

A Definitional Encoding of TLA in Isabelle/HOL

Gudmund Grov & Stephan Merz

May 26, 2024

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	3
1.1 Some operators on sequences	3
1.1.1 Properties of <i>first</i> and <i>second</i>	4
1.1.2 Properties of $(_s)$	4
1.1.3 Properties of $(\#\#)$	5
1.2 Finite and Empty Sequences	5
1.2.1 Properties of <i>emptyseq</i>	6

1.2.2	Properties of <i>Sequence.last</i> and <i>laststate</i>	7
1.3	Stuttering Invariance	7
1.3.1	Properties of <i>nonstutseq</i>	8
1.3.2	Properties of <i>nextnat</i>	8
1.3.3	Properties of <i>nextsuffix</i>	11
1.3.4	Properties of <i>next</i>	11
1.3.5	Properties of \natural	12
1.4	Similarity of Sequences	13
1.4.1	Properties of (\approx)	13
2	Representing Intensional Logic	19
2.1	Abstract Syntax and Definitions	20
2.2	Concrete Syntax	21
2.3	Lemmas and Tactics	24
3	Semantics	26
3.1	Types of Formulas	26
3.2	Semantics of TLA*	27
3.2.1	Concrete Syntax	27
3.3	Abbreviations	28
3.3.1	Concrete Syntax	28
3.4	Properties of Operators	29
3.5	Invariance Under Stuttering	29
3.5.1	Properties of <i>-stutinv</i>	30
3.5.2	Properties of <i>-nstutinv</i>	34
3.5.3	Abbreviations	35
4	Reasoning about PreFormulas	36
4.1	Lemmas about <i>Unchanged</i>	37
4.2	Lemmas about <i>after</i>	37
4.3	Lemmas about <i>before</i>	38
4.4	Some general properties	39
4.5	Unlifting attributes and methods	39
5	A Proof System for TLA*	40
5.1	The Basic Axioms	41
5.2	Derived Theorems	43
5.3	Some other useful derived theorems	46
5.4	Theorems about the eventually operator	52
5.5	Theorems about the leadsto operator	56
5.6	Lemmas about the next operator	63
5.7	Higher Level Derived Rules	65

6	Liveness	67
6.1	Properties of <i>-Enabled</i>	68
6.2	Fairness Properties	69
6.3	Stuttering Invariance	75
7	Representing state in TLA*	75
7.1	Temporal Quantifiers	78
8	A simple illustrative example	79
9	Lamport's Inc example	80
10	Refining a Buffer Specification	93
10.1	Buffer specification	93
10.2	Properties of the buffer	94
10.3	Two FIFO buffers in a row implement a buffer	98

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 ⇒ nat ⇒ 'a seq (**infixl** |_s 60)
where $s \text{ |}_s i \equiv \lambda n. s (n+i)$

definition *tail* :: 'a seq ⇒ 'a seq
where $\text{tail } s \equiv s \text{ |}_s 1$

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

$s \text{ |}_s i$ 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. $s \text{ ## } \sigma$ prefixes the sequence σ by element s .

1.1.1 Properties of *first* and *second*

lemma *first-tail-second*: $\text{first}(\text{tail } s) = \text{second } s$
by (*simp add: first-def second-def tail-def suffix-def*)

1.1.2 Properties of (|_s)

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

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

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

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

lemma *suffix-plus-com*: $s \text{ |}_s m \text{ |}_s n = s \text{ |}_s (m + n)$

proof –

have $s \text{ |}_s n \text{ |}_s m = s \text{ |}_s (m + n)$ **by** (*rule suffix-plus*)

thus $s \text{ |}_s m \text{ |}_s n = s \text{ |}_s (m + n)$ **by** (*simp add: suffix-commute*)

qed

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

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

lemma *tail-suffix-suc*: $s \text{ |}_s (\text{Suc } n) = \text{tail } (s \text{ |}_s n)$
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*: $(\text{first } s) \#\# (\text{tail } s) = s$

proof (*rule ext*)

fix x

show $(\text{first } s \#\# \text{tail } s) x = s x$

by (*simp add: first-def app-def suffix-def tail-def*)

qed

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

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

1.2 Finite and Empty Sequences

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

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

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

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

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

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

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

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 |_s n)*

proof (*clarsimp simp: emptyseq-def*)

fix *i*

from *H* **have** $(s |_s n) i = s 0$ **by** (*simp add: emptyseq-def suffix-def*)

moreover

from *H* **have** $(s |_s n) 0 = s 0$ **by** (*simp add: emptyseq-def suffix-def*)

ultimately

show $(s |_s n) i = (s |_s n) 0$ **by** *simp*

qed

lemma *suc-empty*: **assumes** *H1: emptyseq (s |_s m)* **shows** *emptyseq (s |_s (Suc m))*

proof –

from *H1* **have** *emptyseq ((s |_s m) |_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** $(s |_s n) m = s m$

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** $(s |_s n) = s$

proof (*rule ext*)

fix *m*

from *H* **show** $(s |_s n) m = s m$ **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** $\forall j \geq \text{last } s. s\ j = s\ (\text{last } s)$

proof (*clarify*)

fix j

assume $j: j \geq \text{last } s$

from H **obtain** i **where** $\forall j \geq i. s\ j = s\ i$ (**is** $?P\ i$) **by** (*auto simp: fin-def*)

hence $?P\ (\text{last } s)$ **unfolding** *last-def* **by** (*rule LeastI*)

with j **show** $s\ j = s\ (\text{last } s)$ **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* $s \equiv \text{if emptyseq } s \text{ then } 0 \text{ else LEAST } i. s\ i \neq s\ 0$

definition *nextsuffix* :: $('a\ seq) \Rightarrow ('a\ seq)$

where *nextsuffix* $s \equiv s\ \downarrow_s\ (\text{nextnat } s)$

fun *next* :: $nat \Rightarrow ('a\ seq) \Rightarrow ('a\ seq)$ **where**

next $0 = id$

| *next* $(Suc\ n) = \text{nextsuffix } o\ (\text{next } n)$

definition *collapse* :: $('a\ seq) \Rightarrow ('a\ seq)$ (\natural)

where $\natural\ s \equiv \lambda n. (\text{next } n\ s)\ 0$

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. $\text{next } n$ s is defined by primitive recursion on n using function composition of function nextsuffix . E.g. $\text{next } 3$ s equals $\text{nextsuffix} (\text{nextsuffix} (\text{nextsuffix } s))$. $\text{nextsuffix } s$ 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 : $\text{emptyseq } s$ **shows** $\text{nonstutseq } s$
using H **by** (*auto simp: nonstutseq-def seq-empty-all*)

lemma *notempty-exist-nonstut*:

assumes H : $\neg \text{emptyseq } (s \mid_s m)$ **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 : $n < \text{nextnat } s$ **shows** $s \ n = s \ 0$

proof (*cases emptyseq s*)

assume $\text{emptyseq } s$
hence $\text{nextnat } s = 0$ **by** (*simp add: nextnat-def*)
with H **show** *?thesis* **by** *auto*

next

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

qed

qed

lemma *stutnempty*:

assumes H : $\neg \text{stutstep } s \ n$ **shows** $\neg \text{emptyseq } (s \mid_s n)$
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-nextnat1*:

assumes H : $\neg\ stutstep\ s\ n$ **shows** $nextnat (s \mid_s n) = 1$

proof –

from H **have** h' : $nextnat (s \mid_s n) = (LEAST\ i.\ (s \mid_s n)\ i \neq (s \mid_s n)\ 0)$
by $(auto\ simp\ add:\ nextnat-def\ stutnempty)$

from H **have** $s (Suc\ n) \neq s\ n$ **by** $(auto\ simp\ add:\ stutstep-def)$

hence $(s \mid_s n)\ 1 \neq (s \mid_s n)\ 0$ **(is** $?P\ 1)$ **by** $(auto\ simp\ add:\ suffix-def)$

hence $Least\ ?P \leq 1$ **by** $(rule\ Least-le)$

hence $g1$: $Least\ ?P = 0 \vee Least\ ?P = 1$ **by** $auto$

with h' **have** $g1'$: $nextnat (s \mid_s n) = 0 \vee nextnat (s \mid_s n) = 1$ **by** $auto$

also **have** $nextnat (s \mid_s n) \neq 0$

proof –

from H **have** $\neg\ emptyseq (s \mid_s n)$ **by** $(rule\ stutnempty)$

then **obtain** i **where** $(s \mid_s n)\ i \neq (s \mid_s n)\ 0$ **by** $(auto\ simp\ add:\ emptyseq-def)$

hence $(s \mid_s n)\ (LEAST\ i.\ (s \mid_s n)\ i \neq (s \mid_s n)\ 0) \neq (s \mid_s n)\ 0$ **by** $(rule\ LeastI)$

with h' **have** $g2$: $(s \mid_s n)\ (nextnat (s \mid_s n)) \neq (s \mid_s n)\ 0$ **by** $auto$

show $(nextnat (s \mid_s n)) \neq 0$

proof

assume $(nextnat (s \mid_s n)) = 0$

with $g2$ **show** $False$ **by** $simp$

qed

qed

ultimately **show** $nextnat (s \mid_s n) = 1$ **by** $auto$

qed

lemma *stutstep-notempty-notempty*:

assumes $h1$: $emptyseq (s \mid_s\ Suc\ n)$ **(is** $emptyseq\ ?sn)$

and $h2$: $stutstep\ s\ n$

shows $emptyseq (s \mid_s\ n)$ **(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** $\dots = ?sn\ 0$ **by** $(simp\ add:\ emptyseq-def)$

also **from** $h2$ **have** $\dots = s\ n$ **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 $\text{emptyseq } (s \mid_s \text{Suc } n) = \text{emptyseq } (s \mid_s n)$

using *assms* **by** (*auto elim: stutstep-notempty-notempty suc-empty*)

lemma *stutstep-notempty-sucnextnat*:

assumes *h1: $\neg \text{emptyseq } (s \mid_s n)$* **and** *h2: stutstep s n*

shows $(\text{nextnat } (s \mid_s n)) = \text{Suc } (\text{nextnat } (s \mid_s (\text{Suc } n)))$

proof –

from *h2* **have** *g1: $\neg (s (0+n) \neq s (\text{Suc } n))$* (**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: $s (i+n) \neq s (\text{Suc } n)$* (**is** $?P i$) **by** (*simp add: stutstep-def*)

from *g2 g1* **have** $(\text{LEAST } n. ?P n) = \text{Suc } (\text{LEAST } n. ?P (\text{Suc } n))$ **by** (*rule Least-Suc*)

from *g2 g1* **have** $(\text{LEAST } i. s (i+n) \neq s (\text{Suc } n)) = \text{Suc } (\text{LEAST } i. s ((\text{Suc } i)+n) \neq s (\text{Suc } n))$

by (*rule Least-Suc*)

hence *G1: $(\text{LEAST } i. s (i+n) \neq s (\text{Suc } n)) = \text{Suc } (\text{LEAST } i. s (i+\text{Suc } n) \neq s (\text{Suc } n))$* **by** *auto*

from *h1 h2* **have** $\neg \text{emptyseq } (s \mid_s \text{Suc } n)$ **by** (*simp add: stutstep-empty-suc*)

hence $\text{nextnat } (s \mid_s \text{Suc } n) = (\text{LEAST } i. (s \mid_s \text{Suc } n) i \neq (s \mid_s \text{Suc } n) 0)$

by (*auto simp add: nextnat-def*)

hence *g1: $\text{nextnat } (s \mid_s \text{Suc } n) = (\text{LEAST } i. s (i+(\text{Suc } n)) \neq s (\text{Suc } n))$*

by (*simp add: suffix-def*)

from *h1* **have** $\text{nextnat } (s \mid_s n) = (\text{LEAST } i. (s \mid_s n) i \neq (s \mid_s n) 0)$

by (*auto simp add: nextnat-def*)

hence *g2: $\text{nextnat } (s \mid_s n) = (\text{LEAST } i. s (i+n) \neq s n)$* **by** (*simp add: suffix-def*)

with *h2* **have** *g2': $\text{nextnat } (s \mid_s n) = (\text{LEAST } i. s (i+n) \neq s (\text{Suc } n))$*

by (*auto simp add: stutstep-def*)

from *G1 g1 g2'* **show** *?thesis* **by** *auto*

qed

lemma *nextnat-empty-neq*: **assumes** *H: $\neg \text{emptyseq } s$* **shows** $s (\text{nextnat } s) \neq s 0$

proof –

from *H* **have** *a1: $\text{nextnat } s = (\text{LEAST } i. s i \neq s 0)$* **by** (*simp add: nextnat-def*)

from *H* **obtain** *i* **where** $s i \neq s 0$ **by** (*auto simp: emptyseq-def*)

hence $s (\text{LEAST } i. s i \neq s 0) \neq s 0$ **by** (*rule LeastI*)

with *a1* **show** *?thesis* **by** *auto*

qed

lemma *nextnat-empty-gzero*: **assumes** *H: $\neg \text{emptyseq } s$* **shows** $\text{nextnat } s > 0$

proof –

from *H* **have** *a1: $s (\text{nextnat } s) \neq s 0$* **by** (*rule nextnat-empty-neq*)

have $\text{nextnat } s \neq 0$

proof

assume $\text{nextnat } s = 0$

with *a1* **show** *False* **by** *simp*

qed

thus $\text{nextnat } s > 0$ by *simp*
 qed

1.3.3 Properties of nextsuffix

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

lemma *empty-nextsuffix-id*:
 assumes H : $\text{emptyseq } s$ shows $\text{nextsuffix } s = \text{id } s$
 using H by (*simp add: empty-nextsuffix*)

lemma *notstutstep-nextsuffix1*:
 assumes H : $\neg \text{stutstep } s \ n$ shows $\text{nextsuffix } (s \mid_s n) = s \mid_s (\text{Suc } n)$
proof (*unfold nextsuffix-def*)
 show $(s \mid_s n \mid_s (\text{nextnat } (s \mid_s n))) = s \mid_s (\text{Suc } n)$
proof –
 from H have $\text{nextnat } (s \mid_s n) = 1$ by (*rule notstutstep-nextnat1*)
 hence $(s \mid_s n \mid_s (\text{nextnat } (s \mid_s n))) = s \mid_s n \mid_s 1$ by *auto*
 thus *?thesis* by (*simp add: suffix-def*)
 qed
 qed

1.3.4 Properties of next

lemma *next-suc-suffix*: $\text{next } (\text{Suc } n) \ s = \text{nextsuffix } (\text{next } n \ s)$
 by *simp*

lemma *next-suffix-com*: $\text{nextsuffix } (\text{next } n \ s) = (\text{next } n \ (\text{nextsuffix } s))$
 by (*induct n, auto*)

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

lemma *next-empty*: assumes H : $\text{emptyseq } s$ shows $\text{next } n \ s = s$
proof (*induct n*)

from H show $\text{next } 0 \ s = s$ by *auto*
next
fix n
 assume $a1$: $\text{next } n \ s = s$
 have $\text{next } (\text{Suc } n) \ s = \text{nextsuffix } (\text{next } n \ s)$ by *auto*
 with $a1$ have $\text{next } (\text{Suc } n) \ s = \text{nextsuffix } s$ by *simp*
 with H show $\text{next } (\text{Suc } n) \ s = s$
 by (*simp add: nextsuffix-def nextnat-def*)

qed

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

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

lemma next-ex-id: $\exists i. s \ i = (\text{next } m \ s) \ 0$

proof –

have $\exists i. (s \ |_s \ i) = (\text{next } m \ s)$

proof (induct m)

have $s \ |_s \ 0 = \text{next } 0 \ s$ **by** simp

thus $\exists i. (s \ |_s \ i) = (\text{next } 0 \ s)$..

next

fix m

assume $a1: \exists i. (s \ |_s \ i) = (\text{next } m \ s)$

then obtain i **where** $a1': (s \ |_s \ i) = (\text{next } m \ s)$..

have $\text{next } (\text{Suc } m) \ s = \text{nextsuffix } (\text{next } m \ s)$ **by** auto

hence $\text{next } (\text{Suc } m) \ s = (\text{next } m \ s) \ |_s \ (\text{nextnat } (\text{next } m \ s))$ **by** (simp add: nextsuffix-def)

hence $\exists i. \text{next } (\text{Suc } m) \ s = (\text{next } m \ s) \ |_s \ i$..

then obtain j **where** $\text{next } (\text{Suc } m) \ s = (\text{next } m \ s) \ |_s \ j$..

with $a1'$ **have** $\text{next } (\text{Suc } m) \ s = (s \ |_s \ i) \ |_s \ j$ **by** simp

hence $\text{next } (\text{Suc } m) \ s = (s \ |_s \ (j+i))$ **by** (simp add: suffix-plus)

hence $(s \ |_s \ (j+i)) = \text{next } (\text{Suc } m) \ s$ **by** simp

thus $\exists i. (s \ |_s \ i) = (\text{next } (\text{Suc } m) \ s)$..

qed

then obtain i **where** $(s \ |_s \ i) = (\text{next } m \ s)$..

hence $(s \ |_s \ i) \ 0 = (\text{next } m \ s) \ 0$ **by** auto

hence $s \ i = (\text{next } m \ s) \ 0$ **by** (auto simp add: suffix-def)

thus ?thesis ..

qed

1.3.5 Properties of \natural

lemma emptyseq-collapse-eq: **assumes** $A1: \text{emptyseq } s$ **shows** $\natural \ s = s$

proof (unfold collapse-def, rule ext)

fix n

from $A1$ **have** $\text{next } n \ s = s$ **by** (rule next-empty)

moreover

from $A1$ **have** $s \ n = s \ 0$ **by** (simp add: emptyseq-def)

ultimately

show $(\text{next } n \ s) \ 0 = s \ n$ **by** simp

qed

lemma empty-collapse-empty:

assumes $H: \text{emptyseq } s$ **shows** $\text{emptyseq } (\natural \ s)$

using H **by** (simp add: emptyseq-collapse-eq)

lemma collapse-empty-empty:

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

lemma *collapse-empty-iff-empty* [*simp*]: *emptyseq* (\dagger 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-invariant 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 $\sigma \approx \tau \equiv (\dagger \ \sigma) = (\dagger \ \tau)$

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* $s = \text{first } t$
proof –
from H **have** $(\dagger \ s) \ 0 = (\dagger \ t) \ 0$ **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 : $\text{tail } s \approx \text{tail } t$ **shows** *second* $s = \text{second } t$
proof –
from H **have** *first* (*tail* s) = *first* (*tail* t) **by** (*simp add: sim-first*)
thus *second* $s = \text{second } t$ **by** (*simp add: first-tail-second*)

qed

lemma *seqsimilarI*:

assumes *1*: $\text{first } s = \text{first } t$ and *2*: $\text{nextsuffix } s \approx \text{nextsuffix } t$
shows $s \approx t$

unfolding *seqsimilar-def collapse-def*

proof

fix *n*

show $\text{next } n \ s \ 0 = \text{next } n \ t \ 0$

proof (cases *n*)

assume $n = 0$

with *1* show *thesis* by (*simp add: first-def*)

next

fix *m*

assume $m: n = \text{Suc } m$

from *2* have $\text{next } m \ (\text{nextsuffix } s) \ 0 = \text{next } m \ (\text{nextsuffix } t) \ 0$

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*: $s \approx t$ and *H2*: *emptyseq s* shows *emptyseq t*

proof –

from *H2* have *emptyseq* ($\heartsuit s$) by *simp*

with *H1* have *emptyseq* ($\heartsuit t$) by (*simp add: seqsimilar-def*)

thus *thesis* by *simp*

qed

lemma *seqsim-empty-iff-empty*:

assumes *H*: $s \approx t$ 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 $t \approx s$ 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 $H: \neg (\text{stutstep } s \ n)$ **shows** $(s \mid_s (Suc \ n)) \approx \text{nextsuffix } (s \mid_s \ n)$
using H **by** $(\text{simp add: notstutstep-nextsuffix1})$

lemma *stut-nextsuf-suc*:

assumes $H: \text{stutstep } s \ n$ **shows** $\text{nextsuffix } (s \mid_s \ n) = \text{nextsuffix } (s \mid_s (Suc \ n))$

proof $(\text{cases emptyseq } (s \mid_s \ n))$

case *True*

hence $g1: \text{nextsuffix } (s \mid_s \ n) = (s \mid_s \ n)$ **by** $(\text{simp add: nextsuffix-def nextnat-def})$

from *True* **have** $g2: \text{nextsuffix } (s \mid_s \ Suc \ n) = (s \mid_s \ Suc \ n)$

by $(\text{simp add: suc-empty nextsuffix-def nextnat-def})$

have $(s \mid_s \ n) = (s \mid_s \ Suc \ n)$

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 $(s \mid_s \ n) \ x = (s \mid_s \ Suc \ n) \ x$ **by** $(\text{simp add: suffix-def})$

qed

with $g1 \ g2$ **show** *?thesis* **by** *auto*

next

case *False*

with H **have** $(\text{nextnat } (s \mid_s \ n)) = Suc \ (\text{nextnat } (s \mid_s \ Suc \ n))$

by $(\text{simp add: stutstep-notempty-sucnextnat})$

thus *?thesis*

by $(\text{simp add: nextsuffix-def suffix-plus})$

qed

lemma *seqsim-suffix-seqsim*:

assumes $H: s \approx t$ **shows** $\text{nextsuffix } s \approx \text{nextsuffix } t$

unfolding *seqsimilar-def collapse-def*

proof

fix n

from H **have** $(\text{next } (Suc \ n) \ s) \ 0 = (\text{next } (Suc \ n) \ t) \ 0$

unfolding *seqsimilar-def collapse-def* **by** (rule fun-cong)

thus $\text{next } n \ (\text{nextsuffix } s) \ 0 = \text{next } n \ (\text{nextsuffix } t) \ 0$

by $(\text{simp add: next-suffix-com})$

qed

lemma *seqsim-stutstep*:

assumes $H: \text{stutstep } s \ n$ **shows** $(s \mid_s (Suc \ n)) \approx (s \mid_s \ n)$ **(is** *?sn* **≈** *?s***)**

unfolding *seqsimilar-def collapse-def*

proof

fix m

show $\text{next } m \ (s \mid_s \ Suc \ n) \ 0 = \text{next } m \ (s \mid_s \ n) \ 0$

proof $(\text{cases } m)$

assume $m=0$

with H **show** *?thesis* **by** $(\text{simp add: suffix-def stutstep-def})$

next

fix k

assume $m: m = Suc \ k$

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

lemma *addfeqstut*: $\text{stutstep } ((\text{first } t) \#\# t) 0$
by (*simp add: first-def stutstep-def app-def suffix-def*)

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

lemma *addfirststut*:
assumes H : $\text{first } s = \text{second } s$ **shows** $s \approx \text{tail } s$
proof –
have $g1$: $(\text{first } s) \#\# (\text{tail } s) = s$ **by** (*rule seq-app-first-tail*)
from H **have** $(\text{first } s) = \text{first } (\text{tail } s)$
by (*simp add: first-def second-def tail-def suffix-def*)
hence $(\text{first } s) \#\# (\text{tail } s) \approx (\text{tail } s)$ **by** (*simp add: addfeqsim*)
with $g1$ **show** *?thesis* **by** *simp*
qed

lemma *app-seqsimilar*:
assumes $h1$: $s \approx t$ **shows** $(x \#\# s) \approx (x \#\# t)$
proof (*cases stutstep (x ## s) 0*)
case *True*
from $h1$ **have** $\text{first } s = \text{first } t$ **by** (*rule sim-first*)
with *True* **have** $a2$: $\text{stutstep } (x \#\# t) 0$
by (*simp add: stutstep-def first-def app-def*)
from *True* **have** $((x \#\# s) \mid_s (\text{Suc } 0)) \approx ((x \#\# s) \mid_s 0)$ **by** (*rule seqsim-stutstep*)
hence $\text{tail } (x \#\# s) \approx (x \#\# s)$ **by** (*simp add: tail-def suffix-def*)
hence $g1$: $s \approx (x \#\# s)$ **by** (*simp add: app-def tail-def suffix-def*)
from $a2$ **have** $((x \#\# t) \mid_s (\text{Suc } 0)) \approx ((x \#\# t) \mid_s 0)$ **by** (*rule seqsim-stutstep*)
hence $\text{tail } (x \#\# t) \approx (x \#\# t)$ **by** (*simp add: tail-def suffix-def*)
hence $g2$: $t \approx (x \#\# t)$ **by** (*simp add: app-def tail-def suffix-def*)
from $h1$ $g2$ **have** $s \approx (x \#\# t)$ **by** (*rule seqsim-trans*)
from *this*[*THEN seqsim-sym*] $g1$ **show** $(x \#\# s) \approx (x \#\# t)$
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: ¬ stutstep (x ## t) 0
  by (simp add: stutstep-def first-def app-def)
from False have  $((x \## s) \mid_s (Suc\ 0)) \approx nextsuffix ((x \## s) \mid_s 0)$ 
  by (rule seqsim-notstutstep)
hence  $(tail\ (x \## s)) \approx nextsuffix\ (x \## s)$ 
  by (simp add: tail-def)
hence g1: s ≈ nextsuffix (x ## s) by (simp add: seq-app-tail)
from a2 have  $((x \## t) \mid_s (Suc\ 0)) \approx nextsuffix ((x \## t) \mid_s 0)$ 
  by (rule seqsim-notstutstep)
hence  $(tail\ (x \## t)) \approx nextsuffix\ (x \## t)$  by (simp add: tail-def)
hence g2: t ≈ nextsuffix (x ## t) by (simp add: seq-app-tail)
with h1 have s ≈ nextsuffix (x ## t) by (rule seqsim-trans)
from this[THEN seqsim-sym] g1 have g3: nextsuffix (x ## s) ≈ 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 ≈ t* **shows** $\exists m. ((s \mid_s n) \approx (t \mid_s m))$

proof (*induct n*)

from *H* **have** $((s \mid_s 0) \approx (t \mid_s 0))$ **by** *auto*

thus $\exists m. ((s \mid_s 0) \approx (t \mid_s m))$ **..**

next

fix *n*

assume $\exists m. ((s \mid_s n) \approx (t \mid_s m))$

then obtain *m* **where** *a1': (s |_s n) ≈ (t |_s m)* **..**

show $\exists m. ((s \mid_s (Suc\ n)) \approx (t \mid_s m))$

proof (*cases stutstep s n*)

case *True*

hence $(s \mid_s (Suc\ n)) \approx (s \mid_s n)$ **by** (*rule seqsim-stutstep*)

from *this a1'* **have** $((s \mid_s (Suc\ n)) \approx (t \mid_s m))$ **by** (*rule seqsim-trans*)

thus *?thesis* **..**

next

case *False*

hence $(s \mid_s (Suc\ n)) \approx nextsuffix\ (s \mid_s n)$ **by** (*rule seqsim-notstutstep*)

moreover

from *a1'* **have** $nextsuffix\ (s \mid_s n) \approx nextsuffix\ (t \mid_s m)$

by (*simp add: seqsim-suffix-seqsim*)

ultimately have $(s \mid_s (Suc\ n)) \approx nextsuffix\ (t \mid_s m)$ **by** (*rule seqsim-trans*)

hence $(s \mid_s (Suc\ n)) \approx t \mid_s (m + (nextnat\ (t \mid_s m)))$

by (*simp add: nextsuffix-def suffix-plus-com*)

thus $\exists m. (s \mid_s (Suc\ n)) \approx t \mid_s m$ **..**

qed

qed

lemma *nextnat-le-seqsim*:
assumes $n: n < \text{nextnat } s$ **shows** $s \approx (s \mid_s n)$
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 $s \approx (s \mid_s 0)$ **by** *simp*
next
fix n
assume $a2: n < \text{nextnat } s \implies s \approx (s \mid_s n)$ **and** $a3: \text{Suc } n < \text{nextnat } s$
from $a3$ **have** $g1: s (\text{Suc } n) = s 0$ **by** (*rule nextnat-le-unch*)
from $a3$ **have** $a3': n < \text{nextnat } s$ **by** *simp*
hence $s n = s 0$ **by** (*rule nextnat-le-unch*)
with $g1$ **have** *stutstep s n* **by** (*simp add: stutstep-def*)
hence $g2: (s \mid_s n) \approx (s \mid_s (\text{Suc } n))$ **by** (*rule seqsim-stutstep[THEN seqsim-sym]*)
with $a3' a2$ **show** $s \approx (s \mid_s (\text{Suc } n))$ **by** (*auto elim: seqsim-trans*)
qed
qed

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

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

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

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

$((s \mid_s \text{Suc } n \approx t \mid_s \text{Suc } m) \vee (s \mid_s \text{Suc } n \approx t \mid_s m))$

(**is** $\exists m. ?\text{Sim } n \ m$)

proof (*induct n*)

from H **have** $s \mid_s 0 \approx t \mid_s 0$ **by** *simp*

thus $\exists m. ?\text{Sim } 0 \ m$ **by** (*rule seqsim-suffix-suc*)

next

fix n

assume $\exists m. ?\text{Sim } n \ m$

hence $\exists k. s \mid_s \text{Suc } n \approx t \mid_s k$ **by** *blast*

thus $\exists m. ?\text{Sim } (\text{Suc } n) \ m$ **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 $('w, 'a) \text{ expr} = 'w \Rightarrow 'a$

type-synonym $'w \text{ form} = ('w, \text{bool}) \text{ expr}$

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

definition *Valid* :: $('w::\text{world}) \text{ form} \Rightarrow \text{bool}$

where *Valid* $A \equiv \forall w. A w$

definition *const* :: $'a \Rightarrow ('w::\text{world}, 'a) \text{ expr}$

where *unl-con*: *const* $c w \equiv c$

definition *lift* :: $['a \Rightarrow 'b, ('w::\text{world}, 'a) \text{ expr}] \Rightarrow ('w, 'b) \text{ expr}$

where *unl-lift*: *lift* $f x w \equiv f (x w)$

definition *lift2* :: $['a \Rightarrow 'b \Rightarrow 'c, ('w::\text{world}, 'a) \text{ expr}, ('w, 'b) \text{ expr}] \Rightarrow ('w, 'c) \text{ expr}$

where *unl-lift2*: *lift2* $f x y w \equiv f (x w) (y w)$

definition *lift3* :: $['a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd, ('w::\text{world}, 'a) \text{ expr}, ('w, 'b) \text{ expr}, ('w, 'c) \text{ expr}] \Rightarrow ('w, 'd) \text{ expr}$

where *unl-lift3*: *lift3* $f x y z w \equiv f (x w) (y w) (z w)$

definition *lift4* :: $['a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e, ('w::\text{world}, 'a) \text{ expr}, ('w, 'b) \text{ expr}, ('w, 'c) \text{ expr}, ('w, 'd) \text{ expr}] \Rightarrow ('w, 'e) \text{ expr}$

where *unl-lift4*: *lift4* $f x y z z w \equiv f (x w) (y w) (z w) (z w)$

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 quantification. *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

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

LIFT :: *lift* ⇒ 'a (LIFT -)

$-const :: 'a \Rightarrow lift \quad ((\#-) [1000] 999)$
 $-lift :: ['a, lift] \Rightarrow lift \quad ((-<->) [1000] 999)$
 $-lift2 :: ['a, lift, lift] \Rightarrow lift \quad ((-<-/, ->) [1000] 999)$
 $-lift3 :: ['a, lift, lift, lift] \Rightarrow lift \quad ((-<-/, -, ->) [1000] 999)$
 $-lift4 :: ['a, lift, lift, lift, lift] \Rightarrow lift \quad ((-<-/, -, -, ->) [1000] 999)$

$-liftEqu :: [lift, lift] \Rightarrow lift \quad ((- =/ -) [50,51] 50)$
 $-liftNeq :: [lift, lift] \Rightarrow lift \quad (\mathbf{infixl} \neq 50)$
 $-liftNot :: lift \Rightarrow lift \quad (\neg - [90] 90)$
 $-liftAnd :: [lift, lift] \Rightarrow lift \quad (\mathbf{infixr} \wedge 35)$
 $-liftOr :: [lift, lift] \Rightarrow lift \quad (\mathbf{infixr} \vee 30)$
 $-liftImp :: [lift, lift] \Rightarrow lift \quad (\mathbf{infixr} \longrightarrow 25)$
 $-liftIf :: [lift, lift, lift] \Rightarrow lift \quad ((if (-)/ then (-)/ else (-)) 10)$
 $-liftPlus :: [lift, lift] \Rightarrow lift \quad ((- +/ -) [66,65] 65)$
 $-liftMinus :: [lift, lift] \Rightarrow lift \quad ((- -/ -) [66,65] 65)$
 $-liftTimes :: [lift, lift] \Rightarrow lift \quad ((- */ -) [71,70] 70)$
 $-liftDiv :: [lift, lift] \Rightarrow lift \quad ((- div -) [71,70] 70)$
 $-liftMod :: [lift, lift] \Rightarrow lift \quad ((- mod -) [71,70] 70)$
 $-liftLess :: [lift, lift] \Rightarrow lift \quad ((- / < -) [50, 51] 50)$
 $-liftLeq :: [lift, lift] \Rightarrow lift \quad ((- / \le -) [50, 51] 50)$
 $-liftMem :: [lift, lift] \Rightarrow lift \quad ((- / \in -) [50, 51] 50)$
 $-liftNotMem :: [lift, lift] \Rightarrow lift \quad ((- / \notin -) [50, 51] 50)$
 $-liftFinset :: liftargs \Rightarrow lift \quad (\{-\})$

$-liftPair :: [lift, liftargs] \Rightarrow lift \quad ((1'(-, -')))$

$-liftCons :: [lift, lift] \Rightarrow lift \quad ((- \#/ -) [65,66] 65)$
 $-liftApp :: [lift, lift] \Rightarrow lift \quad ((- @/ -) [65,66] 65)$
 $-liftList :: liftargs \Rightarrow lift \quad ([(-)])$

$-ARAll :: [idts, lift] \Rightarrow lift \quad ((?! -./ -) [0, 10] 10)$
 $-AREx :: [idts, lift] \Rightarrow lift \quad ((? -./ -) [0, 10] 10)$
 $-AREx1 :: [idts, lift] \Rightarrow lift \quad ((?! -./ -) [0, 10] 10)$
 $-RAll :: [idts, lift] \Rightarrow lift \quad ((? \forall -./ -) [0, 10] 10)$
 $-REx :: [idts, lift] \Rightarrow lift \quad ((? \exists -./ -) [0, 10] 10)$
 $-REx1 :: [idts, lift] \Rightarrow lift \quad ((? \exists ! -./ -) [0, 10] 10)$

translations

$-const \quad \Rightarrow \quad \mathbf{CONST} \text{ const}$

translations

$-lift \quad \Rightarrow \quad \mathbf{CONST} \text{ lift}$
 $-lift2 \quad \Rightarrow \quad \mathbf{CONST} \text{ lift2}$
 $-lift3 \quad \Rightarrow \quad \mathbf{CONST} \text{ lift3}$
 $-lift4 \quad \Rightarrow \quad \mathbf{CONST} \text{ lift4}$
 $-Valid \quad \Rightarrow \quad \mathbf{CONST} \text{ Valid}$

translations

$$\begin{aligned}
-RAll\ x\ A &\quad \rightleftharpoons\ Rall\ x.\ A \\
-REx\ x\ A &\quad \rightleftharpoons\ Rex\ x.\ A \\
-REx1\ x\ A &\quad \rightleftharpoons\ Rex!\ x.\ A
\end{aligned}$$

translations

$$\begin{aligned}
-ARAll &\quad \rightarrow\ -RAll \\
-AREx &\quad \rightarrow\ -REx \\
-AREx1 &\quad \rightarrow\ -REx1
\end{aligned}$$

$$\begin{aligned}
w \models A &\quad \rightarrow\ A\ w \\
LIFT\ A &\quad \rightarrow\ A::\Rightarrow\ -
\end{aligned}$$

translations

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

translations

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

$$\begin{aligned}
w \models \neg A &\quad \leftarrow\ -liftNot\ A\ w \\
w \models A \wedge B &\quad \leftarrow\ -liftAnd\ A\ B\ w \\
w \models A \vee B &\quad \leftarrow\ -liftOr\ A\ B\ w \\
w \models A \longrightarrow B &\quad \leftarrow\ -liftImp\ A\ B\ w \\
w \models u = v &\quad \leftarrow\ -liftEqu\ u\ v\ w \\
w \models \forall x.\ A &\quad \leftarrow\ -RAll\ x\ A\ w \\
w \models \exists x.\ A &\quad \leftarrow\ -REx\ x\ A\ w \\
w \models \exists !x.\ A &\quad \leftarrow\ -REx1\ x\ A\ w
\end{aligned}$$

syntax (ASCII)

-Valid	:: lift \Rightarrow bool	((- -) 5)
-holdsAt	:: ['a, lift] \Rightarrow bool	((- = -) [100,10] 10)
-liftNeq	:: [lift, lift] \Rightarrow lift	((- \sim = / -) [50,51] 50)
-liftNot	:: lift \Rightarrow lift	((\sim -) [90] 90)
-liftAnd	:: [lift, lift] \Rightarrow lift	((- & / -) [36,35] 35)
-liftOr	:: [lift, lift] \Rightarrow lift	((- / -) [31,30] 30)
-liftImp	:: [lift, lift] \Rightarrow lift	((- --> / -) [26,25] 25)
-liftLeq	:: [lift, lift] \Rightarrow lift	((- / <= -) [50, 51] 50)
-liftMem	:: [lift, lift] \Rightarrow lift	((- / : -) [50, 51] 50)
-liftNotMem	:: [lift, lift] \Rightarrow lift	((- / \sim : -) [50, 51] 50)
-RAll	:: [idts, lift] \Rightarrow lift	((\exists ALL -./ -) [0, 10] 10)
-REx	:: [idts, lift] \Rightarrow lift	((\exists EX -./ -) [0, 10] 10)
-REx1	:: [idts, lift] \Rightarrow lift	((\exists EX! -./ -) [0, 10] 10)

2.3 Lemmas and Tactics

lemma *intD[dest]*: $\vdash A \Longrightarrow w \models A$

proof -

assume *a*: $\vdash A$

from *a* **have** $\forall w. w \models A$ **by** (*auto simp add: Valid-def*)

thus *?thesis* ..

qed

lemma *intI [intro!]*: **assumes** *P1*: $(\bigwedge w. w \models A)$ **shows** $\vdash A$

using *assms* **by** (*auto simp: Valid-def*)

Basic unlifting introduces a parameter *w* and applies basic rewrites, e.g $\vdash F = G$ becomes $F w = G w$ and $\vdash F \longrightarrow G$ becomes $F w \longrightarrow G w$.

method-setup *int-unlift* = \langle

Scan.succeed (*fn* *ctxt* => *SIMPLE-METHOD'*

 (*resolve-tac* *ctxt* @ $\{$ *thms intI* $\}$ *THEN'* *rewrite-goal-tac* *ctxt* @ $\{$ *thms intensional-rews* $\}$)

\rangle *method to unlift and followed by intensional rewrites*

lemma *inteq-reflection*: **assumes** *P1*: $\vdash x=y$ **shows** $(x \equiv y)$

proof -

from *P1* **have** *P2*: $\forall w. x w = y w$ **by** (*unfold Valid-def unl-lift2*)

hence *P3*: $x=y$ **by** *blast*

thus $x \equiv y$ **by** (*rule eq-reflection*)

qed

lemma *int-simps*:

$\vdash (x=x) = \#True$

$\vdash (\neg \#True) = \#False$

$\vdash (\neg \#False) = \#True$

$\vdash (\neg\neg P) = P$

$\vdash ((\neg P) = P) = \#False$

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

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

method-setup *int-rewrite* = \langle

Scan.succeed (fn ctxt => SIMPLE-METHOD' (rewrite-goal-tac ctxt @ {thms intensional-simps}))

\rangle *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 datatypes¹. 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

```

¹See 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* $F \equiv \lambda s. \forall n. (s \mid_s n) \models F$

definition *nexts* :: ('a::world) formula ⇒ 'a formula
where *nexts* $F \equiv \lambda s. (tail\ s) \models F$

definition *before* :: ('a::world,'b) stfun ⇒ ('a,'b) formfun
where *before* $f \equiv \lambda s. (first\ s) \models f$

definition *after* :: ('a::world,'b) stfun ⇒ ('a,'b) formfun
where *after* $f \equiv \lambda s. (second\ s) \models f$

definition *unch* :: ('a::world,'b) stfun ⇒ 'a formula
where *unch* $v \equiv \lambda s. s \models (after\ v) = (before\ v)$

definition *action* :: ('a::world) formula ⇒ ('a,'b) stfun ⇒ 'a formula
where *action* $P\ v \equiv \lambda s. \forall i. ((s \mid_s i) \models P) \vee ((s \mid_s i) \models unch\ v)$

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 \Rightarrow lift ($(\Box-)$ [90] 90)
 -nexts :: lift \Rightarrow lift ($(\text{Next } -)$ [90] 90)
 -action :: [lift, lift] \Rightarrow lift ($(\Box[-]'-(-))$ [20,1000] 90)

translations

-always \equiv *CONST* always
 -nexts \equiv *CONST* nexts
 -action \equiv *CONST* action
 -before \equiv *CONST* before
 -after \equiv *CONST* after
 -prime \rightarrow *CONST* after
 -unch \equiv *CONST* unch
 TEMP $F \rightarrow (F:: (\text{nat} \Rightarrow -) \Rightarrow -)$

3.3 Abbreviations

Some standard temporal abbreviations, with their concrete syntax.

definition *actrans* :: ('a::world) formula \Rightarrow ('a,'b) stfun \Rightarrow 'a formula
where *actrans* $P v \equiv$ TEMP($P \vee \text{unch } v$)

definition *eventually* :: ('a::world) formula \Rightarrow 'a formula
where *eventually* $F \equiv$ LIFT($\neg\Box(\neg F)$)

definition *angle-action* :: ('a::world) formula \Rightarrow ('a,'b) stfun \Rightarrow 'a formula
where *angle-action* $P v \equiv$ LIFT($\neg\Box[\neg P]-v$)

definition *angle-actrans* :: ('a::world) formula \Rightarrow ('a,'b) stfun \Rightarrow 'a formula
where *angle-actrans* $P v \equiv$ TEMP (\neg *actrans* (LIFT($\neg P$)) v)

definition *leadsto* :: ('a::world) formula \Rightarrow 'a formula \Rightarrow 'a formula
where *leadsto* $P Q \equiv$ LIFT $\Box(P \rightarrow \text{eventually } Q)$

3.3.1 Concrete Syntax

syntax (*ASCII*)

-actrans :: [lift, lift] \Rightarrow lift ($([-]'-(-))$ [20,1000] 90)
 -eventually :: lift \Rightarrow lift ($(\langle\langle\rangle\rangle-)$ [90] 90)
 -angle-action :: [lift, lift] \Rightarrow lift ($(\langle\langle\rangle\rangle\langle\rangle'-(-))$ [20,1000] 90)
 -angle-actrans :: [lift, lift] \Rightarrow lift ($(\langle\rangle\langle\rangle'-(-))$ [20,1000] 90)
 -leadsto :: [lift, lift] \Rightarrow lift ($(-\rightsquigarrow -)$ [26,25] 25)

syntax

-eventually :: lift \Rightarrow lift ($(\Diamond-)$ [90] 90)
 -angle-action :: [lift, lift] \Rightarrow lift ($(\langle\Diamond\rangle'-(-))$ [20,1000] 90)
 -angle-actrans :: [lift, lift] \Rightarrow lift ($(\langle\rangle'-(-))$ [20,1000] 90)
 -leadsto :: [lift, lift] \Rightarrow lift ($(-\rightsquigarrow -)$ [26,25] 25)

translations

-actrans \equiv *CONST actrans*
-eventually \equiv *CONST eventually*
-angle-action \equiv *CONST angle-action*
-angle-actrans \equiv *CONST angle-actrans*
-leadsto \equiv *CONST leadsto*

3.4 Properties of Operators

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

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

lemma *angle-action-defs*: $(w \models \diamond \langle P \rangle \cdot v) = (\exists i. ((w \mid_s i) \models P) \wedge ((w \mid_s i) \models v \$ \neq \$v))$
by (*simp add: angle-action-def action-def unch-def*)

lemma *unch-defs*: $(w \models \text{Unchanged } v) = (((\text{second } w) \models v) = ((\text{first } w) \models v))$
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*: $(w \mid_s a) \models \Box A$

shows $(w \mid_s b) \models \Box A$

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** $(w \mid_s b \mid_s n) \models A$ **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* :: $('a, 'b) \text{ formfun} \Rightarrow \text{bool}$

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: $\square[P]-v$. To show that $\square[P]-v$ is stuttering invariant, it must be shown that a slightly weaker predicate holds for P . For example, if P contains a term of the form $\circ\circ Q$, then it is not a well-formed pre-formula, thus $\square[P]-v$ is not stuttering invariant. This weaker version of stuttering invariance has been named *near stuttering invariance*.

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

syntax

$-stutinv :: \text{lift} \Rightarrow \text{bool} ((STUTINV -) [40] 40)$
 $-nstutinv :: \text{lift} \Rightarrow \text{bool} ((NSTUTINV -) [40] 40)$

translations

$-stutinv \Leftrightarrow \text{CONST } stutinv$
 $-nstutinv \Leftrightarrow \text{CONST } nstutinv$

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 $s t :: \text{nat} \Rightarrow 'a$

assume $a1: \text{first } s = \text{first } t$ **and** $a2: (\text{tail } s) \approx (\text{tail } t)$

have $s \approx t$

proof –

have $tg1: (\text{first } s) \#\# (\text{tail } s) = s$ **by** (*rule seq-app-first-tail*)

have $tg2: (\text{first } t) \#\# (\text{tail } t) = t$ **by** (*rule seq-app-first-tail*)

with $a1$ **have** $tg2': (\text{first } s) \#\# (\text{tail } t) = t$ **by** *simp*

from $a2$ **have** $(\text{first } s) \#\# (\text{tail } s) \approx (\text{first } s) \#\# (\text{tail } t)$ **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** $m: s \mid_s n \approx t \mid_s m$ **by** *blast*
from $a1$ **have** $(t \mid_s m) \models F$ **by** (*simp add: always-def*)
with $H\ m$ **show** $(s \mid_s n) \models F$ **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** $m: t \mid_s n \approx s \mid_s m$ **by** *blast*
from $a1$ **have** $(s \mid_s m) \models F$ **by** (*simp add: always-def*)
with $H\ m$ **show** $(t \mid_s n) \models F$ **by** (*simp add: stutinv-def*)
qed
qed
qed

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

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

fix n
assume $\neg ((s \mid_s n) \models \text{Unchanged } v)$
hence $v: v (s (Suc\ n)) \neq v (s\ n)$
by (*simp add: unch-defs first-def second-def suffix-def*)
from $st[\text{THEN } \text{sim-step}]$ **obtain** m **where**
 $a2': s \mid_s n \approx t \mid_s m$
 $\wedge (s \mid_s Suc\ n \approx t \mid_s Suc\ m \vee s \mid_s Suc\ n \approx t \mid_s m) \dots$
hence $g1: (s \mid_s n \approx t \mid_s m)$ **by** *simp*
hence $g1'': first (s \mid_s n) = first (t \mid_s m)$ **by** (*simp add: sim-first*)
hence $g1': s\ n = t\ m$ **by** (*simp add: suffix-def first-def*)
from $a2'$ **have** $g2: s \mid_s Suc\ n \approx t \mid_s Suc\ m \vee s \mid_s Suc\ n \approx t \mid_s m$ **by** *simp*
from P **have** $a1': ((t \mid_s m) \models P) \vee ((t \mid_s m) \models \text{Unchanged } v)$ **by** (*simp add: action-def*)
from $g2$ **show** $(s \mid_s n) \models P$
proof
assume $s \mid_s Suc\ n \approx t \mid_s m$
hence $first (s \mid_s Suc\ n) = first (t \mid_s m)$ **by** (*simp add: sim-first*)
hence $s (Suc\ n) = t\ m$ **by** (*simp add: suffix-def first-def*)
with $g1' v$ **show** *?thesis* **by** *simp* — by contradiction
next
assume $a3: s \mid_s Suc\ n \approx t \mid_s Suc\ m$
hence $first (s \mid_s Suc\ n) = first (t \mid_s Suc\ m)$ **by** (*simp add: sim-first*)
hence $a3': s (Suc\ n) = t (Suc\ m)$ **by** (*simp add: suffix-def first-def*)
from $a1'$ **show** *?thesis*
proof
assume $(t \mid_s m) \models \text{Unchanged } v$
hence $v (t (Suc\ m)) = v (t\ m)$
by (*simp add: unch-defs first-def second-def suffix-def*)
with $g1' a3' v$ **show** *?thesis* **by** *simp* — again, by contradiction
next
assume $a4: (t \mid_s m) \models P$
from $a3$ **have** $tail (s \mid_s n) \approx tail (t \mid_s m)$ **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** $H: NSTUTINV\ P$ **shows** $STUTINV\ \square[P]-v$

proof (*clarsimp simp: stutinv-def*)

fix $s\ t :: 'a\ seq$

assume $st: s \approx t$

show $(s \models \square[P]-v) = (t \models \square[P]-v)$

proof

assume $t \models \square[P]-v$

with $H\ st$ **show** $s \models \square[P]-v$ **by** (*rule stut-action-lemma*)

next

assume $s \models \square[P]-v$

with $H\ st[\text{THEN } \text{seqsim-sym}]$ **show** $t \models \square[P]-v$ **by** (*rule stut-action-lemma*)

qed

qed

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

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

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

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

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

lemma *stut-noteq*: $\llbracket STUTINV F; STUTINV G \rrbracket \Longrightarrow STUTINV (F \neq G)$
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*: $STUTINV X \Longrightarrow STUTINV (f \langle X \rangle)$
by (*simp add: stutinv-def*)

lemma *stut-fun2*: $\llbracket STUTINV X; STUTINV Y \rrbracket \Longrightarrow STUTINV (f \langle X, Y \rangle)$
by (*simp add: stutinv-def*)

lemma *stut-fun3*: $\llbracket STUTINV X; STUTINV Y; STUTINV Z \rrbracket \Longrightarrow STUTINV (f \langle X, Y, Z \rangle)$
by (*simp add: stutinv-def*)

lemma *stut-fun4*: $\llbracket STUTINV X; STUTINV Y; STUTINV Z; STUTINV W \rrbracket \Longrightarrow STUTINV (f \langle X, Y, Z, W \rangle)$
by (*simp add: stutinv-def*)

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

3.5.2 Properties of $-nstin$

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

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

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

lemma *nstut-and*: $\llbracket NSTUTINV F; NSTUTINV G \rrbracket \implies NSTUTINV (F \wedge G)$
by (*auto simp: nstin-def*)

lemma *nstut-or*: $\llbracket NSTUTINV F; NSTUTINV G \rrbracket \implies NSTUTINV (F \vee G)$
by (*auto simp: nstin-def*)

lemma *nstut-imp*: $\llbracket NSTUTINV F; NSTUTINV G \rrbracket \implies NSTUTINV (F \longrightarrow G)$
by (*auto simp: nstin-def*)

lemma *nstut-eq*: $\llbracket NSTUTINV F; NSTUTINV G \rrbracket \implies NSTUTINV (F = G)$
by (*force simp: nstin-def*)

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

lemma *nstut-noteq*: $\llbracket NSTUTINV F; NSTUTINV G \rrbracket \implies NSTUTINV (F \neq G)$
by (*simp add: nstut-eq nstut-not*)

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

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

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

lemma *nstut-fun1*: $NSTUTINV X \implies NSTUTINV (f \langle X \rangle)$
by (*force simp: nstin-def*)

lemma *nstut-fun2*: $\llbracket NSTUTINV X; NSTUTINV Y \rrbracket \implies NSTUTINV (f \langle X, Y \rangle)$
by (*force simp: nstin-def*)

lemma *nstut-fun3*: $\llbracket NSTUTINV X; NSTUTINV Y; NSTUTINV Z \rrbracket \implies NSTUTINV (f \langle X, Y, Z \rangle)$
by (*force simp: nstin-def*)

lemma *nstut-fun4*: $\llbracket NSTUTINV X; NSTUTINV Y; NSTUTINV Z; NSTUTINV W \rrbracket \implies NSTUTINV (f \langle X, Y, Z, W \rangle)$
by (*force simp: nstin-def*)

lemma *nstut-plus*: $\llbracket NSTUTINV\ x; NSTUTINV\ y \rrbracket \implies NSTUTINV\ (x+y)$
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* \implies *NSTUTINV [P]-v*

by (*simp add: actrans-def nstut-unch nstut-or*)

lemma *stut-eventually*: *STUTINV F* \implies *STUTINV $\Diamond F$*

by (*simp add: eventually-def stut-not stut-always*)

lemma *stut-leadsto*: $\llbracket STUTINV\ F; STUTINV\ G \rrbracket \implies STUTINV\ (F \rightsquigarrow G)$

by (*simp add: leadsto-def stut-always stut-eventually stut-imp*)

lemma *stut-angle-action*: *NSTUTINV P* \implies *STUTINV $\Diamond \langle P \rangle -v$*

by (*simp add: angle-action-def nstut-not stut-action stut-not*)

lemma *nstut-angle-actrans*: *NSTUTINV P* \implies *NSTUTINV $\langle P \rangle -v$*

by (*simp add: angle-actrans-def nstut-not nstut-actrans*)

lemmas *stutinv*s = *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 *nstutinv*s = *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-actrans stutinv-strictly-stronger

nstut-fun1 nstut-fun2 nstut-fun3 nstut-fun4 stutinv[*THEN stutinv-strictly-stronger*]

lemmas *bothstutinv*s = *stutinv*s *nstutinv*s

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  $\Rightarrow$  bool
where PreValid A  $\equiv \forall w. w \models A$ 
```

```
syntax
-PreValid    :: lift  $\Rightarrow$  bool    ((|\sim -) 5)
```

```
translations
-PreValid  $\equiv$  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 inten-
      sional-rews}))
> int-unlift for PreFormulas
```

```
lemma prefq-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: prefq-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** $\bigwedge w. w \models F \implies w \models G$
 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 $|\sim F \wedge \text{Unchanged } id \longrightarrow \bigcirc F$
proof (*pref-unlift*, *clarify*)
fix *s*
assume *a1*: $s \models F$ **and** *a2*: $s \models \text{Unchanged } id$
from *a2* **have** $(id \text{ (second } s) = id \text{ (first } s))$ **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 $\exists (v::'a::\text{world} \Rightarrow 'a). (|\sim F \wedge \text{Unchanged } v \longrightarrow \bigcirc F)$
using *pre-id-unch*[*OF h*] **by** *blast*

lemma *unch-pair*: $|\sim \text{Unchanged } (x,y) = (\text{Unchanged } x \wedge \text{Unchanged } y)$
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 pref-eq-reflection*]

lemma *angle-actrans-sem*: $|\sim \langle F \rangle \cdot v = (F \wedge v\$ \neq \$v)$
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: $|\sim f\langle x \rangle' = f\langle x' \rangle$
by (auto simp: nexts-def before-def after-def)

lemma after-fun2: $|\sim f\langle x, y \rangle' = f\langle x', y' \rangle$
by (auto simp: nexts-def before-def after-def)

lemma after-fun3: $|\sim f\langle x, y, z \rangle' = f\langle x', y', z' \rangle$
by (auto simp: nexts-def before-def after-def)

lemma after-fun4: $|\sim f\langle x, y, z, zz \rangle' = f\langle x', y', z', zz' \rangle$
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: $\vdash \$(f\langle x \rangle) = f\langle \$x \rangle$
by (auto simp: before-def)

lemma before-fun2: $\vdash \$(f\langle x, y \rangle) = f\langle \$x, \$y \rangle$
by (auto simp: before-def)

lemma before-fun3: $\vdash \$(f\langle x, y, z \rangle) = f\langle \$x, \$y, \$z \rangle$
by (auto simp: before-def)

lemma before-fun4: $\vdash \$(f\langle x, y, z, zz \rangle) = f\langle \$x, \$y, \$z, \$zz \rangle$
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*: $|\sim (\langle F \wedge G \rangle -v) = (\langle F \rangle -v \wedge \langle G \rangle -v)$
by (*auto simp: angle-actrans-def actrans-def unch-def*)

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

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

lemma *pref-eq-true*: $|\sim P \implies |\sim P = \#True$
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 *heterogeneous* 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 heterogeneous 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 $\vdash F$ and $\vdash F \longrightarrow G$ shows $\vdash G$
 using *assms[unlifted]* by *auto*

theorem pmp: assumes $|\sim F$ and $|\sim F \longrightarrow G$ shows $|\sim G$
 using *assms[unlifted]* by *auto*

theorem sq: assumes $|\sim P$ shows $\vdash \Box[P]-v$
 using *assms[unlifted]* by (*auto simp: action-def*)

theorem pre: assumes $\vdash F$ shows $|\sim F$
 using *assms* by *auto*

theorem nex: assumes $h1: \vdash F$ shows $|\sim \circ F$
 using *assms* by (*auto simp: nexts-def*)

theorem ax0: $\vdash \# \text{ True}$
 by *auto*

theorem ax1: $\vdash \Box F \longrightarrow F$
proof (*clarsimp simp: always-def*)
 fix w
 assume $\forall n. (w \mid_s n) \models F$
 hence $(w \mid_s 0) \models F$..
 thus $w \models F$ by *simp*
 qed

theorem ax2: $\vdash \Box F \longrightarrow \Box[\Box F]-v$
 by (*auto simp: always-def action-def suffix-plus*)

theorem ax3:
 assumes $H: |\sim F \wedge \text{Unchanged } v \longrightarrow \circ F$
 shows $\vdash \Box[F \longrightarrow \circ F]-v \longrightarrow (F \longrightarrow \Box F)$
proof (*clarsimp simp: always-def*)
 fix $w \ n$
 assume $a1: w \models \Box[F \longrightarrow \circ F]-v$ and $a2: w \models F$
 show $(w \mid_s n) \models F$
proof (*induct n*)
 from $a2$ show $(w \mid_s 0) \models F$ by *simp*
 next
 fix m
 assume $a3: (w \mid_s m) \models F$
 with $a1$ *H[unlifted]* show $(w \mid_s (\text{Suc } m)) \models F$
 by (*auto simp: nexts-def action-def tail-suffix-suc*)
 qed
 qed

theorem ax4: $\vdash \Box[P \longrightarrow Q]-v \longrightarrow (\Box[P]-v \longrightarrow \Box[Q]-v)$
 by (*force simp: action-def*)

```

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

theorem pax0:  $|\sim \# \text{True}$ 
  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)  $\models \text{Unchanged } v$ ) ..
  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:  $\neg ((w |_s i) \models \text{Unchanged } v)$ 
    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*: $\vdash (\forall x. \Box[F x]-v) = (\Box[(\forall x. F x)]-v)$
 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 $|\sim F \wedge \text{Unchanged } v \longrightarrow \bigcirc F$.

theorem *alw2*:
 assumes *h1*: $\vdash F$ and *h2*: $|\sim F \wedge \text{Unchanged } v \longrightarrow \bigcirc F$
 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*: $\vdash \Box[(F \longrightarrow \bigcirc F)]-v$ by (*rule sq*)
 from *h2* have $\vdash \Box[(F \longrightarrow \bigcirc F)]-v \longrightarrow F \longrightarrow \Box F$ 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 $|\sim F \wedge \text{Unchanged } id \longrightarrow \bigcirc F$ 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*: $(\vdash F \implies \vdash G) \implies \vdash \Box F \longrightarrow G$, which can only be proved by induction on the formula structure, in a deep embedding.

theorem *T1[simp-unl]*: $\vdash \Box \Box F = \Box F$

proof (*auto simp: always-def suffix-plus*)

fix $w \ n$

assume $\forall m \ k. (w \mid_s (k+m)) \models F$

hence $(w \mid_s (n+0)) \models F$ **by** *blast*

thus $(w \mid_s n) \models F$ **by** *simp*

qed

theorem *T2[simp-unl]*: $\vdash \Box \Box [P]-v = \Box [P]-v$

proof –

have $1: \mid \sim \Box [P]-v \longrightarrow \circ \Box [P]-v$ **using** *pax4* **by** *force*

hence $\vdash \Box [\Box [P]-v \longrightarrow \circ \Box [P]-v]-v$ **by** (*rule sq*)

moreover

have $\vdash \Box [\Box [P]-v \longrightarrow \circ \Box [P]-v]-v \longrightarrow \Box [P]-v \longrightarrow \Box \Box [P]-v$

by (*rule ax3*) (*auto elim: 1[unlift-rule]*)

moreover

have $\vdash \Box \Box [P]-v \longrightarrow \Box [P]-v$ **by** (*rule ax1*)

ultimately show *?thesis* **by** *force*

qed

theorem *T3[simp-unl]*: $\vdash \Box [[P]-v]-v = \Box [P]-v$

proof –

have $\mid \sim P \longrightarrow [P]-v$ **by** (*auto simp: actrans-def*)

hence $\vdash \Box [(P \longrightarrow [P]-v)]-v$ **by** (*rule sq*)

with *ax4* **have** $\vdash \Box [P]-v \longrightarrow \Box [[P]-v]-v$ **by** *force*

moreover

have $\mid \sim [P]-v \longrightarrow v \neq \$v \longrightarrow P$ **by** (*auto simp: unch-def actrans-def*)

hence $\vdash \Box [[P]-v \longrightarrow v \neq \$v \longrightarrow P]-v$ **by** (*rule sq*)

with *ax5* **have** $\vdash \Box [[P]-v]-v \longrightarrow \Box [P]-v$ **by** (*force intro: ax4[unlift-rule]*)

ultimately show *?thesis* **by** *force*

qed

theorem *M2*:

assumes $h: \mid \sim F \longrightarrow G$

shows $\vdash \Box [F]-v \longrightarrow \Box [G]-v$

using *sq[OF h]* *ax4* **by** *force*

theorem *N1*:

assumes $h: \vdash F \longrightarrow G$

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

theorem $T4: \vdash \Box[P]-v \longrightarrow \Box[[P]-v]-w$

proof –

have $\vdash \Box\Box[P]-v \longrightarrow \Box[\Box\Box[P]-v]-w$ by (rule ax2)

moreover

from pax4 have $|\sim \Box\Box[P]-v \longrightarrow [P]-v$ **unfolding** $T2[int-rewrite]$ **by force**

hence $\vdash \Box[\Box\Box[P]-v]-w \longrightarrow \Box[[P]-v]-w$ **by (rule M2)**

ultimately show *?thesis* **unfolding** $T2[int-rewrite]$ **by (rule lift-imp-trans)**

qed

theorem $T5: \vdash \Box[[P]-w]-v \longrightarrow \Box[[P]-v]-w$

proof –

have $|\sim [[P]-w]-v \longrightarrow [[P]-v]-w$ **by (auto simp: actrans-def)**

hence $\vdash \Box[[P]-w]-v \longrightarrow \Box[[P]-v]-w$ **by (rule M2)**

with $T4$ show *?thesis* **unfolding** $T3[int-rewrite]$ **by (rule lift-imp-trans)**

qed

theorem $T6: \vdash \Box F \longrightarrow \Box[\circ F]-v$

proof –

from ax1 have $|\sim \circ(\Box F \longrightarrow F)$ **by (rule nex)**

with pax2 have $|\sim \circ\Box F \longrightarrow \circ F$ **by force**

with pax3 have $|\sim \Box F \longrightarrow \circ F$ **by (rule pref-imp-trans)**

hence $\vdash \Box[\Box F]-v \longrightarrow \Box[\circ F]-v$ **by (rule M2)**

with ax2 show *?thesis* **by (rule lift-imp-trans)**

qed

theorem $T7:$

assumes $h: |\sim F \wedge \text{Unchanged } v \longrightarrow \circ F$

shows $|\sim (F \wedge \circ\Box F) = \Box F$

proof –

have $\vdash \Box[\circ F \longrightarrow F \longrightarrow \circ F]-v$ **by (rule sq) auto**

with ax4 have $\vdash \Box[\circ F]-v \longrightarrow \Box[(F \longrightarrow \circ F)]-v$ **by force**

with ax3[OF h, unlifted] have $\vdash \Box[\circ F]-v \longrightarrow (F \longrightarrow \Box F)$ **by force**

with pax5 have $|\sim F \wedge \circ\Box F \longrightarrow \Box F$ **by force**

with ax1[of TEMP F, unlifted] pax3[of TEMP F, unlifted] show *?thesis* **by force**

qed

theorem $T8: |\sim \circ(F \wedge G) = (\circ F \wedge \circ G)$

proof –

have $|\sim \circ(F \wedge G) \longrightarrow \circ F$ **by (rule N1) auto**

moreover

have $|\sim \circ(F \wedge G) \longrightarrow \circ G$ **by (rule N1) auto**

moreover

have $\vdash F \longrightarrow G \longrightarrow F \wedge G$ **by auto**

from nex[OF this] have $|\sim \circ F \longrightarrow \circ G \longrightarrow \circ(F \wedge G)$

by (force intro: pax2[unlift-rule])

ultimately show *?thesis* **by force**

qed

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

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

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

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

qed

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

qed

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

5.3 Some other useful derived theorems

theorem P1: $|\sim \Box F \longrightarrow \circ F$
proof –
have $|\sim \circ \Box F \longrightarrow \circ F$ by (*rule N1[OF ax1]*)
with *pax3* show *?thesis* by (*rule pref-imp-trans*)

qed

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

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

qed

theorem P5: $\vdash \Box[P]-v \longrightarrow \Box[\Box[P]-v]-w$

proof –

have $\vdash \Box\Box[P]-v \longrightarrow \Box[\Box[P]-v]-w$ **by** (rule P4)

thus *?thesis* **by** (unfold T2[int-rewrite])

qed

theorem M0: $\vdash \Box F \longrightarrow \Box[F \longrightarrow \circ F]-v$

proof –

from P1 **have** $|\sim \Box F \longrightarrow F \longrightarrow \circ F$ **by** force

hence $\vdash \Box[\Box F]-v \longrightarrow \Box[F \longrightarrow \circ F]-v$ **by** (rule M2)

with ax2 **show** *?thesis* **by** (rule lift-imp-trans)

qed

theorem M1: $\vdash \Box F \longrightarrow \Box[F \wedge \circ F]-v$

proof –

have $|\sim \Box F \longrightarrow F \wedge \circ F$ **by** (rule P2)

hence $\vdash \Box[\Box F]-v \longrightarrow \Box[F \wedge \circ F]-v$ **by** (rule M2)

with ax2 **show** *?thesis* **by** (rule lift-imp-trans)

qed

theorem M3: **assumes** $h: \vdash F$ **shows** $\vdash \Box[\circ F]-v$

using alw[OF h] T6 **by** (rule fmp)

lemma M4: $\vdash \Box[\circ(F \wedge G) = (\circ F \wedge \circ G)]-v$

by (rule sq[OF T8])

theorem M5: $\vdash \Box[\Box[P]-v \longrightarrow \circ\Box[P]-v]-w$

proof (rule sq)

show $|\sim \Box[P]-v \longrightarrow \circ\Box[P]-v$ **by** (auto simp: pax4[unlifted])

qed

theorem M6: $\vdash \Box[F \wedge G]-v \longrightarrow \Box[F]-v \wedge \Box[G]-v$

proof –

have $\vdash \Box[F \wedge G]-v \longrightarrow \Box[F]-v$ **by** (rule M2) *auto*

moreover

have $\vdash \Box[F \wedge G]-v \longrightarrow \Box[G]-v$ **by** (rule M2) *auto*

ultimately show *?thesis* **by** force

qed

theorem M7: $\vdash \Box[F]-v \wedge \Box[G]-v \longrightarrow \Box[F \wedge G]-v$

proof –

have $|\sim F \longrightarrow G \longrightarrow F \wedge G$ **by** *auto*

hence $\vdash \Box[F]-v \longrightarrow \Box[G \longrightarrow F \wedge G]-v$ **by** (rule M2)

with ax4 **show** *?thesis* **by** force

qed

theorem M8: $\vdash \Box[F \wedge G]-v = (\Box[F]-v \wedge \Box[G]-v)$

by (rule int-iffI[OF M6 M7])

theorem M9: $|\sim \Box F \longrightarrow F \wedge \circ \Box F$
using *pre*[*OF ax1*[*of F*]] *pax3*[*of F*] **by** *force*

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

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

theorem M12: $\vdash (\Box [A]-f \wedge \Box [B]-g) = \Box [[A]-f \wedge [B]-g]-(f,g)$

proof –
have $\vdash \Box [A]-f \wedge \Box [B]-g \longrightarrow \Box [[A]-f \wedge [B]-g]-(f,g)$
by (*auto simp: M8[int-rewrite] elim: T4[unlift-rule]*)
moreover
have $|\sim [[A]-f \wedge [B]-g]-(f,g) \longrightarrow [A]-f$
by (*auto simp: actrans-def unch-def all-before-eq all-after-eq*)
hence $\vdash \Box [[A]-f \wedge [B]-g]-(f,g) \longrightarrow \Box [A]-f$ **by** (*rule M11*)
moreover
have $|\sim [[A]-f \wedge [B]-g]-(f,g) \longrightarrow [B]-g$
by (*auto simp: actrans-def unch-def all-before-eq all-after-eq*)
hence $\vdash \Box [[A]-f \wedge [B]-g]-(f,g) \longrightarrow \Box [B]-g$
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 $\vdash ?F \Longrightarrow \vdash \Box ?F$ derived above.

lemmas *STL1* = *alw*

STL2 and STL3 have also already been derived.

lemmas *STL2* = *ax1*

lemmas *STL3* = *T1*

As with the derivation of $\vdash ?F \Longrightarrow \vdash \Box ?F$, 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*: $|\sim G \wedge \text{Unchanged } v \longrightarrow \circ G$
shows $\vdash \Box F \longrightarrow \Box G$

proof –

from $ax1[of\ F]$ $h1$ **have** $\vdash \Box F \longrightarrow G$ **by** (*rule lift-imp-trans*)

moreover

from $h1$ **have** $|\sim \circ F \longrightarrow \circ G$ **by** (*rule N1*)

hence $|\sim \circ F \longrightarrow G \longrightarrow \circ G$ **by** *auto*

hence $\vdash \Box[\circ F]-v \longrightarrow \Box[G \longrightarrow \circ G]-v$ **by** (*rule M2*)

with $T6$ **have** $\vdash \Box F \longrightarrow \Box[G \longrightarrow \circ G]-v$ **by** (*rule lift-imp-trans*)

moreover

from $h2$ **have** $\vdash \Box[G \longrightarrow \circ G]-v \longrightarrow G \longrightarrow \Box G$ **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 $h1\ h2[THEN\ pre-id-unch]$ **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 $h1: |\sim F \wedge Unchanged\ f \longrightarrow \circ F$

and $h2: |\sim G \wedge Unchanged\ g \longrightarrow \circ G$

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 $h3: |\sim (F \wedge G) \wedge Unchanged\ (f,g) \longrightarrow \circ(F \wedge G)$ **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** $\vdash \Box F \wedge \Box G \longrightarrow \Box[\Box F]-(f,g) \wedge \Box[\Box G]-(f,g)$ **by** *force*

with $M8$ **have** $\vdash \Box F \wedge \Box G \longrightarrow \Box[\Box F \wedge \Box G]-(f,g)$ **by** *force*

moreover

from $P1[of\ F]\ P1[of\ G]$ **have** $|\sim \Box F \wedge \Box G \longrightarrow F \wedge G \longrightarrow \circ(F \wedge G)$
unfolding $T8[int-rewrite]$ **by force**
hence $\vdash \Box[\Box F \wedge \Box G]-(f,g) \longrightarrow \Box[F \wedge G \longrightarrow \circ(F \wedge G)]-(f,g)$ **by** (rule $M2$)
from this $ax3[OF\ h3]$ **have** $\vdash \Box[\Box F \wedge \Box G]-(f,g) \longrightarrow F \wedge G \longrightarrow \Box(F \wedge G)$
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** $s \models \Box(F \wedge G) \implies P$
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 : $(w \upharpoonright_s nf) \models \Box F$ **by** (*auto simp: eventually-defs*)

from $a2$ **obtain** ng **where** ng : $(w \upharpoonright_s ng) \models \Box G$ **by** (*auto simp: eventually-defs*)

let $?n = \max\ nf\ ng$

have $nf \leq ?n$ **by** *simp*

from this nf **have** $(w \upharpoonright_s ?n) \models \Box F$ **by** (rule *linalw*)

moreover

have $ng \leq ?n$ **by** *simp*

from this ng **have** $(w \upharpoonright_s ?n) \models \Box G$ **by** (rule *linalw*)

ultimately

have $(w \upharpoonright_s ?n) \models \Box(F \wedge G)$ **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*: $\vdash F = G$ **shows** $\vdash \Box[F]-v = \Box[G]-v$
by (*auto simp: h[int-rewrite]*)

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

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

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

5.4 Theorems about the eventually operator

theorem dualization:
 $\vdash \neg \Box F = \Diamond \neg F$
 $\vdash \neg \Diamond F = \Box \neg F$
 $\vdash \neg \Box[A]-v = \Diamond \langle \neg A \rangle -v$
 $\vdash \neg \Diamond \langle A \rangle -v = \Box[\neg A]-v$
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*: $|\sim \neg F \wedge \text{Unchanged } v \longrightarrow \bigcirc \neg F$
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*: $\vdash \Diamond(\Box[F]-v \wedge \Box[G]-w) = (\Diamond\Box[F]-v \wedge \Diamond\Box[G]-w)$

proof –

have $\vdash (\Diamond\Box(\Box[F]-v \wedge \Box[G]-w)) = \Diamond(\Box\Box[F]-v \wedge \Box\Box[G]-w)$ **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: $\vdash \Diamond \Box Q \longrightarrow \Box [\Diamond Q]-v$
by (*rule lift-imp-trans[OF E20 P4]*)

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

lemma E26: $\vdash \Box \Diamond \langle A \rangle -v \longrightarrow \Box \Diamond A$
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*: $(s \models \Box[(\forall x. F x)]-v) = (\forall x. s \models \Box[(F x)]-v)$
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*: $\vdash (\exists x. \Diamond\langle F x \rangle -v) = \Diamond\langle (\exists x. F x) \rangle -v$

proof –

have $\vdash \neg(\exists x. \Diamond\langle F x \rangle -v) = \neg\Diamond\langle (\exists x. F x) \rangle -v$

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*: $\vdash F \rightsquigarrow F$

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** $\vdash F \rightsquigarrow G$

unfolding *leadsto-def* **by** (*rule alw[OF LT2[OF h]]*)

theorem *LT4*: $\vdash F \longrightarrow (F \rightsquigarrow G) \longrightarrow \Diamond G$

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*: $\vdash \Diamond F \longrightarrow (F \rightsquigarrow G) \longrightarrow \Diamond G$

unfolding *leadsto-def* **using** *LT5[of F G]* **by** *auto*

theorem *LT9[simp-unl]*: $\vdash \Box(F \rightsquigarrow G) = (F \rightsquigarrow G)$

by (*auto simp: leadsto-def*)

theorem *LT7*: $\vdash \Box\Diamond F \longrightarrow (F \rightsquigarrow G) \longrightarrow \Box\Diamond G$

proof –

have $\vdash \Box\Diamond F \longrightarrow \Box((F \rightsquigarrow G) \longrightarrow \Diamond G)$ **by** (*rule STL4[OF LT6]*)

from *lift-imp-trans[OF this MM0]* **show** *?thesis* **by** *simp*

qed

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

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

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

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

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

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

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

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

qed

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

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

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

theorem *LT21*: $\vdash ((\exists x. F x) \rightsquigarrow G) = (\forall x. (F x \rightsquigarrow G))$
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*: $\vdash (F \rightsquigarrow (G \vee H)) \longrightarrow \Box \neg G \longrightarrow (F \rightsquigarrow H)$
proof –
 have $\vdash \Box \neg G \longrightarrow (G \rightsquigarrow H)$ **unfolding** *leadsto-def* **by** (*rule STL4*) *auto*
 thus *?thesis* **by** (*force elim: LT20[unlift-rule]*)
qed

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

theorem *LT24*: $\vdash \Box I \longrightarrow ((P \wedge I) \rightsquigarrow Q) \longrightarrow P \rightsquigarrow Q$
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]*: $\vdash (F \rightsquigarrow \#False) = \Box \neg F$
unfolding *leadsto-def* **proof** (*rule MM1*)
 show $\vdash (F \longrightarrow \Diamond \#False) = \neg F$ **by** *simp*
qed

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

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

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

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

lemma *LT31*:
assumes $h1$: $|\sim P \wedge N \longrightarrow \circ P \vee \circ Q$ **and** $h2$: $|\sim P \wedge \text{Unchanged } v \longrightarrow \circ P$
shows $\vdash \Box N \longrightarrow P \longrightarrow \Box P \vee \Diamond Q$
proof –
from $h1$ [*THEN LT30*] **have** $|\sim N \longrightarrow \Box \neg Q \longrightarrow P \longrightarrow \circ P$ **unfolding** *eventually-def* **by** *auto*
hence $\vdash \Box[N \longrightarrow \Box \neg Q \longrightarrow P \longrightarrow \circ P]-v$ **by** (*rule sq*)
hence $\vdash \Box[N]-v \longrightarrow \Box[\Box \neg Q]-v \longrightarrow \Box[P \longrightarrow \circ P]-v$
by (*force intro: ax4 [unlift-rule]*)
with P_4 **have** $\vdash \Box N \longrightarrow \Box[\Box \neg Q]-v \longrightarrow \Box[P \longrightarrow \circ P]-v$ **by** (*rule lift-imp-trans*)
moreover
have $\vdash \neg \Diamond Q \longrightarrow \Box[\Box \neg Q]-v$ **unfolding** *dualization-rew* **by** (*rule ax2*)
moreover
note $ax3$ [*OF h2*]
ultimately
show *?thesis* **by** *force*
qed

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

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

lemma *AA2*: $\vdash \Box[P]-v \wedge \Diamond \langle Q \rangle -v \longrightarrow \Diamond \langle P \wedge Q \rangle -v$

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

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

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

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

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

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

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

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

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

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

lemma AA15: $\vdash \Diamond P \wedge Q-v \longrightarrow \Diamond P-v$
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: $|\sim P \longrightarrow \circ P \vee \circ Q$

and h2: $|\sim P \wedge A \longrightarrow \circ Q$

and h3: $|\sim P \wedge \text{Unchanged } w \longrightarrow \circ P$

shows $\vdash \Box(\Box P \longrightarrow \Diamond\langle A \rangle\text{-}v) \longrightarrow (P \rightsquigarrow Q)$

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: $|\sim \Diamond\langle \circ F \rangle\text{-}v \longrightarrow \circ \Diamond F$

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 $h1: |\sim P \wedge N \longrightarrow \circ P \vee \circ Q$
and $h2: |\sim P \wedge N \wedge \langle A \rangle\text{-}v \longrightarrow \circ Q$
and $h3: |\sim P \wedge \text{Unchanged } w \longrightarrow \circ P$
shows $\vdash \Box N \wedge \Box(\Box P \longrightarrow \Diamond \langle A \rangle\text{-}v) \longrightarrow (P \rightsquigarrow Q)$
proof –
from $h2$ **have** $|\sim \langle (N \wedge P) \wedge A \rangle\text{-}v \longrightarrow \circ Q$ **by** (*auto simp: angle-actrans-sem[int-rewrite]*)
from *pref-imp-trans[OF this E23]* **have** $\vdash \Diamond \langle (N \wedge P) \wedge A \rangle\text{-}v \longrightarrow \Diamond \langle \Diamond Q \rangle\text{-}v$
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 $|\sim P \wedge N \longrightarrow \circ P \vee \circ Q$
and $|\sim P \wedge N \wedge \langle A \rangle\text{-}v \longrightarrow \circ Q$
and $|\sim P \wedge \text{Unchanged } w \longrightarrow \circ P$
shows $\vdash \Box N \wedge \Box \langle A \rangle\text{-}v \longrightarrow (P \rightsquigarrow Q)$
proof –
have $\vdash \Box \langle A \rangle\text{-}v \longrightarrow \Box(\Box P \longrightarrow \Diamond \langle A \rangle\text{-}v)$ **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]*: $\vdash \Diamond\Diamond\langle A \rangle -v = \Diamond\langle A \rangle -v$
unfolding *eventually-def angle-action-def* **by** *simp*

theorem *AA29*: $\vdash \Box[N] -v \wedge \Box\Diamond\langle A \rangle -v \longrightarrow \Box\Diamond\langle N \wedge A \rangle -v$

proof –

have $\vdash \Box(\Box[N] -v \wedge \Diamond\langle A \rangle -v) \longrightarrow \Box\Diamond\langle N \wedge A \rangle -v$ **by** (*rule STL4[OF AA2]*)

thus *?thesis* **by** (*simp add: STL5[int-rewrite]*)

qed

theorem *AA30[simp-unl]*: $\vdash \Diamond\langle \Diamond\langle P \rangle -f \rangle -f = \Diamond\langle P \rangle -f$

unfolding *angle-action-def* **by** *simp*

theorem *AA31*: $\vdash \Diamond\langle \circ F \rangle -v \longrightarrow \Diamond F$

using *pref-imp-trans[OF AA21 E29]* **by** *auto*

lemma *AA32[simp-unl]*: $\vdash \Box\Diamond\Box[A] -v = \Diamond\Box[A] -v$

using *E21[of TEMP Box[A]-v]* **by** *simp*

lemma *AA33[simp-unl]*: $\vdash \Diamond\Box\Diamond\langle A \rangle -v = \Box\Diamond\langle A \rangle -v$

using *E27[of TEMP Diamond[A]-v]* **by** *simp*

5.6 Lemmas about the next operator

lemma *N2*: **assumes** $h: \vdash F = G$ **shows** $|\sim \circ F = \circ G$

by (*simp add: h[int-rewrite]*)

lemmas *next-and* = *T8*

lemma *next-or*: $|\sim \circ(F \vee G) = (\circ F \vee \circ G)$

proof (*rule pref-iffI*)

have $|\sim \circ((F \vee G) \wedge \neg F) \longrightarrow \circ G$ **by** (*rule N1*) *auto*

thus $|\sim \circ(F \vee G) \longrightarrow \circ F \vee \circ G$ **by** (*auto simp: T8[int-rewrite]*)

next

have $|\sim \circ F \longrightarrow \circ(F \vee G)$ **by** (*rule N1*) *auto*

moreover have $|\sim \circ G \longrightarrow \circ(F \vee G)$ **by** (*rule N1*) *auto*

ultimately show $|\sim \circ F \vee \circ G \longrightarrow \circ(F \vee G)$ **by** *force*

qed

lemma *next-imp*: $|\sim \circ(F \longrightarrow G) = (\circ F \longrightarrow \circ G)$

proof (*rule pref-iffI*)

have $|\sim \circ G \longrightarrow \circ(F \longrightarrow G)$ **by** (*rule N1*) *auto*

moreover have $|\sim \circ\neg F \longrightarrow \circ(F \longrightarrow G)$ **by** (*rule N1*) *auto*

ultimately show $|\sim (\circ F \longrightarrow \circ G) \longrightarrow \circ(F \longrightarrow G)$ **by** *force*

qed (*rule pax2*)

lemmas *next-not* = *pax1*

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

proof –
have $|\sim \circ(F = G) = \circ((F \longrightarrow G) \wedge (G \longrightarrow F))$ **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*: $|\sim \circ(F \neq G) = (\circ F \neq \circ G)$
by (*simp add: next-eq[int-rewrite]*)

lemma *next-const[simp-unl]*: $|\sim \circ\#P = \#P$
proof (*cases P*)

assume P
hence $1: \vdash \#P$ **by** *auto*
hence $|\sim \circ\#P$ **by** (rule *nex*)
with 1 **show** *?thesis* **by** *force*

next

assume $\neg P$
hence $1: \vdash \neg\#P$ **by** *auto*
hence $|\sim \circ\neg\#P$ **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*: $|\sim \circ f\langle x \rangle = f\langle \circ x \rangle$
by (*auto simp: nexts-def*)

lemma *next-fun2*: $|\sim \circ f\langle x, y \rangle = f\langle \circ x, \circ y \rangle$
by (*auto simp: nexts-def*)

lemma *next-fun3*: $|\sim \circ f\langle x, y, z \rangle = f\langle \circ x, \circ y, \circ z \rangle$
by (*auto simp: nexts-def*)

lemma *next-fun4*: $|\sim \circ f\langle x, y, z, zz \rangle = f\langle \circ x, \circ y, \circ z, \circ zz \rangle$
by (*auto simp: nexts-def*)

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

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

lemma *next-exists1*: $|\sim \circ(\exists! x. P x) = (\exists! x. \circ P x)$
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* = *par3*

lemma *t1*: $|\sim \circ \$x = x\$$
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 $|\sim \diamond F \longrightarrow \neg F \longrightarrow \circ \diamond F$
proof –
from *h* **have** *1*: *stutinv (TEMP $\neg F$)* **by** (*rule stut-not*)
have $|\sim \square \neg F = (\neg F \wedge \circ \square \neg F)$ **unfolding** *T7*[*OF pre-id-unch*[*OF 1*], *int-rewrite*]
by *simp*
thus *?thesis* **by** (*auto simp: eventually-def*)
qed

lemma *next-action*: $|\sim \square [P]-v \longrightarrow \circ \square [P]-v$
using *par4*[*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*: $|\sim P \wedge \text{Unchanged } f \longrightarrow \circ P$
shows $\vdash \square P = (P \wedge \square [P \longrightarrow \circ P]-f)$
proof (*rule int-iffI*)
from *ax1*[*of P*] *M0*[*of P f*] **show** $\vdash \square P \longrightarrow P \wedge \square [P \longrightarrow \circ P]-f$ **by** *force*
next
from *ax3*[*OF H*] **show** $\vdash P \wedge \square [P \longrightarrow \circ P]-f \longrightarrow \square P$ **by** *auto*
qed

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

qed

theorem INV1:

assumes $H: |\sim I \wedge [N]-f \longrightarrow \circ I$

shows $\vdash I \wedge \Box[N]-f \longrightarrow \Box I$

proof –

from H have $|\sim [N]-f \longrightarrow I \longrightarrow \circ I$ by *auto*

hence $\vdash \Box[[N]-f]-f \longrightarrow \Box[I \longrightarrow \circ I]-f$ by (rule *M2*)

moreover

from H have $|\sim I \wedge \text{Unchanged } f \longrightarrow \circ I$ by (auto simp: *actrans-def*)

hence $\vdash \Box[I \longrightarrow \circ I]-f \longrightarrow I \longrightarrow \Box I$ by (rule *ax3*)

ultimately show *?thesis* by *force*

qed

theorem INV2: $\vdash \Box I \longrightarrow \Box[N]-f = \Box[N \wedge I \wedge \circ I]-f$

proof –

from *M1*[*of I f*] have $\vdash \Box I \longrightarrow (\Box[N]-f = \Box[N]-f \wedge \Box[I \wedge \circ I]-f)$ by *auto*

thus *?thesis* by (auto simp: *M8*[*int-rewrite*])

qed

lemma R1:

assumes $H: |\sim \text{Unchanged } w \longrightarrow \text{Unchanged } v$

shows $\vdash \Box[F]-w \longrightarrow \Box[F]-v$

proof –

from H have $|\sim [F]-w \longrightarrow [F]-v$ by (auto simp: *actrans-def*)

thus *?thesis* by (rule *M11*)

qed

theorem invmono:

assumes $h1: \vdash I \longrightarrow P$

and $h2: |\sim P \wedge [N]-f \longrightarrow \circ P$

shows $\vdash I \wedge \Box[N]-f \longrightarrow \Box P$

using $h1$ *INV1*[*OF h2*] by *force*

theorem preimpsplit:

assumes $|\sim I \wedge N \longrightarrow Q$

and $|\sim I \wedge \text{Unchanged } v \longrightarrow Q$

shows $|\sim I \wedge [N]-v \longrightarrow Q$

using *assms*[*unlift-rule*] by (auto simp: *actrans-def*)

theorem refinement1:

assumes $h1: \vdash P \longrightarrow Q$

and $h2: |\sim I \wedge \circ I \wedge [A]-f \longrightarrow [B]-g$

shows $\vdash P \wedge \Box I \wedge \Box[A]-f \longrightarrow Q \wedge \Box[B]-g$

proof –

have $\vdash I \longrightarrow \# \text{True}$ by *simp*

from *this h2* have $\vdash \Box I \wedge \Box[A]-f \longrightarrow \Box \# \text{True} \wedge \Box[B]-g$ 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* \Rightarrow *'a formula*

where *enabled* $F \equiv \lambda s. \exists t. ((\text{first } s) \#\# t) \models F$

syntax *-Enabled* :: *lift* \Rightarrow *lift* ((*Enabled* -) [90] 90)

translations *-Enabled* \rightleftharpoons *CONST enabled*

definition *WeakF* :: (*'a::world*) *formula* \Rightarrow (*'a,'b*) *stfun* \Rightarrow *'a formula*

where *WeakF* $F v \equiv \text{TEMP } \Diamond \Box \text{Enabled } \langle F \rangle\text{-}v \longrightarrow \Box \Diamond \langle F \rangle\text{-}v$

definition *StrongF* :: (*'a::world*) *formula* \Rightarrow (*'a,'b*) *stfun* \Rightarrow *'a formula*

where *StrongF* $F v \equiv \text{TEMP } \Box \Diamond \text{Enabled } \langle F \rangle\text{-}v \longrightarrow \Box \Diamond \langle F \rangle\text{-}v$

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*] \Rightarrow *lift* ((*WF* '(-)'(-)) [20,1000] 90)

-*SF* :: [*lift, lift*] \Rightarrow *lift* ((*SF* '(-)'(-)) [20,1000] 90)

-*WFsp* :: [*lift, lift*] \Rightarrow *lift* ((*WF* '(-)'(-)) [20,1000] 90)

-*SFsp* :: [*lift, lift*] \Rightarrow *lift* ((*SF* '(-)'(-)) [20,1000] 90)

translations

- $WF \equiv CONST WeakF$
 - $SF \equiv CONST StrongF$
 - $WFsp \rightarrow CONST WeakF$
 - $SFsp \rightarrow CONST StrongF$

6.1 Properties of $-Enabled$

theorem *enabledI*: $\vdash F \rightarrow Enabled F$

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** $\bigwedge u. (first\ s \#\# u) \models F \implies Q$

shows Q

using *assms unfolding enabled-def by blast*

lemma *enabled-mono*:

assumes $w \models Enabled\ F$ **and** $\vdash F \rightarrow G$

shows $w \models Enabled\ G$

using *assms[unlifted] unfolding enabled-def by blast*

lemma *Enabled-disj1*: $\vdash Enabled\ F \rightarrow Enabled\ (F \vee G)$

by (*auto simp: enabled-def*)

lemma *Enabled-disj2*: $\vdash Enabled\ F \rightarrow Enabled\ (G \vee F)$

by (*auto simp: enabled-def*)

lemma *Enabled-conj1*: $\vdash Enabled\ (F \wedge G) \rightarrow Enabled\ F$

by (*auto simp: enabled-def*)

lemma *Enabled-conj2*: $\vdash Enabled\ (G \wedge F) \rightarrow Enabled\ F$

by (*auto simp: enabled-def*)

lemma *Enabled-disjD*: $\vdash Enabled\ (F \vee G) \rightarrow Enabled\ F \vee Enabled\ G$

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*: $\vdash WF(A)-v = (\Box\Diamond\neg Enabled \langle A \rangle -v \vee \Box\Diamond\langle A \rangle -v)$

proof –

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

qed

lemma *SF-alt*: $\vdash SF(A)-v = (\Diamond\Box\neg Enabled \langle A \rangle -v \vee \Box\Diamond\langle A \rangle -v)$

proof –

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

qed

lemma *alwaysWFI*: $\vdash WF(A)-v \longrightarrow \Box WF(A)-v$

unfolding *WF-alt*[*int-rewrite*] **by** (*rule MM6*)

theorem *WF-always*[*simp-unl*]: $\vdash \Box WF(A)-v = WF(A)-v$

by (*rule int-iffI[OF ax1 alwaysWFI]*)

theorem *WF-eventually*[*simp-unl*]: $\vdash \Diamond WF(A)-v = WF(A)-v$

proof –

have *1*: $\vdash \neg WF(A)-v = (\Diamond\Box Enabled \langle A \rangle -v \wedge \neg \Box\Diamond\langle A \rangle -v)$
by (*auto simp: WeakF-def*)

have $\vdash \Box\neg WF(A)-v = \neg WF(A)-v$

by (*simp add: 1[int-rewrite] STL5[int-rewrite] dualization-rew*)

thus *?thesis*

by (*auto simp: eventually-def*)

qed

lemma *alwaysSFI*: $\vdash SF(A)-v \longrightarrow \Box SF(A)-v$

proof –

have $\vdash \Box\Diamond\Box\neg Enabled \langle A \rangle -v \vee \Box\Diamond\langle A \rangle -v \longrightarrow \Box(\Box\Diamond\Box\neg Enabled \langle A \rangle -v \vee \Box\Diamond\langle A \rangle -v)$
by (*rule MM6*)

thus *?thesis* **unfolding** *SF-alt*[*int-rewrite*] **by** *simp*

qed

theorem *SF-always*[*simp-unl*]: $\vdash \Box SF(A)-v = SF(A)-v$

by (*rule int-iffI[OF ax1 alwaysSFI]*)

theorem *SF-eventually*[*simp-unl*]: $\vdash \Diamond SF(A)-v = SF(A)-v$

proof –

have *1*: $\vdash \neg SF(A)-v = (\Box\Diamond Enabled \langle A \rangle -v \wedge \neg \Box\Diamond\langle A \rangle -v)$
by (*auto simp: StrongF-def*)

have $\vdash \Box\neg SF(A)-v = \neg SF(A)-v$

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*: $\vdash \sim P \wedge N \longrightarrow \circ P \vee \circ Q$

and *h2*: $\vdash \sim P \wedge N \wedge \langle A \rangle -v \longrightarrow \circ Q$

and *h3*: $\vdash P \wedge N \longrightarrow Enabled \langle A \rangle -v$

and *h4*: $\vdash \sim P \wedge Unchanged w \longrightarrow \circ P$

shows $\vdash \Box N \wedge WF(A)-v \longrightarrow (P \rightsquigarrow Q)$

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*: $\vdash \sim P \wedge [N]-v \longrightarrow \circ P \vee \circ Q$

and *h2*: $\vdash \sim P \wedge \langle N \wedge A \rangle -v \longrightarrow \circ Q$

and *h3*: $\vdash P \longrightarrow Enabled \langle A \rangle -v$

and *h4*: $\vdash \sim P \wedge Unchanged v \longrightarrow \circ P$

shows $\vdash \Box [N]-v \wedge WF(A)-v \longrightarrow (P \rightsquigarrow Q)$

proof –

have $\vdash \Box \Box [N]-v \wedge WF(A)-v \longrightarrow (P \rightsquigarrow Q)$

proof (*rule WF1-general*)

from *h1 T9[of N v]* **show** $\vdash \sim P \wedge \Box [N]-v \longrightarrow \circ P \vee \circ Q$ **by force**

next

from *T9[of N v]* **have** $\vdash \sim P \wedge \Box [N]-v \wedge \langle A \rangle -v \longrightarrow P \wedge \langle N \wedge A \rangle -v$

by (*auto simp: actrans-def angle-actrans-def*)
 from *this h2* show $|\sim P \wedge \Box[N]-v \wedge \langle A \rangle -v \longrightarrow \circ Q$ by (*rule pref-imp-trans*)
 next
 from *h3 T9[of N v]* show $\vdash P \wedge \Box[N]-v \longrightarrow \text{Enabled } \langle A \rangle -v$ 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*: $|\sim P \wedge N \longrightarrow \circ P \vee \circ Q$
 and *h2*: $|\sim P \wedge N \wedge \langle A \rangle -v \longrightarrow \circ Q$
 and *h3*: $\vdash \Box P \wedge \Box N \wedge \Box F \longrightarrow \Diamond \text{Enabled } \langle A \rangle -v$
 and *h4*: $|\sim P \wedge \text{Unchanged } w \longrightarrow \circ P$
 shows $\vdash \Box N \wedge \text{SF}(A)-v \wedge \Box F \longrightarrow (P \rightsquigarrow Q)$
proof –
 have $\vdash \Box(\Box N \wedge \text{SF}(A)-v \wedge \Box F) \longrightarrow \Box(\Box P \longrightarrow \Diamond \langle A \rangle -v)$
proof (*rule STL4*)
 have $\vdash \Box(\Box P \wedge \Box N \wedge \Box F) \longrightarrow \Box \Diamond \text{Enabled } \langle A \rangle -v$ by (*rule STL4[OF h3]*)
 hence $\vdash \Box P \wedge \Box N \wedge \Box F \wedge \text{SF}(A)-v \longrightarrow \Box \Diamond \langle A \rangle -v$ by (*auto simp: StrongF-def STL5[int-rewrite]*)
 with *ax1[of TEMP $\Diamond \langle A \rangle -v$]* show $\vdash \Box N \wedge \text{SF}(A)-v \wedge \Box F \longrightarrow \Box P \longrightarrow \Diamond \langle A \rangle -v$
 by *force*
 qed
 hence $\vdash \Box N \wedge \text{SF}(A)-v \wedge \Box F \longrightarrow \Box(\Box P \longrightarrow \Diamond \langle A \rangle -v)$
 by (*simp add: STL5[int-rewrite]*)
 with *AA22[OF h1 h2 h4]* show ?thesis by *force*
 qed

theorem SF1:

assumes *h1*: $|\sim P \wedge [N]-v \longrightarrow \circ P \vee \circ Q$
 and *h2*: $|\sim P \wedge \langle N \wedge A \rangle -v \longrightarrow \circ Q$
 and *h3*: $\vdash \Box P \wedge \Box[N]-v \wedge \Box F \longrightarrow \Diamond \text{Enabled } \langle A \rangle -v$
 and *h4*: $|\sim P \wedge \text{Unchanged } v \longrightarrow \circ P$
 shows $\vdash \Box[N]-v \wedge \text{SF}(A)-v \wedge \Box F \longrightarrow (P \rightsquigarrow Q)$
proof –
 have $\vdash \Box \Box[N]-v \wedge \text{SF}(A)-v \wedge \Box F \longrightarrow (P \rightsquigarrow Q)$
proof (*rule SF1-general*)
 from *h1 T9[of N v]* show $|\sim P \wedge \Box[N]-v \longrightarrow \circ P \vee \circ Q$ 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 $|\sim P \wedge \Box[N]-v \wedge \langle A \rangle -v \longrightarrow \circ Q$ by (*rule pref-imp-trans*)
 next
 from *h3* show $\vdash \Box P \wedge \Box \Box[N]-v \wedge \Box F \longrightarrow \Diamond \text{Enabled } \langle A \rangle -v$ 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 $h2$: $|\sim P \wedge \circ P \wedge \langle N \wedge A \rangle\text{-}f \longrightarrow B$
and $h3$: $\vdash P \wedge \text{Enabled } \langle M \rangle\text{-}g \longrightarrow \text{Enabled } \langle A \rangle\text{-}f$
and $h4$: $\vdash \Box[N \wedge \neg B]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F \wedge \Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \longrightarrow \Diamond\Box P$
shows $\vdash \Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F \longrightarrow WF(M)\text{-}g$

proof –

have $\vdash \Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F \wedge \Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \neg\Box\Diamond\langle M \rangle\text{-}g \longrightarrow \Box\Diamond\langle M \rangle\text{-}g$

proof –

have 1 : $\vdash \Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F \wedge \Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \neg\Box\Diamond\langle M \rangle\text{-}g \longrightarrow \Diamond\Box P$

proof –

have A : $\vdash \Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F \wedge \Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \neg\Box\Diamond\langle M \rangle\text{-}g \longrightarrow \Box(\Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F) \wedge \Diamond\Box(\Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \Box[\neg M]\text{-}g)$

unfolding $STL6$ [*int-rewrite*]

by (*auto simp: STL5*[*int-rewrite*] *dualization-rew*)

have B : $\vdash \Box(\Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F) \wedge \Diamond\Box(\Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \Box[\neg M]\text{-}g)$

\longrightarrow

$\Diamond((\Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F) \wedge \Box(\Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \Box[\neg M]\text{-}g))$

by (*rule SE2*)

from *lift-imp-trans*[OF A B]

have $\vdash \Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F \wedge \Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \neg\Box\Diamond\langle M \rangle\text{-}g \longrightarrow \Diamond((\Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F) \wedge (\Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \Box[\neg M]\text{-}g))$

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 $\vdash \Box[[N]\text{-}f]\text{-}f \longrightarrow \Box[[\neg M]\text{-}g] \longrightarrow [N \wedge \neg B]\text{-}f$ **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** $\vdash (\Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F) \wedge (\Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \Box[\neg M]\text{-}g)$

$\longrightarrow \Diamond\Box P$

by *force*

from $STL4$ -*eve*[OF *this*]

have $\vdash \Diamond((\Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F) \wedge (\Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \Box[\neg M]\text{-}g)) \longrightarrow$

$\Diamond\Box P$ **by** *simp*

ultimately

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

qed

have 2 : $\vdash \Box[N]\text{-}f \wedge WF(A)\text{-}f \wedge \Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge \Diamond\Box P \longrightarrow \Box\Diamond\langle M \rangle\text{-}g$

proof –

have A : $\vdash \Diamond\Box P \wedge \Diamond\Box\text{Enabled } \langle M \rangle\text{-}g \wedge WF(A)\text{-}f \longrightarrow \Box\Diamond\langle A \rangle\text{-}f$

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** $\vdash \Box P \wedge \Box \langle N \wedge A \rangle\text{-}f \longrightarrow \Box \langle (P \wedge \circ P) \wedge (N \wedge A) \rangle\text{-}f$
by (*force intro: AA29[unlift-rule]*)
hence $\vdash \Box(\Box P \wedge \Box \langle N \wedge A \rangle\text{-}f) \longrightarrow \Box \Box \langle (P \wedge \circ P) \wedge (N \wedge A) \rangle\text{-}f$
by (*rule STL4-eve[OF STL4]*)
hence $\vdash \Box \Box P \wedge \Box \langle N \wedge A \rangle\text{-}f \longrightarrow \Box \langle (P \wedge \circ P) \wedge (N \wedge A) \rangle\text{-}f$
by (*simp add: STL6[int-rewrite]*)
with $AA29$ [of N f A]
have $B1$: $\vdash \Box[N]\text{-}f \wedge \Box P \wedge \Box \langle A \rangle\text{-}f \longrightarrow \Box \langle (P \wedge \circ P) \wedge (N \wedge A) \rangle\text{-}f$
by force
from $h2$ **have** $|\sim \langle (P \wedge \circ P) \wedge (N \wedge A) \rangle\text{-}f \longrightarrow \langle N \wedge B \rangle\text{-}f$
by (*auto simp: angle-actrans-sem[unlifted]*)
from $B1$ *this* [THEN $AA25$, THEN $STL4$] **have** $\vdash \Box[N]\text{-}f \wedge \Box P \wedge \Box \langle A \rangle\text{-}f \longrightarrow \Box \langle N \wedge B \rangle\text{-}f$
by (*rule lift-imp-trans*)
moreover
have $\vdash \Box \langle N \wedge B \rangle\text{-}f \longrightarrow \Box \langle M \rangle\text{-}g$ **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

Lampert 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 $h2$: $|\sim P \wedge \circ P \wedge \langle N \wedge A \rangle\text{-}f \longrightarrow B$
and $h3$: $\vdash P \wedge \text{Enabled } \langle M \rangle\text{-}g \longrightarrow \text{Enabled } \langle A \rangle\text{-}f$
and $h4$: $\vdash \Box[N \wedge \neg B]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box \langle \text{Enabled } \langle M \rangle\text{-}g \rangle \longrightarrow \Box P$
shows $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \longrightarrow SF(M)\text{-}g$

proof –

have $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box \langle \text{Enabled } \langle M \rangle\text{-}g \rangle \wedge \neg \Box \langle M \rangle\text{-}g \longrightarrow \Box \langle M \rangle\text{-}g$

proof –

have 1: $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box \langle \text{Enabled } \langle M \rangle\text{-}g \rangle \wedge \neg \Box \langle M \rangle\text{-}g \longrightarrow \Box P$

proof –

have A : $\vdash \Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box \langle \text{Enabled } \langle M \rangle\text{-}g \rangle \wedge \neg \Box \langle M \rangle\text{-}g \longrightarrow \Box(\Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F) \wedge \Box(\Box \langle \text{Enabled } \langle M \rangle\text{-}g \rangle \wedge \Box[\neg M]\text{-}g)$

unfolding $STL6$ [int-rewrite]

by (*auto simp: STL5[int-rewrite] dualization-rew*)

have B : $\vdash \Box(\Box[N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F) \wedge \Box(\Box \langle \text{Enabled } \langle M \rangle\text{-}g \rangle \wedge \Box[\neg M]\text{-}g)$

\longrightarrow

$\diamond((\Box[N]-f \wedge SF(A)-f \wedge \Box F) \wedge \Box(\Box\diamond Enabled \langle M \rangle-g \wedge \Box[\neg M]-g))$
by (rule SE2)
from lift-imp-trans[OF A B]
have $\vdash \Box[N]-f \wedge SF(A)-f \wedge \Box F \wedge \Box\diamond Enabled \langle M \rangle-g \wedge \neg\Box\diamond\langle M \rangle-g \longrightarrow$
 $\diamond((\Box[N]-f \wedge SF(A)-f \wedge \Box F) \wedge (\Box\diamond Enabled \langle M \rangle-g \wedge \Box[\neg M]-g))$
by (simp add: STL5[int-rewrite])
moreover
from h1 **have** $|\sim [N]-f \longrightarrow [\neg M]-g \longrightarrow [N \wedge \neg B]-f$ **by** (auto simp: actrans-def angle-actrans-def)
hence $\vdash \Box[[N]-f]-f \longrightarrow \Box[[\neg M]-g] \longrightarrow [N \wedge \neg B]-f$ **by** (rule M2)
from lift-imp-trans[OF this ax4] **have** $\vdash \Box[N]-f \wedge \Box[\neg M]-g \longrightarrow \Box[N \wedge \neg B]-f$
by (force intro: T4[unlift-rule])
with h4 **have** $\vdash (\Box[N]-f \wedge SF(A)-f \wedge \Box F) \wedge (\Box\diamond Enabled \langle M \rangle-g \wedge \Box[\neg M]-g)$
 $\longrightarrow \diamond\Box P$
by force
from STL4-eve[OF this]
have $\vdash \diamond((\Box[N]-f \wedge SF(A)-f \wedge \Box F) \wedge (\Box\diamond Enabled \langle M \rangle-g \wedge \Box[\neg M]-g)) \longrightarrow$
 $\diamond\Box P$ **by** simp
ultimately
show ?thesis **by** (rule lift-imp-trans)
qed
have 2: $\vdash \Box[N]-f \wedge SF(A)-f \wedge \Box\diamond Enabled \langle M \rangle-g \wedge \diamond\Box P \longrightarrow \Box\diamond\langle M \rangle-g$
proof –
have $\vdash \Box\diamond(P \wedge Enabled \langle M \rangle-g) \wedge SF(A)-f \longrightarrow \Box\diamond\langle A \rangle-f$
using h3[THEN STL4-eve, THEN STL4] **by** (auto simp: StrongF-def)
with E28 **have** A: $\vdash \diamond\Box P \wedge \Box\diamond Enabled \langle M \rangle-g \wedge SF(A)-f \longrightarrow \Box\diamond\langle A \rangle-f$
by force
have B: $\vdash \Box[N]-f \wedge \diamond\Box P \wedge \Box\diamond\langle A \rangle-f \longrightarrow \Box\diamond\langle M \rangle-g$
proof –
from M1[of P f] **have** $\vdash \Box P \wedge \Box\diamond\langle N \wedge A \rangle-f \longrightarrow \Box\diamond((P \wedge \circ P) \wedge (N \wedge A))-f$
by (force intro: AA29[unlift-rule])
hence $\vdash \diamond\Box(\Box P \wedge \Box\diamond\langle N \wedge A \rangle-f) \longrightarrow \diamond\Box\diamond\langle (P \wedge \circ P) \wedge (N \wedge A) \rangle-f$
by (rule STL4-eve[OF STL4])
hence $\vdash \diamond\Box P \wedge \Box\diamond\langle N \wedge A \rangle-f \longrightarrow \Box\diamond\langle (P \wedge \circ P) \wedge (N \wedge A) \rangle-f$
by (simp add: STL6[int-rewrite])
with AA29[of N f A]
have B1: $\vdash \Box[N]-f \wedge \diamond\Box P \wedge \Box\diamond\langle A \rangle-f \longrightarrow \Box\diamond\langle (P \wedge \circ P) \wedge (N \wedge A) \rangle-f$
by force
from h2 **have** $|\sim \langle (P \wedge \circ P) \wedge (N \wedge A) \rangle-f \longrightarrow \langle N \wedge B \rangle-f$
by (auto simp: angle-actrans-sem[unlifted])
from B1 this[THEN AA25, THEN STL4] **have** $\vdash \Box[N]-f \wedge \diamond\Box P \wedge \Box\diamond\langle A \rangle-f$
 $\longrightarrow \Box\diamond\langle N \wedge B \rangle-f$
by (rule lift-imp-trans)
moreover
have $\vdash \Box\diamond\langle N \wedge B \rangle-f \longrightarrow \Box\diamond\langle M \rangle-g$ **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:  $\bigwedge x. \vdash F x \rightsquigarrow (G \vee (\exists y. \#((y,x) \in r) \wedge F y))$ 
  shows  $\vdash F x \rightsquigarrow G$ 
using h1
proof (rule wf-induct)
  fix x
  assume ih:  $\forall y. (y, x) \in r \longrightarrow (\vdash F y \rightsquigarrow G)$ 
  show  $\vdash F x \rightsquigarrow G$ 
  proof -
    from ih have  $\vdash (\exists y. \#((y,x) \in r) \wedge F y) \rightsquigarrow G$ 
      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  $\langle F \rangle$ -v
  by (auto simp: enabled-def stutinv-def dest!: sim-first)

```

```

theorem stut-WF: NSTUTINV F  $\implies$  STUTINV WF(F)-v
  by (auto simp: WeakF-def stut-Enabled bothstutinv)

```

```

theorem stut-SF: NSTUTINV F  $\implies$  STUTINV SF(F)-v
  by (auto simp: StrongF-def stut-Enabled bothstutinv)

```

```

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

```

```

end

```

7 Representing state in TLA*

```

theory State
imports Liveness
begin

```

We adopt the hidden state approach, 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:  $\bigwedge x. v\ x = c \implies Q$ 
  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:  $\bigwedge x. v\ (\text{first } x) = c \implies Q$ 
  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  $c \in \text{range } x$  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  $d \in \text{range } y$  by auto
qed

```

```

lemma base-pair: basevars (x,y)  $\implies$  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 $x::\text{unit}$.

```

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 satisfying 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:  $\bigwedge u. vs\ (first\ u) = c \implies ((first\ s)\ \#\#\ u) \models F$ 
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 equivalence up to a variable, requires state equivalence 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  $\Rightarrow$  temporal)  $\Rightarrow$  temporal      (binder Eex 10)
AAll     :: ('a statefun  $\Rightarrow$  temporal)  $\Rightarrow$  temporal      (binder Aall 10)

```

syntax

```

-EEx     :: [idts, lift]  $\Rightarrow$  lift                       (( $\exists\exists\exists$  -./ -) [0,10] 10)
-AAll    :: [idts, lift]  $\Rightarrow$  lift                       (( $\exists\forall\forall$  -./ -) [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 $eexE: \llbracket s \models (\exists\exists\ x.\ F\ x) ; \text{basevars}\ vs; (!\ x.\ \llbracket \text{basevars}\ (x,vs); s \models F\ x \rrbracket \implies s \models G) \rrbracket \implies (s \models G)$
and $all\text{-def}: \vdash (\forall\forall\ x.\ F\ x) = (\neg(\exists\exists\ x.\ \neg(F\ x)))$
and $eexSTUT: STUTINV\ F\ x \implies STUTINV\ (\exists\exists\ x.\ F\ x)$
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 $eexI\text{-}unl = eexI[\text{unlift-rule}] \text{--- } w \models F\ x \implies w \models (\exists\exists\ x.\ F\ x)$

$tl\text{-}defs$ can be used to unfold TLA definitions into lowest predicate level. This is particularly useful for reasoning about enabledness of formulas.

lemmas $tl\text{-}defs = \text{unch-def before-def after-def first-def second-def suffix-def tail-def nexts-def app-def angle-actrans-def actrans-def}$

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

$Even :: \text{nat set}$
where
 $even\text{-zero}: 0 \in Even$
 $| even\text{-step}: n \in Even \implies Suc\ (Suc\ n) \in Even$

locale *Program* =

fixes $x :: \text{state} \Rightarrow \text{nat}$
and $init :: \text{temporal}$
and $act :: \text{temporal}$
and $phi :: \text{temporal}$
defines $init \equiv TEMP\ \$x = \# 0$
and $act \equiv TEMP\ x' = Suc\ <Suc\ <\$x\ >\ >$
and $phi \equiv TEMP\ init \wedge \Box[act]\text{-}x$

```

lemma (in Program) stutinvprog: STUTINV phi
  by (auto simp: phi-def init-def act-def stutinv nstutinv)

lemma (in Program) inveven:  $\vdash \text{phi} \longrightarrow \Box(\$x \in \# \text{Even})$ 
  unfolding phi-def
proof (rule invmono)
  show  $\vdash \text{init} \longrightarrow \$x \in \# \text{Even}$ 
    by (auto simp: init-def even-zero)
next
  show  $\sim \$x \in \# \text{Even} \wedge [\text{act}]\text{-}x \longrightarrow \circ(\$x \in \# \text{Even})$ 
    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  $x' = \text{Suc}\langle \$x \rangle \wedge y' = \$y$ 
  and m2  $\equiv$  TEMP  $y' = \text{Suc}\langle \$y \rangle \wedge x' = \$x$ 
  and Live  $\equiv$  TEMP  $WF(m1)\text{-}(x,y) \wedge WF(m2)\text{-}(x,y)$ 
  and phi  $\equiv$  TEMP  $(\text{init} \wedge \Box[m1 \vee m2]\text{-}(x,y) \wedge \text{Live})$ 
  assumes bvar: basevars  $(x,y)$ 

```

```

lemma (in Firstprogram) STUTINV phi
  by (auto simp: phi-def init-def m1-def m2-def Live-def stutinv nstutinv lives-
tutinv)

```

```

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

```


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

lemma (in *Firstprogram*) *enabled-m2*: $\vdash \text{Enabled } \langle m2 \rangle \text{-}(x,y)$

proof (*clarify*)

fix *s*

show $s \models \text{Enabled } \langle m2 \rangle \text{-}(x,y)$

by (rule base-enabled[OF bvar]) (auto simp: m2-def tla-defs)

qed

locale *Secondprogram* = *Firstprogram* +

fixes *sem* :: *state* \Rightarrow *nat*

and *pc1* :: *state* \Rightarrow *pcount*

and *pc2* :: *state* \Rightarrow *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* \equiv *LIFT* (*x,y,sem,pc1,pc2*)

and *initPsi* \equiv *TEMP* $\$pc1 = \# a \wedge \$pc2 = \# a \wedge \$x = \# 0 \wedge \$y = \# 0 \wedge$
 $\$sem = \# 1$

and *alpha1* \equiv *TEMP* $\$pc1 = \# a \wedge \# 0 < \$sem \wedge pc1\$ = \# b \wedge sem\$ = \sem
 $- \# 1 \wedge \text{Unchanged } (x,y,pc2)$

and *alpha2* \equiv *TEMP* $\$pc2 = \# a \wedge \# 0 < \$sem \wedge pc2' = \# b \wedge sem\$ = \sem
 $- \# 1 \wedge \text{Unchanged } (x,y,pc1)$

and *beta1* \equiv *TEMP* $\$pc1 = \# b \wedge pc1' = \# g \wedge x' = \text{Suc}\langle \$x \rangle \wedge \text{Unchanged}$
 $(y,sem,pc2)$

and *beta2* \equiv *TEMP* $\$pc2 = \# b \wedge pc2' = \# g \wedge y' = \text{Suc}\langle \$y \rangle \wedge \text{Unchanged}$
 $(x,sem,pc1)$

and *gamma1* \equiv *TEMP* $\$pc1 = \# g \wedge pc1' = \# a \wedge sem' = \text{Suc}\langle \$sem \rangle \wedge \text{Un-$
 $\text{changed } (x,y,pc2)$

and *gamma2* \equiv *TEMP* $\$pc2 = \# g \wedge pc2' = \# a \wedge sem' = \text{Suc}\langle \$sem \rangle \wedge \text{Un-$
 $\text{changed } (x,y,pc1)$

and *n1* \equiv *TEMP* (*alpha1* \vee *beta1* \vee *gamma1*)

and *n2* \equiv *TEMP* (*alpha2* \vee *beta2* \vee *gamma2*)

and *Live2* \equiv *TEMP* *SF*(*n1*)-*vars* \wedge *SF*(*n2*)-*vars*

and *psi* \equiv *TEMP* (*initPsi* \wedge \Box [*n1* \vee *n2*]-*vars* \wedge *Live2*)

and *I* \equiv *TEMP* ($\$sem = \# 1 \wedge \$pc1 = \# a \wedge \$pc2 = \# a$)

$\vee (\$sem = \# 0 \wedge ((\$pc1 = \# a \wedge \$pc2 \in \{\# b, \# g\})$

$\vee (\$pc2 = \# a \wedge \$pc1 \in \{\# b, \# g\})))$

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 $step: |\sim I \wedge [n1 \vee n2]\text{-vars} \longrightarrow \bigcirc I$ **by** (*force simp: actrans-def*)

from *init step* **have** $goal: \vdash initPsi \wedge \Box [n1 \vee n2]\text{-vars} \longrightarrow \Box I$ **by** (*rule invmono*)

have $\vdash initPsi \wedge \Box [n1 \vee n2]\text{-vars} \wedge Live2 \implies \vdash initPsi \wedge \Box [n1 \vee n2]\text{-vars}$

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*: $\vdash psi \longrightarrow init \wedge \Box [m1 \vee m2]\text{-(}x,y)$

proof –

have $\vdash initPsi \wedge \Box I \wedge \Box [n1 \vee n2]\text{-vars} \longrightarrow init \wedge \Box [m1 \vee m2]\text{-(}x,y)$

proof (*rule refinement1*)

show $\vdash initPsi \longrightarrow init$ **by** (*auto simp: initPsi-def init-def*)

next

show $|\sim I \wedge \bigcirc I \wedge [n1 \vee n2]\text{-vars} \longrightarrow [m1 \vee m2]\text{-(}x,y)$

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*: $|\sim \langle n1 \rangle\text{-vars} = n1$

proof –

have $|\sim n1 \longrightarrow \langle n1 \rangle\text{-vars}$ **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  $s :: \text{state seq}$ 
assume  $pc1 (s\ 0) = a$  and  $0 < sem (s\ 0)$ 
thus  $s \models \text{Enabled } \alpha1$ 
  by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)
qed

```

```

lemma (in Secondprogram) enab-beta1:  $\vdash \$pc1 = \#b \longrightarrow \text{Enabled } \beta1$ 
proof (clarsimp simp: tla-defs)
  fix  $s :: \text{state seq}$ 
  assume  $pc1 (s\ 0) = b$ 
  thus  $s \models \text{Enabled } \beta1$ 
    by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)
qed

```

```

lemma (in Secondprogram) enab-gamma1:  $\vdash \$pc1 = \#g \longrightarrow \text{Enabled } \gamma1$ 
proof (clarsimp simp: tla-defs)
  fix  $s :: \text{state seq}$ 
  assume  $pc1 (s\ 0) = g$ 
  thus  $s \models \text{Enabled } \gamma1$ 
    by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)
qed

```

```

lemma (in Secondprogram) enab-n1:
   $\vdash \text{Enabled } \langle n1 \rangle\text{-vars} = (\$pc1 = \#a \longrightarrow \# 0 < \$sem)$ 
unfolding n1-ch[int-rewrite] proof (rule int-iffI)
  show  $\vdash \text{Enabled } n1 \longrightarrow \$pc1 = \#a \longrightarrow \# 0 < \$sem$ 
    by (auto elim!: enabledE simp: Sact2-defs tla-defs)
next
  show  $\vdash (\$pc1 = \#a \longrightarrow \# 0 < \$sem) \longrightarrow \text{Enabled } n1$ 
  proof (clarsimp simp: tla-defs)
    fix  $s :: \text{state seq}$ 
    assume  $pc1 (s\ 0) = a \longrightarrow 0 < sem (s\ 0)$ 
    thus  $s \models \text{Enabled } n1$ 
      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:  $|\sim \langle n2 \rangle\text{-vars} = n2$ 
proof -
  have  $|\sim n2 \longrightarrow \langle n2 \rangle\text{-vars}$  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:  $\vdash \$pc2 = \#a \longrightarrow \# 0 < \$sem \longrightarrow$ 

```

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

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

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

lemma (*in Secondprogram*) *enab-n2*:
 $\vdash \text{Enabled } \langle n2 \rangle\text{-vars} = (\$pc2 = \#a \longrightarrow \# 0 < \$sem)$
unfolding *n2-ch[int-rewrite]* **proof** (*rule int-iffI*)
show $\vdash \text{Enabled } n2 \longrightarrow \$pc2 = \#a \longrightarrow \# 0 < \sem
by (*auto elim!: enabledE simp: Sact2-defs tla-defs*)
next
show $\vdash (\$pc2 = \#a \longrightarrow \# 0 < \$sem) \longrightarrow \text{Enabled } n2$
proof (*clarsimp simp: tla-defs*)
fix $s :: \text{state seq}$
assume $pc2 (s 0) = a \longrightarrow 0 < sem (s 0)$
thus $s \models \text{Enabled } n2$
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*: $\vdash \text{psi} \longrightarrow \text{SF}(m1)\text{-(}x,y\text{)}$
proof –
have $\vdash \square[n1 \vee n2]\text{-vars} \wedge \text{SF}(n1)\text{-vars} \wedge \square(I \wedge \text{SF}(n2)\text{-vars}) \longrightarrow \text{SF}(m1)\text{-(}x,y\text{)}$
proof (*rule SF2*)

Rule *SF2* requires us to choose a helpful action (whose effect implies $\langle m1 \rangle\text{-}(x,y)$) 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  $\mid\sim \langle (n1 \vee n2) \wedge beta1 \rangle\text{-vars} \longrightarrow \langle m1 \rangle\text{-}(x,y)$ 
  by (auto simp: beta1-def m1-def vars-def tla-defs)
next
show  $\mid\sim \$pc1 = \#b \wedge \circ(\$pc1 = \#b) \wedge \langle (n1 \vee n2) \wedge n1 \rangle\text{-vars} \longrightarrow beta1$ 
  by (auto simp: n1-def alpha1-def beta1-def gamma1-def tla-defs)
next
show  $\vdash \$pc1 = \#b \wedge Enabled \langle m1 \rangle\text{-}(x, y) \longrightarrow Enabled \langle n1 \rangle\text{-vars}$ 
  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  $\vdash \square[(n1 \vee n2) \wedge \neg beta1]\text{-vars}$ 
   $\wedge SF(n1)\text{-vars} \wedge \square(I \wedge SF(n2)\text{-vars}) \wedge \square\Diamond Enabled \langle m1 \rangle\text{-}(x, y)$ 
   $\longrightarrow \Diamond\square(\$pc1 = \#b)$ 
proof -
have  $\vdash \square\square[(n1 \vee n2) \wedge \neg beta1]\text{-vars} \longrightarrow \square(\$pc1 = \#b \longrightarrow \square(\$pc1 = \#b))$ 
proof (rule STL4)
  have  $\mid\sim \$pc1 = \#b \wedge [(n1 \vee n2) \wedge \neg beta1]\text{-vars} \longrightarrow \circ(\$pc1 = \#b)$ 
  by (auto simp: Sact2-defs vars-def tla-defs)
  from this[THEN INV1]
  show  $\vdash \square[(n1 \vee n2) \wedge \neg beta1]\text{-vars} \longrightarrow \$pc1 = \#b \longrightarrow \square(\$pc1 = \#b)$ 
by auto
qed
hence  $1: \vdash \square[(n1 \vee n2) \wedge \neg beta1]\text{-vars} \longrightarrow \Diamond(\$pc1 = \#b) \longrightarrow \Diamond\square(\$pc1 = \#b)$ 
by (force intro: E31[unlift-rule])
have  $\vdash \square[(n1 \vee n2) \wedge \neg beta1]\text{-vars} \wedge SF(n1)\text{-vars} \wedge \square(I \wedge SF(n2)\text{-vars})$ 
   $\longrightarrow \Diamond(\$pc1 = \#b)$ 
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  $?F = LIFT (\square[(n1 \vee n2) \wedge \neg beta1]\text{-vars} \wedge SF(n1)\text{-vars} \wedge \square(I \wedge SF(n2)\text{-vars}))$ 

```

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

```

have  $ga: \vdash ?F \longrightarrow (\$pc1 = \#g \rightsquigarrow \$pc1 = \#a)$ 
proof (rule SF1)
  show  $\mid\sim \$pc1 = \#g \wedge [(n1 \vee n2) \wedge \neg beta1]\text{-vars} \longrightarrow \circ(\$pc1 = \#g) \vee \circ(\$pc1 = \#a)$ 

```

```

      by (auto simp: Sact2-defs vars-def tla-defs)
    next
      show  $\sim \$pc1 = \#g \wedge (((n1 \vee n2) \wedge \neg beta1) \wedge n1)$ -vars  $\longrightarrow \circ(\$pc1 = \#a)$ 
      by (auto simp: Sact2-defs vars-def tla-defs)
    next
      show  $\sim \$pc1 = \#g \wedge Unchanged\ vars \longrightarrow \circ(\$pc1 = \#g)$ 
      by (auto simp: vars-def tla-defs)
    next
      have  $\vdash \$pc1 = \#g \longrightarrow Enabled \langle n1 \rangle$ -vars
      unfolding enab-n1[int-rewrite] by (auto simp: tla-defs)
      hence  $\vdash \square(\$pc1 = \#g) \longrightarrow Enabled \langle n1 \rangle$ -vars
      by (rule lift-imp-trans[OF ax1])
      hence  $\vdash \square(\$pc1 = \#g) \longrightarrow \diamond Enabled \langle n1 \rangle$ -vars
      by (rule lift-imp-trans[OF E3])
    thus  $\vdash \square(\$pc1 = \#g) \wedge \square[(n1 \vee n2) \wedge \neg beta1]$ -vars  $\wedge \square(I \wedge SF(n2))$ -vars
       $\longrightarrow \diamond Enabled \langle n1 \rangle$ -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:  $\vdash ?F \longrightarrow (\$pc1 = \#a \rightsquigarrow \$pc1 = \#b)$ 
      proof (rule SF1)
        show  $\sim \$pc1 = \#a \wedge [(n1 \vee n2) \wedge \neg beta1]$ -vars  $\longrightarrow \circ(\$pc1 = \#a) \vee \circ(\$pc1 = \#b)$ 
        by (auto simp: Sact2-defs vars-def tla-defs)
      next
        show  $\sim \$pc1 = \#a \wedge (((n1 \vee n2) \wedge \neg beta1) \wedge n1)$ -vars  $\longrightarrow \circ(\$pc1 = \#b)$ 
        by (auto simp: Sact2-defs vars-def tla-defs)
      next
        show  $\sim \$pc1 = \#a \wedge Unchanged\ vars \longrightarrow \circ(\$pc1 = \#a)$ 
        by (auto simp: vars-def tla-defs)
      next

```

We establish a suitable leadsto-chain.

```

      let  $?G = LIFT \square[(n1 \vee n2) \wedge \neg beta1]$ -vars  $\wedge SF(n2)$ -vars  $\wedge \square(\$pc1 = \#a \wedge I)$ 
      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:  $\vdash ?G \longrightarrow (\$pc2 = \#b \rightsquigarrow \$pc2 = \#g)$ 
      proof (rule SF1)
        show  $\sim \$pc2 = \#b \wedge [(n1 \vee n2) \wedge \neg beta1]$ -vars  $\longrightarrow \circ(\$pc2 = \#b) \vee \circ(\$pc2 = \#g)$ 
        by (auto simp: Sact2-defs vars-def tla-defs)

```

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

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

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

```

($pc2 = #a)
  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  $\vdash ((\$pc2 = \#a \vee \$pc2 = \#b \vee \$pc2 = \#g) \rightsquigarrow \$pc2 = \#a) \longrightarrow$ 
 $\diamond(\$pc2 = \#a)$ 
  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  $\vdash \diamond(\$pc2 = \#a \wedge \$pc1 = \#a \wedge I) \longrightarrow \diamond Enabled \langle n1 \rangle\text{-vars}$ 
  unfolding enab-n1[int-rewrite] by (rule STL4-eve) (auto simp: I-def
tla-defs)
  ultimately
  show  $\vdash \square(\$pc1 = \#a) \wedge \square[(n1 \vee n2) \wedge \neg beta1]\text{-vars} \wedge \square(I \wedge SF(n2)\text{-vars})$ 
 $\longrightarrow \diamond Enabled \langle n1 \rangle\text{-vars}$ 
  by (force simp: STL5[int-rewrite])
  qed
  from  $ga\ ab$  have  $\vdash ?F \longrightarrow (\$pc1 = \#g \rightsquigarrow \$pc1 = \#b)$ 
  by (force elim: LT13[unlift-rule])
  with  $ab$  have  $\vdash ?F \longrightarrow ((\$pc1 = \#a \vee \$pc1 = \#b \vee \$pc1 = \#g) \rightsquigarrow \$pc1$ 
 $= \#b)$ 
  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  $\vdash ((\$pc1 = \#a \vee \$pc1 = \#b \vee \$pc1 = \#g) \rightsquigarrow \$pc1 = \#b) \longrightarrow$ 
 $\diamond(\$pc1 = \#b)$ 
  by (rule fmp[OF - LT4])
  ultimately show ?thesis by (rule lift-imp-trans)
  qed
  with 1 show ?thesis by force
  qed
  with  $psiI$  show ?thesis unfolding  $psi\text{-def}$   $Live2\text{-def}$   $STL5[int-rewrite]$  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*) $psi\text{-fair-}m2: \vdash psi \longrightarrow SF(m2)\text{-}(x,y)$

proof –

have $\vdash \Box[n1 \vee n2]\text{-vars} \wedge SF(n2)\text{-vars} \wedge \Box(I \wedge SF(n1)\text{-vars}) \longrightarrow SF(m2)\text{-}(x,y)$
proof (rule *SF2*)

Rule *SF2* requires us to choose a helpful action (whose effect implies $\langle m2 \rangle\text{-}(x,y)$) 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 $|\sim \langle (n1 \vee n2) \wedge beta2 \rangle\text{-vars} \longrightarrow \langle m2 \rangle\text{-}(x,y)$
by (*auto simp: beta2-def m2-def vars-def tla-defs*)
next
show $|\sim \$pc2 = \#b \wedge \Box(\$pc2 = \#b) \wedge \langle (n1 \vee n2) \wedge n2 \rangle\text{-vars} \longrightarrow beta2$
by (*auto simp: n2-def alpha2-def beta2-def gamma2-def tla-defs*)
next
show $\vdash \$pc2 = \#b \wedge Enabled \langle m2 \rangle\text{-}(x, y) \longrightarrow Enabled \langle n2 \rangle\text{-vars}$
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 $\vdash \Box[(n1 \vee n2) \wedge \neg beta2]\text{-vars}$
 $\wedge SF(n2)\text{-vars} \wedge \Box(I \wedge SF(n1)\text{-vars}) \wedge \Box \Diamond Enabled \langle m2 \rangle\text{-}(x, y)$
 $\longrightarrow \Diamond \Box(\$pc2 = \#b)$
proof –
have $\vdash \Box \Box[(n1 \vee n2) \wedge \neg beta2]\text{-vars} \longrightarrow \Box(\$pc2 = \#b \longrightarrow \Box(\$pc2 = \#b))$
proof (rule *STL4*)
have $|\sim \$pc2 = \#b \wedge [(n1 \vee n2) \wedge \neg beta2]\text{-vars} \longrightarrow \Box(\$pc2 = \#b)$
by (*auto simp: Sact2-defs vars-def tla-defs*)
from *this[THEN INV1]*
show $\vdash \Box[(n1 \vee n2) \wedge \neg beta2]\text{-vars} \longrightarrow \$pc2 = \#b \longrightarrow \Box(\$pc2 = \#b)$
by *auto*
qed
hence $1: \vdash \Box[(n1 \vee n2) \wedge \neg beta2]\text{-vars} \longrightarrow \Diamond(\$pc2 = \#b) \longrightarrow \Diamond \Box(\$pc2 = \#b)$
by (*force intro: E31[unlift-rule]*)
have $\vdash \Box[(n1 \vee n2) \wedge \neg beta2]\text{-vars} \wedge SF(n2)\text{-vars} \wedge \Box(I \wedge SF(n1)\text{-vars})$
 $\longrightarrow \Diamond(\$pc2 = \#b)$
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 $?F = LIFT (\Box[(n1 \vee n2) \wedge \neg beta2]\text{-vars} \wedge SF(n2)\text{-vars} \wedge \Box(I \wedge SF(n1)\text{-vars}))$

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

```

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

```

We establish a suitable leadsto-chain.

```

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

```

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

```

have  $bg1: \vdash ?G \longrightarrow (\$pc1 = \#b \rightsquigarrow \$pc1 = \#g)$ 
proof (rule SF1)
  show  $|\sim \$pc1 = \#b \wedge [(n1 \vee n2) \wedge \neg beta2]\text{-vars} \longrightarrow \circ(\$pc1 = \#b)$ 
 $\vee \circ(\$pc1 = \#g)$ 
  by (auto simp: Sact2-defs vars-def tla-defs)
next
  show  $|\sim \$pc1 = \#b \wedge \langle ((n1 \vee n2) \wedge \neg beta2) \wedge n1 \rangle\text{-vars} \longrightarrow \circ(\$pc1$ 
 $= \#g)$ 
  by (auto simp: n1-def alpha1-def beta1-def gamma1-def vars-def
tla-defs)
next
  show  $|\sim \$pc1 = \#b \wedge \text{Unchanged vars} \longrightarrow \circ(\$pc1 = \#b)$ 
  by (auto simp: vars-def tla-defs)
next
  have  $\vdash \$pc1 = \#b \longrightarrow \text{Enabled } \langle n1 \rangle\text{-vars}$ 
  unfolding enab-n1[int-rewrite] by (auto simp: tla-defs)
  hence  $\vdash \square(\$pc1 = \#b) \longrightarrow \text{Enabled } \langle n1 \rangle\text{-vars}$ 
  by (rule lift-imp-trans[OF ax1])
  hence  $\vdash \square(\$pc1 = \#b) \longrightarrow \diamond \text{Enabled } \langle n1 \rangle\text{-vars}$ 
  by (rule lift-imp-trans[OF - E3])
  thus  $\vdash \square(\$pc1 = \#b) \wedge \square[(n1 \vee n2) \wedge \neg beta2]\text{-vars} \wedge \square(\$pc2 = \#a$ 
 $\wedge I)$ 
   $\longrightarrow \diamond \text{Enabled } \langle n1 \rangle\text{-vars}$ 
  by auto
qed

```

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

```

have  $ga1: \vdash ?G \longrightarrow (\$pc1 = \#g \rightsquigarrow \$pc1 = \#a)$ 
proof (rule SF1)
  show  $|\sim \$pc1 = \#g \wedge [(n1 \vee n2) \wedge \neg beta2]\text{-vars} \longrightarrow \circ(\$pc1 = \#g)$ 
 $\vee \circ(\$pc1 = \#a)$ 
  by (auto simp: Sact2-defs vars-def tla-defs)
next
  show  $|\sim \$pc1 = \#g \wedge \langle ((n1 \vee n2) \wedge \neg beta2) \wedge n1 \rangle\text{-vars} \longrightarrow \circ(\$pc1$ 
 $= \#a)$ 
  by (auto simp: n1-def alpha1-def beta1-def gamma1-def vars-def
tla-defs)
next
  show  $|\sim \$pc1 = \#g \wedge \text{Unchanged vars} \longrightarrow \circ(\$pc1 = \#g)$ 
  by (auto simp: vars-def tla-defs)
next
  have  $\vdash \$pc1 = \#g \longrightarrow \text{Enabled } \langle n1 \rangle\text{-vars}$ 
  unfolding enab-n1[int-rewrite] by (auto simp: tla-defs)
  hence  $\vdash \square(\$pc1 = \#g) \longrightarrow \text{Enabled } \langle n1 \rangle\text{-vars}$ 
  by (rule lift-imp-trans[OF ax1])
  hence  $\vdash \square(\$pc1 = \#g) \longrightarrow \diamond \text{Enabled } \langle n1 \rangle\text{-vars}$ 
  by (rule lift-imp-trans[OF - E3])
  thus  $\vdash \square(\$pc1 = \#g) \wedge \square[(n1 \vee n2) \wedge \neg beta2]\text{-vars} \wedge \square(\$pc2 = \#a$ 
 $\wedge I)$ 

```

$\longrightarrow \diamond Enabled \langle n1 \rangle\text{-vars}$
by auto
qed
with $bg1$ **have** $\vdash ?G \longrightarrow (\$pc1 = \#b \rightsquigarrow \$pc1 = \#a)$
by (*force elim: LT13[unlift-rule]*)
with $ga1$ **have** $\vdash ?G \longrightarrow (\$pc1 = \#a \vee \$pc1 = \#b \vee \$pc1 = \#g) \rightsquigarrow$
 $(\$pc1 = \#a)$
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 $s :: \text{state seq}$
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 $\vdash ((\$pc1 = \#a \vee \$pc1 = \#b \vee \$pc1 = \#g) \rightsquigarrow \$pc1 = \#a) \longrightarrow$
 $\diamond(\$pc1 = \#a)$
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 $\vdash \square(\$pc2 = \#a) \wedge \square[(n1 \vee n2) \wedge \neg beta2]\text{-vars} \wedge \square(I \wedge SF(n1)\text{-vars})$
 $\longrightarrow \diamond Enabled \langle n2 \rangle\text{-vars}$
by (*force simp: STL5[int-rewrite]*)
qed
from $ga ab$ **have** $\vdash ?F \longrightarrow (\$pc2 = \#g \rightsquigarrow \$pc2 = \#b)$
by (*force elim: LT13[unlift-rule]*)
with ab **have** $\vdash ?F \longrightarrow ((\$pc2 = \#a \vee \$pc2 = \#b \vee \$pc2 = \#g) \rightsquigarrow \$pc2$
 $= \#b)$
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 $s :: \text{state seq}$
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 $\vdash ((\$pc2 = \#a \vee \$pc2 = \#b \vee \$pc2 = \#g) \rightsquigarrow \$pc2 = \#b) \longrightarrow$
 $\diamond(\$pc2 = \#b)$
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:  $\vdash \textit{psi} \longrightarrow \textit{phi}$ 
  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  $\Rightarrow$  'a list statefun  $\Rightarrow$  'a statefun  $\Rightarrow$  temporal
where BInit ic q oc  $\equiv$  TEMP $q = #[]
       $\wedge$  $ic = $oc — initial condition of buffer

```

```

definition Enq :: 'a statefun  $\Rightarrow$  'a list statefun  $\Rightarrow$  'a statefun  $\Rightarrow$  temporal
where Enq ic q oc  $\equiv$  TEMP ic$  $\neq$  $ic
       $\wedge$  q$ = $q @ [ ic$ ]
       $\wedge$  oc$ = $oc — enqueue a new value

```

```

definition Deq :: 'a statefun  $\Rightarrow$  'a list statefun  $\Rightarrow$  'a statefun  $\Rightarrow$  temporal
where Deq ic q oc  $\equiv$  TEMP # 0 < length<$q>
       $\wedge$  oc$ = hd<$q>
       $\wedge$  q$ = tl<$q>
       $\wedge$  ic$ = $ic — dequeue value at front

```

```

definition Nxt :: 'a statefun  $\Rightarrow$  'a list statefun  $\Rightarrow$  'a statefun  $\Rightarrow$  temporal
where Nxt ic q oc  $\equiv$  TEMP (Enq ic q oc  $\vee$  Deq ic q oc)

```

— internal specification with buffer visible

definition $ISpec :: 'a\ statefun \Rightarrow 'a\ list\ statefun \Rightarrow 'a\ statefun \Rightarrow temporal$

where $ISpec\ ic\ q\ oc \equiv TEMP\ BInit\ ic\ q\ oc$
 $\wedge \square[Nxt\ ic\ q\ oc]-(ic,q,oc)$
 $\wedge WF(Deq\ ic\ q\ oc)-(ic,q,oc)$

— external specification: buffer hidden

definition $Spec :: 'a\ statefun \Rightarrow 'a\ statefun \Rightarrow temporal$

where $Spec\ ic\ oc == TEMP\ (\exists\exists\ q.\ ISpec\ ic\ q\ oc)$

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 $noreps :: 'a\ list \Rightarrow bool$

where $noreps\ xs \equiv \forall\ i < length\ xs - 1.\ xs!\ i \neq xs!\ (Suc\ i)$

definition $BInv :: 'a\ statefun \Rightarrow 'a\ list\ statefun \Rightarrow 'a\ statefun \Rightarrow temporal$

where $BInv\ ic\ q\ oc \equiv TEMP\ List.last<\$oc\ \#\ \$q> = \$ic \wedge noreps<\$oc\ \#\ \$q>$

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** $(simp\ add: bothstutinvs\ livestutinvs)$

lemma $Spec-stutinv: STUTINV\ Spec\ ic\ oc$

unfolding $buffer-defs$ **by** $(simp\ add: bothstutinvs\ livestutinvs\ eexSTUT)$

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 $xs \neq []$ **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 $xs \neq []$ with 1 show *False*
 by (auto simp: neq-Nil-conv)
 qed
 qed (auto)

lemma *Deq-visible*:

assumes $v \vdash \text{Unchanged } v \longrightarrow \text{Unchanged } q$
 shows $|\sim \langle \text{Deq } ic\ q\ oc \rangle \cdot v = \text{Deq } ic\ q\ oc$
proof (auto simp: tla-defs)
 fix w
 assume $deq: w \models \text{Deq } ic\ q\ oc$ and $unch: v(w(Suc\ 0)) = v(w\ 0)$
 from $unch\ v[\text{unlifted}]$ 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*: $\vdash \text{Enabled } \langle \text{Deq } ic\ q\ oc \rangle \cdot (ic, q, oc) \longrightarrow \$q \sim = \#[]$
 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]: $\text{noreps } []$
 by (auto simp: noreps-def)

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

lemma *noreps-cons* [simp]:
 $\text{noreps } (x \# xs) = (\text{noreps } xs \wedge (xs = [] \vee x \neq hd\ xs))$

proof (auto simp: noreps-singleton)
 assume $cons: \text{noreps } (x \# xs)$
 show $\text{noreps } xs$
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 = 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*]:

$$\text{noreps } (xs @ ys) = (\text{noreps } xs \wedge \text{noreps } ys \wedge (xs = [] \vee ys = [] \vee \text{List.last } xs \neq \text{hd } ys))$$

```

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))
  qed

```



```

    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 ≠ [] and 3: ys ≠ []
  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 ≠ hd ys
show noreps (xs @ ys)
proof (cases xs = [] ∨ 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  $(xs @ ys)!i = ys!(i - \text{length } xs)$ 
  by (simp add: nth-append)
moreover
from xsi have  $Suc\ i - \text{length } xs = Suc\ (i - \text{length } xs)$  by auto
with xsi have  $(xs @ ys)!(Suc\ i) = ys!(Suc\ (i - \text{length } xs))$ 
  by (simp add: nth-append)
moreover
from i xsi have  $i - \text{length } xs < \text{length } ys - 1$  by auto
with 2 have  $ys!(i - \text{length } xs) \neq ys!(Suc\ (i - \text{length } xs))$ 
  by (auto simp: noreps-def)
moreover
note eq
ultimately show False by simp
qed
qed
qed
qed
qed

```

lemma *ISpec-BInv-lemma*:

```

 $\vdash BInit\ ic\ q\ oc \wedge \square[Nxt\ ic\ q\ oc]-(ic, q, oc) \longrightarrow \square(BInv\ ic\ q\ oc)$ 
proof (rule invmono)
  show  $\vdash BInit\ ic\ q\ oc \longrightarrow BInv\ ic\ q\ oc$ 
    by (auto simp: BInit-def BInv-def)
next
  have enq:  $\sim Enq\ ic\ q\ oc \longrightarrow BInv\ ic\ q\ oc \longrightarrow \circ(BInv\ ic\ q\ oc)$ 
    by (auto simp: Enq-def BInv-def tla-defs)
  have deq:  $\sim Deq\ ic\ q\ oc \longrightarrow BInv\ ic\ q\ oc \longrightarrow \circ(BInv\ ic\ q\ oc)$ 
    by (auto simp: Deq-def BInv-def tla-defs neq-Nil-conv)
  have unch:  $\sim Unchanged\ (ic, q, oc) \longrightarrow BInv\ ic\ q\ oc \longrightarrow \circ(BInv\ ic\ q\ oc)$ 
    by (auto simp: BInv-def tla-defs)
  show  $\sim BInv\ ic\ q\ oc \wedge [Nxt\ ic\ q\ oc]-(ic, q, oc) \longrightarrow \circ(BInv\ ic\ q\ oc)$ 
    by (auto simp: Nxt-def actrans-def
      elim: enq[unlift-rule] deq[unlift-rule] unch[unlift-rule])
qed

```

```

theorem ISpec-BInv:  $\vdash ISpec\ ic\ q\ oc \longrightarrow \square(BInv\ ic\ q\ oc)$ 
  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 $DBSpec \equiv TEMP\ ISpec\ inp\ q1\ mid$
 $\wedge\ ISpec\ mid\ q2\ out$
 $\wedge\ \Box[\neg(Enq\ inp\ q1\ mid \wedge Deq\ mid\ q2\ out)]-vars$

The proof rules of TLA are geared towards specifications of the form $Init \wedge \Box[Next]-vars \wedge L$, 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 *FullNext*
where $FullNext \equiv TEMP\ (Enq\ inp\ q1\ mid \wedge Unchanged\ (q2, out)$
 $\vee\ Deq\ inp\ q1\ mid \wedge Enq\ mid\ q2\ out$
 $\vee\ Deq\ mid\ q2\ out \wedge Unchanged\ (inp, q1))$

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

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

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

lemmas $db-defs = buffer-defs\ DBSpec-def\ FullInit-def\ FullNext-def\ FullSpec-def$
 $qc-def\ vars-def$

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

lemma *FullSpec-stutinv: STUTINV FullSpec*
unfolding *db-defs* **by** (*simp add: bothstutinus 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:*

$\vdash \square BInv\ inp\ q1\ mid$
 $\wedge \square [Nxt\ inp\ q1\ mid]-(inp, q1, mid)$
 $\wedge \square [Nxt\ mid\ q2\ out]-(mid, q2, out)$
 $\wedge \square [\neg(Enq\ inp\ q1\ mid \wedge Deq\ mid\ q2\ out)]-vars$
 $\longrightarrow \square [FullNxt]-vars$
(is $\vdash \square ?inv \wedge ?nxts \longrightarrow \square [FullNxt]-vars$ **)**

proof –

have $\vdash \square [Nxt\ inp\ q1\ mid]-(inp, q1, mid)$
 $\wedge \square [Nxt\ mid\ q2\ out]-(mid, q2, out)$
 $\longrightarrow \square [[Nxt\ inp\ q1\ mid]-(inp, q1, mid)$
 $\wedge [Nxt\ mid\ q2\ out]-(mid, q2, out)]-((inp, q1, mid), (mid, q2, out))$
(is $\vdash ?tmp \longrightarrow \square [?b1b2]-?vs$ **)**
by (*auto simp: M12[int-rewrite]*)

moreover

have $\vdash \square [?b1b2]-?vs \longrightarrow \square [?b1b2]-vars$
by (*rule R1, auto simp: vars-def tla-defs*)

ultimately

have *1:* $\vdash \square [Nxt\ inp\ q1\ mid]-(inp, q1, mid)$
 $\wedge \square [Nxt\ mid\ q2\ out]-(mid, q2, out)$
 $\longrightarrow \square [[Nxt\ inp\ q1\ mid]-(inp, q1, mid)$
 $\wedge [Nxt\ mid\ q2\ out]-(mid, q2, out)]-vars$

by *force*

have *2:* $\vdash \square [?b1b2]-vars \wedge \square [\neg(Enq\ inp\ q1\ mid \wedge Deq\ mid\ q2\ out)]-vars$
 $\longrightarrow \square [?b1b2 \wedge \neg(Enq\ inp\ q1\ mid \wedge Deq\ mid\ q2\ out)]-vars$

(is $\vdash ?tmp2 \longrightarrow \square [?mid]-vars$ **)**

by (*simp add: M8[int-rewrite]*)

have $\vdash ?inv \longrightarrow \#True$ **by** *auto*

moreover

have $|\sim ?inv \wedge \circ ?inv \wedge [?mid]-vars \longrightarrow [FullNxt]-vars$

proof –

have $|\sim ?inv \wedge ?mid \longrightarrow [FullNxt]-vars$

proof –

have *A:* $|\sim Nxt\ inp\ q1\ mid$
 $\longrightarrow [Nxt\ mid\ q2\ out]-(mid, q2, out)$
 $\longrightarrow \neg(Enq\ inp\ q1\ mid \wedge Deq\ mid\ q2\ out)$
 $\longrightarrow ?inv$
 $\longrightarrow FullNxt$

```

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)
  qed

```

```

      intro: DBNxt-then-FullNxt[unlift-rule])
    ultimately show ?thesis by force
  qed
  have 3:  $\vdash DBSpec \longrightarrow WF(Deq\ inp\ q1\ mid)\text{-vars}$ 
  proof -
    have 31:  $\vdash Unchanged\ vars \longrightarrow Unchanged\ q1$ 
      by (auto simp: vars-def tla-defs)
    have 32:  $\vdash Unchanged\ (inp,q1,mid) \longrightarrow Unchanged\ q1$ 
      by (auto simp: tla-defs)
    have deg:  $|\sim \langle Deq\ inp\ q1\ mid \rangle\text{-vars} = \langle Deq\ inp\ q1\ mid \rangle\text{-}(inp,q1,mid)$ 
      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
        deg[int-rewrite] deg[THEN AA26,int-rewrite])
  qed
  have 4:  $\vdash DBSpec \longrightarrow WF(Deq\ mid\ q2\ out)\text{-vars}$ 
  proof -
    have 41:  $\vdash Unchanged\ vars \longrightarrow Unchanged\ q2$ 
      by (auto simp: vars-def tla-defs)
    have 42:  $\vdash Unchanged\ (mid,q2,out) \longrightarrow Unchanged\ q2$ 
      by (auto simp: tla-defs)
    have deg:  $|\sim \langle Deq\ mid\ q2\ out \rangle\text{-vars} = \langle Deq\ mid\ q2\ out \rangle\text{-}(mid,q2,out)$ 
      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
        deg[int-rewrite] deg[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*: $\vdash FullInit \longrightarrow BInit\ inp\ qc\ out$
 by (auto simp: db-defs tla-defs)

lemma *Full-step-simulation*:
 $|\sim [FullNxt]\text{-vars} \longrightarrow [Nxt\ inp\ qc\ out]\text{-}(inp,qc,out)$
 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:  $\vdash \text{Enabled } \langle \text{Deq } \text{inp } q1 \text{ mid} \rangle\text{-vars} = (\$q1 \neq \#\ [])$ 
proof –
  have 1:  $|\sim \langle \text{Deq } \text{inp } q1 \text{ mid} \rangle\text{-vars} = \text{Deq } \text{inp } q1 \text{ mid}$ 
    by (rule Deq-visible, auto simp: vars-def tla-defs)
  have  $\vdash \text{Enabled } (\text{Deq } \text{inp } q1 \text{ mid}) = (\$q1 \neq \#\ [])$ 
    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:  $\vdash \text{Enabled } \langle \text{Deq } \text{mid } q2 \text{ out} \rangle\text{-vars} = (\$q2 \neq \#\ [])$ 
proof –
  have 1:  $|\sim \langle \text{Deq } \text{mid } q2 \text{ out} \rangle\text{-vars} = \text{Deq } \text{mid } q2 \text{ out}$ 
    by (rule Deq-visible, auto simp: vars-def tla-defs)
  have  $\vdash \text{Enabled } (\text{Deq } \text{mid } q2 \text{ out}) = (\$q2 \neq \#\ [])$ 
    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:
   $\vdash \square[\text{FullNxt}]\text{-vars} \wedge \text{WF}(\text{Deq } \text{mid } q2 \text{ out})\text{-vars} \wedge \square \text{WF}(\text{Deq } \text{inp } q1 \text{ mid})\text{-vars}$ 
   $\longrightarrow \text{WF}(\text{Deq } \text{inp } qc \text{ out})\text{-(inp, qc, out)}$ 
proof (rule WF2)
  – the helpful action is the Deq action of the second queue
  show  $|\sim \langle \text{FullNxt} \wedge \text{Deq } \text{mid } q2 \text{ out} \rangle\text{-vars} \longrightarrow \langle \text{Deq } \text{inp } qc \text{ out} \rangle\text{-(inp, qc, out)}$ 
    by (auto simp: db-defs tla-defs)
next
  – the helpful condition is the second queue being non-empty
  show  $|\sim (\$q2 \neq \#\ []) \wedge \circ(\$q2 \neq \#\ []) \wedge \langle \text{FullNxt} \wedge \text{Deq } \text{mid } q2 \text{ out} \rangle\text{-vars}$ 
     $\longrightarrow \text{Deq } \text{mid } q2 \text{ out}$ 
    by (auto simp: tla-defs)
next
  show  $\vdash \$q2 \neq \#\ [] \wedge \text{Enabled } \langle \text{Deq } \text{inp } qc \text{ out} \rangle\text{-(inp, qc, out)}$ 
     $\longrightarrow \text{Enabled } \langle \text{Deq } \text{mid } q2 \text{ out} \rangle\text{-vars}$ 
    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 $\vdash \Box\Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-vars}$
 $\longrightarrow \Box(\$q2 \neq \#[] \longrightarrow \Box(\$q2 \neq \#[]))$
proof (*rule STL4*)
have $|\sim \$q2 \neq \#[] \wedge [FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-vars}$
 $\longrightarrow \circ(\$q2 \neq \#[])$
by (*auto simp: db-defs tla-defs*)
from *this*[*THEN INV1*]
show $\vdash \Box[FullNxt \wedge \neg Deq\ mid\ q2\ out]\text{-vars}$
 $\longrightarrow (\$q2 \neq \#[] \longrightarrow \Box(\$q2 \neq \#[]))$
by *auto*
qed
hence 1: $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-vars}$
 $\longrightarrow \diamond(\$q2 \neq \#[]) \longrightarrow \diamond\Box(\$q2 \neq \#[])$
by (*force intro: E31[unlift-rule]*)
have 2: $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-vars}$
 $\wedge WF(Deq\ inp\ q1\ mid)\text{-vars}$
 $\longrightarrow (Enabled\ \langle Deq\ inp\ qc\ out \rangle\text{-}(inp, qc, out) \rightsquigarrow \$q2 \neq \#[])$
proof -
have *qc*: $\vdash (\$qc \neq \#[]) = (\$q1 \neq \#[] \vee \$q2 \neq \#[])$
by (*auto simp: qc-def tla-defs*)
have $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-vars} \wedge WF(Deq\ inp\ q1\ mid)\text{-vars}$
 $\longrightarrow (\$q1 \neq \#[] \rightsquigarrow \$q2 \neq \#[])$
proof (*rule WF1*)
show $|\sim \$q1 \neq \#[] \wedge [FullNxt \wedge \neg Deq\ mid\ q2\ out]\text{-vars}$
 $\longrightarrow \circ(\$q1 \neq \#[]) \vee \circ(\$q2 \neq \#[])$
by (*auto simp: db-defs tla-defs*)
next
show $|\sim \$q1 \neq \#[]$
 $\wedge \langle (FullNxt \wedge \neg Deq\ mid\ q2\ out) \wedge Deq\ inp\ q1\ mid \rangle\text{-vars} \longrightarrow$
 $\circ(\$q2 \neq \#[])$
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 $|\sim \$q1 \neq \#[] \wedge Unchanged\ vars \longrightarrow \circ(\$q1 \neq \#[])$
by (*auto simp: vars-def tla-defs*)
qed
hence $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-vars}$
 $\wedge WF(Deq\ inp\ q1\ mid)\text{-vars}$
 $\longrightarrow (\$qc \neq \#[] \rightsquigarrow \$q2 \neq \#[])$
by (*auto simp: qc[int-rewrite] LT17[int-rewrite] LT1[int-rewrite]*)
moreover
have $\vdash Enabled\ \langle Deq\ inp\ qc\ out \rangle\text{-}(inp, qc, out) \rightsquigarrow \$qc \neq \#[]$
by (*rule Deq-enabledE[THEN LT3]*)
ultimately show *?thesis* **by** (*force elim: LT13[unlift-rule]*)
qed
with *LT6*
have $\vdash \Box[FullNxt \wedge \neg(Deq\ mid\ q2\ out)]\text{-vars}$

$$\begin{aligned} & \wedge WF(Deq\ inp\ q1\ mid)\text{-vars} \\ & \wedge \diamond Enabled \langle Deq\ inp\ qc\ out \rangle\text{-}(inp, qc, out) \\ & \longrightarrow \diamond(\$q2 \neq \#[]) \\ \text{by force} \\ \text{with 1 E16} \\ \text{show } & \vdash \square[FullNext \wedge \neg(Deq\ mid\ q2\ out)]\text{-vars} \\ & \wedge WF(Deq\ mid\ q2\ out)\text{-vars} \\ & \wedge \square WF(Deq\ inp\ q1\ mid)\text{-vars} \\ & \wedge \diamond \square Enabled \langle Deq\ inp\ qc\ out \rangle\text{-}(inp, qc, out) \\ & \longrightarrow \diamond \square(\$q2 \neq \#[]) \\ \text{by force} \\ \text{qed} \end{aligned}$$

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: eeI[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 $tl\text{a}^+$ 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.