eq stut-noteq stut-not
stut-all stut-ex stut-eventually stut-leadsto stut-angle-action stut-const
stut-fun1 stut-fun2 stut-fun3 stut-fun4

lemmas *nstutinvs* = *nstut-after nstut-nexts nstut-actrans*
nstut-unch nstut-and nstut-or nstut-imp nstut-eq nstut-noteq nstut-not
nstut-all nstut-ex nstut-angle-acttrans stutinv-strictly-stronger
nstut-fun1 nstut-fun2 nstut-fun3 nstut-fun4 stutinvs[*THEN stutinv-strictly-stronger*]

lemmas *bothstutinvs* = *stutinvs nstutinvs*

end

4 Reasoning about PreFormulas

theory *PreFormulas*
imports *Semantics*
begin

Semantic separation of formulas and pre-formulas requires a deep embedding. We introduce a syntactically distinct notion of validity, written $|^\sim A$, for pre-formulas. Although it is semantically identical to $\vdash A$, it helps users distinguish pre-formulas from formulas in TLA* proofs.

definition *PreValid* :: *('w::world) form* ⇒ *bool*
where *PreValid* $A \equiv \forall\ w.\ w \models A$

syntax
  -PreValid :: *lift* ⇒ *bool* (‹(|~ -)› *5*)

translations
  -PreValid ⇌ *CONST PreValid*

lemma *prefD*[*dest*]: $|^\sim A \Longrightarrow w \models A$
  by (*simp add*: *PreValid-def*)

lemma *prefI*[*intro*!]: $(\bigwedge w.\ w \models A) \Longrightarrow |^\sim A$
  by (*simp add*: *PreValid-def*)

method-setup *pref-unlift* = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD′
    (resolve-tac ctxt @{thms prefI} THEN′ rewrite-goal-tac ctxt @{thms intensional-rews}))
› *int-unlift for PreFormulas*

lemma *prefeq-reflection*: **assumes** *P1*: $|^\sim x{=}y$ **shows** $(x \equiv y)$
using *P1* **by** (*intro eq-reflection*) *force*

lemma *pref-True*[*simp*]: $|^\sim$ *#True*
  by *auto*

lemma *pref-eq*: $|^\sim X = Y \Longrightarrow X = Y$
  by (*auto simp*: *prefeq-reflection*)

lemma *pref-iffI*:
  assumes $|^\sim F \longrightarrow G$ **and** $|^\sim G \longrightarrow F$
  shows $|^\sim F = G$
  using *assms* **by** *force*

lemma *pref-iffD1*: **assumes** $|^\sim F = G$ **shows** $|^\sim F \longrightarrow G$

using *assms* **by** *auto*

lemma *pref-iffD2*: **assumes** $|^\sim F = G$ **shows** $|^\sim G \longrightarrow F$
using *assms* **by** *auto*

lemma *unl-pref-imp*:
assumes $|^\sim F \longrightarrow G$ **shows** $\bigwedge w.\ w \models F \Longrightarrow w \models G$
using *assms* **by** *auto*

lemma *pref-imp-trans*:
assumes $|^\sim F \longrightarrow G$ **and** $|^\sim G \longrightarrow H$
shows $|^\sim F \longrightarrow H$
using *assms* **by** *force*

4.1 Lemmas about *Unchanged*

Many of the TLA* axioms only require a state function witness which leaves the state space unchanged. An obvious witness is the *id* function. The lemmas require that the given formula is invariant under stuttering.

lemma *pre-id-unch*: **assumes** *h*: *stutinv F*
shows $|^\sim F \wedge$ *Unchanged id* $\longrightarrow \bigcirc F$
proof (*pref-unlift*, *clarify*)
fix *s*
assume *a1*: $s \models F$ **and** *a2*: $s \models$ *Unchanged id*
from *a2* **have** (*id* (*second s*) = *id* (*first s*)) **by** (*simp add*: *unch-defs*)
hence $s \approx$ (*tail s*) **by** (*simp add*: *addfirststut*)
with *h a1* **have** (*tail s*) $\models F$ **by** (*simp add*: *stutinv-def*)
thus $s \models \bigcirc F$ **by** (*unfold nexts-def*)
qed

lemma *pre-ex-unch*:
assumes *h*: *stutinv F*
shows $\exists (v{::}'a{::}world \Rightarrow 'a).\ (\ |^\sim F \wedge$ *Unchanged v* $\longrightarrow \bigcirc F)$
using *pre-id-unch*[*OF h*] **by** *blast*

lemma *unch-pair*: $|^\sim$ *Unchanged* (*x*,*y*) = (*Unchanged x* $\wedge$ *Unchanged y*)
by (*auto simp*: *unch-def before-def after-def nexts-def*)

lemmas *unch-eq1* = *unch-pair*[*THEN pref-eq*]
lemmas *unch-eq2* = *unch-pair*[*THEN prefeq-reflection*]

lemma *angle-actrans-sem*: $|^\sim \langle F\rangle\text{-}v = (F \wedge v\$ \neq \$v)$
by (*auto simp*: *angle-actrans-def actrans-def unch-def*)

lemmas *angle-actrans-sem-eq* = *angle-actrans-sem*[*THEN pref-eq*]

4.2 Lemmas about *after*

lemma *after-const*: $|^\sim (\#c)` = \#c$

by (*auto simp*: *nexts-def before-def after-def*)

lemma *after-fun1*: $|^{\sim}\ f{<}x{>}` = f{<}x`{>}$
by (*auto simp*: *nexts-def before-def after-def*)

lemma *after-fun2*: $|^{\sim}\ f{<}x,y{>}` = f\ {<}x`,y`{>}$
by (*auto simp*: *nexts-def before-def after-def*)

lemma *after-fun3*: $|^{\sim}\ f{<}x,y,z{>}` = f\ {<}x`,y`,z`{>}$
by (*auto simp*: *nexts-def before-def after-def*)

lemma *after-fun4*: $|^{\sim}\ f{<}x,y,z,zz{>}` = f\ {<}x`,y`,z`,zz`{>}$
by (*auto simp*: *nexts-def before-def after-def*)

lemma *after-forall*: $|^{\sim}\ (\forall\ x.\ P\ x)` = (\forall\ x.\ (P\ x)`)$
by (*auto simp*: *nexts-def before-def after-def*)

lemma *after-exists*: $|^{\sim}\ (\exists\ x.\ P\ x)` = (\exists\ x.\ (P\ x)`)$
by (*auto simp*: *nexts-def before-def after-def*)

lemma *after-exists1*: $|^{\sim}\ (\exists!\ x.\ P\ x)` = (\exists!\ x.\ (P\ x)`)$
by (*auto simp*: *nexts-def before-def after-def*)

lemmas *all-after* = *after-const after-fun1 after-fun2 after-fun3 after-fun4 after-forall after-exists after-exists1*

lemmas *all-after-unl* = *all-after*[*THEN prefD*]
lemmas *all-after-eq* = *all-after*[*THEN prefeq-reflection*]

4.3 Lemmas about *before*

lemma *before-const*: $\vdash \$(\#c) = \#c$
by (*auto simp*: *before-def*)

lemma *before-fun1*: $\vdash \$(f{<}x{>}) = f\ {<}\$x{>}$
by (*auto simp*: *before-def*)

lemma *before-fun2*: $\vdash \$(f{<}x,y{>}) = f\ {<}\$x,\$y{>}$
by (*auto simp*: *before-def*)

lemma *before-fun3*: $\vdash \$(f{<}x,y,z{>}) = f\ {<}\$x,\$y,\$z{>}$
by (*auto simp*: *before-def*)

lemma *before-fun4*: $\vdash \$(f{<}x,y,z,zz{>}) = f\ {<}\$x,\$y,\$z,\$zz{>}$
by (*auto simp*: *before-def*)

lemma *before-forall*: $\vdash \$(\forall\ x.\ P\ x) = (\forall\ x.\ \$(P\ x))$
by (*auto simp*: *before-def*)

lemma *before-exists*: $\vdash \$(\exists\ x.\ P\ x) = (\exists\ x.\ \$(P\ x))$
by (*auto simp*: *before-def*)

lemma *before-exists1*: $\vdash \$(\exists!\ x.\ P\ x) = (\exists!\ x.\ \$(P\ x))$
by (*auto simp*: *before-def*)

lemmas *all-before* = *before-const before-fun1 before-fun2 before-fun3 before-fun4 before-forall before-exists before-exists1*

lemmas *all-before-unl* = *all-before*[*THEN intD*]
lemmas *all-before-eq* = *all-before*[*THEN inteq-reflection*]

4.4 Some general properties

lemma *angle-actrans-conj*: $|^\sim\ (\langle F \wedge G\rangle\text{-}v) = (\langle F\rangle\text{-}v \wedge \langle G\rangle\text{-}v)$
by (*auto simp*: *angle-actrans-def actrans-def unch-def*)

lemma *angle-actrans-disj*: $|^\sim\ (\langle F \vee G\rangle\text{-}v) = (\langle F\rangle\text{-}v \vee \langle G\rangle\text{-}v)$
by (*auto simp*: *angle-actrans-def actrans-def unch-def*)

lemma *int-eq-true*: $\vdash P \Longrightarrow \vdash P = \#True$
by *auto*

lemma *pref-eq-true*: $|^\sim\ P \Longrightarrow |^\sim\ P = \#True$
by *auto*

4.5 Unlifting attributes and methods

Attribute which unlifts an intensional formula or preformula

ML ‹
```
fun unl-rewr ctxt thm =
    let
      val unl = (thm RS @{thm intD}) handle THM - => (thm RS @{thm prefD})
                                     handle THM - => thm
       val rewr = rewrite-rule ctxt @{thms intensional-rews}
     in
       unl |> rewr
     end;
```
›
attribute-setup *unlifted* = ‹
Scan.succeed (*Thm.rule-attribute* [] (*unl-rewr o Context.proof-of*))
› *unlift intensional formulas*

attribute-setup *unlift-rule* = ‹
Scan.succeed
(*Thm.rule-attribute* []
(*Context.proof-of* #> (*fn ctxt* => *Object-Logic.rulify ctxt o unl-rewr ctxt*)))
› *unlift and rulify intensional formulas*

Attribute which turns an intensional formula or preformula into a rewrite rule. Formulas F that are not equalities are turned into $F \equiv \#True$.

```
ML ‹
fun int-rewr thm =
  (thm RS @{thm inteq-reflection})
    handle THM - => (thm RS @{thm prefeq-reflection})
    handle THM - => ((thm RS @{thm int-eq-true}) RS @{thm inteq-reflection})
   handle THM - => ((thm RS @{thm pref-eq-true}) RS @{thm prefeq-reflection});
›

attribute-setup simp-unl = ‹
    Attrib.add-del
      (Thm.declaration-attribute
        (fn th => Simplifier.map-ss (Simplifier.add-simp (int-rewr th))))
      (K (NONE, NONE))  (* note only adding -- removing is ignored *)
› add thm unlifted from rewrites from intensional formulas or preformulas

attribute-setup int-rewrite = ‹Scan.succeed (Thm.rule-attribute [] (fn - => int-rewr))›
  produce rewrites from intensional formulas or preformulas

end
```

5 A Proof System for TLA*

theory *Rules*
imports *PreFormulas*
begin

We prove soundness of the proof system of TLA*, from which the system verification rules from Lamport's original TLA paper will be derived. This theory is still state-independent, thus state-dependent enableness proofs, required for proofs based on fairness assumptions, and flexible quantification, are not discussed here.

The TLA* paper [8] suggest both a *hetereogeneous* and a *homogenous* proof system for TLA*. The homogeneous version eliminates the auxiliary definitions from the *Preformula* theory, creating a single provability relation. This axiomatisation is based on the fact that a pre-formula can only be used via the *sq* rule. In a nutshell, *sq* is applied to *pax1* to *pax5*, and *nex*, *pre* and *pmp* are changed to accommodate this. It is argued that while the hetereogenous version is easier to understand, the homogenous system avoids the introduction of an auxiliary provability relation. However, the price to pay is that reasoning about pre-formulas (in particular, actions) has to be performed in the scope of temporal operators such as $\Box[P]$-v, which is notationally quite heavy, We prefer here the heterogeneous approach, which exposes the pre-formulas and lets us use standard HOL rules more directly.

5.1 The Basic Axioms

```
theorem fmp: assumes ⊢ F and ⊢ F ⟶ G shows ⊢ G
  using assms[unlifted] by auto

theorem pmp: assumes |~ F and |~ F ⟶ G shows |~ G
  using assms[unlifted] by auto

theorem sq: assumes |~ P shows ⊢ □[P]-v
  using assms[unlifted] by (auto simp: action-def)

theorem pre: assumes ⊢ F shows |~ F
  using assms by auto

theorem nex: assumes h1: ⊢ F shows |~ ○F
  using assms by (auto simp: nexts-def)

theorem ax0: ⊢ # True
  by auto

theorem ax1: ⊢ □F ⟶ F
proof (clarsimp simp: always-def)
  fix w
  assume ∀ n. (w |s n) ⊨ F
  hence (w |s 0) ⊨ F ..
  thus w ⊨ F by simp
qed

theorem ax2: ⊢ □F ⟶ □[□F]-v
  by (auto simp: always-def action-def suffix-plus)

theorem ax3:
  assumes H: |~ F ∧ Unchanged v ⟶ ○F
  shows ⊢ □[F ⟶ ○F]-v ⟶ (F ⟶ □F)
proof (clarsimp simp: always-def)
  fix w n
  assume a1: w ⊨ □[F ⟶ ○F]-v and  a2: w ⊨ F
  show (w |s n) ⊨ F
  proof (induct n)
    from a2 show (w |s 0) ⊨ F by simp
  next
    fix m
    assume a3: (w |s m) ⊨ F
    with a1 H[unlifted] show (w |s (Suc m)) ⊨ F
      by (auto simp: nexts-def action-def tail-suffix-suc)
  qed
qed

theorem ax4: ⊢ □[P ⟶ Q]-v ⟶ (□[P]-v ⟶ □[Q]-v)
  by (force simp: action-def)
```

theorem *ax5*: $\vdash \Box[v` \neq \$v]$-$v$
  by (*auto simp*: *action-def unch-def*)

theorem *pax0*: $|^{\sim}$ # *True*
  by *auto*

theorem *pax1* [*simp-unl*]: $|^{\sim}$ $(\bigcirc\neg F) = (\neg\bigcirc F)$
  by (*auto simp*: *nexts-def*)

theorem *pax2*: $|^{\sim}$ $\bigcirc(F \longrightarrow G) \longrightarrow (\bigcirc F \longrightarrow \bigcirc G)$
  by (*auto simp*: *nexts-def*)

theorem *pax3*: $|^{\sim}$ $\Box F \longrightarrow \bigcirc\Box F$
  by (*auto simp*: *always-def nexts-def tail-def suffix-plus*)

theorem *pax4*: $|^{\sim}$ $\Box[P]$-v = $([P]$-$v \wedge \bigcirc\Box[P]$-$v)$
proof (*auto*)
  fix w
  assume $w \models \Box[P]$-v
  from *this*[*unfolded action-def*] **have** $((w\ |_s\ 0) \models P) \vee ((w\ |_s\ 0) \models$ *Unchanged* $v)$ **..**
  thus $w \models [P]$-v **by** (*simp add*: *actrans-def*)
next
  fix w
  assume $w \models \Box[P]$-v
  thus $w \models \bigcirc\Box[P]$-$v$ **by** (*auto simp*: *nexts-def action-def tail-def suffix-plus*)
next
  fix w
  assume *1*: $w \models [P]$-v **and** *2*: $w \models \bigcirc\Box[P]$-$v$
  show $w \models \Box[P]$-v
  proof (*auto simp*: *action-def*)
    fix i
    assume *3*: $\neg\ ((w\ |_s\ i) \models$ *Unchanged* $v)$
    show $(w\ |_s\ i) \models P$
    proof (*cases* i)
      assume $i = 0$
      with *1 3* **show** *?thesis* **by** (*simp add*: *actrans-def*)
    next
      fix j
      assume $i =$ *Suc* j
      with *2 3* **show** *?thesis* **by** (*auto simp*: *nexts-def action-def tail-def suffix-plus*)
    qed
  qed
qed

theorem *pax5*: $|^{\sim}$ $\bigcirc\Box F \longrightarrow \Box[\bigcirc F]$-$v$
  by (*auto simp*: *nexts-def always-def action-def tail-def suffix-plus*)

Theorem to show that universal quantification distributes over the always operator. Since the TLA* paper only addresses the propositional fragment, this theorem does not appear there.

theorem *allT*: $\vdash (\forall\, x.\ \Box(F\ x)) = (\Box(\forall\, x.\ F\ x))$
  by (*auto simp*: *always-def*)

theorem *allActT*: $\vdash (\forall\, x.\ \Box[F\ x]\text{-}v) = (\Box[(\forall\, x.\ F\ x)]\text{-}v)$
  by (*force simp*: *action-def*)

5.2 Derived Theorems

This section includes some derived theorems based on the axioms, taken from the TLA* paper [8]. We mimic the proofs given there and avoid semantic reasoning whenever possible.

The *alw* theorem of [8] states that if F holds in all worlds then it always holds, i.e. $F \models \Box F$. However, the derivation of this theorem (using the proof rules above) relies on access of the set of free variables (FV), which is not available in a shallow encoding.

However, we can prove a similar rule *alw2* using an additional hypothesis $|^{\sim}\ F \wedge \mathit{Unchanged}\ v \longrightarrow \bigcirc F$.

theorem *alw2*:
  assumes *h1*: $\vdash F$ **and** *h2*: $|^{\sim}\ F \wedge \mathit{Unchanged}\ v \longrightarrow \bigcirc F$
  shows $\vdash \Box F$
proof −
  from *h1* **have** *g2*: $|^{\sim}\ \bigcirc F$ **by** (*rule nex*)
  hence *g3*: $|^{\sim}\ F \longrightarrow \bigcirc F$ **by** *auto*
  hence *g4*:$\vdash \Box[(F \longrightarrow \bigcirc F)]\text{-}v$ **by** (*rule sq*)
  from *h2* **have** $\vdash \Box[(F \longrightarrow \bigcirc F)]\text{-}v \longrightarrow F \longrightarrow \Box F$ **by** (*rule ax3*)
  with *g4*[*unlifted*] **have** *g5*: $\vdash F \longrightarrow \Box F$ **by** *auto*
  with *h1*[*unlifted*] **show** *?thesis* **by** *auto*
qed

Similar theorem, assuming that F is stuttering invariant.

theorem *alw3*:
  assumes *h1*: $\vdash F$ **and** *h2*: *stutinv F*
  shows $\vdash \Box F$
proof −
  from *h2* **have** $|^{\sim}\ F \wedge \mathit{Unchanged}\ id \longrightarrow \bigcirc F$ **by** (*rule pre-id-unch*)
  with *h1* **show** *?thesis* **by** (*rule alw2*)
qed

In a deep embedding, we could prove that all (proper) TLA* formulas are stuttering invariant and then get rid of the second hypothesis of rule *alw3*. In fact, the rule is even true for pre-formulas, as shown by the following rule, whose proof relies on semantical reasoning.

theorem *alw*: **assumes** *H1*: $\vdash F$ **shows** $\vdash \Box F$
  using *H1* **by** (*auto simp*: *always-def*)

theorem *alw-valid-iff-valid*: $(\vdash \Box F) = (\vdash F)$
proof
  assume $\vdash \Box F$
  from *this ax1* **show** $\vdash F$ **by** (*rule fmp*)
qed (*rule alw*)

[8] proves the following theorem using the deduction theorem of TLA*: $(\vdash F \Longrightarrow \vdash G) \Longrightarrow \vdash []F \longrightarrow G$, which can only be proved by induction on the formula structure, in a deep embedding.

theorem *T1*[*simp-unl*]: $\vdash \Box\Box F = []F$
proof (*auto simp*: *always-def suffix-plus*)
  fix *w n*
  assume $\forall\, m\ k.\ (w\ |_s\ (k+m)) \models F$
  hence $(w\ |_s\ (n+0)) \models F$ **by** *blast*
  thus $(w\ |_s\ n) \models F$ **by** *simp*
qed

theorem *T2*[*simp-unl*]: $\vdash \Box\Box[P]\text{-}v = \Box[P]\text{-}v$
proof −
  have *1*: $|^\sim\ \Box[P]\text{-}v \longrightarrow \bigcirc\Box[P]\text{-}v$ **using** *pax4* **by** *force*
  hence $\vdash \Box[\Box[P]\text{-}v \longrightarrow \bigcirc\Box[P]\text{-}v]\text{-}v$ **by** (*rule sq*)
  moreover
  have $\vdash \Box[\ \Box[P]\text{-}v \longrightarrow \bigcirc\Box[P]\text{-}v\ ]\text{-}v \longrightarrow \Box[P]\text{-}v \longrightarrow \Box\Box[P]\text{-}v$
    by (*rule ax3*) (*auto elim*: *1*[*unlift-rule*])
  moreover
  have $\vdash \Box\Box[P]\text{-}v \longrightarrow \Box[P]\text{-}v$ **by** (*rule ax1*)
  ultimately show *?thesis* **by** *force*
qed

theorem *T3*[*simp-unl*]: $\vdash \Box[[P]\text{-}v]\text{-}v = \Box[P]\text{-}v$
proof −
  have $|^\sim\ P \longrightarrow [P]\text{-}v$ **by** (*auto simp*: *actrans-def*)
  hence $\vdash \Box[(P \longrightarrow [P]\text{-}v)]\text{-}v$ **by** (*rule sq*)
  with *ax4* **have** $\vdash \Box[P]\text{-}v \longrightarrow \Box[[P]\text{-}v]\text{-}v$ **by** *force*
  moreover
  have $|^\sim\ [P]\text{-}v \longrightarrow v\text{`}\neq \$v \longrightarrow P$ **by** (*auto simp*: *unch-def actrans-def*)
  hence $\vdash \Box[[P]\text{-}v \longrightarrow v\text{`}\neq \$v \longrightarrow P]\text{-}v$ **by** (*rule sq*)
  with *ax5* **have** $\vdash \Box[[P]\text{-}v]\text{-}v \longrightarrow \Box[P]\text{-}v$ **by** (*force intro*: *ax4*[*unlift-rule*])
  ultimately show *?thesis* **by** *force*
qed

theorem *M2*:
  assumes *h*: $|^\sim\ F \longrightarrow G$
  shows $\vdash \Box[F]\text{-}v \longrightarrow \Box[G]\text{-}v$
  using *sq*[*OF h*] *ax4* **by** *force*

theorem *N1*:
  assumes *h*: $\vdash F \longrightarrow G$

shows $|^\sim \bigcirc F \longrightarrow \bigcirc G$
by (*rule pmp*[*OF nex*[*OF h*] *pax2*])

theorem *T4*: $\vdash \Box[P]\text{-}v \longrightarrow \Box[[P]\text{-}v]\text{-}w$
proof −
have $\vdash \Box\Box[P]\text{-}v \longrightarrow \Box[\Box\Box[P]\text{-}v]\text{-}w$ **by** (*rule ax2*)
moreover
from *pax4* **have** $|^\sim \Box\Box[P]\text{-}v \longrightarrow [P]\text{-}v$ **unfolding** *T2*[*int-rewrite*] **by** *force*
hence $\vdash \Box[\Box\Box[P]\text{-}v]\text{-}w \longrightarrow \Box[[P]\text{-}v]\text{-}w$ **by** (*rule M2*)
ultimately show *?thesis* **unfolding** *T2*[*int-rewrite*] **by** (*rule lift-imp-trans*)
qed

theorem *T5*: $\vdash \Box[[P]\text{-}w]\text{-}v \longrightarrow \Box[[P]\text{-}v]\text{-}w$
proof −
have $|^\sim [[P]\text{-}w]\text{-}v \longrightarrow [[P]\text{-}v]\text{-}w$ **by** (*auto simp*: *actrans-def*)
hence $\vdash \Box[[[P]\text{-}w]\text{-}v]\text{-}w \longrightarrow \Box[[[P]\text{-}v]\text{-}w]\text{-}w$ **by** (*rule M2*)
with *T4* **show** *?thesis* **unfolding** *T3*[*int-rewrite*] **by** (*rule lift-imp-trans*)
qed

theorem *T6*: $\vdash \Box F \longrightarrow \Box[\bigcirc F]\text{-}v$
proof −
from *ax1* **have** $|^\sim \bigcirc(\Box F \longrightarrow F)$ **by** (*rule nex*)
with *pax2* **have** $|^\sim \bigcirc\Box F \longrightarrow \bigcirc F$ **by** *force*
with *pax3* **have** $|^\sim \Box F \longrightarrow \bigcirc F$ **by** (*rule pref-imp-trans*)
hence $\vdash \Box[\Box F]\text{-}v \longrightarrow \Box[\bigcirc F]\text{-}v$ **by** (*rule M2*)
with *ax2* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

theorem *T7*:
assumes *h*: $|^\sim F \wedge \mathit{Unchanged}\ v \longrightarrow \bigcirc F$
shows $|^\sim (F \wedge \bigcirc\Box F) = \Box F$
proof −
have $\vdash \Box[\bigcirc F \longrightarrow F \longrightarrow \bigcirc F]\text{-}v$ **by** (*rule sq*) *auto*
with *ax4* **have** $\vdash \Box[\bigcirc F]\text{-}v \longrightarrow \Box[(F \longrightarrow \bigcirc F)]\text{-}v$ **by** *force*
with *ax3*[*OF h*, *unlifted*] **have** $\vdash \Box[\bigcirc F]\text{-}v \longrightarrow (F \longrightarrow \Box F)$ **by** *force*
with *pax5* **have** $|^\sim F \wedge \bigcirc\Box F \longrightarrow \Box F$ **by** *force*
with *ax1*[*of TEMP F*,*unlifted*] *pax3*[*of TEMP F*,*unlifted*] **show** *?thesis* **by** *force*
qed

theorem *T8*: $|^\sim \bigcirc(F \wedge G) = (\bigcirc F \wedge \bigcirc G)$
proof −
have $|^\sim \bigcirc(F \wedge G) \longrightarrow \bigcirc F$ **by** (*rule N1*) *auto*
moreover
have $|^\sim \bigcirc(F \wedge G) \longrightarrow \bigcirc G$ **by** (*rule N1*) *auto*
moreover
have $\vdash F \longrightarrow G \longrightarrow F \wedge G$ **by** *auto*
from *nex*[*OF this*] **have** $|^\sim \bigcirc F \longrightarrow \bigcirc G \longrightarrow \bigcirc(F \wedge G)$
by (*force intro*: *pax2*[*unlift-rule*])
ultimately show *?thesis* **by** *force*

qed

lemma *T9*: $|^{\sim}$ $\Box[A]\text{-}v \longrightarrow [A]\text{-}v$
  using *pax4* **by** *force*

theorem *H1*:
  assumes *h1*: $\vdash \Box[P]\text{-}v$ **and** *h2*: $\vdash \Box[P \longrightarrow Q]\text{-}v$
  shows $\vdash \Box[Q]\text{-}v$
  using *assms ax4*[*unlifted*] **by** *force*

theorem *H2*: **assumes** *h1*: $\vdash F$ **shows** $\vdash \Box[F]\text{-}v$
  using *h1* **by** (*blast dest*: *pre sq*)

theorem *H3*:
  assumes *h1*: $\vdash \Box[P \longrightarrow Q]\text{-}v$ **and** *h2*: $\vdash \Box[Q \longrightarrow R]\text{-}v$
  shows $\vdash \Box[P \longrightarrow R]\text{-}v$
proof −
  have $|^{\sim}$ $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow R) \longrightarrow (P \longrightarrow R)$ **by** *auto*
  hence $\vdash \Box[(P \longrightarrow Q) \longrightarrow (Q \longrightarrow R) \longrightarrow (P \longrightarrow R)]\text{-}v$ **by** (*rule sq*)
  with *h1* **have** $\vdash \Box[(Q \longrightarrow R) \longrightarrow (P \longrightarrow R)]\text{-}v$ **by** (*rule H1*)
  with *h2* **show** *?thesis* **by** (*rule H1*)
qed

theorem *H4*: $\vdash \Box[[P]\text{-}v \longrightarrow P]\text{-}v$
proof −
  have $|^{\sim}$ $v` \neq \$v \longrightarrow ([P]\text{-}v \longrightarrow P)$ **by** (*auto simp*: *unch-def actrans-def*)
  hence $\vdash \Box[v` \neq \$v \longrightarrow ([P]\text{-}v \longrightarrow P)]\text{-}v$ **by** (*rule sq*)
  with *ax5* **show** *?thesis* **by** (*rule H1*)
qed

theorem *H5*: $\vdash \Box[\Box F \longrightarrow \bigcirc\Box F]\text{-}v$
  by (*rule pax3*[*THEN sq*])

5.3 Some other useful derived theorems

theorem *P1*: $|^{\sim}$ $\Box F \longrightarrow \bigcirc F$
proof −
  have $|^{\sim}$ $\bigcirc\Box F \longrightarrow \bigcirc F$ **by** (*rule N1*[*OF ax1*])
  with *pax3* **show** *?thesis* **by** (*rule pref-imp-trans*)
qed

theorem *P2*: $|^{\sim}$ $\Box F \longrightarrow F \wedge \bigcirc F$
  using *ax1*[*of F*] *P1*[*of F*] **by** *force*

theorem *P4*: $\vdash \Box F \longrightarrow \Box[F]\text{-}v$
proof −
  have $\vdash \Box[\Box F]\text{-}v \longrightarrow \Box[F]\text{-}v$ **by** (*rule M2*[*OF pre*[*OF ax1*]])
  with *ax2* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

theorem *P5*: $\vdash \Box[P]\text{-}v \longrightarrow \Box[\Box[P]\text{-}v]\text{-}w$
proof –
  have $\vdash \Box\Box[P]\text{-}v \longrightarrow \Box[\Box[P]\text{-}v]\text{-}w$ **by** (*rule P4*)
  thus *?thesis* **by** (*unfold T2*[*int-rewrite*])
qed

theorem *M0*: $\vdash \Box F \longrightarrow \Box[F \longrightarrow \bigcirc F]\text{-}v$
proof –
  from *P1* **have** $|^{\sim} \Box F \longrightarrow F \longrightarrow \bigcirc F$ **by** *force*
  hence $\vdash \Box[\Box F]\text{-}v \longrightarrow \Box[F \longrightarrow \bigcirc F]\text{-}v$ **by** (*rule M2*)
  with *ax2* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

theorem *M1*: $\vdash \Box F \longrightarrow \Box[F \wedge \bigcirc F]\text{-}v$
proof –
  have $|^{\sim} \Box F \longrightarrow F \wedge \bigcirc F$ **by** (*rule P2*)
  hence $\vdash \Box[\Box F]\text{-}v \longrightarrow \Box[F \wedge \bigcirc F]\text{-}v$ **by** (*rule M2*)
  with *ax2* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

theorem *M3*: **assumes** *h*: $\vdash F$ **shows** $\vdash \Box[\bigcirc F]\text{-}v$
  using *alw*[*OF h*] *T6* **by** (*rule fmp*)

lemma *M4*: $\vdash \Box[\bigcirc(F \wedge G) = (\bigcirc F \wedge \bigcirc G)]\text{-}v$
  by (*rule sq*[*OF T8*])

theorem *M5*: $\vdash \Box[\ \Box[P]\text{-}v \longrightarrow \bigcirc\Box[P]\text{-}v\ ]\text{-}w$
proof (*rule sq*)
  show $|^{\sim} \Box[P]\text{-}v \longrightarrow \bigcirc\Box[P]\text{-}v$ **by** (*auto simp*: *pax4*[*unlifted*])
qed

theorem *M6*: $\vdash \Box[F \wedge G]\text{-}v \longrightarrow \Box[F]\text{-}v \wedge \Box[G]\text{-}v$
proof –
  have $\vdash \Box[F \wedge G]\text{-}v \longrightarrow \Box[F]\text{-}v$ **by** (*rule M2*) *auto*
  moreover
  have $\vdash \Box[F \wedge G]\text{-}v \longrightarrow \Box[G]\text{-}v$ **by** (*rule M2*) *auto*
  ultimately show *?thesis* **by** *force*
qed

theorem *M7*: $\vdash \Box[F]\text{-}v \wedge \Box[G]\text{-}v \longrightarrow \Box[F \wedge G]\text{-}v$
proof –
  have $|^{\sim} F \longrightarrow G \longrightarrow F \wedge G$ **by** *auto*
  hence $\vdash \Box[F]\text{-}v \longrightarrow \Box[G \longrightarrow F \wedge G]\text{-}v$ **by** (*rule M2*)
  with *ax4* **show** *?thesis* **by** *force*
qed

theorem *M8*: $\vdash \Box[F \wedge G]\text{-}v = (\Box[F]\text{-}v \wedge \Box[G]\text{-}v)$
  by (*rule int-iffI*[*OF M6 M7*])

theorem *M9*: $|^{\sim}\ \square F \longrightarrow F \wedge \bigcirc\square F$
  using *pre*[*OF ax1*[*of F*]] *pax3*[*of F*] **by** *force*

theorem *M10*:
  assumes *h*: $|^{\sim}\ F \wedge$ *Unchanged v* $\longrightarrow \bigcirc F$
  shows $|^{\sim}\ F \wedge \bigcirc\square F \longrightarrow \square F$
  using *T7*[*OF h*] **by** *auto*

theorem *M11*:
  assumes *h*: $|^{\sim}\ [A]\text{-}f \longrightarrow [B]\text{-}g$
  shows $\vdash \square[A]\text{-}f \longrightarrow \square[B]\text{-}g$
proof −
  from *h* **have** $\vdash \square[[A]\text{-}f]\text{-}g \longrightarrow \square[[B]\text{-}g]\text{-}g$ **by** (*rule M2*)
  with *T4* **show** *?thesis* **by** *force*
qed

theorem *M12*: $\vdash (\square[A]\text{-}f \wedge \square[B]\text{-}g) = \square[[A]\text{-}f \wedge [B]\text{-}g]\text{-}(f,g)$
proof −
  have $\vdash \square[A]\text{-}f \wedge \square[B]\text{-}g \longrightarrow \square[[A]\text{-}f \wedge [B]\text{-}g]\text{-}(f,g)$
    by (*auto simp*: *M8*[*int-rewrite*] *elim*: *T4*[*unlift-rule*])
  moreover
  have $|^{\sim}\ [[A]\text{-}f \wedge [B]\text{-}g]\text{-}(f,g) \longrightarrow [A]\text{-}f$
    by (*auto simp*: *actrans-def unch-def all-before-eq all-after-eq*)
  hence $\vdash \square[[A]\text{-}f \wedge [B]\text{-}g]\text{-}(f,g) \longrightarrow \square[A]\text{-}f$ **by** (*rule M11*)
  moreover
  have $|^{\sim}\ [[A]\text{-}f \wedge [B]\text{-}g]\text{-}(f,g) \longrightarrow [B]\text{-}g$
    by (*auto simp*: *actrans-def unch-def all-before-eq all-after-eq*)
  hence $\vdash \square[[A]\text{-}f \wedge [B]\text{-}g]\text{-}(f,g) \longrightarrow \square[B]\text{-}g$
    by (*rule M11*)
  ultimately show *?thesis* **by** *force*
qed

We now derive Lamport's 6 simple temporal logic rules (STL1)-(STL6) [5]. Firstly, STL1 is the same as $\vdash$ *?F* $\Longrightarrow \vdash \square$*?F* derived above.

lemmas *STL1* = *alw*

STL2 and STL3 have also already been derived.

lemmas *STL2* = *ax1*

lemmas *STL3* = *T1*

As with the derivation of $\vdash$ *?F* $\Longrightarrow \vdash \square$*?F*, a purely syntactic derivation of (STL4) relies on an additional argument – either using *Unchanged* or *STUTINV*.

theorem *STL4-2*:
  assumes *h1*: $\vdash F \longrightarrow G$ **and** *h2*: $|^{\sim}\ G \wedge$ *Unchanged v* $\longrightarrow \bigcirc G$
  shows $\vdash \square F \longrightarrow \square G$

proof −
  from $ax1[of\ F]$ $h1$ **have** $\vdash \Box F \longrightarrow G$ **by** (*rule lift-imp-trans*)
  moreover
  from $h1$ **have** $|^{\sim} \bigcirc F \longrightarrow \bigcirc G$ **by** (*rule N1*)
  hence $|^{\sim} \bigcirc F \longrightarrow G \longrightarrow \bigcirc G$ **by** *auto*
  hence $\vdash \Box[\bigcirc F]\text{-}v \longrightarrow \Box[G \longrightarrow \bigcirc G]\text{-}v$ **by** (*rule M2*)
  with $T6$ **have** $\vdash \Box F \longrightarrow \Box[G \longrightarrow \bigcirc G]\text{-}v$ **by** (*rule lift-imp-trans*)
  moreover
  from $h2$ **have** $\vdash \Box[G \longrightarrow \bigcirc G]\text{-}v \longrightarrow G \longrightarrow \Box G$ **by** (*rule ax3*)
  ultimately
  show *?thesis* **by** *force*
qed

lemma *STL4-3*:
  assumes $h1$: $\vdash F \longrightarrow G$ **and** $h2$: $STUTINV\ G$
  shows $\vdash \Box F \longrightarrow \Box G$
using $h1$ $h2[THEN\ pre\text{-}id\text{-}unch]$ **by** (*rule STL4-2*)

Of course, the original rule can be derived semantically

lemma *STL4*: **assumes** h: $\vdash F \longrightarrow G$ **shows** $\vdash \Box F \longrightarrow \Box G$
  using h **by** (*force simp*: *always-def*)

Dual rule for $\Diamond$

lemma *STL4-eve*: **assumes** h: $\vdash F \longrightarrow G$ **shows** $\vdash \Diamond F \longrightarrow \Diamond G$
  using h **by** (*force simp*: *eventually-defs*)

Similarly, a purely syntactic derivation of (STL5) requires extra hypotheses.

theorem *STL5-2*:
  assumes $h1$: $|^{\sim} F \wedge Unchanged\ f \longrightarrow \bigcirc F$
    and $h2$: $|^{\sim} G \wedge Unchanged\ g \longrightarrow \bigcirc G$
  shows $\vdash \Box(F \wedge G) = (\Box F \wedge \Box G)$
proof (*rule int-iffI*)
  have $\vdash F \wedge G \longrightarrow F$ **by** *auto*
  from *this* $h1$ **have** $\vdash \Box(F \wedge G) \longrightarrow \Box F$ **by** (*rule STL4-2*)
  moreover
  have $\vdash F \wedge G \longrightarrow G$ **by** *auto*
  from *this* $h2$ **have** $\vdash \Box(F \wedge G) \longrightarrow \Box G$ **by** (*rule STL4-2*)
  ultimately show $\vdash \Box(F \wedge G) \longrightarrow \Box F \wedge \Box G$ **by** *force*
next
  have $|^{\sim} Unchanged\ (f,g) \longrightarrow Unchanged\ f \wedge Unchanged\ g$ **by** (*auto simp*: *unch-defs*)
  with $h1[unlifted]$ $h2[unlifted]$ $T8[of\ F\ G,\ unlifted]$
  have $h3$: $|^{\sim} (F \wedge G) \wedge Unchanged\ (f,g) \longrightarrow \bigcirc(F \wedge G)$ **by** *force*
  from $ax1[of\ F]$ $ax1[of\ G]$ **have** $\vdash \Box F \wedge \Box G \longrightarrow F \wedge G$ **by** *force*
  moreover
  from $ax2[of\ F]$ $ax2[of\ G]$ **have** $\vdash \Box F \wedge \Box G \longrightarrow \Box[\Box F]\text{-}(f,g) \wedge \Box[\Box G]\text{-}(f,g)$ **by** *force*
  with $M8$ **have** $\vdash \Box F \wedge \Box G \longrightarrow \Box[\Box F \wedge \Box G]\text{-}(f,g)$ **by** *force*
  moreover

from *P1*[*of F*] *P1*[*of G*] **have** $|^{\sim}$ $\Box F \wedge \Box G \longrightarrow F \wedge G \longrightarrow \bigcirc(F \wedge G)$
unfolding *T8*[*int-rewrite*] **by** *force*
hence $\vdash \Box[\ \Box F \wedge \Box G\ ]\text{-}(f,g) \longrightarrow \Box[F \wedge G \longrightarrow \bigcirc(F \wedge G)]\text{-}(f,g)$ **by** (*rule M2*)
from *this ax3*[*OF h3*] **have** $\vdash \Box[\ \Box F \wedge \Box G\ ]\text{-}(f,g) \longrightarrow F \wedge G \longrightarrow \Box(F \wedge G)$
by (*rule lift-imp-trans*)
ultimately show $\vdash \Box F \wedge \Box G \longrightarrow \Box(F \wedge G)$ **by** *force*
qed

theorem *STL5-21*:
assumes *h1*: *stutinv F* **and** *h2*: *stutinv G*
shows $\vdash \Box(F \wedge G) = (\Box F \wedge \Box G)$
using *h1*[*THEN pre-id-unch*] *h2*[*THEN pre-id-unch*] **by** (*rule STL5-2*)

We also derive STL5 semantically.

lemma *STL5*: $\vdash \Box(F \wedge G) = (\Box F \wedge \Box G)$
by (*auto simp*: *always-def*)

Elimination rule corresponding to *STL5* in unlifted form.

lemma *box-conjE*:
assumes $s \models \Box F$ **and** $s \models \Box G$ **and** $s \models \Box(F \wedge G) \Longrightarrow P$
shows *P*
using *assms* **by** (*auto simp*: *STL5*[*unlifted*])

lemma *box-thin*:
assumes *h1*: $s \models \Box F$ **and** *h2*: *PROP W*
shows *PROP W*
using *h2* **.**

Finally, we derive STL6 (only semantically)

lemma *STL6*: $\vdash \Diamond\Box(F \wedge G) = (\Diamond\Box F \wedge \Diamond\Box G)$
proof *auto*
fix *w*
assume *a1*: $w \models \Diamond\Box F$ **and** *a2*: $w \models \Diamond\Box G$
from *a1* **obtain** *nf* **where** *nf*: $(w\ |_s\ nf) \models \Box F$ **by** (*auto simp*: *eventually-defs*)
from *a2* **obtain** *ng* **where** *ng*: $(w\ |_s\ ng) \models \Box G$ **by** (*auto simp*: *eventually-defs*)
let *?n = max nf ng*
have $nf \leq ?n$ **by** *simp*
from *this nf* **have** $(w\ |_s\ ?n) \models \Box F$ **by** (*rule linalw*)
moreover
have $ng \leq ?n$ **by** *simp*
from *this ng* **have** $(w\ |_s\ ?n) \models \Box G$ **by** (*rule linalw*)
ultimately
have $(w\ |_s\ ?n) \models \Box(F \wedge G)$ **by** (*rule box-conjE*)
thus $w \models \Diamond\Box(F \wedge G)$ **by** (*auto simp*: *eventually-defs*)
next
fix *w*
assume *h*: $w \models \Diamond\Box(F \wedge G)$
have $\vdash F \wedge G \longrightarrow F$ **by** *auto*
hence $\vdash \Diamond\Box(F \wedge G) \longrightarrow \Diamond\Box F$ **by** (*rule STL4-eve*[*OF STL4*])

with h **show** $w \models \Diamond\Box F$ **by** *auto*
next
fix w
assume h: $w \models \Diamond\Box(F \wedge G)$
have $\vdash F \wedge G \longrightarrow G$ **by** *auto*
hence $\vdash \Diamond\Box(F \wedge G) \longrightarrow \Diamond\Box G$ **by** (*rule STL4-eve*[*OF STL4*])
with h **show** $w \models \Diamond\Box G$ **by** *auto*
qed

lemma *MM0*: $\vdash \Box(F \longrightarrow G) \longrightarrow \Box F \longrightarrow \Box G$
proof −
have $\vdash \Box(F \wedge (F \longrightarrow G)) \longrightarrow \Box G$ **by** (*rule STL4*) *auto*
thus *?thesis* **by** (*auto simp*: *STL5*[*int-rewrite*])
qed

lemma *MM1*: **assumes** h: $\vdash F = G$ **shows** $\vdash \Box F = \Box G$
by (*auto simp*: *h*[*int-rewrite*])

theorem *MM2*: $\vdash \Box A \wedge \Box(B \longrightarrow C) \longrightarrow \Box(A \wedge B \longrightarrow C)$
proof −
have $\vdash \Box(A \wedge (B \longrightarrow C)) \longrightarrow \Box(A \wedge B \longrightarrow C)$ **by** (*rule STL4*) *auto*
thus *?thesis* **by** (*auto simp*: *STL5*[*int-rewrite*])
qed

theorem *MM3*: $\vdash \Box\neg A \longrightarrow \Box(A \wedge B \longrightarrow C)$
by (*rule STL4*) *auto*

theorem *MM4*[*simp-unl*]: $\vdash \Box\#F = \#F$
proof (*cases F*)
assume F
hence *1*: $\vdash \#F$ **by** *auto*
hence $\vdash \Box\#F$ **by** (*rule alw*)
with *1* **show** *?thesis* **by** *force*
next
assume $\neg F$
hence *1*: $\vdash \neg\ \#F$ **by** *auto*
from *ax1* **have** $\vdash \neg\ \#F \longrightarrow \neg\ \Box\#F$ **by** (*rule lift-imp-neg*)
with *1* **show** *?thesis* **by** *force*
qed

theorem *MM4b*[*simp-unl*]: $\vdash \Box\neg\#F = \neg\#F$
proof −
have $\vdash \neg\#F = \#(\neg F)$ **by** *auto*
hence $\vdash \Box\neg\#F = \Box\#(\neg F)$ **by** (*rule MM1*)
thus *?thesis* **by** *auto*
qed

theorem *MM5*: $\vdash \Box F \vee \Box G \longrightarrow \Box(F \vee G)$
proof −

have $\vdash \Box F \longrightarrow \Box(F \vee G)$ **by** (*rule STL4*) *auto*
moreover
have $\vdash \Box G \longrightarrow \Box(F \vee G)$ **by** (*rule STL4*) *auto*
ultimately show *?thesis* **by** *force*
qed

theorem *MM6*: $\vdash \Box F \vee \Box G \longrightarrow \Box(\Box F \vee \Box G)$
proof $-$
have $\vdash \Box\Box F \vee \Box\Box G \longrightarrow \Box(\Box F \vee \Box G)$ **by** (*rule MM5*)
thus *?thesis* **by** *simp*
qed

lemma *MM10*:
assumes *h*: $|^{\sim}\ F = G$ **shows** $\vdash \Box[F]\text{-}v = \Box[G]\text{-}v$
by (*auto simp*: *h*[*int-rewrite*])

lemma *MM9*:
assumes *h*: $\vdash F = G$ **shows** $\vdash \Box[F]\text{-}v = \Box[G]\text{-}v$
by (*rule MM10*[*OF pre*[*OF h*]])

theorem *MM11*: $\vdash \Box[\neg(P \wedge Q)]\text{-}v \longrightarrow \Box[P]\text{-}v \longrightarrow \Box[P \wedge \neg Q]\text{-}v$
proof $-$
have $\vdash \Box[\neg(P \wedge Q)]\text{-}v \longrightarrow \Box[P \longrightarrow P \wedge \neg Q]\text{-}v$ **by** (*rule M2*) *auto*
from *this ax4* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

theorem *MM12*[*simp-unl*]: $\vdash \Box[\Box[P]\text{-}v]\text{-}v = \Box[P]\text{-}v$
proof (*rule int-iffI*)
have $|^{\sim}\ \Box[P]\text{-}v \longrightarrow [P]\text{-}v$ **by** (*auto simp*: *pax4*[*unlifted*])
hence $\vdash \Box[\Box[P]\text{-}v]\text{-}v \longrightarrow \Box[[P]\text{-}v]\text{-}v$ **by** (*rule M2*)
thus $\vdash \Box[\Box[P]\text{-}v]\text{-}v \longrightarrow \Box[P]\text{-}v$ **by** (*unfold T3*[*int-rewrite*])
next
have $\vdash \Box\Box[P]\text{-}v \longrightarrow \Box[\Box\Box[P]\text{-}v]\text{-}v$ **by** (*rule ax2*)
thus $\vdash \Box[P]\text{-}v \longrightarrow \Box[\Box[P]\text{-}v]\text{-}v$ **by** *auto*
qed

5.4 Theorems about the eventually operator

theorem *dualization*:
$\vdash \neg\Box F = \Diamond\neg F$
$\vdash \neg\Diamond F = \Box\neg F$
$\vdash \neg\Box[A]\text{-}v = \Diamond\langle\neg A\rangle\text{-}v$
$\vdash \neg\Diamond\langle A\rangle\text{-}v = \Box[\neg A]\text{-}v$
unfolding *eventually-def angle-action-def* **by** *simp-all*

lemmas *dualization-rew* = *dualization*[*int-rewrite*]
lemmas *dualization-unl* = *dualization*[*unlifted*]

theorem *E1*: $\vdash \Diamond(F \vee G) = (\Diamond F \vee \Diamond G)$

proof −
  have $\vdash \Box\neg(F \vee G) = \Box(\neg F \wedge \neg G)$ **by** (*rule MM1*) *auto*
  thus *?thesis* **unfolding** *eventually-def STL5*[*int-rewrite*] **by** *force*
qed

theorem *E3*: $\vdash F \longrightarrow \Diamond F$
  unfolding *eventually-def* **by** (*force dest*: *ax1*[*unlift-rule*])

theorem *E4*: $\vdash \Box F \longrightarrow \Diamond F$
  by (*rule lift-imp-trans*[*OF ax1 E3*])

theorem *E5*: $\vdash \Box F \longrightarrow \Box\Diamond F$
proof −
  have $\vdash \Box\Box F \longrightarrow \Box\Diamond F$ **by** (*rule STL4*[*OF E4*])
  thus *?thesis* **by** *simp*
qed

theorem *E6*: $\vdash \Box F \longrightarrow \Diamond\Box F$
  using *E4*[*of TEMP* $\Box F$] **by** *simp*

theorem *E7*:
  assumes *h*: $|\sim \neg F \wedge Unchanged\ v \longrightarrow \bigcirc\neg F$
  shows $|\sim \Diamond F \longrightarrow F \vee \bigcirc\Diamond F$
proof −
  from *h* **have** $|\sim \neg F \wedge \bigcirc\Box\neg F \longrightarrow \Box\neg F$ **by** (*rule M10*)
  thus *?thesis* **by** (*auto simp*: *eventually-def*)
qed

theorem *E8*: $\vdash \Diamond(F \longrightarrow G) \longrightarrow \Box F \longrightarrow \Diamond G$
proof −
  have $\vdash \Box(F \wedge \neg G) \longrightarrow \Box\neg(F \longrightarrow G)$ **by** (*rule STL4*) *auto*
  thus *?thesis* **unfolding** *eventually-def STL5*[*int-rewrite*] **by** *auto*
qed

theorem *E9*: $\vdash \Box(F \longrightarrow G) \longrightarrow \Diamond F \longrightarrow \Diamond G$
proof −
  have $\vdash \Box(F \longrightarrow G) \longrightarrow \Box(\neg G \longrightarrow \neg F)$ **by** (*rule STL4*) *auto*
  with *MM0*[*of TEMP* $\neg G$ *TEMP* $\neg F$] **show** *?thesis* **unfolding** *eventually-def*
by *force*
qed

theorem *E10*[*simp-unl*]: $\vdash \Diamond\Diamond F = \Diamond F$
  by (*simp add*: *eventually-def*)

theorem *E22*:
  assumes *h*: $\vdash F = G$
  shows $\vdash \Diamond F = \Diamond G$
  by (*auto simp*: *h*[*int-rewrite*])

theorem *E15*[*simp-unl*]: $\vdash \Diamond\#F = \#F$
  by (*simp add*: *eventually-def*)

theorem *E15b*[*simp-unl*]: $\vdash \Diamond\neg\#F = \neg\#F$
  by (*simp add*: *eventually-def*)

theorem *E16*: $\vdash \Diamond\Box F \longrightarrow \Diamond F$
  by (*rule STL4-eve*[*OF ax1*])

An action version of STL6

lemma *STL6-act*: $\vdash \Diamond(\Box[F]\text{-}v \wedge \Box[G]\text{-}w) = (\Diamond\Box[F]\text{-}v \wedge \Diamond\Box[G]\text{-}w)$
proof −
  have $\vdash (\Diamond\Box(\Box[F]\text{-}v \wedge \Box[G]\text{-}w)) = \Diamond(\Box\Box[F]\text{-}v \wedge \Box\Box[G]\text{-}w)$ **by** (*rule E22*[*OF STL5*])
  thus *?thesis* **by** (*auto simp*: *STL6*[*int-rewrite*])
qed

lemma *SE1*: $\vdash \Box F \wedge \Diamond G \longrightarrow \Diamond(\Box F \wedge G)$
proof −
  have $\vdash \Box\neg(\Box F \wedge G) \longrightarrow \Box(\Box F \longrightarrow \neg G)$ **by** (*rule STL4*) *auto*
  with *MM0* **show** *?thesis* **by** (*force simp*: *eventually-def*)
qed

lemma *SE2*: $\vdash \Box F \wedge \Diamond G \longrightarrow \Diamond(F \wedge G)$
proof −
  have $\vdash \Box F \wedge G \longrightarrow F \wedge G$ **by** (*auto elim*: *ax1*[*unlift-rule*])
  hence $\vdash \Diamond(\Box F \wedge G) \longrightarrow \Diamond(F \wedge G)$ **by** (*rule STL4-eve*)
  with *SE1* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

lemma *SE3*: $\vdash \Box F \wedge \Diamond G \longrightarrow \Diamond(G \wedge F)$
proof −
  have $\vdash \Diamond(F \wedge G) \longrightarrow \Diamond(G \wedge F)$ **by** (*rule STL4-eve*) *auto*
  with *SE2* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

lemma *SE4*:
  assumes *h1*: $s \models \Box F$ **and** *h2*: $s \models \Diamond G$ **and** *h3*: $\vdash \Box F \wedge G \longrightarrow H$
  shows $s \models \Diamond H$
  using *h1 h2 h3*[*THEN STL4-eve*] *SE1* **by** *force*

theorem *E17*: $\vdash \Box\Diamond\Box F \longrightarrow \Box\Diamond F$
  by (*rule STL4*[*OF STL4-eve*[*OF ax1*]])

theorem *E18*: $\vdash \Box\Diamond\Box F \longrightarrow \Diamond\Box F$
  by (*rule ax1*)

theorem *E19*: $\vdash \Diamond\Box F \longrightarrow \Box\Diamond\Box F$
proof −

have $\vdash (\Box F \wedge \neg\Box F) = \#False$ **by** *auto*
hence $\vdash \Diamond\Box(\Box F \wedge \neg\Box F) = \Diamond\Box\#False$ **by** (*rule E22*[*OF MM1*])
thus *?thesis* **unfolding** *STL6*[*int-rewrite*] **by** (*auto simp*: *eventually-def*)
qed

theorem *E20*: $\vdash \Diamond\Box F \longrightarrow \Box\Diamond F$
by (*rule lift-imp-trans*[*OF E19 E17*])

theorem *E21*[*simp-unl*]: $\vdash \Box\Diamond\Box F = \Diamond\Box F$
by (*rule int-iffI*[*OF E18 E19*])

theorem *E27*[*simp-unl*]: $\vdash \Diamond\Box\Diamond F = \Box\Diamond F$
using *E21* **unfolding** *eventually-def* **by** *force*

lemma *E28*: $\vdash \Diamond\Box F \wedge \Box\Diamond G \longrightarrow \Box\Diamond(F \wedge G)$
proof –
have $\vdash \Diamond\Box(\Box F \wedge \Diamond G) \longrightarrow \Diamond\Box\Diamond(F \wedge G)$ **by** (*rule STL4-eve*[*OF STL4*[*OF SE2*]])
thus *?thesis* **by** (*simp add*: *STL6*[*int-rewrite*])
qed

lemma *E23*: $|^\sim \circ F \longrightarrow \Diamond F$
using *P1* **by** (*force simp*: *eventually-def*)

lemma *E24*: $\vdash \Diamond\Box Q \longrightarrow \Box[\Diamond Q]\text{-}v$
by (*rule lift-imp-trans*[*OF E20 P4*])

lemma *E25*: $\vdash \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond A$
using *P4* **by** (*force simp*: *eventually-def angle-action-def*)

lemma *E26*: $\vdash \Box\Diamond\langle A\rangle\text{-}v \longrightarrow \Box\Diamond A$
by (*rule STL4*[*OF E25*])

lemma *allBox*: $(s \models \Box(\forall x.\ F\ x)) = (\forall x.\ s \models \Box(F\ x))$
unfolding *allT*[*unlifted*] **..**

lemma *E29*: $|^\sim \circ\Diamond F \longrightarrow \Diamond F$
unfolding *eventually-def* **using** *pax3* **by** *force*

lemma *E30*:
assumes *h1*: $\vdash F \longrightarrow \Box F$ **and** *h2*: $\vdash \Diamond F$
shows $\vdash \Diamond\Box F$
using *h2 h1*[*THEN STL4-eve*] **by** (*rule fmp*)

lemma *E31*: $\vdash \Box(F \longrightarrow \Box F) \wedge \Diamond F \longrightarrow \Diamond\Box F$
proof –
have $\vdash \Box(F \longrightarrow \Box F) \wedge \Diamond F \longrightarrow \Diamond(\Box(F \longrightarrow \Box F) \wedge F)$ **by** (*rule SE1*)
moreover
have $\vdash \Box(F \longrightarrow \Box F) \wedge F \longrightarrow \Box F$ **using** *ax1*[*of TEMP* $F \longrightarrow \Box F$] **by** *auto*
hence $\vdash \Diamond(\Box(F \longrightarrow \Box F) \wedge F) \longrightarrow \Diamond\Box F$ **by** (*rule STL4-eve*)

ultimately show *?thesis* **by** (*rule lift-imp-trans*)
qed

lemma *allActBox*: $(s \models \Box[(\forall x.\ F\ x)]\text{-}v) = (\forall x.\ s \models \Box[(F\ x)]\text{-}v)$
unfolding *allActT*[*unlifted*] **..**

theorem *exEE*: $\vdash (\exists x.\ \Diamond(F\ x)) = \Diamond(\exists x.\ F\ x)$
proof −
have $\vdash \neg(\exists\ x.\ \Diamond(F\ x)) = \neg\Diamond(\exists\ x.\ F\ x)$
by (*auto simp*: *eventually-def Not-Rex*[*int-rewrite*] *allBox*)
thus *?thesis* **by** *force*
qed

theorem *exActE*: $\vdash (\exists x.\ \Diamond\langle F\ x\rangle\text{-}v) = \Diamond\langle(\exists x.\ F\ x)\rangle\text{-}v$
proof −
have $\vdash \neg(\exists x.\ \Diamond\langle F\ x\rangle\text{-}v) = \neg\Diamond\langle(\exists x.\ F\ x)\rangle\text{-}v$
by (*auto simp*: *angle-action-def Not-Rex*[*int-rewrite*] *allActBox*)
thus *?thesis* **by** *force*
qed

5.5 Theorems about the leadsto operator

theorem *LT1*: $\vdash F \leadsto F$
unfolding *leadsto-def* **by** (*rule alw*[*OF E3*])

theorem *LT2*: **assumes** *h*: $\vdash F \longrightarrow G$ **shows** $\vdash F \longrightarrow \Diamond G$
by (*rule lift-imp-trans*[*OF h E3*])

theorem *LT3*: **assumes** *h*: $\vdash F \longrightarrow G$ **shows** $\vdash F \leadsto G$
unfolding *leadsto-def* **by** (*rule alw*[*OF LT2*[*OF h*]])

theorem *LT4*: $\vdash F \longrightarrow (F \leadsto G) \longrightarrow \Diamond G$
unfolding *leadsto-def* **using** *ax1*[*of TEMP F* $\longrightarrow \Diamond G$] **by** *auto*

theorem *LT5*: $\vdash \Box(F \longrightarrow \Diamond G) \longrightarrow \Diamond F \longrightarrow \Diamond G$
using *E9*[*of F TEMP* $\Diamond G$] **by** *simp*

theorem *LT6*: $\vdash \Diamond F \longrightarrow (F \leadsto G) \longrightarrow \Diamond G$
unfolding *leadsto-def* **using** *LT5*[*of F G*] **by** *auto*

theorem *LT9*[*simp-unl*]: $\vdash \Box(F \leadsto G) = (F \leadsto G)$
by (*auto simp*: *leadsto-def*)

theorem *LT7*: $\vdash \Box\Diamond F \longrightarrow (F \leadsto G) \longrightarrow \Box\Diamond G$
proof −
have $\vdash \Box\Diamond F \longrightarrow \Box((F \leadsto G) \longrightarrow \Diamond G)$ **by** (*rule STL4*[*OF LT6*])
from *lift-imp-trans*[*OF this MM0*] **show** *?thesis* **by** *simp*
qed

theorem *LT8*: $\vdash \Box\Diamond G \longrightarrow (F \leadsto G)$
  unfolding *leadsto-def* **by** (*rule STL4*) *auto*

theorem *LT13*: $\vdash (F \leadsto G) \longrightarrow (G \leadsto H) \longrightarrow (F \leadsto H)$
proof −
  have $\vdash \Diamond G \longrightarrow (G \leadsto H) \longrightarrow \Diamond H$ **by** (*rule LT6*)
  hence $\vdash \Box(F \longrightarrow \Diamond G) \longrightarrow \Box((G \leadsto H) \longrightarrow (F \longrightarrow \Diamond H))$ **by** (*intro STL4*) *auto*
  from *lift-imp-trans*[*OF this MM0*] **show** *?thesis* **by** (*simp add*: *leadsto-def*)
qed

theorem *LT11*: $\vdash (F \leadsto G) \longrightarrow (F \leadsto (G \vee H))$
proof −
  have $\vdash G \leadsto (G \vee H)$ **by** (*rule LT3*) *auto*
  with *LT13*[*of F G TEMP* $(G \vee H)$] **show** *?thesis* **by** *force*
qed

theorem *LT12*: $\vdash (F \leadsto H) \longrightarrow (F \leadsto (G \vee H))$
proof −
  have $\vdash H \leadsto (G \vee H)$ **by** (*rule LT3*) *auto*
  with *LT13*[*of F H TEMP* $(G \vee H)$] **show** *?thesis* **by** *force*
qed

theorem *LT14*: $\vdash ((F \vee G) \leadsto H) \longrightarrow (F \leadsto H)$
  unfolding *leadsto-def* **by** (*rule STL4*) *auto*

theorem *LT15*: $\vdash ((F \vee G) \leadsto H) \longrightarrow (G \leadsto H)$
  unfolding *leadsto-def* **by** (*rule STL4*) *auto*

theorem *LT16*: $\vdash (F \leadsto H) \longrightarrow (G \leadsto H) \longrightarrow ((F \vee G) \leadsto H)$
proof −
  have $\vdash \Box(F \longrightarrow \Diamond H) \longrightarrow \Box((G \longrightarrow \Diamond H) \longrightarrow (F \vee G \longrightarrow \Diamond H))$ **by** (*rule STL4*)
auto
  from *lift-imp-trans*[*OF this MM0*] **show** *?thesis* **by** (*unfold leadsto-def*)
qed

theorem *LT17*: $\vdash ((F \vee G) \leadsto H) = ((F \leadsto H) \wedge (G \leadsto H))$
  by (*auto elim*: *LT14*[*unlift-rule*] *LT15*[*unlift-rule*]
          LT16[*unlift-rule*])

theorem *LT10*:
  assumes *h*: $\vdash (F \wedge \neg G) \leadsto G$
  shows $\vdash F \leadsto G$
proof −
  from *h* **have** $\vdash ((F \wedge \neg G) \vee G) \leadsto G$
    by (*auto simp*: *LT17*[*int-rewrite*] *LT1*[*int-rewrite*])
  moreover
  have $\vdash F \leadsto ((F \wedge \neg G) \vee G)$ **by** (*rule LT3*, *auto*)
  ultimately
  show *?thesis* **by** (*force elim*: *LT13*[*unlift-rule*])

qed

theorem *LT18*: $\vdash (A \leadsto (B \vee C)) \longrightarrow (B \leadsto D) \longrightarrow (C \leadsto D) \longrightarrow (A \leadsto D)$
proof –
  have $\vdash (B \leadsto D) \longrightarrow (C \leadsto D) \longrightarrow ((B \vee C) \leadsto D)$ **by** (*rule LT16*)
  thus *?thesis* **by** (*force elim*: *LT13*[*unlift-rule*])
qed

theorem *LT19*: $\vdash (A \leadsto (D \vee B)) \longrightarrow (B \leadsto D) \longrightarrow (A \leadsto D)$
  using *LT18*[*of A D B D*] *LT1*[*of D*] **by** *force*

theorem *LT20*: $\vdash (A \leadsto (B \vee D)) \longrightarrow (B \leadsto D) \longrightarrow (A \leadsto D)$
  using *LT18*[*of A B D D*] *LT1*[*of D*] **by** *force*

theorem *LT21*: $\vdash ((\exists x.\ F\ x) \leadsto G) = (\forall x.\ (F\ x \leadsto G))$
proof –
  have $\vdash \Box((\exists x.\ F\ x) \longrightarrow \Diamond G) = \Box(\forall x.\ (F\ x \longrightarrow \Diamond G))$ **by** (*rule MM1*) *auto*
  thus *?thesis* **by** (*unfold leadsto-def allT*[*int-rewrite*])
qed

theorem *LT22*: $\vdash (F \leadsto (G \vee H)) \longrightarrow \Box\neg G \longrightarrow (F \leadsto H)$
proof –
  have $\vdash \Box\neg G \longrightarrow (G \leadsto H)$ **unfolding** *leadsto-def* **by** (*rule STL4*) *auto*
  thus *?thesis* **by** (*force elim*: *LT20*[*unlift-rule*])
qed

lemma *LT23*: $|^\sim\ (P \longrightarrow \bigcirc Q) \longrightarrow (P \longrightarrow \Diamond Q)$
  by (*auto dest*: *E23*[*unlift-rule*])

theorem *LT24*: $\vdash \Box I \longrightarrow ((P \wedge I) \leadsto Q) \longrightarrow P \leadsto Q$
proof –
  have $\vdash \Box I \longrightarrow \Box((P \wedge I \longrightarrow \Diamond Q) \longrightarrow (P \longrightarrow \Diamond Q))$ **by** (*rule STL4*) *auto*
  from *lift-imp-trans*[*OF this MM0*] **show** *?thesis* **by** (*unfold leadsto-def*)
qed

theorem *LT25*[*simp-unl*]: $\vdash (F \leadsto \#False) = \Box\neg F$
unfolding *leadsto-def* **proof** (*rule MM1*)
  show $\vdash (F \longrightarrow \Diamond\#False) = \neg F$ **by** *simp*
qed

lemma *LT28*:
  assumes *h*: $|^\sim\ P \longrightarrow \bigcirc P \vee \bigcirc Q$
  shows $|^\sim\ (P \longrightarrow \bigcirc P) \vee \Diamond Q$
  using *h E23*[*of Q*] **by** *force*

lemma *LT29*:
  assumes *h1*: $|^\sim\ P \longrightarrow \bigcirc P \vee \bigcirc Q$ **and** *h2*: $|^\sim\ P \wedge Unchanged\ v \longrightarrow \bigcirc P$
  shows $\vdash P \longrightarrow \Box P \vee \Diamond Q$
proof –

from *h1*[*THEN LT28*] **have** $|{\sim}\ \Box\neg Q \longrightarrow (P \longrightarrow \bigcirc P)$ **unfolding** *eventually-def* **by** *auto*
hence $\vdash \Box[\Box\neg Q]\text{-}v \longrightarrow \Box[P \longrightarrow \bigcirc P]\text{-}v$ **by** (*rule M2*)
moreover
have $\vdash \neg\Diamond Q \longrightarrow \Box[\Box\neg Q]\text{-}v$ **unfolding** *dualization-rew* **by** (*rule ax2*)
moreover
note *ax3*[*OF h2*]
ultimately
show *?thesis* **by** *force*
qed

lemma *LT30*:
assumes *h*: $|{\sim}\ P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
shows $|{\sim}\ N \longrightarrow (P \longrightarrow \bigcirc P) \vee \Diamond Q$
using *h E23* **by** *force*

lemma *LT31*:
assumes *h1*: $|{\sim}\ P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$ **and** *h2*: $|{\sim}\ P \wedge \mathit{Unchanged}\ v \longrightarrow \bigcirc P$
shows$\vdash \Box N \longrightarrow P \longrightarrow \Box P \vee \Diamond Q$
proof –
from *h1*[*THEN LT30*] **have** $|{\sim}\ N \longrightarrow \Box\neg Q \longrightarrow P \longrightarrow \bigcirc P$ **unfolding** *eventually-def* **by** *auto*
hence $\vdash \Box[N \longrightarrow \Box\neg Q \longrightarrow P \longrightarrow \bigcirc P]\text{-}v$ **by** (*rule sq*)
hence $\vdash \Box[N]\text{-}v \longrightarrow \Box[\Box\neg Q]\text{-}v \longrightarrow \Box[P \longrightarrow \bigcirc P]\text{-}v$
by (*force intro*: *ax4*[*unlift-rule*])
with *P4* **have** $\vdash \Box N \longrightarrow \Box[\Box\neg Q]\text{-}v \longrightarrow \Box[P \longrightarrow \bigcirc P]\text{-}v$ **by** (*rule lift-imp-trans*)
moreover
have $\vdash \neg\Diamond Q \longrightarrow \Box[\Box\neg Q]\text{-}v$ **unfolding** *dualization-rew* **by** (*rule ax2*)
moreover
note *ax3*[*OF h2*]
ultimately
show *?thesis* **by** *force*
qed

lemma *LT33*: $\vdash ((\#P \wedge F) \leadsto G) = (\#P \longrightarrow (F \leadsto G))$
by (*cases P*, *auto simp*: *leadsto-def*)

lemma *AA1*: $\vdash \Box[\#\mathit{False}]\text{-}v \longrightarrow \neg\Diamond\langle Q\rangle\text{-}v$
unfolding *dualization-rew* **by** (*rule M2*) *auto*

lemma *AA2*: $\vdash \Box[P]\text{-}v \wedge \Diamond\langle Q\rangle\text{-}v \longrightarrow \Diamond\langle P \wedge Q\rangle\text{-}v$
proof –
have $\vdash \Box[P \longrightarrow {\sim}(P \wedge Q) \longrightarrow \neg Q]\text{-}v$ **by** (*rule sq*) (*auto simp*: *actrans-def*)
hence $\vdash \Box[P]\text{-}v \longrightarrow \Box[{\sim}(P \wedge Q)]\text{-}v \longrightarrow \Box[\neg Q]\text{-}v$
by (*force intro*: *ax4*[*unlift-rule*])
thus *?thesis* **by** (*auto simp*: *angle-action-def*)
qed

lemma *AA3*: $\vdash \Box P \wedge \Box[P \longrightarrow Q]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond Q$

proof −
  have $\vdash \Box P \wedge \Box[P \longrightarrow Q]\text{-}v \longrightarrow \Box[P \wedge (P \longrightarrow Q)]\text{-}v$
    by (*auto dest*: *P4*[*unlift-rule*] *simp*: *M8*[*int-rewrite*])
  moreover
  have $\vdash \Box[P \wedge (P \longrightarrow Q)]\text{-}v \longrightarrow \Box[Q]\text{-}v$ **by** (*rule M2*) *auto*
  ultimately have $\vdash \Box P \wedge \Box[P \longrightarrow Q]\text{-}v \longrightarrow \Box[Q]\text{-}v$ **by** (*rule lift-imp-trans*)
  moreover
  have $\vdash \Diamond(Q \wedge A) \longrightarrow \Diamond Q$ **by** (*rule STL4-eve*) *auto*
  hence $\vdash \Diamond\langle Q \wedge A\rangle\text{-}v \longrightarrow \Diamond Q$ **by** (*force dest*: *E25*[*unlift-rule*])
  with *AA2* **have** $\vdash \Box[Q]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond Q$ **by** (*rule lift-imp-trans*)
  ultimately show *?thesis* **by** *force*
qed

lemma *AA4*: $\vdash \Diamond\langle\langle A\rangle\text{-}v\rangle\text{-}w \longrightarrow \Diamond\langle\langle A\rangle\text{-}w\rangle\text{-}v$
  unfolding *angle-action-def angle-actrans-def* **using** *T5* **by** *force*

lemma *AA7*: **assumes** *h*: $|\sim\ F \longrightarrow G$ **shows** $\vdash \Diamond\langle F\rangle\text{-}v \longrightarrow \Diamond\langle G\rangle\text{-}v$
proof −
  from *h* **have** $\vdash \Box[\neg G]\text{-}v \longrightarrow \Box[\neg F]\text{-}v$ **by** (*intro M2*) *auto*
  thus *?thesis* **unfolding** *angle-action-def* **by** *force*
qed

lemma *AA6*: $\vdash \Box[P \longrightarrow Q]\text{-}v \wedge \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle Q\rangle\text{-}v$
proof −
  have $\vdash \Diamond\langle(P \longrightarrow Q) \wedge P\rangle\text{-}v \longrightarrow \Diamond\langle Q\rangle\text{-}v$ **by** (*rule AA7*) *auto*
  with *AA2* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

lemma *AA8*: $\vdash \Box[P]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle\Box[P]\text{-}v \wedge A\rangle\text{-}v$
proof −
  have $\vdash \Box[\Box[P]\text{-}v]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle\Box[P]\text{-}v \wedge A\rangle\text{-}v$ **by** (*rule AA2*)
  with *P5* **show** *?thesis* **by** *force*
qed

lemma *AA9*: $\vdash \Box[P]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle[P]\text{-}v \wedge A\rangle\text{-}v$
proof −
  have $\vdash \Box[[P]\text{-}v]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle[P]\text{-}v \wedge A\rangle\text{-}v$ **by** (*rule AA2*)
  thus *?thesis* **by** *simp*
qed

lemma *AA10*: $\vdash \neg(\Box[P]\text{-}v \wedge \Diamond\langle\neg P\rangle\text{-}v)$
  unfolding *angle-action-def* **by** *auto*

lemma *AA11*: $\vdash \neg\Diamond\langle v\$ = \$v\rangle\text{-}v$
  unfolding *dualization-rew* **by** (*rule ax5*)

lemma *AA15*: $\vdash \Diamond\langle P \wedge Q\rangle\text{-}v \longrightarrow \Diamond\langle P\rangle\text{-}v$
  by (*rule AA7*) *auto*

lemma *AA16*: $\vdash \Diamond\langle P \wedge Q\rangle\text{-}v \longrightarrow \Diamond\langle Q\rangle\text{-}v$
  by (*rule AA7*) *auto*

lemma *AA13*: $\vdash \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle v\$ \neq \$v\rangle\text{-}v$
proof −
  have $\vdash \Box[v\$ \neq \$v]\text{-}v \wedge \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle v\$ \neq \$v \wedge P\rangle\text{-}v$ **by** (*rule AA2*)
  hence $\vdash \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle v\$ \neq \$v \wedge P\rangle\text{-}v$ **by** (*simp add*: *ax5*[*int-rewrite*])
  from *this AA15* **show** *?thesis* **by** (*rule lift-imp-trans*)
qed

lemma *AA14*: $\vdash \Diamond\langle P \vee Q\rangle\text{-}v = (\Diamond\langle P\rangle\text{-}v \vee \Diamond\langle Q\rangle\text{-}v)$
proof −
  have $\vdash \Box[\neg(P \vee Q)]\text{-}v = \Box[\neg P \wedge \neg Q]\text{-}v$ **by** (*rule MM10*) *auto*
  hence $\vdash \Box[\neg(P \vee Q)]\text{-}v = (\Box[\neg P]\text{-}v \wedge \Box[\neg Q]\text{-}v)$ **by** (*unfold M8*[*int-rewrite*])
  thus *?thesis* **unfolding** *angle-action-def* **by** *auto*
qed

lemma *AA17*: $\vdash \Diamond\langle [P]\text{-}v \wedge A\rangle\text{-}v \longrightarrow \Diamond\langle P \wedge A\rangle\text{-}v$
proof −
  have $\vdash \Box[v\$ \neq \$v \wedge \neg(P \wedge A)]\text{-}v \longrightarrow \Box[\neg([P]\text{-}v \wedge A)]\text{-}v$
    by (*rule M2*) (*auto simp*: *actrans-def unch-def*)
  with *ax5*[*of v*] **show** *?thesis*
    unfolding *angle-action-def M8*[*int-rewrite*] **by** *force*
qed

lemma *AA19*: $\vdash \Box P \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle P \wedge A\rangle\text{-}v$
  using *P4* **by** (*force intro*: *AA2*[*unlift-rule*])

lemma *AA20*:
  assumes *h1*: $|^\sim\ P \longrightarrow \bigcirc P \vee \bigcirc Q$
    and *h2*: $|^\sim\ P \wedge A \longrightarrow \bigcirc Q$
    and *h3*: $|^\sim\ P \wedge \mathit{Unchanged}\ w \longrightarrow \bigcirc P$
  shows $\vdash \Box(\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v) \longrightarrow (P \leadsto Q)$
proof −
  from *h2 E23* **have** $|^\sim\ P \wedge A \longrightarrow \Diamond Q$ **by** *force*
  hence $\vdash \Diamond\langle P \wedge A\rangle\text{-}v \longrightarrow \Diamond\langle \Diamond Q\rangle\text{-}v$ **by** (*rule AA7*)
  with *E25*[*of TEMP* $\Diamond Q$ *v*] **have** $\vdash \Diamond\langle P \wedge A\rangle\text{-}v \longrightarrow \Diamond Q$ **by** *force*
  with *AA19* **have** $\vdash \Box P \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond Q$ **by** (*rule lift-imp-trans*)
  with *LT29*[*OF h1 h3*] **have** $\vdash (\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v) \longrightarrow (P \longrightarrow \Diamond Q)$ **by** *force*
  thus *?thesis* **unfolding** *leadsto-def* **by** (*rule STL4*)
qed

lemma *AA21*: $|^\sim\ \Diamond\langle \bigcirc F\rangle\text{-}v \longrightarrow \bigcirc\Diamond F$
  using *pax5*[*of TEMP* $\neg F$ *v*] **unfolding** *angle-action-def eventually-def* **by** *auto*

theorem *AA24*[*simp-unl*]: $\vdash \Diamond\langle\langle P\rangle\text{-}f\rangle\text{-}f = \Diamond\langle P\rangle\text{-}f$
  unfolding *angle-action-def angle-actrans-def* **by** *simp*

lemma *AA22*:

assumes *h1*: $|^{\sim}\ P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
and *h2*: $|^{\sim}\ P \wedge N \wedge \langle A\rangle\text{-}v \longrightarrow \bigcirc Q$
and *h3*: $|^{\sim}\ P \wedge \mathit{Unchanged}\ w \longrightarrow \bigcirc P$
shows $\vdash \Box N \wedge \Box(\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v) \longrightarrow (P \leadsto Q)$
proof −
from *h2* **have** $|^{\sim}\ \langle(N \wedge P) \wedge A\rangle\text{-}v \longrightarrow \bigcirc Q$ **by** (*auto simp*: *angle-actrans-sem*[*int-rewrite*])
from *pref-imp-trans*[*OF this E23*] **have** $\vdash \Diamond\langle\langle(N \wedge P) \wedge A\rangle\text{-}v\rangle\text{-}v \longrightarrow \Diamond\langle\Diamond Q\rangle\text{-}v$
by (*rule AA7*)
hence $\vdash \Diamond\langle(N \wedge P) \wedge A\rangle\text{-}v \longrightarrow \Diamond Q$ **by** (*force dest*: *E25*[*unlift-rule*])
with *AA19* **have** $\vdash \Box(N \wedge P) \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond Q$ **by** (*rule lift-imp-trans*)
hence $\vdash \Box N \wedge \Box P \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond Q$ **by** (*auto simp*: *STL5*[*int-rewrite*])
with *LT31*[*OF h1 h3*] **have** $\vdash \Box N \wedge (\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v) \longrightarrow (P \longrightarrow \Diamond Q)$ **by** *force*
hence $\vdash \Box(\Box N \wedge (\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v)) \longrightarrow \Box(P \longrightarrow \Diamond Q)$ **by** (*rule STL4*)
thus *?thesis* **by** (*simp add*: *leadsto-def STL5*[*int-rewrite*])
qed

lemma *AA23*:
assumes $|^{\sim}\ P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
and $|^{\sim}\ P \wedge N \wedge \langle A\rangle\text{-}v \longrightarrow \bigcirc Q$
and $|^{\sim}\ P \wedge \mathit{Unchanged}\ w \longrightarrow \bigcirc P$
shows $\vdash \Box N \wedge \Box\Diamond\langle A\rangle\text{-}v \longrightarrow (P \leadsto Q)$
proof −
have $\vdash \Box\Diamond\langle A\rangle\text{-}v \longrightarrow \Box(\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v)$ **by** (*rule STL4*) *auto*
with *AA22*[*OF assms*] **show** *?thesis* **by** *force*
qed

lemma *AA25*:
assumes *h*: $|^{\sim}\ \langle P\rangle\text{-}v \longrightarrow \langle Q\rangle\text{-}w$
shows $\vdash \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle Q\rangle\text{-}w$
proof −
from *h* **have** $\vdash \Diamond\langle\langle P\rangle\text{-}v\rangle\text{-}v \longrightarrow \Diamond\langle\langle P\rangle\text{-}w\rangle\text{-}v$
by (*intro AA7*) (*auto simp*: *angle-actrans-def actrans-def*)
with *AA4* **have** $\vdash \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle\langle P\rangle\text{-}v\rangle\text{-}w$ **by** *force*
from *this AA7*[*OF h*] **have** $\vdash \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle\langle Q\rangle\text{-}w\rangle\text{-}w$ **by** (*rule lift-imp-trans*)
thus *?thesis* **by** *simp*
qed

lemma *AA26*:
assumes *h*: $|^{\sim}\ \langle A\rangle\text{-}v = \langle B\rangle\text{-}w$
shows $\vdash \Diamond\langle A\rangle\text{-}v = \Diamond\langle B\rangle\text{-}w$
proof −
from *h* **have** $|^{\sim}\ \langle A\rangle\text{-}v \longrightarrow \langle B\rangle\text{-}w$ **by** *auto*
hence $\vdash \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle B\rangle\text{-}w$ **by** (*rule AA25*)
moreover
from *h* **have** $|^{\sim}\ \langle B\rangle\text{-}w \longrightarrow \langle A\rangle\text{-}v$ **by** *auto*
hence $\vdash \Diamond\langle B\rangle\text{-}w \longrightarrow \Diamond\langle A\rangle\text{-}v$ **by** (*rule AA25*)
ultimately
show *?thesis* **by** *force*
qed

theorem *AA28* [*simp-unl*]: $\vdash \Diamond\Diamond\langle A\rangle\text{-}v = \Diamond\langle A\rangle\text{-}v$
  unfolding *eventually-def angle-action-def* **by** *simp*

theorem *AA29*: $\vdash \Box[N]\text{-}v \wedge \Box\Diamond\langle A\rangle\text{-}v \longrightarrow \Box\Diamond\langle N \wedge A\rangle\text{-}v$
proof −
  have $\vdash \Box(\Box[N]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v) \longrightarrow \Box\Diamond\langle N \wedge A\rangle\text{-}v$ **by** (*rule STL4* [*OF AA2*])
  thus *?thesis* **by** (*simp add*: *STL5* [*int-rewrite*])
qed

theorem *AA30* [*simp-unl*]: $\vdash \Diamond\langle\Diamond\langle P\rangle\text{-}f\rangle\text{-}f = \Diamond\langle P\rangle\text{-}f$
  unfolding *angle-action-def* **by** *simp*

theorem *AA31*: $\vdash \Diamond\langle\bigcirc F\rangle\text{-}v \longrightarrow \Diamond F$
  using *pref-imp-trans* [*OF AA21 E29*] **by** *auto*

lemma *AA32* [*simp-unl*]: $\vdash \Box\Diamond\Box[A]\text{-}v = \Diamond\Box[A]\text{-}v$
  using *E21* [*of TEMP* $\Box[A]\text{-}v$] **by** *simp*

lemma *AA33* [*simp-unl*]: $\vdash \Diamond\Box\Diamond\langle A\rangle\text{-}v = \Box\Diamond\langle A\rangle\text{-}v$
  using *E27* [*of TEMP* $\Diamond\langle A\rangle\text{-}v$] **by** *simp*

5.6 Lemmas about the next operator

lemma *N2*: **assumes** *h*: $\vdash F = G$ **shows** $|\!\sim \bigcirc F = \bigcirc G$
  by (*simp add*: *h* [*int-rewrite*])

lemmas *next-and* = *T8*

lemma *next-or*: $|\!\sim \bigcirc(F \vee G) = (\bigcirc F \vee \bigcirc G)$
proof (*rule pref-iffI*)
  have $|\!\sim \bigcirc((F \vee G) \wedge \neg F) \longrightarrow \bigcirc G$ **by** (*rule N1*) *auto*
  thus $|\!\sim \bigcirc(F \vee G) \longrightarrow \bigcirc F \vee \bigcirc G$ **by** (*auto simp*: *T8* [*int-rewrite*])
next
  have $|\!\sim \bigcirc F \longrightarrow \bigcirc(F \vee G)$ **by** (*rule N1*) *auto*
  moreover have $|\!\sim \bigcirc G \longrightarrow \bigcirc(F \vee G)$ **by** (*rule N1*) *auto*
  ultimately show $|\!\sim \bigcirc F \vee \bigcirc G \longrightarrow \bigcirc(F \vee G)$ **by** *force*
qed

lemma *next-imp*: $|\!\sim \bigcirc(F \longrightarrow G) = (\bigcirc F \longrightarrow \bigcirc G)$
proof (*rule pref-iffI*)
  have $|\!\sim \bigcirc G \longrightarrow \bigcirc(F \longrightarrow G)$ **by** (*rule N1*) *auto*
  moreover have $|\!\sim \bigcirc\neg F \longrightarrow \bigcirc(F \longrightarrow G)$ **by** (*rule N1*) *auto*
  ultimately show $|\!\sim (\bigcirc F \longrightarrow \bigcirc G) \longrightarrow \bigcirc(F \longrightarrow G)$ **by** *force*
qed (*rule pax2*)

lemmas *next-not* = *pax1*

lemma *next-eq*: $|\!\sim \bigcirc(F = G) = (\bigcirc F = \bigcirc G)$

proof −
  have $|^{\sim}\ \bigcirc(F = G) = \bigcirc((F \longrightarrow G) \wedge (G \longrightarrow F))$ **by** (*rule N2*) *auto*
  from *this*[*int-rewrite*] **show** *?thesis*
    by (*auto simp*: *next-and*[*int-rewrite*] *next-imp*[*int-rewrite*])
qed

lemma *next-noteq*: $|^{\sim}\ \bigcirc(F \neq G) = (\bigcirc F \neq \bigcirc G)$
  by (*simp add*: *next-eq*[*int-rewrite*])

lemma *next-const*[*simp-unl*]: $|^{\sim}\ \bigcirc\#P = \#P$
proof (*cases P*)
  assume P
  hence *1*: $\vdash \#P$ **by** *auto*
  hence $|^{\sim}\ \bigcirc\#P$ **by** (*rule nex*)
  with *1* **show** *?thesis* **by** *force*
next
  assume $\neg P$
  hence *1*: $\vdash \neg\#P$ **by** *auto*
  hence $|^{\sim}\ \bigcirc\neg\#P$ **by** (*rule nex*)
  with *1* **show** *?thesis* **by** *force*
qed

The following are proved semantically because they are essentially first-order theorems.

lemma *next-fun1*: $|^{\sim}\ \bigcirc f{<}x{>} = f{<}\bigcirc x{>}$
  by (*auto simp*: *nexts-def*)

lemma *next-fun2*: $|^{\sim}\ \bigcirc f{<}x,y{>} = f{<}\bigcirc x,\bigcirc y{>}$
  by (*auto simp*: *nexts-def*)

lemma *next-fun3*: $|^{\sim}\ \bigcirc f{<}x,y,z{>} = f{<}\bigcirc x,\bigcirc y,\bigcirc z{>}$
  by (*auto simp*: *nexts-def*)

lemma *next-fun4*: $|^{\sim}\ \bigcirc f{<}x,y,z,zz{>} = f{<}\bigcirc x,\bigcirc y,\bigcirc z,\bigcirc zz{>}$
  by (*auto simp*: *nexts-def*)

lemma *next-forall*: $|^{\sim}\ \bigcirc(\forall\ x.\ P\ x) = (\forall\ x.\ \bigcirc\ P\ x)$
  by (*auto simp*: *nexts-def*)

lemma *next-exists*: $|^{\sim}\ \bigcirc(\exists\ x.\ P\ x) = (\exists\ x.\ \bigcirc\ P\ x)$
  by (*auto simp*: *nexts-def*)

lemma *next-exists1*: $|^{\sim}\ \bigcirc(\exists!\ x.\ P\ x) = (\exists!\ x.\ \bigcirc\ P\ x)$
  by (*auto simp*: *nexts-def*)

Rewrite rules to push the "next" operator inward over connectives. (Note that axiom *pax1* and theorem *next-const* are anyway active as rewrite rules.)

lemmas *next-commutes*[*int-rewrite*] =
  next-and next-or next-imp next-eq

next-fun1 next-fun2 next-fun3 next-fun4
next-forall next-exists next-exists1

lemmas *ifs-eq*[*int-rewrite*] = *after-fun3 next-fun3 before-fun3*

lemmas *next-always* = *pax3*

lemma *t1*: $|^{\sim}$ $\bigcirc\$x = x\$$
by (*simp add*: *before-def after-def nexts-def first-tail-second*)

Theorem *next-eventually* should not be used "blindly".

lemma *next-eventually*:
assumes *h*: *stutinv F*
shows $|^{\sim}$ $\Diamond F \longrightarrow \neg F \longrightarrow \bigcirc\Diamond F$
proof −
from *h* **have** *1*: *stutinv* (*TEMP* $\neg F$) **by** (*rule stut-not*)
have $|^{\sim}$ $\Box\neg F = (\neg F \wedge \bigcirc\Box\neg F)$ **unfolding** *T7*[*OF pre-id-unch*[*OF 1*], *int-rewrite*]
by *simp*
thus *?thesis* **by** (*auto simp*: *eventually-def*)
qed

lemma *next-action*: $|^{\sim}$ $\Box[P]$-$v \longrightarrow \bigcirc\Box[P]$-$v$
using *pax4*[*of P v*] **by** *auto*

5.7 Higher Level Derived Rules

In most verification tasks the low-level rules discussed above are not used directly. Here, we derive some higher-level rules more suitable for verification. In particular, variants of Lamport's rules *TLA1*, *TLA2*, *INV1* and *INV2* are derived, where $|^{\sim}$ is used where appropriate.

theorem *TLA1*:
assumes *H*: $|^{\sim}$ $P \wedge$ *Unchanged f* $\longrightarrow \bigcirc P$
shows $\vdash \Box P = (P \wedge \Box[P \longrightarrow \bigcirc P]$-$f)$
proof (*rule int-iffI*)
from *ax1*[*of P*] *M0*[*of P f*] **show** $\vdash \Box P \longrightarrow P \wedge \Box[P \longrightarrow \bigcirc P]$-$f$ **by** *force*
next
from *ax3*[*OF H*] **show** $\vdash P \wedge \Box[P \longrightarrow \bigcirc P]$-$f \longrightarrow \Box P$ **by** *auto*
qed

theorem *TLA2*:
assumes *h1*: $\vdash P \longrightarrow Q$
and *h2*: $|^{\sim}$ $P \wedge \bigcirc P \wedge [A]$-$f \longrightarrow [B]$-$g$
shows $\vdash \Box P \wedge \Box[A]$-$f \longrightarrow \Box Q \wedge \Box[B]$-$g$
proof −
from *h2* **have** $\vdash \Box[P \wedge \bigcirc P \wedge [A]$-$f]$-$g \longrightarrow \Box[[B]$-$g]$-$g$ **by** (*rule M2*)
hence $\vdash \Box[P \wedge \bigcirc P]$-$g \wedge \Box[[A]$-$f]$-$g \longrightarrow \Box[B]$-$g$ **by** (*auto simp add*: *M8*[*int-rewrite*])
with *M1*[*of P g*] *T4*[*of A f g*] **have** $\vdash \Box P \wedge \Box[A]$-$f \longrightarrow \Box[B]$-$g$ **by** *force*
with *h1*[*THEN STL4*] **show** *?thesis* **by** *force*

qed

theorem *INV1*:
  assumes *H*: $|^{\sim}$ $I \wedge [N]\text{-}f \longrightarrow \bigcirc I$
  shows $\vdash I \wedge \Box[N]\text{-}f \longrightarrow \Box I$
proof −
  from *H* **have** $|^{\sim}$ $[N]\text{-}f \longrightarrow I \longrightarrow \bigcirc I$ **by** *auto*
  hence $\vdash \Box[[N]\text{-}f]\text{-}f \longrightarrow \Box[I \longrightarrow \bigcirc I]\text{-}f$ **by** (*rule M2*)
  moreover
  from *H* **have** $|^{\sim}$ $I \wedge$ *Unchanged f* $\longrightarrow \bigcirc I$ **by** (*auto simp*: *actrans-def*)
  hence $\vdash \Box[I \longrightarrow \bigcirc I]\text{-}f \longrightarrow I \longrightarrow \Box I$ **by** (*rule ax3*)
  ultimately show *?thesis* **by** *force*
qed

theorem *INV2*: $\vdash \Box I \longrightarrow \Box[N]\text{-}f = \Box[N \wedge I \wedge \bigcirc I]\text{-}f$
proof −
  from *M1*[*of I f*] **have** $\vdash \Box I \longrightarrow (\Box[N]\text{-}f = \Box[N]\text{-}f \wedge \Box[I \wedge \bigcirc I]\text{-}f)$ **by** *auto*
  thus *?thesis* **by** (*auto simp*: *M8*[*int-rewrite*])
qed

lemma *R1*:
  assumes *H*: $|^{\sim}$ *Unchanged w* $\longrightarrow$ *Unchanged v*
  shows $\vdash \Box[F]\text{-}w \longrightarrow \Box[F]\text{-}v$
proof −
  from *H* **have** $|^{\sim}$ $[F]\text{-}w \longrightarrow [F]\text{-}v$ **by** (*auto simp*: *actrans-def*)
  thus *?thesis* **by** (*rule M11*)
qed

theorem *invmono*:
  assumes *h1*: $\vdash I \longrightarrow P$
    and *h2*: $|^{\sim}$ $P \wedge [N]\text{-}f \longrightarrow \bigcirc P$
  shows $\vdash I \wedge \Box[N]\text{-}f \longrightarrow \Box P$
  using *h1 INV1*[*OF h2*] **by** *force*

theorem *preimpsplit*:
  assumes $|^{\sim}$ $I \wedge N \longrightarrow Q$
    and $|^{\sim}$ $I \wedge$ *Unchanged v* $\longrightarrow Q$
  shows $|^{\sim}$ $I \wedge [N]\text{-}v \longrightarrow Q$
  using *assms*[*unlift-rule*] **by** (*auto simp*: *actrans-def*)

theorem *refinement1*:
  assumes *h1*: $\vdash P \longrightarrow Q$
    and *h2*: $|^{\sim}$ $I \wedge \bigcirc I \wedge [A]\text{-}f \longrightarrow [B]\text{-}g$
  shows $\vdash P \wedge \Box I \wedge \Box[A]\text{-}f \longrightarrow Q \wedge \Box[B]\text{-}g$
proof −
  have $\vdash I \longrightarrow$ #*True* **by** *simp*
  from *this h2* **have** $\vdash \Box I \wedge \Box[A]\text{-}f \longrightarrow \Box$#*True* $\wedge \Box[B]\text{-}g$ **by** (*rule TLA2*)
  with *h1* **show** *?thesis* **by** *force*
qed

theorem *inv-join*:
  assumes $\vdash P \longrightarrow \Box Q$ **and** $\vdash P \longrightarrow \Box R$
  shows $\vdash P \longrightarrow \Box(Q \wedge R)$
  using *assms*[*unlift-rule*] **unfolding** *STL5*[*int-rewrite*] **by** *force*

lemma *inv-cases*: $\vdash \Box(A \longrightarrow B) \wedge \Box(\neg A \longrightarrow B) \longrightarrow \Box B$
proof −
  have $\vdash \Box((A \longrightarrow B) \wedge (\neg A \longrightarrow B)) \longrightarrow \Box B$ **by** (*rule STL4*) *auto*
  thus *?thesis* **by** (*simp add*: *STL5*[*int-rewrite*])
qed

end

6 Liveness

theory *Liveness*
imports *Rules*
begin

This theory derives proof rules for liveness properties.

definition *enabled* :: *'a formula ⇒ 'a formula*
where *enabled F* ≡ λ *s*. ∃ *t*. ((*first s*) ## *t*) ⊨ *F*

syntax *-Enabled* :: *lift ⇒ lift* (‹(*Enabled* -)› [*90*] *90*)

translations *-Enabled* ⇌ *CONST enabled*

definition *WeakF* :: (*'a::world*) *formula* ⇒ (*'a,'b*) *stfun* ⇒ *'a formula*
where *WeakF F v* ≡ *TEMP* ◇□*Enabled* ⟨*F*⟩-*v* ⟶ □◇⟨*F*⟩-*v*

definition *StrongF* :: (*'a::world*) *formula* ⇒ (*'a,'b*) *stfun* ⇒ *'a formula*
where *StrongF F v* ≡ *TEMP* □◇*Enabled* ⟨*F*⟩-*v* ⟶ □◇⟨*F*⟩-*v*

Lamport's TLA defines the above notions for actions. In TLA*, (pre-)formulas generalise TLA's actions and the above definition is the natural generalisation of enabledness to pre-formulas. In particular, we have chosen to define *enabled* such that it yields itself a temporal formula, although its value really just depends on the first state of the sequence it is evaluated over. Then, the definitions of weak and strong fairness are exactly as in TLA.

syntax
 -WF :: [*lift,lift*] ⇒ *lift* (‹(*WF*'(-')'-(-))› [*20,1000*] *90*)
 -SF :: [*lift,lift*] ⇒ *lift* (‹(*SF*'(-')'-(-))› [*20,1000*] *90*)
 -WFsp :: [*lift,lift*] ⇒ *lift* (‹(*WF* '(-')'-(-))› [*20,1000*] *90*)
 -SFsp :: [*lift,lift*] ⇒ *lift* (‹(*SF* '(-')'-(-))› [*20,1000*] *90*)

translations

-*WF* $\rightleftharpoons$ *CONST WeakF*
-*SF* $\rightleftharpoons$ *CONST StrongF*
-*WFsp* $\rightharpoonup$ *CONST WeakF*
-*SFsp* $\rightharpoonup$ *CONST StrongF*

6.1 Properties of -*Enabled*

theorem *enabledI*: $\vdash$ *F* $\longrightarrow$ *Enabled F*
proof (*clarsimp*)
  fix *w*
  assume *w* $\models$ *F*
  with *seq-app-first-tail*[*of w*] **have** ((*first w*) ## *tail w*) $\models$ *F* **by** *simp*
  thus *w* $\models$ *Enabled F* **by** (*auto simp*: *enabled-def*)
qed

theorem *enabledE*:
  assumes *s* $\models$ *Enabled F* **and** $\bigwedge u.$ (*first s* ## *u*) $\models$ *F* $\Longrightarrow$ *Q*
  shows *Q*
  using *assms* **unfolding** *enabled-def* **by** *blast*

lemma *enabled-mono*:
  assumes *w* $\models$ *Enabled F* **and** $\vdash$ *F* $\longrightarrow$ *G*
  shows *w* $\models$ *Enabled G*
  using *assms*[*unlifted*] **unfolding** *enabled-def* **by** *blast*

lemma *Enabled-disj1*: $\vdash$ *Enabled F* $\longrightarrow$ *Enabled* (*F* $\vee$ *G*)
  by (*auto simp*: *enabled-def*)

lemma *Enabled-disj2*: $\vdash$ *Enabled F* $\longrightarrow$ *Enabled* (*G* $\vee$ *F*)
  by (*auto simp*: *enabled-def*)

lemma *Enabled-conj1*: $\vdash$ *Enabled* (*F* $\wedge$ *G*) $\longrightarrow$ *Enabled F*
  by (*auto simp*: *enabled-def*)

lemma *Enabled-conj2*: $\vdash$ *Enabled* (*G* $\wedge$ *F*) $\longrightarrow$ *Enabled F*
  by (*auto simp*: *enabled-def*)

lemma *Enabled-disjD*: $\vdash$ *Enabled* (*F* $\vee$ *G*) $\longrightarrow$ *Enabled F* $\vee$ *Enabled G*
  by (*auto simp*: *enabled-def*)

lemma *Enabled-disj*: $\vdash$ *Enabled* (*F* $\vee$ *G*) = (*Enabled F* $\vee$ *Enabled G*)
  by (*auto simp*: *enabled-def*)

lemmas *enabled-disj-rew* = *Enabled-disj*[*int-rewrite*]

lemma *Enabled-ex*: $\vdash$ *Enabled* ($\exists$ *x*. *F x*) = ($\exists$ *x*. *Enabled* (*F x*))
  by (*force simp*: *enabled-def*)

6.2 Fairness Properties

lemma *WF-alt*: $\vdash$ $WF(A)\text{-}v = (\Box\Diamond\neg Enabled\ \langle A\rangle\text{-}v \vee \Box\Diamond\langle A\rangle\text{-}v)$
proof $-$
  have $\vdash$ $WF(A)\text{-}v = (\neg\Diamond\Box\ Enabled\ \langle A\rangle\text{-}v \vee \Box\Diamond\langle A\rangle\text{-}v)$ **by** (*auto simp*: *WeakF-def*)
  thus *?thesis* **by** (*simp add*: *dualization-rew*)
qed

lemma *SF-alt*: $\vdash$ $SF(A)\text{-}v = (\Diamond\Box\neg Enabled\ \langle A\rangle\text{-}v \vee \Box\Diamond\langle A\rangle\text{-}v)$
proof $-$
  have $\vdash$ $SF(A)\text{-}v = (\neg\Box\Diamond\ Enabled\ \langle A\rangle\text{-}v \vee \Box\Diamond\langle A\rangle\text{-}v)$ **by** (*auto simp*: *StrongF-def*)
  thus *?thesis* **by** (*simp add*: *dualization-rew*)
qed

lemma *alwaysWFI*: $\vdash$ $WF(A)\text{-}v \longrightarrow \Box WF(A)\text{-}v$
  unfolding *WF-alt*[*int-rewrite*] **by** (*rule MM6*)

theorem *WF-always*[*simp-unl*]: $\vdash$ $\Box WF(A)\text{-}v = WF(A)\text{-}v$
  by (*rule int-iffI*[*OF ax1 alwaysWFI*])

theorem *WF-eventually*[*simp-unl*]: $\vdash$ $\Diamond WF(A)\text{-}v = WF(A)\text{-}v$
proof $-$
  have *1*: $\vdash$ $\neg WF(A)\text{-}v = (\Diamond\Box Enabled\ \langle A\rangle\text{-}v \wedge \neg\ \Box\Diamond\langle A\rangle\text{-}v)$
    by (*auto simp*: *WeakF-def*)
  have $\vdash$ $\Box\neg WF(A)\text{-}v = \neg WF(A)\text{-}v$
    by (*simp add*: *1*[*int-rewrite*] *STL5*[*int-rewrite*] *dualization-rew*)
  thus *?thesis*
    by (*auto simp*: *eventually-def*)
qed

lemma *alwaysSFI*: $\vdash$ $SF(A)\text{-}v \longrightarrow \Box SF(A)\text{-}v$
proof $-$
  have $\vdash$ $\Box\Diamond\Box\neg Enabled\ \langle A\rangle\text{-}v \vee \Box\Diamond\langle A\rangle\text{-}v \longrightarrow \Box(\Box\Diamond\Box\neg Enabled\ \langle A\rangle\text{-}v \vee \Box\Diamond\langle A\rangle\text{-}v)$
    by (*rule MM6*)
  thus *?thesis* **unfolding** *SF-alt*[*int-rewrite*] **by** *simp*
qed

theorem *SF-always*[*simp-unl*]: $\vdash$ $\Box SF(A)\text{-}v = SF(A)\text{-}v$
  by (*rule int-iffI*[*OF ax1 alwaysSFI*])

theorem *SF-eventually*[*simp-unl*]: $\vdash$ $\Diamond SF(A)\text{-}v = SF(A)\text{-}v$
proof $-$
  have *1*: $\vdash$ $\neg SF(A)\text{-}v = (\Box\Diamond Enabled\ \langle A\rangle\text{-}v \wedge \neg\ \Box\Diamond\langle A\rangle\text{-}v)$
    by (*auto simp*: *StrongF-def*)
  have $\vdash$ $\Box\neg SF(A)\text{-}v = \neg SF(A)\text{-}v$
    by (*simp add*: *1*[*int-rewrite*] *STL5*[*int-rewrite*] *dualization-rew*)
  thus *?thesis*
    by (*auto simp*: *eventually-def*)
qed

theorem *SF-imp-WF*: $\vdash$ *SF* (A)-*v* $\longrightarrow$ *WF* (A)-*v*
unfolding *WeakF-def StrongF-def* **by** (*auto dest*: *E20*[*unlift-rule*])

lemma *enabled-WFSF*: $\vdash$ $\Box$*Enabled* $\langle F\rangle$-*v* $\longrightarrow$ (*WF*(F)-*v* = *SF*(F)-*v*)
proof −
have $\vdash$ $\Box$*Enabled* $\langle F\rangle$-*v* $\longrightarrow$ $\Diamond\Box$*Enabled* $\langle F\rangle$-*v* **by** (*rule E3*)
hence $\vdash$ $\Box$*Enabled* $\langle F\rangle$-*v* $\longrightarrow$ *WF*(F)-*v* $\longrightarrow$ *SF*(F)-*v* **by** (*auto simp*: *WeakF-def StrongF-def*)
moreover
have $\vdash$ $\Box$*Enabled* $\langle F\rangle$-*v* $\longrightarrow$ $\Box\Diamond$*Enabled* $\langle F\rangle$-*v* **by** (*rule STL4*[*OF E3*])
hence $\vdash$ $\Box$*Enabled* $\langle F\rangle$-*v* $\longrightarrow$ *SF*(F)-*v* $\longrightarrow$ *WF*(F)-*v* **by** (*auto simp*: *WeakF-def StrongF-def*)
ultimately show *?thesis* **by** *force*
qed

theorem *WF1-general*:
assumes *h1*: $|\!\sim$ $P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
and *h2*: $|\!\sim$ $P \wedge N \wedge \langle A\rangle$-*v* $\longrightarrow \bigcirc Q$
and *h3*: $\vdash$ $P \wedge N \longrightarrow$ *Enabled* $\langle A\rangle$-*v*
and *h4*: $|\!\sim$ $P \wedge$ *Unchanged w* $\longrightarrow \bigcirc P$
shows $\vdash$ $\Box N \wedge$ *WF*(A)-*v* $\longrightarrow (P \leadsto Q)$
proof −
have $\vdash$ $\Box(\Box N \wedge$ *WF*(A)-*v*$) \longrightarrow \Box(\Box P \longrightarrow \Diamond\langle A\rangle$-*v*$)$
proof (*rule STL4*)
have $\vdash$ $\Box(P \wedge N) \longrightarrow \Diamond\Box$*Enabled* $\langle A\rangle$-*v* **by** (*rule lift-imp-trans*[*OF h3*[*THEN STL4*] *E3*])
hence $\vdash$ $\Box P \wedge \Box N \wedge$ *WF*(A)-*v* $\longrightarrow \Box\Diamond\langle A\rangle$-*v* **by** (*auto simp*: *WeakF-def STL5*[*int-rewrite*])
with *ax1*[*of TEMP* $\Diamond\langle A\rangle$-*v*] **show** $\vdash$ $\Box N \wedge$ *WF*(A)-*v* $\longrightarrow \Box P \longrightarrow \Diamond\langle A\rangle$-*v* **by** *force*
qed
hence $\vdash$ $\Box N \wedge$ *WF*(A)-*v* $\longrightarrow \Box(\Box P \longrightarrow \Diamond\langle A\rangle$-*v*$)$
by (*simp add*: *STL5*[*int-rewrite*])
with *AA22*[*OF h1 h2 h4*] **show** *?thesis* **by** *force*
qed

Lamport's version of the rule is derived as a special case.

theorem *WF1*:
assumes *h1*: $|\!\sim$ $P \wedge [N]$-*v* $\longrightarrow \bigcirc P \vee \bigcirc Q$
and *h2*: $|\!\sim$ $P \wedge \langle N \wedge A\rangle$-*v* $\longrightarrow \bigcirc Q$
and *h3*: $\vdash$ $P \longrightarrow$ *Enabled* $\langle A\rangle$-*v*
and *h4*: $|\!\sim$ $P \wedge$ *Unchanged v* $\longrightarrow \bigcirc P$
shows $\vdash$ $\Box[N]$-*v* $\wedge$ *WF*(A)-*v* $\longrightarrow (P \leadsto Q)$
proof −
have $\vdash$ $\Box\Box[N]$-*v* $\wedge$ *WF*(A)-*v* $\longrightarrow (P \leadsto Q)$
proof (*rule WF1-general*)
from *h1 T9*[*of N v*] **show** $|\!\sim$ $P \wedge \Box[N]$-*v* $\longrightarrow \bigcirc P \vee \bigcirc Q$ **by** *force*
next
from *T9*[*of N v*] **have** $|\!\sim$ $P \wedge \Box[N]$-*v* $\wedge \langle A\rangle$-*v* $\longrightarrow P \wedge \langle N \wedge A\rangle$-*v*

by (*auto simp*: *actrans-def angle-actrans-def*)
from *this h2* **show** $|\sim\ P \wedge \Box[N]\text{-}v \wedge \langle A\rangle\text{-}v \longrightarrow \bigcirc Q$ **by** (*rule pref-imp-trans*)
next
from *h3 T9*[*of N v*] **show** $\vdash P \wedge \Box[N]\text{-}v \longrightarrow$ *Enabled* $\langle A\rangle\text{-}v$ **by** *force*
qed (*rule h4*)
thus *?thesis* **by** *simp*
qed

The corresponding rule for strong fairness has an additional hypothesis $\Box F$, which is typically a conjunction of other fairness properties used to prove that the helpful action eventually becomes enabled.

theorem *SF1-general*:
assumes *h1*: $|\sim\ P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
and *h2*: $|\sim\ P \wedge N \wedge \langle A\rangle\text{-}v \longrightarrow \bigcirc Q$
and *h3*: $\vdash \Box P \wedge \Box N \wedge \Box F \longrightarrow \Diamond$*Enabled* $\langle A\rangle\text{-}v$
and *h4*: $|\sim\ P \wedge$ *Unchanged w* $\longrightarrow \bigcirc P$
shows $\vdash \Box N \wedge SF(A)\text{-}v \wedge \Box F \longrightarrow (P \leadsto Q)$
proof –
have $\vdash \Box(\Box N \wedge SF(A)\text{-}v \wedge \Box F) \longrightarrow \Box(\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v)$
proof (*rule STL4*)
have $\vdash \Box(\Box P \wedge \Box N \wedge \Box F) \longrightarrow \Box\Diamond$*Enabled* $\langle A\rangle\text{-}v$ **by** (*rule STL4*[*OF h3*])
hence $\vdash \Box P \wedge \Box N \wedge \Box F \wedge SF(A)\text{-}v \longrightarrow \Box\Diamond\langle A\rangle\text{-}v$ **by** (*auto simp*: *StrongF-def STL5*[*int-rewrite*])
with *ax1*[*of TEMP* $\Diamond\langle A\rangle\text{-}v$] **show** $\vdash \Box N \wedge SF(A)\text{-}v \wedge \Box F \longrightarrow \Box P \longrightarrow \Diamond\langle A\rangle\text{-}v$ **by** *force*
qed
hence $\vdash \Box N \wedge SF(A)\text{-}v \wedge \Box F \longrightarrow \Box(\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v)$
by (*simp add*: *STL5*[*int-rewrite*])
with *AA22*[*OF h1 h2 h4*] **show** *?thesis* **by** *force*
qed

theorem *SF1*:
assumes *h1*: $|\sim\ P \wedge [N]\text{-}v \longrightarrow \bigcirc P \vee \bigcirc Q$
and *h2*: $|\sim\ P \wedge \langle N \wedge A\rangle\text{-}v \longrightarrow \bigcirc Q$
and *h3*: $\vdash \Box P \wedge \Box[N]\text{-}v \wedge \Box F \longrightarrow \Diamond$*Enabled* $\langle A\rangle\text{-}v$
and *h4*: $|\sim\ P \wedge$ *Unchanged v* $\longrightarrow \bigcirc P$
shows $\vdash \Box[N]\text{-}v \wedge SF(A)\text{-}v \wedge \Box F \longrightarrow (P \leadsto Q)$
proof –
have $\vdash \Box\Box[N]\text{-}v \wedge SF(A)\text{-}v \wedge \Box F \longrightarrow (P \leadsto Q)$
proof (*rule SF1-general*)
from *h1 T9*[*of N v*] **show** $|\sim\ P \wedge \Box[N]\text{-}v \longrightarrow \bigcirc P \vee \bigcirc Q$ **by** *force*
next
from *T9*[*of N v*] **have** $|\sim\ P \wedge \Box[N]\text{-}v \wedge \langle A\rangle\text{-}v \longrightarrow P \wedge \langle N \wedge A\rangle\text{-}v$
by (*auto simp*: *actrans-def angle-actrans-def*)
from *this h2* **show** $|\sim\ P \wedge \Box[N]\text{-}v \wedge \langle A\rangle\text{-}v \longrightarrow \bigcirc Q$ **by** (*rule pref-imp-trans*)
next
from *h3* **show** $\vdash \Box P \wedge \Box\Box[N]\text{-}v \wedge \Box F \longrightarrow \Diamond$*Enabled* $\langle A\rangle\text{-}v$ **by** *simp*
qed (*rule h4*)
thus *?thesis* **by** *simp*

qed

Lamport proposes the following rule as an introduction rule for *WF* formulas.

theorem *WF2*:
  assumes *h1*: $|\sim\ \langle N \wedge B\rangle\text{-}f \longrightarrow \langle M\rangle\text{-}g$
    and *h2*: $|\sim\ P \wedge \bigcirc P \wedge \langle N \wedge A\rangle\text{-}f \longrightarrow B$
    and *h3*: $\vdash P \wedge \mathit{Enabled}\ \langle M\rangle\text{-}g \longrightarrow \mathit{Enabled}\ \langle A\rangle\text{-}f$
    and *h4*: $\vdash \Box[N \wedge \neg B]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F \wedge \Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \longrightarrow \Diamond\Box P$
  shows $\vdash \Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F \longrightarrow \mathit{WF}(M)\text{-}g$
proof –
 have $\vdash \Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F \wedge \Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \neg\Box\Diamond\langle M\rangle\text{-}g \longrightarrow \Box\Diamond\langle M\rangle\text{-}g$
  proof –
   have *1*: $\vdash \Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F \wedge \Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \neg\Box\Diamond\langle M\rangle\text{-}g \longrightarrow$
$\Diamond\Box P$
   proof –
    have *A*: $\vdash \Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F \wedge \Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \neg\Box\Diamond\langle M\rangle\text{-}g \longrightarrow$
      $\Box(\Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F) \wedge \Diamond\Box(\Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g)$
     unfolding *STL6*[*int-rewrite*]
     by (*auto simp*: *STL5*[*int-rewrite*] *dualization-rew*)
    have *B*: $\vdash \Box(\Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F) \wedge \Diamond\Box(\Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g)$
$\longrightarrow$
      $\Diamond((\Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F) \wedge \Box(\Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g))$
     by (*rule SE2*)
    from *lift-imp-trans*[*OF A B*]
    have $\vdash \Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F \wedge \Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \neg\Box\Diamond\langle M\rangle\text{-}g \longrightarrow$
     $\Diamond((\Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F) \wedge (\Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g))$
     by (*simp add*: *STL5*[*int-rewrite*])
    moreover
    from *h1* **have** $|\sim\ [N]\text{-}f \longrightarrow [\neg M]\text{-}g \longrightarrow [N \wedge \neg B]\text{-}f$ **by** (*auto simp*: *actrans-def*
angle-actrans-def)
    hence $\vdash \Box[[N]\text{-}f]\text{-}f \longrightarrow \Box[[\neg M]\text{-}g \longrightarrow [N \wedge \neg B]\text{-}f]\text{-}f$ **by** (*rule M2*)
    from *lift-imp-trans*[*OF this ax4*] **have** $\vdash \Box[N]\text{-}f \wedge \Box[\neg M]\text{-}g \longrightarrow \Box[N \wedge \neg B]\text{-}f$
     by (*force intro*: *T4*[*unlift-rule*])
    with *h4* **have** $\vdash (\Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F) \wedge (\Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g)$
$\longrightarrow \Diamond\Box P$
     by *force*
    from *STL4-eve*[*OF this*]
    have $\vdash \Diamond((\Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Box F) \wedge (\Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g)) \longrightarrow$
$\Diamond\Box P$ **by** *simp*
    ultimately
    show *?thesis* **by** (*rule lift-imp-trans*)
   qed
   have *2*: $\vdash \Box[N]\text{-}f \wedge \mathit{WF}(A)\text{-}f \wedge \Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Diamond\Box P \longrightarrow \Box\Diamond\langle M\rangle\text{-}g$
   proof –
    have *A*: $\vdash \Diamond\Box P \wedge \Diamond\Box \mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \mathit{WF}(A)\text{-}f \longrightarrow \Box\Diamond\langle A\rangle\text{-}f$
     using *h3*[*THEN STL4*, *THEN STL4-eve*] **by** (*auto simp*: *STL6*[*int-rewrite*]
WeakF-def)
    have *B*: $\vdash \Box[N]\text{-}f \wedge \Diamond\Box P \wedge \Box\Diamond\langle A\rangle\text{-}f \longrightarrow \Box\Diamond\langle M\rangle\text{-}g$

proof –
from *M1*[*of P f*] **have** $\vdash \Box P \wedge \Box\Diamond\langle N \wedge A\rangle\text{-}f \longrightarrow \Box\Diamond\langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f$
by (*force intro*: *AA29*[*unlift-rule*])
hence $\vdash \Diamond\Box(\Box P \wedge \Box\Diamond\langle N \wedge A\rangle\text{-}f) \longrightarrow \Diamond\Box\Box\Diamond\langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f$
by (*rule STL4-eve*[*OF STL4*])
hence $\vdash \Diamond\Box P \wedge \Box\Diamond\langle N \wedge A\rangle\text{-}f \longrightarrow \Box\Diamond\langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f$
by (*simp add*: *STL6*[*int-rewrite*])
with *AA29*[*of N f A*]
have *B1*: $\vdash \Box[N]\text{-}f \wedge \Diamond\Box P \wedge \Box\Diamond\langle A\rangle\text{-}f \longrightarrow \Box\Diamond\langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f$
by *force*
from *h2* **have** $|{\sim}\ \langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f \longrightarrow \langle N \wedge B\rangle\text{-}f$
by (*auto simp*: *angle-actrans-sem*[*unlifted*])
from *B1 this*[*THEN AA25*, *THEN STL4*] **have** $\vdash \Box[N]\text{-}f \wedge \Diamond\Box P \wedge \Box\Diamond\langle A\rangle\text{-}f \longrightarrow \Box\Diamond\langle N \wedge B\rangle\text{-}f$
by (*rule lift-imp-trans*)
moreover
have $\vdash \Box\Diamond\langle N \wedge B\rangle\text{-}f \longrightarrow \Box\Diamond\langle M\rangle\text{-}g$ **by** (*rule h1*[*THEN AA25*, *THEN STL4*])
ultimately show *?thesis* **by** (*rule lift-imp-trans*)
qed
from *A B* **show** *?thesis* **by** *force*
qed
from *1 2* **show** *?thesis* **by** *force*
qed
thus *?thesis* **by** (*auto simp*: *WeakF-def*)
qed

Lamport proposes an analogous theorem for introducing strong fairness, and its proof is very similar, in fact, it was obtained by copy and paste, with minimal modifications.

theorem *SF2*:
assumes *h1*: $|{\sim}\ \langle N \wedge B\rangle\text{-}f \longrightarrow \langle M\rangle\text{-}g$
and *h2*: $|{\sim}\ P \wedge \bigcirc P \wedge \langle N \wedge A\rangle\text{-}f \longrightarrow B$
and *h3*: $\vdash P \wedge \mathit{Enabled}\ \langle M\rangle\text{-}g \longrightarrow \mathit{Enabled}\ \langle A\rangle\text{-}f$
and *h4*: $\vdash \Box[N \wedge \neg B]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box\Diamond\mathit{Enabled}\ \langle M\rangle\text{-}g \longrightarrow \Diamond\Box P$
shows $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \longrightarrow SF(M)\text{-}g$
proof –
have $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box\Diamond\mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \neg\Box\Diamond\langle M\rangle\text{-}g \longrightarrow \Box\Diamond\langle M\rangle\text{-}g$
proof –
have *1*: $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box\Diamond\mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \neg\Box\Diamond\langle M\rangle\text{-}g \longrightarrow \Diamond\Box P$
proof –
have *A*: $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box\Diamond\mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \neg\Box\Diamond\langle M\rangle\text{-}g \longrightarrow$
$\Box(\Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F) \wedge \Diamond\Box(\Box\Diamond\mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g)$
unfolding *STL6*[*int-rewrite*]
by (*auto simp*: *STL5*[*int-rewrite*] *dualization-rew*)
have *B*: $\vdash \Box(\Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F) \wedge \Diamond\Box(\Box\Diamond\mathit{Enabled}\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g) \longrightarrow$

$\Diamond((\Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F) \wedge \Box(\Box\Diamond Enabled\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g))$
by (*rule SE2*)
from *lift-imp-trans*[*OF A B*]
have $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box\Diamond Enabled\ \langle M\rangle\text{-}g \wedge \neg\Box\Diamond\langle M\rangle\text{-}g \longrightarrow$
$\Diamond((\Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F) \wedge (\Box\Diamond Enabled\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g))$
by (*simp add*: *STL5*[*int-rewrite*])
moreover
from *h1* **have** $|\sim\ [N]\text{-}f \longrightarrow [\neg M]\text{-}g \longrightarrow [N \wedge \neg B]\text{-}f$ **by** (*auto simp*: *actrans-def angle-actrans-def*)
hence $\vdash \Box[[N]\text{-}f]\text{-}f \longrightarrow \Box[[\neg M]\text{-}g \longrightarrow [N \wedge \neg B]\text{-}f]\text{-}f$ **by** (*rule M2*)
from *lift-imp-trans*[*OF this ax4*] **have** $\vdash \Box[N]\text{-}f \wedge \Box[\neg M]\text{-}g \longrightarrow \Box[N \wedge \neg B]\text{-}f$
by (*force intro*: *T4*[*unlift-rule*])
with *h4* **have** $\vdash (\Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F) \wedge (\Box\Diamond Enabled\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g) \longrightarrow \Diamond\Box P$
by *force*
from *STL4-eve*[*OF this*]
have $\vdash \Diamond((\Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F) \wedge (\Box\Diamond Enabled\ \langle M\rangle\text{-}g \wedge \Box[\neg M]\text{-}g)) \longrightarrow \Diamond\Box P$ **by** *simp*
ultimately
show *?thesis* **by** (*rule lift-imp-trans*)
qed
have *2*: $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box\Diamond Enabled\ \langle M\rangle\text{-}g \wedge \Diamond\Box P \longrightarrow \Box\Diamond\langle M\rangle\text{-}g$
proof –
have $\vdash \Box\Diamond(P \wedge Enabled\ \langle M\rangle\text{-}g) \wedge SF(A)\text{-}f \longrightarrow \Box\Diamond\langle A\rangle\text{-}f$
using *h3*[*THEN STL4-eve*, *THEN STL4*] **by** (*auto simp*: *StrongF-def*)
with *E28* **have** *A*: $\vdash \Diamond\Box P \wedge \Box\Diamond Enabled\ \langle M\rangle\text{-}g \wedge SF(A)\text{-}f \longrightarrow \Box\Diamond\langle A\rangle\text{-}f$
by *force*
have *B*: $\vdash \Box[N]\text{-}f \wedge \Diamond\Box P \wedge \Box\Diamond\langle A\rangle\text{-}f \longrightarrow \Box\Diamond\langle M\rangle\text{-}g$
proof –
from *M1*[*of P f*] **have** $\vdash \Box P \wedge \Box\Diamond\langle N \wedge A\rangle\text{-}f \longrightarrow \Box\Diamond\langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f$
by (*force intro*: *AA29*[*unlift-rule*])
hence $\vdash \Diamond\Box(\Box P \wedge \Box\Diamond\langle N \wedge A\rangle\text{-}f) \longrightarrow \Diamond\Box\Box\Diamond\langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f$
by (*rule STL4-eve*[*OF STL4*])
hence $\vdash \Diamond\Box P \wedge \Box\Diamond\langle N \wedge A\rangle\text{-}f \longrightarrow \Box\Diamond\langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f$
by (*simp add*: *STL6*[*int-rewrite*])
with *AA29*[*of N f A*]
have *B1*: $\vdash \Box[N]\text{-}f \wedge \Diamond\Box P \wedge \Box\Diamond\langle A\rangle\text{-}f \longrightarrow \Box\Diamond\langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f$ **by** *force*
from *h2* **have** $|\sim\ \langle(P \wedge \bigcirc P) \wedge (N \wedge A)\rangle\text{-}f \longrightarrow \langle N \wedge B\rangle\text{-}f$
by (*auto simp*: *angle-actrans-sem*[*unlifted*])
from *B1 this*[*THEN AA25*, *THEN STL4*] **have** $\vdash \Box[N]\text{-}f \wedge \Diamond\Box P \wedge \Box\Diamond\langle A\rangle\text{-}f \longrightarrow \Box\Diamond\langle N \wedge B\rangle\text{-}f$
by (*rule lift-imp-trans*)
moreover
have $\vdash \Box\Diamond\langle N \wedge B\rangle\text{-}f \longrightarrow \Box\Diamond\langle M\rangle\text{-}g$ **by** (*rule h1*[*THEN AA25*, *THEN STL4*])
ultimately show *?thesis* **by** (*rule lift-imp-trans*)
qed

```
    from A B show ?thesis by force
  qed
  from 1 2 show ?thesis by force
 qed
 thus ?thesis by (auto simp: StrongF-def)
qed
```

This is the lattice rule from TLA

```
theorem wf-leadsto:
 assumes h1: wf r
    and h2: ⋀x. ⊢ F x ⇝ (G ∨ (∃ y. #((y,x) ∈ r) ∧ F y))
 shows      ⊢ F x ⇝ G
using h1
proof (rule wf-induct)
 fix x
 assume ih: ∀ y. (y, x) ∈ r ⟶ (⊢ F y ⇝ G)
 show ⊢ F x ⇝ G
 proof –
  from ih have ⊢ (∃ y. #((y,x) ∈ r) ∧ F y) ⇝ G
   by (force simp: LT21[int-rewrite] LT33[int-rewrite])
  with h2 show ?thesis by (force intro: LT19[unlift-rule])
 qed
qed
```

6.3 Stuttering Invariance

```
theorem stut-Enabled: STUTINV Enabled ⟨F⟩-v
 by (auto simp: enabled-def stutinv-def dest!: sim-first)

theorem stut-WF: NSTUTINV F ⟹ STUTINV WF(F)-v
 by (auto simp: WeakF-def stut-Enabled bothstutinvs)

theorem stut-SF: NSTUTINV F ⟹ STUTINV SF(F)-v
 by (auto simp: StrongF-def stut-Enabled bothstutinvs)

lemmas livestutinv = stut-WF stut-SF stut-Enabled

end
```

7 Representing state in TLA*

```
theory State
imports Liveness
begin
```

We adopt the hidden state appraoch, as used in the existing Isabelle/HOL TLA embedding [7]. This approach is also used in [3]. Here, a state space is defined by its projections, and everything else is unknown. Thus, a variable is a projection of the state space, and has the same type as a state function.

Moreover, strong typing is achieved, since the projection function may have any result type. To achieve this, the state space is represented by an undefined type, which is an instance of the *world* class to enable use with the *Intensional* theory.

typedecl *state*

instance *state* :: *world* **..**

type-synonym *'a statefun* = (*state*,*'a*) *stfun*
type-synonym *statepred* = *bool statefun*
type-synonym *'a tempfun* = (*state*,*'a*) *formfun*
type-synonym *temporal* = *state formula*

Formalizing type state would require formulas to be tagged with their underlying state space and would result in a system that is much harder to use. (Unlike Hoare logic or Unity, TLA has quantification over state variables, and therefore one usually works with different state spaces within a single specification.) Instead, state is just an anonymous type whose only purpose is to provide Skolem constants. Moreover, we do not define a type of state variables separate from that of arbitrary state functions, again in order to simplify the definition of flexible quantification later on. Nevertheless, we need to distinguish state variables, mainly to define the enabledness of actions. The user identifies (tuples of) "base" state variables in a specification via the "meta predicate" *basevars*, which is defined here.

definition *stvars* :: *'a statefun* ⇒ *bool*
where *basevars-def*: *stvars* ≡ *surj*

syntax
PRED :: *lift* ⇒ *'a* (‹*PRED* -›)
-stvars :: *lift* ⇒ *bool* (‹*basevars* -›)

translations
PRED P ⇀ (*P*::*state* => -)
-stvars ⇌ *CONST stvars*

Base variables may be assigned arbitrary (type-correct) values. In the following lemma, note that *vs* may be a tuple of variables. The correct identification of base variables is up to the user who must take care not to introduce an inconsistency. For example, *basevars* (x, x) would definitely be inconsistent.

lemma *basevars*: *basevars vs* ⟹ ∃ *u*. *vs u* = *c*
proof (*unfold basevars-def surj-def*)
assume ∀ *y*. ∃ *x*. *y* = *vs x*
then obtain *x* **where** *c* = *vs x* **by** *blast*
thus ∃ *u*. *vs u* = *c* **by** *blast*
qed

lemma *baseE*:
  assumes *H1*: *basevars v* **and** *H2*:$\bigwedge x.\ v\ x = c \Longrightarrow Q$
  shows *Q*
  using *H1*[*THEN basevars*] *H2* **by** *auto*

A variant written for sequences rather than single states.

lemma *first-baseE*:
  assumes *H1*: *basevars v* **and** *H2*: $\bigwedge x.\ v\ (\mathit{first}\ x) = c \Longrightarrow Q$
  shows *Q*
  using *H1*[*THEN basevars*] *H2* **by** (*force simp*: *first-def*)

lemma *base-pair1*:
  assumes *h*: *basevars* (*x*,*y*)
  shows *basevars x*
proof (*auto simp*: *basevars-def*)
  fix *c*
  from *h*[*THEN basevars*] **obtain** *s* **where** (*LIFT* (*x*,*y*)) *s* = (*c*, *arbitrary*) **by** *auto*
  thus $c \in \mathit{range}\ x$ **by** *auto*
qed

lemma *base-pair2*:
  assumes *h*: *basevars* (*x*,*y*)
  shows *basevars y*
proof (*auto simp*: *basevars-def*)
  fix *d*
  from *h*[*THEN basevars*] **obtain** *s* **where** (*LIFT* (*x*,*y*)) *s* = (*arbitrary*, *d*) **by** *auto*
  thus $d \in \mathit{range}\ y$ **by** *auto*
qed

lemma *base-pair*: *basevars* (*x*,*y*) $\Longrightarrow$ *basevars x* $\wedge$ *basevars y*
  by (*auto elim*: *base-pair1 base-pair2*)

Since the *unit* type has just one value, any state function of unit type satisfies the predicate *basevars*. The following theorem can sometimes be useful because it gives a trivial solution for *basevars* premises.

lemma *unit-base*: *basevars* (v::*state* $\Rightarrow$ *unit*)
  by (*auto simp*: *basevars-def*)

A pair of the form (*x*,*x*) will generally not satisfy the predicate *basevars* – except for pathological cases such as *x*::*unit*.

lemma
  fixes *x* :: *state* $\Rightarrow$ *bool*
  assumes *h1*: *basevars* (*x*,*x*)
  shows *False*
proof –

from *h1* **have** $\exists\, u.\ (LIFT\ (x,x))\ u = (False,True)$ **by** (*rule basevars*)
thus *False* **by** *auto*
qed

lemma
fixes $x :: state \Rightarrow nat$
assumes *h1*: *basevars* (x,x)
shows *False*
proof –
from *h1* **have** $\exists\, u.\ (LIFT\ (x,x))\ u = (0,1)$ **by** (*rule basevars*)
thus *False* **by** *auto*
qed

The following theorem reduces the reasoning about the existence of a state sequence satisfiyng an enabledness predicate to finding a suitable value c at the successor state for the base variables of the specification. This rule is intended for reasoning about standard TLA specifications, where *Enabled* is applied to actions, not arbitrary pre-formulas.

lemma *base-enabled*:
assumes *h1*: *basevars vs*
and *h2*: $\bigwedge u.\ vs\ (first\ u) = c \Longrightarrow ((first\ s)\ \#\#\ u) \models F$
shows $s \models Enabled\ F$
using *h1* **proof** (*rule first-baseE*)
fix *t*
assume $vs\ (first\ t) = c$
hence $((first\ s)\ \#\#\ t) \models F$ **by** (*rule h2*)
thus $s \models Enabled\ F$ **unfolding** *enabled-def* **by** *blast*
qed

7.1 Temporal Quantifiers

In [5], Lamport gives a stuttering invariant definition of quantification over (flexible) variables. It relies on similarity of two sequences (as supported in our *TLA.Sequence* theory), and equivalence of two sequences up to a variable (the bound variable). However, sequence equaivalence up to a variable, requires state equaivalence up to a variable. Our state representation above does not support this, hence we cannot encode Lamport's definition in our theory. Thus, we need to axiomatise quantification over (flexible) variables. Note that with a state representation supporting this, our theory should allow such an encoding.

consts
EEx :: ('a statefun ⇒ temporal) ⇒ temporal (**binder** ‹*Eex* › *10*)
AAll :: ('a statefun ⇒ temporal) ⇒ temporal (**binder** ‹*Aall* › *10*)

syntax
-EEx :: [*idts*, *lift*] => *lift* (‹(*3*∃∃ -./ -)› [*0*,*10*] *10*)
-AAll :: [*idts*, *lift*] => *lift* (‹(*3*∀∀ -./ -)› [*0*,*10*] *10*)

translations
  -EEx v A == *Eex v. A*
  -AAll v A == *Aall v. A*

axiomatization where
  eexI: $\vdash F\ x \longrightarrow (\exists\exists\ x.\ F\ x)$
and *eexE*: $[\![s \models (\exists\exists\ x.\ F\ x)$; *basevars vs*; (!! x. $[\![$ *basevars* (x,vs); $s \models F\ x\ ]\!] \Longrightarrow s \models G)]\!]$
    $\Longrightarrow (s \models G)$
and *all-def*: $\vdash (\forall\forall\ x.\ F\ x) = (\neg(\exists\exists\ x.\ \neg(F\ x)))$
and *eexSTUT*: $STUTINV\ F\ x \Longrightarrow STUTINV\ (\exists\exists\ x.\ F\ x)$
and *history*: $\vdash (I \wedge \Box[A]\text{-}v) = (\exists\exists\ h.\ (\$h = ha) \wedge I \wedge \Box[A \wedge h\$=hb]\text{-}(h,v))$

lemmas *eexI-unl* = *eexI*[*unlift-rule*] — $w \models F\ x \Longrightarrow w \models (\exists\exists\ x.\ F\ x)$

tla-defs can be used to unfold TLA definitions into lowest predicate level. This is particularly useful for reasoning about enabledness of formulas.

lemmas *tla-defs* = *unch-def before-def after-def first-def second-def suffix-def tail-def nexts-def app-def angle-actrans-def actrans-def*

end

8 A simple illustrative example

theory *Even*
imports *State*
begin

A trivial example illustrating invariant proofs in the logic, and how Isabelle/HOL can help with specification. It proves that x is always even in a program where x is initialized as 0 and always incremented by 2.

inductive-set
  Even :: *nat set*
where
  even-zero: *0* ∈ *Even*
| *even-step*: $n \in Even \Longrightarrow Suc\ (Suc\ n) \in Even$

locale *Program* =
  fixes x :: *state* ⇒ *nat*
  and *init* :: *temporal*
  and *act* :: *temporal*
  and *phi* :: *temporal*
  defines *init* ≡ *TEMP* $\$x = \#\ 0$
  and *act* ≡ *TEMP* $x` = Suc<Suc<\$x>>$
  and *phi* ≡ *TEMP init* ∧ $\Box[act]\text{-}x$

lemma (**in** *Program*) *stutinvprog*: *STUTINV phi*
  by (*auto simp*: *phi-def init-def act-def stutinvs nstutinvs*)

lemma (**in** *Program*) *inveven*: $\vdash$ *phi* $\longrightarrow$ $\Box$($\$x \in \#$ *Even*)
  unfolding *phi-def*
proof (*rule invmono*)
  show $\vdash$ *init* $\longrightarrow$ $\$x \in \#Even$
    by (*auto simp*: *init-def even-zero*)
next
  show $|^\sim$ $\$x \in \#Even \wedge [act]$-$x \longrightarrow \bigcirc(\$x \in \#Even)$
    by (*auto simp*: *act-def even-step tla-defs*)
qed

end

9 Lamport's Inc example

theory *Inc*
imports *State*
begin

This example illustrates use of the embedding by mechanising the running example of Lamports original TLA paper [5].

datatype *pcount* = *a* | *b* | *g*

locale *Firstprogram* =
  fixes *x* :: *state* $\Rightarrow$ *nat*
  and *y* :: *state* $\Rightarrow$ *nat*
  and *init* :: *temporal*
  and *m1* :: *temporal*
  and *m2* :: *temporal*
  and *phi* :: *temporal*
  and *Live* :: *temporal*
  defines *init* $\equiv$ *TEMP* $\$x = \#\ 0 \wedge \$y = \#\ 0$
  and *m1* $\equiv$ *TEMP* $x` = Suc<\$x> \wedge y` = \$y$
  and *m2* $\equiv$ *TEMP* $y` = Suc<\$y> \wedge x` = \$x$
  and *Live* $\equiv$ *TEMP WF*(*m1*)-(*x*,*y*) $\wedge$ *WF*(*m2*)-(*x*,*y*)
  and *phi* $\equiv$ *TEMP* (*init* $\wedge$ $\Box$[*m1* $\vee$ *m2*]-(*x*,*y*) $\wedge$ *Live*)
  assumes *bvar*: *basevars* (*x*,*y*)

lemma (**in** *Firstprogram*) *STUTINV phi*
  by (*auto simp*: *phi-def init-def m1-def m2-def Live-def stutinvs nstutinvs livestutinv*)

lemma (**in** *Firstprogram*) *enabled-m1*: $\vdash$ *Enabled* $\langle$*m1*$\rangle$-(*x*,*y*)
proof (*clarify*)
  fix *s*
  show *s* $\models$ *Enabled* $\langle$*m1*$\rangle$-(*x*,*y*)

by (*rule base-enabled*[*OF bvar*]) (*auto simp*: *m1-def tla-defs*)
qed

lemma (**in** *Firstprogram*) *enabled-m2*: ⊢ *Enabled* ⟨*m2*⟩-(*x*,*y*)
proof (*clarify*)
fix *s*
show *s* ⊨ *Enabled* ⟨*m2*⟩-(*x*,*y*)
by (*rule base-enabled*[*OF bvar*]) (*auto simp*: *m2-def tla-defs*)
qed

locale *Secondprogram* = *Firstprogram* +
fixes *sem* :: *state* ⇒ *nat*
and *pc1* :: *state* ⇒ *pcount*
and *pc2* :: *state* ⇒ *pcount*
and *vars*
and *initPsi* :: *temporal*
and *alpha1* :: *temporal*
and *alpha2* :: *temporal*
and *beta1* :: *temporal*
and *beta2* :: *temporal*
and *gamma1* :: *temporal*
and *gamma2* :: *temporal*
and *n1* :: *temporal*
and *n2* :: *temporal*
and *Live2* :: *temporal*
and *psi* :: *temporal*
and *I* :: *temporal*
defines *vars* ≡ *LIFT* (*x*,*y*,*sem*,*pc1*,*pc2*)
and *initPsi* ≡ *TEMP* $*pc1* = # *a* ∧ $*pc2* = # *a* ∧ $*x* = # *0* ∧ $*y* = # *0* ∧ $*sem* = # *1*
and *alpha1* ≡ *TEMP* $*pc1* =#*a* ∧ # *0* < $*sem* ∧ *pc1*$ = #*b* ∧ *sem*$ = $*sem* − # *1* ∧ *Unchanged* (*x*,*y*,*pc2*)
and *alpha2* ≡ *TEMP* $*pc2* =#*a* ∧ # *0* < $*sem* ∧ *pc2*‘ = #*b* ∧ *sem*$ = $*sem* − # *1* ∧ *Unchanged* (*x*,*y*,*pc1*)
and *beta1* ≡ *TEMP* $*pc1* =#*b* ∧ *pc1*‘ = #*g* ∧ *x*‘ = *Suc*<$*x*> ∧ *Unchanged* (*y*,*sem*,*pc2*)
and *beta2* ≡ *TEMP* $*pc2* =#*b* ∧ *pc2*‘ = #*g* ∧ *y*‘ = *Suc*<$*y*> ∧ *Unchanged* (*x*,*sem*,*pc1*)
and *gamma1* ≡ *TEMP* $*pc1* =#*g* ∧ *pc1*‘ = #*a* ∧ *sem*‘ = *Suc*<$*sem*> ∧ *Unchanged* (*x*,*y*,*pc2*)
and *gamma2* ≡ *TEMP* $*pc2* =#*g* ∧ *pc2*‘ = #*a* ∧ *sem*‘ = *Suc*<$*sem*> ∧ *Unchanged* (*x*,*y*,*pc1*)
and *n1* ≡ *TEMP* (*alpha1* ∨ *beta1* ∨ *gamma1*)
and *n2* ≡ *TEMP* (*alpha2* ∨ *beta2* ∨ *gamma2*)
and *Live2* ≡ *TEMP SF*(*n1*)-*vars* ∧ *SF*(*n2*)-*vars*
and *psi* ≡ *TEMP* (*initPsi* ∧ □[*n1* ∨ *n2*]-*vars* ∧ *Live2*)
and *I* ≡ *TEMP* ($*sem* = # *1* ∧ $*pc1* = #*a* ∧ $*pc2* = # *a*)
∨ ($*sem* = # *0* ∧ (($*pc1* = #*a* ∧ $*pc2* ∈ {#*b* , #*g*})
∨ ($*pc2* = #*a* ∧ $*pc1* ∈ {#*b* , #*g*})))

assumes *bvar2*: *basevars vars*

lemmas (**in** *Secondprogram*) *Sact2-defs* = *n1-def n2-def alpha1-def beta1-def gamma1-def alpha2-def beta2-def gamma2-def*

Proving invariants is the basis of every effort of system verification. We show that *I* is an inductive invariant of specification *psi*.

lemma (**in** *Secondprogram*) *psiI*: $\vdash$ *psi* $\longrightarrow \Box I$
proof −
  have *init*: $\vdash$ *initPsi* $\longrightarrow I$ **by** (*auto simp*: *initPsi-def I-def*)
  have $|^\sim$ $I \wedge$ *Unchanged vars* $\longrightarrow \bigcirc I$ **by** (*auto simp*: *I-def vars-def tla-defs*)
  moreover
  have $|^\sim$ $I \wedge n1 \longrightarrow \bigcirc I$ **by** (*auto simp*: *I-def Sact2-defs tla-defs*)
  moreover
  have $|^\sim$ $I \wedge n2 \longrightarrow \bigcirc I$ **by** (*auto simp*: *I-def Sact2-defs tla-defs*)
 ultimately have *step*: $|^\sim$ $I \wedge [n1 \vee n2]$-*vars* $\longrightarrow \bigcirc I$ **by** (*force simp*: *actrans-def*)
 from *init step* **have** *goal*: $\vdash$ *initPsi* $\wedge \Box[n1 \vee n2]$-*vars* $\longrightarrow \Box I$ **by** (*rule invmono*)
  have $\vdash$ *initPsi* $\wedge \Box[n1 \vee n2]$-*vars* $\wedge$ *Live2* ==> $\vdash$ *initPsi* $\wedge \Box[n1 \vee n2]$-*vars*
   by *auto*
  with *goal* **show** *?thesis* **unfolding** *psi-def* **by** *auto*
qed

Using this invariant we now prove step simulation, i.e. the safety part of the refinement proof.

theorem (**in** *Secondprogram*) *step-simulation*: $\vdash$ *psi* $\longrightarrow$ *init* $\wedge \Box[m1 \vee m2]$-(x,y)
proof −
  have $\vdash$ *initPsi* $\wedge \Box I \wedge \Box[n1 \vee n2]$-*vars* $\longrightarrow$ *init* $\wedge \Box[m1 \vee m2]$-(x,y)
  proof (*rule refinement1*)
    show $\vdash$ *initPsi* $\longrightarrow$ *init* **by** (*auto simp*: *initPsi-def init-def*)
  next
    show $|^\sim$ $I \wedge \bigcirc I \wedge [n1 \vee n2]$-*vars* $\longrightarrow [m1 \vee m2]$-$(x,y)$
      by (*auto simp*: *I-def m1-def m2-def vars-def Sact2-defs tla-defs*)
  qed
  with *psiI* **show** *?thesis* **unfolding** *psi-def* **by** *force*
qed

Liveness proofs require computing the enabledness conditions of actions. The first lemma below shows that all steps are visible, i.e. they change at least one variable.

lemma (**in** *Secondprogram*) *n1-ch*: $|^\sim$ $\langle n1 \rangle$-*vars* = *n1*
proof −
  have $|^\sim$ $n1 \longrightarrow \langle n1 \rangle$-*vars* **by** (*auto simp*: *Sact2-defs tla-defs vars-def*)
  thus *?thesis* **by** (*auto simp*: *angle-actrans-sem*[*int-rewrite*])
qed

lemma (**in** *Secondprogram*) *enab-alpha1*: $\vdash$ \$*pc1* = #*a* $\longrightarrow$ # *0* < \$*sem* $\longrightarrow$ *Enabled alpha1*
proof (*clarsimp simp*: *tla-defs*)

fix *s* :: *state seq*
assume *pc1* (*s 0*) = *a* **and** *0* < *sem* (*s 0*)
thus *s* ⊨ *Enabled alpha1*
by (*intro base-enabled*[*OF bvar2*]) (*auto simp*: *Sact2-defs tla-defs vars-def*)
qed

lemma (**in** *Secondprogram*) *enab-beta1*: ⊢ $pc1 = #b ⟶ *Enabled beta1*
proof (*clarsimp simp*: *tla-defs*)
fix *s* :: *state seq*
assume *pc1* (*s 0*) = *b*
thus *s* ⊨ *Enabled beta1*
by (*intro base-enabled*[*OF bvar2*]) (*auto simp*: *Sact2-defs tla-defs vars-def*)
qed

lemma (**in** *Secondprogram*) *enab-gamma1*: ⊢ $pc1 = #g ⟶ *Enabled gamma1*
proof (*clarsimp simp*: *tla-defs*)
fix *s* :: *state seq*
assume *pc1* (*s 0*) = *g*
thus *s* ⊨ *Enabled gamma1*
by (*intro base-enabled*[*OF bvar2*]) (*auto simp*: *Sact2-defs tla-defs vars-def*)
qed

lemma (**in** *Secondprogram*) *enab-n1*:
⊢ *Enabled* ⟨*n1*⟩-*vars* = ($pc1 = #a ⟶ # 0 < $sem)
unfolding *n1-ch*[*int-rewrite*] **proof** (*rule int-iffI*)
show ⊢ *Enabled n1* ⟶ $pc1 = #a ⟶ # 0 < $sem
by (*auto elim*!: *enabledE simp*: *Sact2-defs tla-defs*)
next
show ⊢ ($pc1 = #a ⟶ # 0 < $sem) ⟶ *Enabled n1*
proof (*clarsimp simp*: *tla-defs*)
fix *s* :: *state seq*
assume *pc1* (*s 0*) = *a* ⟶ *0* < *sem* (*s 0*)
thus *s* ⊨ *Enabled n1*
using *enab-alpha1*[*unlift-rule*]
enab-beta1[*unlift-rule*]
enab-gamma1[*unlift-rule*]
by (*cases pc1* (*s 0*)) (*force simp*: *n1-def Enabled-disj*[*int-rewrite*] *tla-defs*)+
qed
qed

The analogous properties for the second process are obtained by copy and paste.

lemma (**in** *Secondprogram*) *n2-ch*: |~ ⟨*n2*⟩-*vars* = *n2*
proof −
have |~ *n2* ⟶ ⟨*n2*⟩-*vars* **by** (*auto simp*: *Sact2-defs tla-defs vars-def*)
thus *?thesis* **by** (*auto simp*: *angle-actrans-sem*[*int-rewrite*])
qed

lemma (**in** *Secondprogram*) *enab-alpha2*: ⊢ $pc2 = #a ⟶ # 0 < $sem ⟶

Enabled alpha2
proof (*clarsimp simp*: *tla-defs*)
  fix *s* :: *state seq*
  assume $pc2\ (s\ 0) = a$ **and** $0 < sem\ (s\ 0)$
  thus $s \models$ *Enabled alpha2*
    by (*intro base-enabled*[*OF bvar2*]) (*auto simp*: *Sact2-defs tla-defs vars-def*)
qed

lemma (**in** *Secondprogram*) *enab-beta2*: $\vdash \$pc2 = \#b \longrightarrow$ *Enabled beta2*
proof (*clarsimp simp*: *tla-defs*)
  fix *s* :: *state seq*
  assume $pc2\ (s\ 0) = b$
  thus $s \models$ *Enabled beta2*
    by (*intro base-enabled*[*OF bvar2*]) (*auto simp*: *Sact2-defs tla-defs vars-def*)
qed

lemma (**in** *Secondprogram*) *enab-gamma2*: $\vdash \$pc2 = \#g \longrightarrow$ *Enabled gamma2*
proof (*clarsimp simp*: *tla-defs*)
  fix *s* :: *state seq*
  assume $pc2\ (s\ 0) = g$
  thus $s \models$ *Enabled gamma2*
    by (*intro base-enabled*[*OF bvar2*]) (*auto simp*: *Sact2-defs tla-defs vars-def*)
qed

lemma (**in** *Secondprogram*) *enab-n2*:
  $\vdash$ *Enabled* $\langle n2 \rangle$-*vars* $= (\$pc2 = \#a \longrightarrow \#\ 0 < \$sem)$
unfolding *n2-ch*[*int-rewrite*] **proof** (*rule int-iffI*)
  show $\vdash$ *Enabled n2* $\longrightarrow \$pc2 = \#a \longrightarrow \#\ 0 < \$sem$
    by (*auto elim*!: *enabledE simp*: *Sact2-defs tla-defs*)
next
  show $\vdash (\$pc2 = \#a \longrightarrow \#\ 0 < \$sem) \longrightarrow$ *Enabled n2*
  proof (*clarsimp simp*: *tla-defs*)
    fix *s* :: *state seq*
    assume $pc2\ (s\ 0) = a \longrightarrow 0 < sem\ (s\ 0)$
    thus $s \models$ *Enabled n2*
      using *enab-alpha2*[*unlift-rule*]
            enab-beta2[*unlift-rule*]
            enab-gamma2[*unlift-rule*]
      by (*cases pc2* (*s 0*)) (*force simp*: *n2-def Enabled-disj*[*int-rewrite*] *tla-defs*)+
  qed
qed

We use rule *SF2* to prove that *psi* implements strong fairness for the abstract action *m1*. Since strong fairness implies weak fairness, it follows that *psi* refines the liveness condition of *phi*.

lemma (**in** *Secondprogram*) *psi-fair-m1*: $\vdash psi \longrightarrow SF(m1)\text{-}(x,y)$
proof −
  have $\vdash \Box[n1 \lor n2]\text{-}vars \land SF(n1)\text{-}vars \land \Box(I \land SF(n2)\text{-}vars) \longrightarrow SF(m1)\text{-}(x,y)$
  proof (*rule SF2*)

Rule *SF2* requires us to choose a helpful action (whose effect implies $\langle m1\rangle$-(x,y)) and a persistent condition, which will eventually remain true if the helpful action is never executed. In our case, the helpful action is *beta1* and the persistent condition is $pc1 = b$.

show $|\sim \langle(n1 \lor n2) \land beta1\rangle\text{-}vars \longrightarrow \langle m1\rangle\text{-}(x,y)$
by (*auto simp*: *beta1-def m1-def vars-def tla-defs*)
next
show $|\sim \$pc1 = \#b \land \bigcirc(\$pc1 = \#b) \land \langle(n1 \lor n2) \land n1\rangle\text{-}vars \longrightarrow beta1$
by (*auto simp*: *n1-def alpha1-def beta1-def gamma1-def tla-defs*)
next
show $\vdash \$pc1 = \#b \land Enabled\ \langle m1\rangle\text{-}(x,\ y) \longrightarrow Enabled\ \langle n1\rangle\text{-}vars$
unfolding *enab-n1*[*int-rewrite*] **by** *auto*
next

The difficult part of the proof is showing that the persistent condition will eventually always be true if the helpful action is never executed. We show that (1) whenever the condition becomes true it remains so and (2) eventually the condition must be true.

show $\vdash \Box[(n1 \lor n2) \land \neg\ beta1]\text{-}vars$
$\land\ SF(n1)\text{-}vars \land \Box(I \land SF(n2)\text{-}vars) \land \Box\Diamond Enabled\ \langle m1\rangle\text{-}(x,\ y)$
$\longrightarrow \Diamond\Box(\$pc1 = \#b)$
proof −
have $\vdash \Box\Box[(n1 \lor n2) \land \neg\ beta1]\text{-}vars \longrightarrow \Box(\$pc1 = \#b \longrightarrow \Box(\$pc1 = \#b))$
proof (*rule STL4*)
have $|\sim \$pc1 = \#b \land [(n1 \lor n2) \land \neg\ beta1]\text{-}vars \longrightarrow \bigcirc(\$pc1 = \#b)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
from *this*[*THEN INV1*]
show $\vdash \Box[(n1 \lor n2) \land \neg\ beta1]\text{-}vars \longrightarrow \$pc1 = \#b \longrightarrow \Box(\$pc1 = \#b)$
by *auto*
qed
hence *1*: $\vdash \Box[(n1 \lor n2) \land \neg\ beta1]\text{-}vars \longrightarrow \Diamond(\$pc1 = \#b) \longrightarrow \Diamond\Box(\$pc1 = \#b)$
by (*force intro*: *E31*[*unlift-rule*])
have $\vdash \Box[(n1 \lor n2) \land \neg\ beta1]\text{-}vars \land SF(n1)\text{-}vars \land \Box(I \land SF(n2)\text{-}vars)$
$\longrightarrow \Diamond(\$pc1 = \#b)$
proof −

The plan of the proof is to show that from any state where $pc1 = g$ one eventually reaches $pc1 = a$, from where one eventually reaches $pc1 = b$. The result follows by combining leadsto properties.

let $?F = LIFT\ (\Box[(n1 \lor n2) \land \neg\ beta1]\text{-}vars \land SF(n1)\text{-}vars \land \Box(I \land SF(n2)\text{-}vars))$

Showing that $pc1 = g$ leads to $pc1 = a$ is a simple application of rule *SF1* because the first process completely controls this transition.

have *ga*: $\vdash ?F \longrightarrow (\$pc1 = \#g \leadsto \$pc1 = \#a)$
proof (*rule SF1*)
show $|\sim \$pc1 = \#g \land [(n1 \lor n2) \land \neg\ beta1]\text{-}vars \longrightarrow \bigcirc(\$pc1 = \#g) \lor \bigcirc(\$pc1 = \#a)$

by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show $|\sim$ $\$pc1 = \#g \wedge \langle((n1 \vee n2) \wedge \neg\ beta1) \wedge n1\rangle\text{-}vars \longrightarrow \bigcirc(\$pc1 = \#a)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show $|\sim$ $\$pc1 = \#g \wedge Unchanged\ vars \longrightarrow \bigcirc(\$pc1 = \#g)$
by (*auto simp*: *vars-def tla-defs*)
next
have $\vdash \$pc1 = \#g \longrightarrow Enabled\ \langle n1\rangle\text{-}vars$
unfolding *enab-n1*[*int-rewrite*] **by** (*auto simp*: *tla-defs*)
hence $\vdash \Box(\$pc1 = \#g) \longrightarrow Enabled\ \langle n1\rangle\text{-}vars$
by (*rule lift-imp-trans*[*OF ax1*])
hence $\vdash \Box(\$pc1 = \#g) \longrightarrow \Diamond Enabled\ \langle n1\rangle\text{-}vars$
by (*rule lift-imp-trans*[*OF - E3*])
thus $\vdash \Box(\$pc1 = \#g) \wedge \Box[(n1 \vee n2) \wedge \neg\ beta1]\text{-}vars \wedge \Box(I \wedge SF(n2)\text{-}vars) \longrightarrow \Diamond Enabled\ \langle n1\rangle\text{-}vars$
by *auto*
qed

The proof that $pc1 = a$ leads to $pc1 = b$ follows the same basic schema. However, showing that $n1$ is eventually enabled requires reasoning about the second process, which must liberate the critical section.

have *ab*: $\vdash\ ?F \longrightarrow (\$pc1 = \#a \leadsto \$pc1 = \#b)$
proof (*rule SF1*)
show $|\sim$ $\$pc1 = \#a \wedge [(n1 \vee n2) \wedge \neg\ beta1]\text{-}vars \longrightarrow \bigcirc(\$pc1 = \#a) \vee \bigcirc(\$pc1 = \#b)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show $|\sim$ $\$pc1 = \#a \wedge \langle((n1 \vee n2) \wedge \neg\ beta1) \wedge n1\rangle\text{-}vars \longrightarrow \bigcirc(\$pc1 = \#b)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show $|\sim$ $\$pc1 = \#a \wedge Unchanged\ vars \longrightarrow \bigcirc(\$pc1 = \#a)$
by (*auto simp*: *vars-def tla-defs*)
next

We establish a suitable leadsto-chain.

let $?G = LIFT\ \Box[(n1 \vee n2) \wedge \neg\ beta1]\text{-}vars \wedge SF(n2)\text{-}vars \wedge \Box(\$pc1 = \#a \wedge I)$
have $\vdash\ ?G \longrightarrow \Diamond(\$pc2 = \#a \wedge \$pc1 = \#a \wedge I)$
proof −

Rule *SF1* takes us from $pc2 = b$ to $pc2 = g$.

have *bg2*: $\vdash\ ?G \longrightarrow (\$pc2 = \#b \leadsto \$pc2 = \#g)$
proof (*rule SF1*)
show $|\sim$ $\$pc2 = \#b \wedge [(n1 \vee n2) \wedge \neg beta1]\text{-}vars \longrightarrow \bigcirc(\$pc2 = \#b) \vee \bigcirc(\$pc2 = \#g)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)

next
show $|{\sim}\ \$pc2 = \#b \wedge \langle((n1 \vee n2) \wedge \neg beta1) \wedge n2\rangle\text{-}vars \longrightarrow \bigcirc(\$pc2 = \#g)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show $|{\sim}\ \$pc2 = \#b \wedge Unchanged\ vars \longrightarrow \bigcirc(\$pc2 = \#b)$
by (*auto simp*: *vars-def tla-defs*)
next
have $\vdash \$pc2 = \#b \longrightarrow Enabled\ \langle n2\rangle\text{-}vars$
unfolding *enab-n2*[*int-rewrite*] **by** (*auto simp*: *tla-defs*)
hence $\vdash \Box(\$pc2 = \#b) \longrightarrow Enabled\ \langle n2\rangle\text{-}vars$
by (*rule lift-imp-trans*[*OF ax1*])
hence $\vdash \Box(\$pc2 = \#b) \longrightarrow \Diamond Enabled\ \langle n2\rangle\text{-}vars$
by (*rule lift-imp-trans*[*OF - E3*])
thus $\vdash \Box(\$pc2 = \#b) \wedge \Box[(n1 \vee n2) \wedge \neg beta1]\text{-}vars \wedge \Box(\$pc1 = \#a \wedge I)$
$\longrightarrow \Diamond Enabled\ \langle n2\rangle\text{-}vars$
by *auto*
qed

Similarly, $pc2 = b$ leads to $pc2 = g$.

have *ga2*: $\vdash\ ?G \longrightarrow (\$pc2 = \#g \leadsto \$pc2 = \#a)$
proof (*rule SF1*)
show $|{\sim}\ \$pc2 = \#g \wedge [(n1 \vee n2) \wedge \neg beta1]\text{-}vars \longrightarrow \bigcirc(\$pc2 = \#g) \vee \bigcirc(\$pc2 = \#a)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show $|{\sim}\ \$pc2 = \#g \wedge \langle((n1 \vee n2) \wedge \neg beta1) \wedge n2\rangle\text{-}vars \longrightarrow \bigcirc(\$pc2 = \#a)$
by (*auto simp*: *n2-def alpha2-def beta2-def gamma2-def vars-def tla-defs*)
next
show $|{\sim}\ \$pc2 = \#g \wedge Unchanged\ vars \longrightarrow \bigcirc(\$pc2 = \#g)$
by (*auto simp*: *vars-def tla-defs*)
next
have $\vdash \$pc2 = \#g \longrightarrow Enabled\ \langle n2\rangle\text{-}vars$
unfolding *enab-n2*[*int-rewrite*] **by** (*auto simp*: *tla-defs*)
hence $\vdash \Box(\$pc2 = \#g) \longrightarrow Enabled\ \langle n2\rangle\text{-}vars$
by (*rule lift-imp-trans*[*OF ax1*])
hence $\vdash \Box(\$pc2 = \#g) \longrightarrow \Diamond Enabled\ \langle n2\rangle\text{-}vars$
by (*rule lift-imp-trans*[*OF - E3*])
thus $\vdash \Box(\$pc2 = \#g) \wedge \Box[(n1 \vee n2) \wedge \neg beta1]\text{-}vars \wedge \Box(\$pc1 = \#a \wedge I)$
$\longrightarrow \Diamond Enabled\ \langle n2\rangle\text{-}vars$
by *auto*
qed
with *bg2* **have** $\vdash\ ?G \longrightarrow (\$pc2 = \#b \leadsto \$pc2 = \#a)$
by (*force elim*: *LT13*[*unlift-rule*])
with *ga2* **have** $\vdash\ ?G \longrightarrow (\$pc2 = \#a \vee \$pc2 = \#b \vee \$pc2 = \#g) \leadsto$

($\$pc2 = \#a$)
unfolding *LT17*[*int-rewrite*] *LT1*[*int-rewrite*] **by** *force*
moreover
have $\vdash \$pc2 = \#a \lor \$pc2 = \#b \lor \$pc2 = \#g$
proof (*clarsimp simp*: *tla-defs*)
fix *s* :: *state seq*
assume $pc2\ (s\ 0) \neq a$ **and** $pc2\ (s\ 0) \neq g$
thus $pc2\ (s\ 0) = b$ **by** (*cases pc2* (*s 0*)) *auto*
qed
hence $\vdash ((\$pc2 = \#a \lor \$pc2 = \#b \lor \$pc2 = \#g) \leadsto \$pc2 = \#a) \longrightarrow \Diamond(\$pc2 = \#a)$
by (*rule fmp*[*OF* - *LT4*])
ultimately
have $\vdash$ *?G* $\longrightarrow \Diamond(\$pc2 = \#a)$ **by** *force*
thus *?thesis* **by** (*auto intro*!: *SE3*[*unlift-rule*])
qed
moreover
have $\vdash \Diamond(\$pc2 = \#a \land \$pc1 = \#a \land I) \longrightarrow \Diamond Enabled\ \langle n1 \rangle\text{-}vars$
unfolding *enab-n1*[*int-rewrite*] **by** (*rule STL4-eve*) (*auto simp*: *I-def tla-defs*)
ultimately
show $\vdash \Box(\$pc1 = \#a) \land \Box[(n1 \lor n2) \land \neg\ beta1]\text{-}vars \land \Box(I \land SF(n2)\text{-}vars)$
$\longrightarrow \Diamond Enabled\ \langle n1 \rangle\text{-}vars$
by (*force simp*: *STL5*[*int-rewrite*])
qed
from *ga ab* **have** $\vdash$ *?F* $\longrightarrow (\$pc1 = \#g \leadsto \$pc1 = \#b)$
by (*force elim*: *LT13*[*unlift-rule*])
with *ab* **have** $\vdash$ *?F* $\longrightarrow ((\$pc1 = \#a \lor \$pc1 = \#b \lor \$pc1 = \#g) \leadsto \$pc1 = \#b)$
unfolding *LT17*[*int-rewrite*] *LT1*[*int-rewrite*] **by** *force*
moreover
have $\vdash \$pc1 = \#a \lor \$pc1 = \#b \lor \$pc1 = \#g$
proof (*clarsimp simp*: *tla-defs*)
fix *s* :: *state seq*
assume $pc1\ (s\ 0) \neq a$ **and** $pc1\ (s\ 0) \neq g$
thus $pc1\ (s\ 0) = b$ **by** (*cases pc1* (*s 0*), *auto*)
qed
hence $\vdash ((\$pc1 = \#a \lor \$pc1 = \#b \lor \$pc1 = \#g) \leadsto \$pc1 = \#b) \longrightarrow \Diamond(\$pc1 = \#b)$
by (*rule fmp*[*OF* - *LT4*])
ultimately show *?thesis* **by** (*rule lift-imp-trans*)
qed
with *1* **show** *?thesis* **by** *force*
qed
qed
with *psiI* **show** *?thesis* **unfolding** *psi-def Live2-def STL5*[*int-rewrite*] **by** *force*
qed

In the same way we prove that *psi* implements strong fairness for the abstract action *m1*. The proof is obtained by copy and paste from the previous one.

lemma (**in** *Secondprogram*) *psi-fair-m2*: $\vdash$ *psi* $\longrightarrow$ *SF*(*m2*)-(*x*,*y*)
proof –
  have $\vdash \Box$[*n1* $\vee$ *n2*]-*vars* $\wedge$ *SF*(*n2*)-*vars* $\wedge$ $\Box$(*I* $\wedge$ *SF*(*n1*)-*vars*) $\longrightarrow$ *SF*(*m2*)-(*x*,*y*)
  proof (*rule SF2*)

Rule *SF2* requires us to choose a helpful action (whose effect implies $\langle$*m2*$\rangle$-(*x*,*y*)) and a persistent condition, which will eventually remain true if the helpful action is never executed. In our case, the helpful action is *beta2* and the persistent condition is *pc2* = *b*.

    show |~ $\langle$(*n1* $\vee$ *n2*) $\wedge$ *beta2*$\rangle$-*vars* $\longrightarrow$ $\langle$*m2*$\rangle$-(*x*,*y*)
      by (*auto simp*: *beta2-def m2-def vars-def tla-defs*)
  next
    show |~ $*pc2* = #*b* $\wedge$ $\bigcirc$($*pc2* = #*b*) $\wedge$ $\langle$(*n1* $\vee$ *n2*) $\wedge$ *n2*$\rangle$-*vars* $\longrightarrow$ *beta2*
      by (*auto simp*: *n2-def alpha2-def beta2-def gamma2-def tla-defs*)
  next
    show $\vdash$ $*pc2* = #*b* $\wedge$ *Enabled* $\langle$*m2*$\rangle$-(*x*, *y*) $\longrightarrow$ *Enabled* $\langle$*n2*$\rangle$-*vars*
      unfolding *enab-n2*[*int-rewrite*] **by** *auto*
  next

The difficult part of the proof is showing that the persistent condition will eventually always be true if the helpful action is never executed. We show that (1) whenever the condition becomes true it remains so and (2) eventually the condition must be true.

    show $\vdash \Box$[(*n1* $\vee$ *n2*) $\wedge$ $\neg$ *beta2*]-*vars*
        $\wedge$ *SF*(*n2*)-*vars* $\wedge$ $\Box$(*I* $\wedge$ *SF*(*n1*)-*vars*) $\wedge$ $\Box\Diamond$*Enabled* $\langle$*m2*$\rangle$-(*x*, *y*)
        $\longrightarrow$ $\Diamond\Box$($*pc2* = #*b*)
    proof –
     have $\vdash \Box\Box$[(*n1* $\vee$ *n2*) $\wedge$ $\neg$ *beta2*]-*vars* $\longrightarrow$ $\Box$($*pc2* = #*b* $\longrightarrow$ $\Box$($*pc2* = #*b*))
      proof (*rule STL4*)
        have |~ $*pc2* = #*b* $\wedge$ [(*n1* $\vee$ *n2*) $\wedge$ $\neg$ *beta2*]-*vars* $\longrightarrow$ $\bigcirc$($*pc2* = #*b*)
          by (*auto simp*: *Sact2-defs vars-def tla-defs*)
        from *this*[*THEN INV1*]
        show $\vdash \Box$[(*n1* $\vee$ *n2*) $\wedge$ $\neg$ *beta2*]-*vars* $\longrightarrow$ $*pc2* = #*b* $\longrightarrow$ $\Box$($*pc2* = #*b*)
by *auto*
      qed
      hence *1*: $\vdash \Box$[(*n1* $\vee$ *n2*) $\wedge$ $\neg$ *beta2*]-*vars* $\longrightarrow$ $\Diamond$($*pc2* = #*b*) $\longrightarrow$ $\Diamond\Box$($*pc2* = #*b*)
        by (*force intro*: *E31*[*unlift-rule*])
      have $\vdash \Box$[(*n1* $\vee$ *n2*) $\wedge$ $\neg$ *beta2*]-*vars* $\wedge$ *SF*(*n2*)-*vars* $\wedge$ $\Box$(*I* $\wedge$ *SF*(*n1*)-*vars*)
          $\longrightarrow$ $\Diamond$($*pc2* = #*b*)
      proof –

The plan of the proof is to show that from any state where *pc2* = *g* one eventually reaches *pc2* = *a*, from where one eventually reaches *pc2* = *b*. The result follows by combining leadsto properties.

        let *?F* = *LIFT* ($\Box$[(*n1* $\vee$ *n2*) $\wedge$ $\neg$ *beta2*]-*vars* $\wedge$ *SF*(*n2*)-*vars* $\wedge$ $\Box$(*I* $\wedge$ *SF*(*n1*)-*vars*))

Showing that $pc2 = g$ leads to $pc2 = a$ is a simple application of rule *SF1* because the second process completely controls this transition.

have *ga*: $\vdash$ *?F* $\longrightarrow$ ($\$pc2 = \#g \leadsto \$pc2 = \#a$)
proof (*rule SF1*)
show $|\sim$ $\$pc2 = \#g \wedge [(n1 \vee n2) \wedge \neg\ beta2]\text{-}vars \longrightarrow \bigcirc(\$pc2 = \#g) \vee \bigcirc(\$pc2 = \#a)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show $|\sim$ $\$pc2 = \#g \wedge \langle((n1 \vee n2) \wedge \neg\ beta2) \wedge n2\rangle\text{-}vars \longrightarrow \bigcirc(\$pc2 = \#a)$
by (*auto simp*: *n2-def alpha2-def beta2-def gamma2-def vars-def tla-defs*)
next
show $|\sim$ $\$pc2 = \#g \wedge Unchanged\ vars \longrightarrow \bigcirc(\$pc2 = \#g)$
by (*auto simp*: *vars-def tla-defs*)
next
have $\vdash \$pc2 = \#g \longrightarrow Enabled\ \langle n2\rangle\text{-}vars$
unfolding *enab-n2*[*int-rewrite*] **by** (*auto simp*: *tla-defs*)
hence $\vdash \Box(\$pc2 = \#g) \longrightarrow Enabled\ \langle n2\rangle\text{-}vars$
by (*rule lift-imp-trans*[*OF ax1*])
hence $\vdash \Box(\$pc2 = \#g) \longrightarrow \Diamond Enabled\ \langle n2\rangle\text{-}vars$
by (*rule lift-imp-trans*[*OF - E3*])
thus $\vdash \Box(\$pc2 = \#g) \wedge \Box[(n1 \vee n2) \wedge \neg\ beta2]\text{-}vars \wedge \Box(I \wedge SF(n1)\text{-}vars)$
$\longrightarrow \Diamond Enabled\ \langle n2\rangle\text{-}vars$
by *auto*
qed

The proof that $pc2 = a$ leads to $pc2 = b$ follows the same basic schema. However, showing that *n2* is eventually enabled requires reasoning about the second process, which must liberate the critical section.

have *ab*: $\vdash$ *?F* $\longrightarrow$ ($\$pc2 = \#a \leadsto \$pc2 = \#b$)
proof (*rule SF1*)
show $|\sim$ $\$pc2 = \#a \wedge [(n1 \vee n2) \wedge \neg\ beta2]\text{-}vars \longrightarrow \bigcirc(\$pc2 = \#a) \vee \bigcirc(\$pc2 = \#b)$
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show $|\sim$ $\$pc2 = \#a \wedge \langle((n1 \vee n2) \wedge \neg\ beta2) \wedge n2\rangle\text{-}vars \longrightarrow \bigcirc(\$pc2 = \#b)$
by (*auto simp*: *n2-def alpha2-def beta2-def gamma2-def vars-def tla-defs*)
next
show $|\sim$ $\$pc2 = \#a \wedge Unchanged\ vars \longrightarrow \bigcirc(\$pc2 = \#a)$
by (*auto simp*: *vars-def tla-defs*)
next

We establish a suitable leadsto-chain.

let $?G = LIFT\ \Box[(n1 \vee n2) \wedge \neg\ beta2]\text{-}vars \wedge SF(n1)\text{-}vars \wedge \Box(\$pc2 = \#a \wedge I)$
have $\vdash\ ?G \longrightarrow \Diamond(\$pc1 = \#a \wedge \$pc2 = \#a \wedge I)$
proof −

Rule *SF1* takes us from $pc1 = b$ to $pc1 = g$.

have *bg1*: $\vdash$ *?G* $\longrightarrow$ ($pc1 = #b ⇝ $pc1 = #g)
proof (*rule SF1*)
show |~ $pc1 = #b ∧ [(n1 ∨ n2) ∧ ¬beta2]-vars ⟶ ○($pc1 = #b) ∨ ○($pc1 = #g)
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show |~ $pc1 = #b ∧ ⟨((n1 ∨ n2) ∧ ¬beta2) ∧ n1⟩-vars ⟶ ○($pc1 = #g)
by (*auto simp*: *n1-def alpha1-def beta1-def gamma1-def vars-def tla-defs*)
next
show |~ $pc1 = #b ∧ *Unchanged vars* ⟶ ○($pc1 = #b)
by (*auto simp*: *vars-def tla-defs*)
next
have ⊢ $pc1 = #b ⟶ *Enabled* ⟨n1⟩-vars
unfolding *enab-n1*[*int-rewrite*] **by** (*auto simp*: *tla-defs*)
hence ⊢ □($pc1 = #b) ⟶ *Enabled* ⟨n1⟩-vars
by (*rule lift-imp-trans*[*OF ax1*])
hence ⊢ □($pc1 = #b) ⟶ ◇*Enabled* ⟨n1⟩-vars
by (*rule lift-imp-trans*[*OF - E3*])
thus ⊢ □($pc1 = #b) ∧ □[(n1 ∨ n2) ∧ ¬beta2]-vars ∧ □($pc2 = #a ∧ I)
⟶ ◇*Enabled* ⟨n1⟩-vars
by *auto*
qed

Similarly, *pc1* = *b* leads to *pc1* = *g*.

have *ga1*: ⊢ *?G* ⟶ ($pc1 = #g ⇝ $pc1 = #a)
proof (*rule SF1*)
show |~ $pc1 = #g ∧ [(n1 ∨ n2) ∧ ¬beta2]-vars ⟶ ○($pc1 = #g) ∨ ○($pc1 = #a)
by (*auto simp*: *Sact2-defs vars-def tla-defs*)
next
show |~ $pc1 = #g ∧ ⟨((n1 ∨ n2) ∧ ¬beta2) ∧ n1⟩-vars ⟶ ○($pc1 = #a)
by (*auto simp*: *n1-def alpha1-def beta1-def gamma1-def vars-def tla-defs*)
next
show |~ $pc1 = #g ∧ *Unchanged vars* ⟶ ○($pc1 = #g)
by (*auto simp*: *vars-def tla-defs*)
next
have ⊢ $pc1 = #g ⟶ *Enabled* ⟨n1⟩-vars
unfolding *enab-n1*[*int-rewrite*] **by** (*auto simp*: *tla-defs*)
hence ⊢ □($pc1 = #g) ⟶ *Enabled* ⟨n1⟩-vars
by (*rule lift-imp-trans*[*OF ax1*])
hence ⊢ □($pc1 = #g) ⟶ ◇*Enabled* ⟨n1⟩-vars
by (*rule lift-imp-trans*[*OF - E3*])
thus ⊢ □($pc1 = #g) ∧ □[(n1 ∨ n2) ∧ ¬beta2]-vars ∧ □($pc2 = #a ∧ I)

$\longrightarrow \Diamond Enabled\ \langle n1\rangle\text{-}vars$
by *auto*
qed
with *bg1* **have** $\vdash\ ?G \longrightarrow (\$pc1 = \#b \leadsto \$pc1 = \#a)$
by (*force elim*: *LT13*[*unlift-rule*])
with *ga1* **have** $\vdash\ ?G \longrightarrow (\$pc1 = \#a \vee \$pc1 = \#b \vee \$pc1 = \#g) \leadsto (\$pc1 = \#a)$
unfolding *LT17*[*int-rewrite*] *LT1*[*int-rewrite*] **by** *force*
moreover
have $\vdash \$pc1 = \#a \vee \$pc1 = \#b \vee \$pc1 = \#g$
proof (*clarsimp simp*: *tla-defs*)
fix *s* :: *state seq*
assume $pc1\ (s\ 0) \neq a$ **and** $pc1\ (s\ 0) \neq g$
thus $pc1\ (s\ 0) = b$ **by** (*cases pc1* (*s 0*)) *auto*
qed
hence $\vdash ((\$pc1 = \#a \vee \$pc1 = \#b \vee \$pc1 = \#g) \leadsto \$pc1 = \#a) \longrightarrow \Diamond(\$pc1 = \#a)$
by (*rule fmp*[*OF - LT4*])
ultimately
have $\vdash\ ?G \longrightarrow \Diamond(\$pc1 = \#a)$ **by** *force*
thus *?thesis* **by** (*auto intro*!: *SE3*[*unlift-rule*])
qed
moreover
have $\vdash \Diamond(\$pc1 = \#a \wedge \$pc2 = \#a \wedge I) \longrightarrow \Diamond Enabled\ \langle n2\rangle\text{-}vars$
unfolding *enab-n2*[*int-rewrite*] **by** (*rule STL4-eve*) (*auto simp*: *I-def tla-defs*)
ultimately
show $\vdash \Box(\$pc2 = \#a) \wedge \Box[(n1 \vee n2) \wedge \neg\ beta2]\text{-}vars \wedge \Box(I \wedge SF(n1)\text{-}vars)$
$\longrightarrow \Diamond Enabled\ \langle n2\rangle\text{-}vars$
by (*force simp*: *STL5*[*int-rewrite*])
qed
from *ga ab* **have** $\vdash\ ?F \longrightarrow (\$pc2 = \#g \leadsto \$pc2 = \#b)$
by (*force elim*: *LT13*[*unlift-rule*])
with *ab* **have** $\vdash\ ?F \longrightarrow ((\$pc2 = \#a \vee \$pc2 = \#b \vee \$pc2 = \#g) \leadsto \$pc2 = \#b)$
unfolding *LT17*[*int-rewrite*] *LT1*[*int-rewrite*] **by** *force*
moreover
have $\vdash \$pc2 = \#a \vee \$pc2 = \#b \vee \$pc2 = \#g$
proof (*clarsimp simp*: *tla-defs*)
fix *s* :: *state seq*
assume $pc2\ (s\ 0) \neq a$ **and** $pc2\ (s\ 0) \neq g$
thus $pc2\ (s\ 0) = b$ **by** (*cases pc2* (*s 0*)) *auto*
qed
hence $\vdash ((\$pc2 = \#a \vee \$pc2 = \#b \vee \$pc2 = \#g) \leadsto \$pc2 = \#b) \longrightarrow \Diamond(\$pc2 = \#b)$
by (*rule fmp*[*OF - LT4*])
ultimately show *?thesis* **by** (*rule lift-imp-trans*)
qed
with *1* **show** *?thesis* **by** *force*

qed
qed
with *psiI* **show** *?thesis* **unfolding** *psi-def Live2-def STL5*[*int-rewrite*] **by** *force*
qed

We can now prove the main theorem, which states that *psi* implements *phi*.

theorem (**in** *Secondprogram*) *impl*: ⊢ *psi* ⟶ *phi*
unfolding *phi-def Live-def*
by (*auto dest*: *step-simulation*[*unlift-rule*]
lift-imp-trans[*OF psi-fair-m1 SF-imp-WF*, *unlift-rule*]
lift-imp-trans[*OF psi-fair-m2 SF-imp-WF*, *unlift-rule*])

end

10 Refining a Buffer Specification

theory *Buffer*
imports *State*
begin

We specify a simple FIFO buffer and prove that two FIFO buffers in a row implement a FIFO buffer.

10.1 Buffer specification

The following definitions all take three parameters: a state function representing the input channel of the FIFO buffer, another representing the internal queue, and a third one representing the output channel. These parameters will be instantiated later in the definition of the double FIFO.

definition *BInit* :: *'a statefun* ⇒ *'a list statefun* ⇒ *'a statefun* ⇒ *temporal*
where *BInit ic q oc* ≡ *TEMP* $*q* = #[]
∧ $*ic* = $*oc* — initial condition of buffer

definition *Enq* :: *'a statefun* ⇒ *'a list statefun* ⇒ *'a statefun* ⇒ *temporal*
where *Enq ic q oc* ≡ *TEMP ic*$ ≠ $*ic*
∧ *q*$ = $*q* @ [*ic*$]
∧ *oc*$ = $*oc* — enqueue a new value

definition *Deq* :: *'a statefun* ⇒ *'a list statefun* ⇒ *'a statefun* ⇒ *temporal*
where *Deq ic q oc* ≡ *TEMP* # *0* < *length*<$*q*>
∧ *oc*$ = *hd*<$*q*>
∧ *q*$ = *tl*<$*q*>
∧ *ic*$ = $*ic* — dequeue value at front

definition *Nxt* :: *'a statefun* ⇒ *'a list statefun* ⇒ *'a statefun* ⇒ *temporal*
where *Nxt ic q oc* ≡ *TEMP* (*Enq ic q oc* ∨ *Deq ic q oc*)

— internal specification with buffer visible
definition *ISpec* :: $'a$ *statefun* ⇒ $'a$ *list statefun* ⇒ $'a$ *statefun* ⇒ *temporal*
where *ISpec ic q oc* ≡ *TEMP BInit ic q oc*
∧ □[*Nxt ic q oc*]-(*ic*,*q*,*oc*)
∧ *WF*(*Deq ic q oc*)-(*ic*,*q*,*oc*)

— external specification: buffer hidden
definition *Spec* :: $'a$ *statefun* ⇒ $'a$ *statefun* ⇒ *temporal*
where *Spec ic oc* == *TEMP* (∃∃ *q*. *ISpec ic q oc*)

10.2 Properties of the buffer

The buffer never enqueues the same element twice. We therefore have the following invariant:

- any two subsequent elements in the queue are different, and the last element in the queue is different from the value of the output channel,
- if the queue is non-empty then the last element in the queue is the value that appears on the input channel,
- if the queue is empty then the values on the output and input channels are equal.

The following auxiliary predicate *noreps* is true if no two subsequent elements in a list are identical.

definition *noreps* :: $'a$ *list* ⇒ *bool*
where *noreps xs* ≡ ∀ *i* < *length xs* − *1*. *xs*!*i* ≠ *xs*!(*Suc i*)

definition *BInv* :: $'a$ *statefun* ⇒ $'a$ *list statefun* ⇒ $'a$ *statefun* ⇒ *temporal*
where *BInv ic q oc* ≡ *TEMP List.last*<\$*oc* # \$*q*> = \$*ic* ∧ *noreps*<\$*oc* # \$*q*>

lemmas *buffer-defs* = *BInit-def Enq-def Deq-def Nxt-def*
ISpec-def Spec-def BInv-def

lemma *ISpec-stutinv*: *STUTINV* (*ISpec ic q oc*)
unfolding *buffer-defs* **by** (*simp add*: *bothstutinvs livestutinv*)

lemma *Spec-stutinv*: *STUTINV Spec ic oc*
unfolding *buffer-defs* **by** (*simp add*: *bothstutinvs livestutinv eexSTUT*)

A lemma about lists that is useful in the following

lemma *tl-self-iff-empty*[*simp*]: (*tl xs* = *xs*) = (*xs* = [])
proof
assume *1*: *tl xs* = *xs*
show *xs* = []
proof (*rule ccontr*)
assume *xs* ≠ [] **with** *1* **show** *False*

by (*auto simp*: *neq-Nil-conv*)
qed
qed (*auto*)

lemma *tl-self-iff-empty'*[*simp*]: (*xs* = *tl xs*) = (*xs* = [])
proof
assume *1*: *xs* = *tl xs*
show *xs* = []
proof (*rule ccontr*)
assume *xs* ≠ [] **with** *1* **show** *False*
by (*auto simp*: *neq-Nil-conv*)
qed
qed (*auto*)

lemma *Deq-visible*:
assumes *v*: ⊢ *Unchanged v* ⟶ *Unchanged q*
shows |~ <*Deq ic q oc*>-*v* = *Deq ic q oc*
proof (*auto simp*: *tla-defs*)
fix *w*
assume *deq*: *w* ⊨ *Deq ic q oc* **and** *unch*: *v* (*w* (*Suc 0*)) = *v* (*w 0*)
from *unch v*[*unlifted*] **have** *q* (*w* (*Suc 0*)) = *q* (*w 0*)
by (*auto simp*: *tla-defs*)
with *deq* **show** *False* **by** (*auto simp*: *Deq-def tla-defs*)
qed

lemma *Deq-enabledE*: ⊢ *Enabled* <*Deq ic q oc*>-(*ic*,*q*,*oc*) ⟶ $*q* ~= #[]
by (*auto elim*!: *enabledE simp*: *Deq-def tla-defs*)

We now prove that *BInv* is an invariant of the Buffer specification.

We need several lemmas about *noreps* that are used in the invariant proof.

lemma *noreps-empty* [*simp*]: *noreps* []
by (*auto simp*: *noreps-def*)

lemma *noreps-singleton*: *noreps* [*x*] — special case of following lemma
by (*auto simp*: *noreps-def*)

lemma *noreps-cons* [*simp*]:
noreps (*x* # *xs*) = (*noreps xs* ∧ (*xs* = [] ∨ *x* ≠ *hd xs*))
proof (*auto simp*: *noreps-singleton*)
assume *cons*: *noreps* (*x* # *xs*)
show *noreps xs*
proof (*auto simp*: *noreps-def*)
fix *i*
assume *i*: *i* < *length xs* − *Suc 0* **and** *eq*: *xs*!*i* = *xs*!(*Suc i*)
from *i* **have** *Suc i* < *length* (*x*#*xs*) − *1* **by** *auto*
moreover
from *eq* **have** (*x*#*xs*)!(*Suc i*) = (*x*#*xs*)!(*Suc* (*Suc i*)) **by** *auto*
moreover
note *cons*

```
    ultimately show False by (auto simp: noreps-def)
  qed
next
  assume 1: noreps (hd xs # xs) and 2: xs ≠ []
  from 2 obtain x xxs where xs = x # xxs by (cases xs, auto)
  with 1 show False by (auto simp: noreps-def)
next
  assume 1: noreps xs and 2: x ≠ hd xs
  show noreps (x # xs)
  proof (auto simp: noreps-def)
    fix i
    assume i: i < length xs and eq: (x # xs)!i = xs!i
    from i obtain y ys where xs: xs = y # ys by (cases xs, auto)
    show False
    proof (cases i)
      assume i = 0
      with eq 2 xs show False by auto
    next
      fix k
      assume k: i = Suc k
      with i eq xs 1 show False by (auto simp: noreps-def)
    qed
  qed
qed

lemma noreps-append [simp]:
  noreps (xs @ ys) =
   (noreps xs ∧ noreps ys ∧ (xs = [] ∨ ys = [] ∨ List.last xs ≠ hd ys))
proof auto
  assume 1: noreps (xs @ ys)
  show noreps xs
  proof (auto simp: noreps-def)
    fix i
    assume i: i < length xs − Suc 0 and eq: xs!i = xs!(Suc i)
    from i have i < length (xs @ ys) − Suc 0 by auto
    moreover
    from i eq have (xs @ ys)!i = (xs@ys)!(Suc i) by (auto simp: nth-append)
    moreover note 1
    ultimately show False by (auto simp: noreps-def)
  qed
next
  assume 1: noreps (xs @ ys)
  show noreps ys
  proof (auto simp: noreps-def)
    fix i
    assume i: i < length ys − Suc 0 and eq: ys!i = ys!(Suc i)
    from i have i + length xs < length (xs @ ys) − Suc 0 by auto
    moreover
    from i eq have (xs @ ys)!(i+length xs) = (xs@ys)!(Suc (i + length xs))
```

by (*auto simp*: *nth-append*)
moreover note *1*
ultimately show *False* **by** (*auto simp*: *noreps-def*)
qed
next
assume *1*: *noreps* (*xs* @ *ys*) **and** *2*: *xs* $\neq$ [] **and** *3*: *ys* $\neq$ []
and *4*: *List.last xs* = *hd ys*
from *2* **obtain** *x xxs* **where** *xs*: *xs* = *x* # *xxs* **by** (*cases xs*, *auto*)
from *3* **obtain** *y yys* **where** *ys*: *ys* = *y* # *yys* **by** (*cases ys*, *auto*)
from *xs ys* **have** *5*: *length xxs* < *length* (*xs* @ *ys*) − *1* **by** *auto*
from *4 xs ys* **have** (*xs* @ *ys*) ! (*length xxs*) = (*xs* @ *ys*) ! (*Suc* (*length xxs*))
by (*auto simp*: *nth-append last-conv-nth*)
with *5 1* **show** *False* **by** (*auto simp*: *noreps-def*)
next
assume *1*: *noreps xs* **and** *2*: *noreps ys* **and** *3*: *List.last xs* $\neq$ *hd ys*
show *noreps* (*xs* @ *ys*)
proof (*cases xs* = [] $\vee$ *ys* = [])
case *True*
with *1 2* **show** *?thesis* **by** *auto*
next
case *False*
then obtain *x xxs* **where** *xs*: *xs* = *x* # *xxs* **by** (*cases xs*, *auto*)
from *False* **obtain** *y yys* **where** *ys*: *ys* = *y* # *yys* **by** (*cases ys*, *auto*)
show *?thesis*
proof (*auto simp*: *noreps-def*)
fix *i*
assume *i*: *i* < *length xs* + *length ys* − *Suc 0*
and *eq*: (*xs* @ *ys*)!*i* = (*xs* @ *ys*)!(*Suc i*)
show *False*
proof (*cases i* < *length xxs*)
case *True*
hence *i* < *length* (*x* # *xxs*) **by** *simp*
hence *xsi*: ((*x* # *xxs*) @ *ys*)!*i* = (*x* # *xxs*)!*i*
unfolding *nth-append* **by** *simp*
from *True* **have** (*xxs* @ *ys*)!*i* = *xxs*!*i* **by** (*auto simp*: *nth-append*)
with *True xsi eq 1 xs* **show** *False* **by** (*auto simp*: *noreps-def*)
next
assume *i2*: ¬(*i* < *length xxs*)
show *False*
proof (*cases i* = *length xxs*)
case *True*
with *xs* **have** *xsi*: (*xs* @ *ys*)!*i* = *List.last xs*
by (*auto simp*: *nth-append last-conv-nth*)
from *True xs ys* **have** (*xs* @ *ys*)!(*Suc i*) = *y*
by (*auto simp*: *nth-append*)
with *3 ys eq xsi* **show** *False* **by** *simp*
next
case *False*
with *i2 xs* **have** *xsi*: ¬(*i* < *length xs*) **by** *auto*

hence $(xs\ @\ ys)!i = ys!(i - length\ xs)$
  by (*simp add*: *nth-append*)
moreover
from *xsi* **have** $Suc\ i - length\ xs = Suc\ (i - length\ xs)$ **by** *auto*
with *xsi* **have** $(xs\ @\ ys)!(Suc\ i) = ys!(Suc\ (i - length\ xs))$
  by (*simp add*: *nth-append*)
moreover
from *i xsi* **have** $i - length\ xs < length\ ys - 1$ **by** *auto*
with *2* **have** $ys!(i - length\ xs) \neq ys!(Suc\ (i - length\ xs))$
  by (*auto simp*: *noreps-def*)
moreover
note *eq*
ultimately show *False* **by** *simp*
qed
qed
qed
qed
qed

lemma *ISpec-BInv-lemma*:
$\vdash BInit\ ic\ q\ oc \wedge \Box[Nxt\ ic\ q\ oc]\text{-}(ic,q,oc) \longrightarrow \Box(BInv\ ic\ q\ oc)$
proof (*rule invmono*)
  show $\vdash BInit\ ic\ q\ oc \longrightarrow BInv\ ic\ q\ oc$
    by (*auto simp*: *BInit-def BInv-def*)
next
  have *enq*: $|^\sim\ Enq\ ic\ q\ oc \longrightarrow BInv\ ic\ q\ oc \longrightarrow \bigcirc(BInv\ ic\ q\ oc)$
    by (*auto simp*: *Enq-def BInv-def tla-defs*)
  have *deq*: $|^\sim\ Deq\ ic\ q\ oc \longrightarrow BInv\ ic\ q\ oc \longrightarrow \bigcirc(BInv\ ic\ q\ oc)$
    by (*auto simp*: *Deq-def BInv-def tla-defs neq-Nil-conv*)
  have *unch*: $|^\sim\ Unchanged\ (ic,q,oc) \longrightarrow BInv\ ic\ q\ oc \longrightarrow \bigcirc(BInv\ ic\ q\ oc)$
    by (*auto simp*: *BInv-def tla-defs*)
  show $|^\sim\ BInv\ ic\ q\ oc \wedge [Nxt\ ic\ q\ oc]\text{-}(ic,\ q,\ oc) \longrightarrow \bigcirc(BInv\ ic\ q\ oc)$
    by (*auto simp*: *Nxt-def actrans-def*
        elim: *enq*[*unlift-rule*] *deq*[*unlift-rule*] *unch*[*unlift-rule*])
qed

theorem *ISpec-BInv*: $\vdash ISpec\ ic\ q\ oc \longrightarrow \Box(BInv\ ic\ q\ oc)$
  by (*auto simp*: *ISpec-def intro*: *ISpec-BInv-lemma*[*unlift-rule*])

10.3 Two FIFO buffers in a row implement a buffer

locale *DBuffer* =
  fixes *inp* :: *'a statefun* — input channel for double FIFO
    and *mid* :: *'a statefun* — channel linking the two buffers
    and *out* :: *'a statefun* — output channel for double FIFO
    and *q1* :: *'a list statefun* — inner queue of first FIFO
    and *q2* :: *'a list statefun* — inner queue of second FIFO
    and *vars*
  defines $vars \equiv LIFT\ (inp,mid,out,q1,q2)$

assumes *DB-base*: *basevars vars*
begin

We need to specify the behavior of two FIFO buffers in a row. Intuitively, that specification is just the conjunction of two buffer specifications, where the first buffer has input channel *inp* and output channel *mid* whereas the second one receives from *mid* and outputs on *out*. However, this conjunction allows a simultaneous enqueue action of the first buffer and dequeue of the second one. It would not implement the previous buffer specification, which excludes such simultaneous enqueueing and dequeueing (it is written in "interleaving style"). We could relax the specification of the FIFO buffer above, which is esthetically pleasant, but non-interleaving specifications are usually hard to get right and to understand. We therefore impose an interleaving constraint on the specification of the double buffer, which requires that enqueueing and dequeueing do not happen simultaneously.

definition *DBSpec*
where $DBSpec \equiv TEMP\ ISpec\ inp\ q1\ mid$
$\quad\wedge\ ISpec\ mid\ q2\ out$
$\quad\wedge\ \Box[\neg(Enq\ inp\ q1\ mid \wedge Deq\ mid\ q2\ out)]\text{-}vars$

The proof rules of TLA are geared towards specifications of the form $Init \wedge \Box[Next]\text{-}vars \wedge L$, and we prove that *DBSpec* corresponds to a specification in this form, which we now define.

definition *FullInit*
where $FullInit \equiv TEMP\ (BInit\ inp\ q1\ mid \wedge BInit\ mid\ q2\ out)$

definition *FullNxt*
where $FullNxt \equiv TEMP\ (Enq\ inp\ q1\ mid \wedge Unchanged\ (q2,out)$
$\quad\vee\ Deq\ inp\ q1\ mid \wedge Enq\ mid\ q2\ out$
$\quad\vee\ Deq\ mid\ q2\ out \wedge Unchanged\ (inp,q1))$

definition *FullSpec*
where $FullSpec \equiv TEMP\ FullInit$
$\quad\wedge\ \Box[FullNxt]\text{-}vars$
$\quad\wedge\ WF(Deq\ inp\ q1\ mid)\text{-}vars$
$\quad\wedge\ WF(Deq\ mid\ q2\ out)\text{-}vars$

The concatenation of the two queues will serve as the refinement mapping.

definition *qc* :: *'a list statefun*
where $qc \equiv LIFT\ (q2\ @\ q1)$

lemmas *db-defs* = *buffer-defs DBSpec-def FullInit-def FullNxt-def FullSpec-def qc-def vars-def*

lemma *DBSpec-stutinv*: *STUTINV DBSpec*
unfolding *db-defs* **by** (*simp add*: *bothstutinvs livestutinv*)

lemma *FullSpec-stutinv*: *STUTINV FullSpec*
  unfolding *db-defs* **by** (*simp add*: *bothstutinvs livestutinv*)

We prove that *DBSpec* implies *FullSpec*. (The converse implication also holds but is not needed for our implementation proof.)

The following lemma is somewhat more bureaucratic than we'd like it to be. It shows that the conjunction of the next-state relations, together with the invariant for the first queue, implies the full next-state relation of the combined queues.

```
lemma DBNxt-then-FullNxt:
  ⊢ □BInv inp q1 mid
      ∧ □[Nxt inp q1 mid]-(inp,q1,mid)
      ∧ □[Nxt mid q2 out]-(mid,q2,out)
      ∧ □[¬(Enq inp q1 mid ∧ Deq mid q2 out)]-vars
      ⟶ □[FullNxt]-vars
  (is ⊢ □?inv ∧ ?nxts ⟶ □[FullNxt]-vars)
proof −
  have ⊢ □[Nxt inp q1 mid]-(inp,q1,mid)
       ∧ □[Nxt mid q2 out]-(mid,q2,out)
       ⟶ □[  [Nxt inp q1 mid]-(inp,q1,mid)
            ∧ [Nxt mid q2 out]-(mid,q2,out)]-((inp,q1,mid),(mid,q2,out))
    (is ⊢ ?tmp ⟶ □[?b1b2]-?vs)
    by (auto simp: M12[int-rewrite])
  moreover
  have ⊢ □[?b1b2]-?vs ⟶ □[?b1b2]-vars
    by (rule R1, auto simp: vars-def tla-defs)
  ultimately
  have 1: ⊢ □[Nxt inp q1 mid]-(inp,q1,mid)
          ∧ □[Nxt mid q2 out]-(mid,q2,out)
          ⟶ □[  [Nxt inp q1 mid]-(inp,q1,mid)
               ∧ [Nxt mid q2 out]-(mid,q2,out) ]-vars
    by force
  have 2: ⊢ □[?b1b2]-vars ∧ □[¬(Enq inp q1 mid ∧ Deq mid q2 out)]-vars
            ⟶ □[?b1b2 ∧ ¬(Enq inp q1 mid ∧ Deq mid q2 out)]-vars
    (is ⊢ ?tmp2 ⟶ □[?mid]-vars)
    by (simp add: M8[int-rewrite])
  have ⊢ ?inv ⟶ #True by auto
  moreover
  have |~ ?inv ∧ ○?inv ∧ [?mid]-vars ⟶ [FullNxt]-vars
  proof −
    have |~ ?inv ∧ ?mid ⟶ [FullNxt]-vars
    proof −
      have A: |~ Nxt inp q1 mid
                ⟶ [Nxt mid q2 out]-(mid,q2,out)
                ⟶ ¬(Enq inp q1 mid ∧ Deq mid q2 out)
                ⟶ ?inv
                ⟶ FullNxt
```

```
    proof −
      have enq: |~ Enq inp q1 mid
                 ∧ [Nxt mid q2 out]-(mid,q2,out)
                 ∧ ¬(Deq mid q2 out)
                 ⟶ Unchanged (q2,out)
        by (auto simp: db-defs tla-defs)
      have deq1: |~ Deq inp q1 mid ⟶ ?inv ⟶ mid$ ≠ $mid
        by (auto simp: Deq-def BInv-def)
      have deq2: |~ Deq mid q2 out ⟶ mid$ = $mid
        by (auto simp: Deq-def)
      have deq: |~ Deq inp q1 mid
                 ∧ [Nxt mid q2 out]-(mid,q2,out)
                 ∧ ?inv
                 ⟶ Enq mid q2 out
        by (force simp: Nxt-def tla-defs
                  dest: deq1[unlift-rule] deq2[unlift-rule])
      with enq show ?thesis by (force simp: Nxt-def FullNxt-def)
    qed
    have B: |~ Nxt mid q2 out
             ⟶ Unchanged (inp,q1,mid)
             ⟶ FullNxt
      by (auto simp: db-defs tla-defs)
    have C: ⊢ Unchanged (inp,q1,mid)
          ⟶ Unchanged (mid,q2,out)
          ⟶ Unchanged vars
      by (auto simp: vars-def tla-defs)
    show ?thesis
      by (force simp: actrans-def
                dest: A[unlift-rule] B[unlift-rule] C[unlift-rule])
  qed
  thus ?thesis by (auto simp: tla-defs)
 qed
 ultimately
 have ⊢ □?inv ∧ □[?mid]-vars ⟶ □#True ∧ □[FullNxt]-vars
   by (rule TLA2)
 with 1 2 show ?thesis by force
qed
```

It is now easy to show that *DBSpec* refines *FullSpec*.

```
theorem DBSpec-impl-FullSpec: ⊢ DBSpec ⟶ FullSpec
proof −
  have 1: ⊢ DBSpec ⟶ FullInit
    by (auto simp: DBSpec-def FullInit-def ISpec-def)
  have 2: ⊢ DBSpec ⟶ □[FullNxt]-vars
  proof −
    have ⊢ DBSpec ⟶ □(BInv inp q1 mid)
      by (auto simp: DBSpec-def intro: ISpec-BInv[unlift-rule])
    moreover have ⊢ DBSpec ∧ □(BInv inp q1 mid) ⟶ □[FullNxt]-vars
      by (auto simp: DBSpec-def ISpec-def
```

intro: *DBNxt-then-FullNxt*[*unlift-rule*])
ultimately show *?thesis* **by** *force*
qed
have *3*: ⊢ *DBSpec* ⟶ *WF*(*Deq inp q1 mid*)-*vars*
proof −
have *31*: ⊢ *Unchanged vars* ⟶ *Unchanged q1*
by (*auto simp*: *vars-def tla-defs*)
have *32*: ⊢ *Unchanged* (*inp*,*q1*,*mid*) ⟶ *Unchanged q1*
by (*auto simp*: *tla-defs*)
have *deq*: |~ ⟨*Deq inp q1 mid*⟩-*vars* = ⟨*Deq inp q1 mid*⟩-(*inp*,*q1*,*mid*)
by (*simp add*: *Deq-visible*[*OF 31*, *int-rewrite*]
Deq-visible[*OF 32*, *int-rewrite*])
show *?thesis*
by (*auto simp*: *DBSpec-def ISpec-def WeakF-def*
deq[*int-rewrite*] *deq*[*THEN AA26*,*int-rewrite*])
qed
have *4*: ⊢ *DBSpec* ⟶ *WF*(*Deq mid q2 out*)-*vars*
proof −
have *41*: ⊢ *Unchanged vars* ⟶ *Unchanged q2*
by (*auto simp*: *vars-def tla-defs*)
have *42*: ⊢ *Unchanged* (*mid*,*q2*,*out*) ⟶ *Unchanged q2*
by (*auto simp*: *tla-defs*)
have *deq*: |~ ⟨*Deq mid q2 out*⟩-*vars* = ⟨*Deq mid q2 out*⟩-(*mid*,*q2*,*out*)
by (*simp add*: *Deq-visible*[*OF 41*, *int-rewrite*]
Deq-visible[*OF 42*, *int-rewrite*])
show *?thesis*
by (*auto simp*: *DBSpec-def ISpec-def WeakF-def*
deq[*int-rewrite*] *deq*[*THEN AA26*,*int-rewrite*])
qed
show *?thesis*
by (*auto simp*: *FullSpec-def*
elim: *1*[*unlift-rule*] *2*[*unlift-rule*] *3*[*unlift-rule*]
4[*unlift-rule*])
qed

We now prove that two FIFO buffers in a row (as specified by formula *FullSpec*) implement a FIFO buffer whose internal queue is the concatenation of the two buffers. We start by proving step simulation.

lemma *FullInit*: ⊢ *FullInit* ⟶ *BInit inp qc out*
by (*auto simp*: *db-defs tla-defs*)

lemma *Full-step-simulation*:
|~ [*FullNxt*]-*vars* ⟶ [*Nxt inp qc out*]-(*inp*,*qc*,*out*)
by (*auto simp*: *db-defs tla-defs*)

The liveness condition requires that the combined buffer eventually performs a *Deq* action on the output channel if it contains some element. The idea is to use the fairness hypothesis for the first buffer to prove that in that case, eventually the queue of the second buffer will be non-empty, and that it must therefore eventually dequeue some element.

The first step is to establish the enabledness conditions for the two *Deq* actions of the implementation.

lemma *Deq1-enabled*: $\vdash$ *Enabled* $\langle$*Deq inp q1 mid*$\rangle$-*vars* = ($*q1* $\neq$ #[])
proof $-$
  have *1*: $|\sim$ $\langle$*Deq inp q1 mid*$\rangle$-*vars* = *Deq inp q1 mid*
    by (*rule Deq-visible*, *auto simp*: *vars-def tla-defs*)
  have $\vdash$ *Enabled* (*Deq inp q1 mid*) = ($*q1* $\neq$ #[])
    by (*force simp*: *Deq-def tla-defs vars-def*
        intro: *base-enabled*[*OF DB-base*] *elim*!: *enabledE*)
  thus *?thesis* **by** (*simp add*: *1*[*int-rewrite*])
qed

lemma *Deq2-enabled*: $\vdash$ *Enabled* $\langle$*Deq mid q2 out*$\rangle$-*vars* = ($*q2* $\neq$ #[])
proof $-$
  have *1*: $|\sim$ $\langle$*Deq mid q2 out*$\rangle$-*vars* = *Deq mid q2 out*
    by (*rule Deq-visible*, *auto simp*: *vars-def tla-defs*)
  have $\vdash$ *Enabled* (*Deq mid q2 out*) = ($*q2* $\neq$ #[])
    by (*force simp*: *Deq-def tla-defs vars-def*
        intro: *base-enabled*[*OF DB-base*] *elim*!: *enabledE*)
  thus *?thesis* **by** (*simp add*: *1*[*int-rewrite*])
qed

We now use rule *WF2* to prove that the combined buffer (behaving according to specification *FullSpec*) implements the fairness condition of the single buffer under the refinement mapping.

lemma *Full-fairness*:
  $\vdash$ $\Box$[*FullNxt*]-*vars* $\wedge$ *WF*(*Deq mid q2 out*)-*vars* $\wedge$ $\Box$*WF*(*Deq inp q1 mid*)-*vars*
    $\longrightarrow$ *WF*(*Deq inp qc out*)-(*inp*,*qc*,*out*)
proof (*rule WF2*)
  — the helpful action is the *Deq* action of the second queue
  show $|\sim$ $\langle$*FullNxt* $\wedge$ *Deq mid q2 out*$\rangle$-*vars* $\longrightarrow$ $\langle$*Deq inp qc out*$\rangle$-(*inp*,*qc*,*out*)
    by (*auto simp*: *db-defs tla-defs*)
next
  — the helpful condition is the second queue being non-empty
  show $|\sim$ ($*q2* $\neq$ #[]) $\wedge$ $\bigcirc$($*q2* $\neq$ #[]) $\wedge$ $\langle$*FullNxt* $\wedge$ *Deq mid q2 out*$\rangle$-*vars*
      $\longrightarrow$ *Deq mid q2 out*
    by (*auto simp*: *tla-defs*)
next
  show $\vdash$ $*q2* $\neq$ #[] $\wedge$ *Enabled* $\langle$*Deq inp qc out*$\rangle$-(*inp*, *qc*, *out*)
      $\longrightarrow$ *Enabled* $\langle$*Deq mid q2 out*$\rangle$-*vars*
    unfolding *Deq2-enabled*[*int-rewrite*] **by** *auto*
next

The difficult part of the proof is to show that the helpful condition will eventually always be true provided that the combined dequeue action is eventually always enabled and that the helpful action is never executed. We prove that (1) the helpful condition persists and (2) that it must eventually become true.

have $\vdash \Box\Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-}vars$
$\quad\longrightarrow \Box(\$q2 \neq \#[] \longrightarrow \Box(\$q2 \neq \#[]))$
proof (*rule STL4*)
  have $|\!\sim\ \$q2 \neq \#[] \wedge [FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-}vars$
$\quad\longrightarrow \bigcirc(\$q2 \neq \#[])$
    by (*auto simp*: *db-defs tla-defs*)
  from *this*[*THEN INV1*]
  show $\vdash \Box[FullNxt \wedge \neg\ Deq\ mid\ q2\ out]\text{-}vars$
$\quad\longrightarrow (\$q2 \neq \#[] \longrightarrow \Box(\$q2 \neq \#[]))$
    by *auto*
qed
hence *1*: $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-}vars$
$\quad\longrightarrow \Diamond(\$q2 \neq \#[]) \longrightarrow \Diamond\Box(\$q2 \neq \#[])$
  by (*force intro*: *E31*[*unlift-rule*])
have *2*: $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-}vars$
$\quad\wedge\ WF(Deq\ inp\ q1\ mid)\text{-}vars$
$\quad\longrightarrow (Enabled\ \langle Deq\ inp\ qc\ out\rangle\text{-}(inp,\ qc,\ out) \leadsto \$q2 \neq \#[])$
proof −
  have *qc*: $\vdash (\$qc \neq \#[]) = (\$q1 \neq \#[] \vee \$q2 \neq \#[])$
    by (*auto simp*: *qc-def tla-defs*)
  have $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-}vars \wedge WF(Deq\ inp\ q1\ mid)\text{-}vars$
$\quad\longrightarrow (\$q1 \neq \#[] \leadsto \$q2 \neq \#[])$
  proof (*rule WF1*)
    show $|\!\sim\ \$q1 \neq \#[] \wedge [FullNxt \wedge \neg\ Deq\ mid\ q2\ out]\text{-}vars$
$\quad\longrightarrow \bigcirc(\$q1 \neq \#[]) \vee \bigcirc(\$q2 \neq \#[])$
      by (*auto simp*: *db-defs tla-defs*)
  next
    show $|\!\sim\ \$q1 \neq \#[]$
$\quad\wedge\ \langle(FullNxt \wedge \neg\ Deq\ mid\ q2\ out) \wedge Deq\ inp\ q1\ mid\rangle\text{-}vars \longrightarrow$
$\quad\bigcirc(\$q2 \neq \#[])$
      by (*auto simp*: *db-defs tla-defs*)
  next
    show $\vdash \$q1 \neq \#[] \longrightarrow Enabled\ \langle Deq\ inp\ q1\ mid\rangle\text{-}vars$
      by (*simp add*: *Deq1-enabled*[*int-rewrite*])
  next
    show $|\!\sim\ \$q1 \neq \#[] \wedge Unchanged\ vars \longrightarrow \bigcirc(\$q1 \neq \#[])$
      by (*auto simp*: *vars-def tla-defs*)
  qed
  hence $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-}vars$
$\quad\wedge\ WF(Deq\ inp\ q1\ mid)\text{-}vars$
$\quad\longrightarrow (\$qc \neq \#[] \leadsto \$q2 \neq \#[])$
    by (*auto simp*: *qc*[*int-rewrite*] *LT17*[*int-rewrite*] *LT1*[*int-rewrite*])
  moreover
  have $\vdash Enabled\ \langle Deq\ inp\ qc\ out\rangle\text{-}(inp,\ qc,\ out) \leadsto \$qc \neq \#[]$
    by (*rule Deq-enabledE*[*THEN LT3*])
  ultimately show *?thesis* **by** (*force elim*: *LT13*[*unlift-rule*])
qed
with *LT6*
have $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-}vars$

$\wedge$ *WF*(*Deq inp q1 mid*)*-vars*
$\wedge$ $\Diamond$*Enabled* $\langle$*Deq inp qc out*$\rangle$-(*inp*, *qc*, *out*)
$\longrightarrow$ $\Diamond$($\$q2 \neq \#[]$)
by *force*
with *1 E16*
show $\vdash$ $\Box$[*FullNxt* $\wedge$ $\neg$(*Deq mid q2 out*)]*-vars*
$\wedge$ *WF*(*Deq mid q2 out*)*-vars*
$\wedge$ $\Box$*WF*(*Deq inp q1 mid*)*-vars*
$\wedge$ $\Diamond\Box$ *Enabled* $\langle$*Deq inp qc out*$\rangle$-(*inp*, *qc*, *out*)
$\longrightarrow$ $\Diamond\Box$($\$q2 \neq \#[]$)
by *force*
qed

Putting everything together, we obtain that *FullSpec* refines the Buffer specification under the refinement mapping.

theorem *FullSpec-impl-ISpec*: $\vdash$ *FullSpec* $\longrightarrow$ *ISpec inp qc out*
unfolding *FullSpec-def ISpec-def*
using *FullInit Full-step-simulation*[*THEN M11*] *Full-fairness*
by *force*

theorem *FullSpec-impl-Spec*: $\vdash$ *FullSpec* $\longrightarrow$ *Spec inp out*
unfolding *Spec-def* **using** *FullSpec-impl-ISpec*
by (*force intro*: *eexI*[*unlift-rule*])

By transitivity, two buffers in a row also implement a single buffer.

theorem *DBSpec-impl-Spec*: $\vdash$ *DBSpec* $\longrightarrow$ *Spec inp out*
by (*rule lift-imp-trans*[*OF DBSpec-impl-FullSpec FullSpec-impl-Spec*])

end — locale DBuffer

end

References

[1] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the tla$^+$ proof system. In J. Giesl and R. Hähnle, editors, *5th Intl. Joint Conf. Automated Reasoning (IJCAR 2010)*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148, Edinburgh, UK, 2010. Springer.

[2] M. Devillers, D. Griffioen, and O. Müller. Possibly Infinite Sequences in Theorem Provers: A comparative study. In E. L. Gunter and A. P. Felty, editors, *10th International Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104. Springer, August 1997.

[3] S. O. Ehmety and L. C. Paulson. Representing Component States in Higher-Order Logic. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics*, pages 151–158, 2001.

[4] G. Grov. *Reasoning about Correctness Properties of a Coordination Programming Language*. PhD thesis, Heriot-Watt University, March 2009.

[5] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[6] L. Lamport. *Specifying Systems — The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Reading, Massachusetts, 2002.

[7] S. Merz. An Encoding of TLA in Isabelle. http://www.pst.informatik.uni-muenchen.de/˜merz/isabelle/. Part of the Isabelle distribution., 1998.

[8] S. Merz. A More Complete TLA. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1226–1244, Toulouse, France, Sept. 1999. Springer-Verlag.

[9] M. Wenzel. Using Axiomatic Type Classes in Isabelle, May 2000.

[10] M. Wildmoser and T. Nipkow. Certifying Machine Code Safety: Shallow versus Deep Embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.