

Generalised Reactive Processes in Isabelle/UTP

Simon Foster

Samuel Canham

May 11, 2020

Abstract

Hoare and He’s UTP theory of reactive processes provides a unifying foundation for the semantics of process calculi and reactive programming. A reactive process is a form of UTP relation which can refer to both state variables and also a trace history of events. In their original presentation, a trace was modelled solely by a discrete sequence of events. Here, we generalise the trace model using “trace algebra”, which characterises traces abstractly using cancellative monoids, and thus enables application of the theory to a wider family of computational models, including hybrid computation. We recast the reactive healthiness conditions in this setting, and prove all the associated distributivity laws. We tackle parallel composition of reactive processes using the “parallel-by-merge” scheme from UTP. We also identify the associated theory of “reactive relations”, and use it to define generic reactive laws, a Hoare logic, and a weakest precondition calculus.

Contents

1	Trace Algebras	2
1.1	Ordered Semigroups	2
1.2	Monoid Subclasses	3
1.3	Trace Algebras	4
1.4	Models	7
2	Reactive Processes Core Definitions	8
2.1	Alphabet and Signature	8
2.2	Reactive Lemmas	10
2.3	Trace contribution lens	10
3	Events for Reactive Processes	11
3.1	Events	12
3.2	Channels	12
3.2.1	Operators	12
4	Reactive Healthiness Conditions	13
4.1	R1: Events cannot be undone	13
4.2	R2: No dependence upon trace history	16
4.3	R3: No activity while predecessor is waiting	25
4.4	R4: The trace strictly increases	26
4.5	R5: The trace does not increase	27
4.6	RP laws	28
4.7	UTP theories	28

5	Reactive Parallel-by-Merge	29
6	Reactive Relations	33
6.1	Healthiness Conditions	33
6.2	Reactive relational operators	34
6.3	Unrestriction and substitution laws	36
6.4	Closure laws	37
6.5	Reactive relational calculus	42
6.6	UTP theory	46
6.7	Instantaneous Reactive Relations	47
7	Reactive Conditions	47
7.1	Healthiness Conditions	47
7.2	Closure laws	49
8	Reactive Programs	51
8.1	Stateful reactive alphabet	52
8.2	State Lifting	53
8.3	Reactive Program Operators	54
8.3.1	State Substitution	54
8.3.2	Assignment	55
8.3.3	Conditional	56
8.3.4	Assumptions	57
8.3.5	State Abstraction	58
8.3.6	Reactive Frames and Extensions	59
8.4	Stateful Reactive specifications	62
9	Reactive Weakest Preconditions	64
10	Reactive Hoare Logic	67
11	Meta-theory for Generalised Reactive Processes	68

1 Trace Algebras

```

theory Trace-Algebra
  imports
    UTP-Toolkit.List-Extra
    UTP-Toolkit.Positive
begin

```

Trace algebras provide a useful way in the UTP of characterising different notions of trace history. They can characterise notions as diverse as discrete event sequences and piecewise continuous functions, as employed by hybrid systems. For more information, please see our journal publication [4].

1.1 Ordered Semigroups

```

class ordered-semigroup = semigroup-add + order +
  assumes add-left-mono:  $a \leq b \implies c + a \leq c + b$ 
  and add-right-mono:  $a \leq b \implies a + c \leq b + c$ 
begin

```

```

lemma add-mono:
   $a \leq b \implies c \leq d \implies a + c \leq b + d$ 
  using local.add-left-mono local.add-right-mono local.order.trans by blast

end

```

1.2 Monoid Subclasses

```

class left-cancel-monoid = monoid-add +
  assumes add-left-imp-eq:  $a + b = a + c \implies b = c$ 

class right-cancel-monoid = monoid-add +
  assumes add-right-imp-eq:  $b + a = c + a \implies b = c$ 

```

Positive Monoids

```

class monoid-pos = monoid-add +
  assumes zero-sum-left:  $a + b = 0 \implies a = 0$ 
begin

lemma zero-sum-right:  $a + b = 0 \implies b = 0$ 
  by (metis local.add-0-left local.zero-sum-left)

lemma zero-sum:  $a + b = 0 \iff a = 0 \wedge b = 0$ 
  by (metis local.add-0-right zero-sum-right)

```

end

```

context monoid-add
begin

```

An additive monoid gives rise to natural notions of order, which we here define.

```

definition monoid-le (infix  $\leq_m$  50)
where  $a \leq_m b \iff (\exists c. b = a + c)$ 

```

We can also define a subtraction operator that remove a prefix from a monoid, if possible.

```

definition monoid-subtract (infixl  $-_m$  65)
where  $a -_m b = (\text{if } (b \leq_m a) \text{ then } \text{THE } c. a = b + c \text{ else } 0)$ 

```

We derive some basic properties of the preorder

```

lemma monoid-le-least-zero:  $0 \leq_m a$ 
  by (simp add: monoid-le-def)

lemma monoid-le-add:  $a \leq_m a + b$ 
  by (auto simp add: monoid-le-def)

lemma monoid-le-refl:  $a \leq_m a$ 
  by (simp add: monoid-le-def, metis add.right-neutral)

lemma monoid-le-trans:  $\llbracket a \leq_m b; b \leq_m c \rrbracket \implies a \leq_m c$ 
  by (metis add.assoc monoid-le-def)

lemma monoid-le-add-left-mono:  $a \leq_m b \implies c + a \leq_m c + b$ 
  using add-assoc by (auto simp add: monoid-le-def)

```

end

```

class ordered-monoid-pos = monoid-pos + ord +
  assumes le-is-monoid-le:  $a \leq b \longleftrightarrow (a \leq_m b)$ 
  and less-iff:  $a < b \longleftrightarrow a \leq b \wedge \neg (b \leq a)$ 
begin

  subclass preorder
  proof
    fix x y z :: 'a
    show  $(x < y) = (x \leq y \wedge \neg y \leq x)$ 
      by (simp add: local.less-iff)
    show  $x \leq x$ 
      by (simp add: local.le-is-monoid-le local.monoid-le-reft)
    show  $x \leq y \implies y \leq z \implies x \leq z$ 
      using local.le-is-monoid-le local.monoid-le-trans by blast
  qed

```

end

1.3 Trace Algebras

A pre-trace algebra is based on a left-cancellative monoid with the additional property that plus has no additive inverse. The latter is required to ensure that there are no “negative traces”. A pre-trace algebra has all the trace algebra axioms, but does not export the definitions of (\leq) and $(-)$.

```

class pre-trace = left-cancel-monoid + monoid-pos
begin

```

From our axiom set, we can derive a variety of properties of the monoid order

```

lemma monoid-le-antisym:
  assumes  $a \leq_m b$   $b \leq_m a$ 
  shows  $a = b$ 
proof -
  obtain a' where a':  $b = a + a'$ 
    using assms(1) monoid-le-def by auto

  obtain b' where b':  $a = b + b'$ 
    using assms(2) monoid-le-def by auto

  have  $b' = (b' + a' + b')$ 
    by (metis a' add-assoc b' local.add-left-imp-eq)

  hence  $a' + b' = 0$ 
    by (metis add-assoc local.add-0-right local.add-left-imp-eq)

  hence  $a' = 0$   $b' = 0$ 
    by (simp add: zero-sum)+

  with a' b' show ?thesis
    by simp
qed

```

The monoid minus operator is also the inverse of plus in this context, as expected.

```

lemma add-monoid-diff-cancel-left [simp]:  $(a + b) -_m a = b$ 
  apply (simp add: monoid-subtract-def monoid-le-add)
  apply (rule the-equality)
  apply (simp)
  using local.add-left-imp-eq apply blast
done

```

Iterating a trace

```

fun tr-iter :: nat  $\Rightarrow$  'a  $\Rightarrow$  'a where
  tr-iter-0: tr-iter 0 t = 0 |
  tr-iter-Suc: tr-iter (Suc n) t = tr-iter n t + t

```

```

lemma tr-iter-empty [simp]: tr-iter m 0 = 0
  by (induct m, simp-all)

```

end

We now construct the trace algebra by also exporting the order and minus operators.

```

class trace = pre-trace + ord + minus +
  assumes le-is-monoid-le:  $a \leq b \longleftrightarrow (a \leq_m b)$ 
  and less-iff:  $a < b \longleftrightarrow a \leq b \wedge \neg (b \leq a)$ 
  and minus-def:  $a - b = a -_m b$ 
begin

```

Next we prove all the trace algebra lemmas.

```

lemma le-iff-add:  $a \leq b \longleftrightarrow (\exists c. b = a + c)$ 
  by (simp add: local.le-is-monoid-le local.monoid-le-def)

```

```

lemma least-zero [simp]:  $0 \leq a$ 
  by (simp add: local.le-is-monoid-le local.monoid-le-least-zero)

```

```

lemma le-add [simp]:  $a \leq a + b$ 
  by (simp add: le-is-monoid-le local.monoid-le-add)

```

```

lemma not-le-minus [simp]:  $\neg (a \leq b) \implies b - a = 0$ 
  by (simp add: le-is-monoid-le local.minus-def local.monoid-subtract-def)

```

```

lemma add-diff-cancel-left [simp]:  $(a + b) - a = b$ 
  by (simp add: minus-def)

```

```

lemma diff-zero [simp]:  $a - 0 = a$ 
  by (metis local.add-0-left local.add-diff-cancel-left)

```

```

lemma diff-cancel [simp]:  $a - a = 0$ 
  by (metis local.add-0-right local.add-diff-cancel-left)

```

```

lemma add-left-mono:  $a \leq b \implies c + a \leq c + b$ 
  by (simp add: local.le-is-monoid-le local.monoid-le-add-left-mono)

```

```

lemma add-le-imp-le-left:  $c + a \leq c + b \implies a \leq b$ 
  by (auto simp add: le-iff-add, metis add-assoc local.add-diff-cancel-left)

```

```

lemma add-diff-cancel-left' [simp]:  $(c + a) - (c + b) = a - b$ 
proof (cases  $b \leq a$ )
  case True thus ?thesis

```

```

    by (metis add-assoc local.add-diff-cancel-left local.le-iff-add)
next
case False thus ?thesis
  using local.add-le-imp-le-left not-le-minus by blast
qed

lemma minus-zero-eq:  $\llbracket b \leq a; a - b = 0 \rrbracket \implies a = b$ 
  using local.le-iff-add local.monoid-le-def by auto

lemma diff-add-cancel-left':  $a \leq b \implies a + (b - a) = b$ 
  using local.le-iff-add local.monoid-le-def by auto

lemma add-left-strict-mono:  $\llbracket a + b < a + c \rrbracket \implies b < c$ 
  using local.add-le-imp-le-left local.add-left-mono local.less-iff by blast

lemma sum-minus-left:  $c \leq a \implies (a + b) - c = (a - c) + b$ 
  by (metis add-assoc diff-add-cancel-left' local.add-monoid-diff-cancel-left local.minus-def)

lemma neq-zero-impl-greater:
   $x \neq 0 \implies 0 < x$ 
  using le-is-monoid-le less-iff monoid-le-antisym monoid-le-least-zero by auto

lemma minus-cancel-le:
   $\llbracket x \leq y; y \leq z \rrbracket \implies y - x \leq z - x$ 
  using add-assoc le-iff-add by auto

lemma sum-minus-right:  $c \geq a \implies a + b - c = b - (c - a)$ 
  by (metis diff-add-cancel-left' local.add-diff-cancel-left')

lemma minus-gr-zero-iff [simp]:
   $0 < x - y \iff y < x$ 
  by (metis diff-cancel le-is-monoid-le least-zero less-iff minus-zero-eq monoid-le-antisym not-le-minus)

lemma le-zero-iff [simp]:  $x \leq 0 \iff x = 0$ 
  using local.le-iff-add local.zero-sum by auto

lemma minus-assoc [simp]:  $x - y - z = x - (y + z)$ 
  by (metis diff-add-cancel-left' le-add local.add-0-right local.add-diff-cancel-left' local.zero-sum minus-cancel-le not-le-minus)

```

end

```

class trace-split = trace +
  assumes
    sum-eq-sum-conv:  $(a + b) = (c + d) \implies \exists e. a = c + e \wedge e + b = d \vee a + e = c \wedge b = e + d$ 
    —  $?a + ?b = ?c + ?d \implies \exists e. ?a = ?c + e \wedge e + ?b = ?d \vee ?a + e = ?c \wedge ?b = e + ?d$  shows
    how two equal traces that are each composed of two subtraces, can be expressed in terms of each other.
begin

```

The set subtraces of a common trace c is totally ordered.

```

lemma le-common-total:  $\llbracket a \leq c; b \leq c \rrbracket \implies a \leq b \vee b \leq a$ 
  by (metis diff-add-cancel-left' le-add local.sum-eq-sum-conv)

lemma le-sum-cases:  $a \leq b + c \implies a \leq b \vee b \leq a$ 
  by (simp add: le-common-total)

```

```

lemma le-sum-cases':
   $a \leq b + c \implies a \leq b \vee b \leq a \wedge a - b \leq c$ 
  by (auto, metis le-sum-cases, metis minus-def le-is-monoid-le add-monoid-diff-cancel-left monoid-le-def sum-eq-sum-conv)

lemma le-sum-iff:  $a \leq b + c \iff a \leq b \vee b \leq a \wedge a - b \leq c$ 
  by (metis le-sum-cases' add-monoid-diff-cancel-left le-is-monoid-le minus-def monoid-le-add-left-monoid-le-def monoid-le-trans)

```

end

Trace algebra give rise to a partial order on traces.

```

instance trace  $\subseteq$  order
  apply (intro-classes)
  apply (simp-all add: less-iff le-is-monoid-le monoid-le-refl)
  using monoid-le-trans apply blast
  apply (simp add: monoid-le-antisym)
  done

```

1.4 Models

Lists form a trace algebra.

```

instantiation list :: (type) monoid-add
begin

```

```

  definition zero-list :: 'a list where zero-list = []
  definition plus-list :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list where plus-list = (@)

```

```

instance
  by (intro-classes, simp-all add: zero-list-def plus-list-def)

```

end

```

lemma monoid-le-list:
   $(xs :: 'a list) \leq_m ys \iff xs \leq ys$ 
  apply (simp add: monoid-le-def plus-list-def)
  apply (meson Prefix-Order.prefixE Prefix-Order.prefixI)
  done

```

```

lemma monoid-subtract-list:
   $(xs :: 'a list) -_m ys = xs - ys$ 
  apply (auto simp add: monoid-subtract-def monoid-le-list minus-list-def less-eq-list-def)
  apply (rule the-equality)
  apply (simp-all add: zero-list-def plus-list-def prefix-drop)
  done

```

```

instance list :: (type) trace-split
  apply (intro-classes, simp-all add: zero-list-def plus-list-def monoid-le-def monoid-subtract-list)
  using Prefix-Order.prefixE apply blast
  apply (simp add: less-list-def)
  apply (simp add: append-eq-append-conv2)
  done

```

```

lemma monoid-le-nat:

```

```

(x :: nat) ≤m y ↔ x ≤ y
by (simp add: monoid-le-def nat-le-iff-add)

```

lemma *monoid-subtract-nat*:

```

(x :: nat) −m y = x − y
by (auto simp add: monoid-subtract-def monoid-le-nat)

```

instance *nat :: trace-split*

```

  apply (intro-classes, simp-all add: monoid-subtract-nat)
  apply (simp add: nat-le-iff-add monoid-le-def)
  apply linarith+
  apply (metis Nat.diff-add-assoc Nat.diff-add-assoc2 add-diff-cancel-right' add-le-cancel-left add-le-cancel-right
    add-less-mono cancel-ab-semigroup-add-class.add-diff-cancel-left' less-irrefl not-le)
  done

```

Positives form a trace algebra.

instance *pos :: (linordered-semidom) trace-split*

proof (*intro-classes, simp-all*)

```

  fix a b c d :: 'a pos
  show a + b = 0 ⇒ a = 0
  by (transfer, simp add: add-nonneg-eq-0-iff)
  show a + b = c + d ⇒ ∃ e. a = c + e ∧ e + b = d ∨ a + e = c ∧ b = e + d

```

```

  apply (cases c ≤ a)
  apply (metis (no-types, lifting) cancel-semigroup-add-class.add-left-imp-eq le-add-diff-inverse semiring-normalization-
    semiring-normalization-rules(21))
  done

```

```

  show (a < b) = (a ≤ b ∧ ¬ b ≤ a)
  by auto
  show le-def: ∧ a b :: 'a pos. (a ≤ b) = (a ≤m b)
  by (auto simp add: monoid-le-def, metis le-add-diff-inverse)
  show a − b = a −m b
  apply (auto simp add: monoid-subtract-def le-def[THEN sym])
  apply (rule sym)
  apply (rule the-equality)
  apply (simp-all)
  apply (transfer, simp)
  done

```

qed

end

2 Reactive Processes Core Definitions

theory *utp-rea-core*

imports

Trace-Algebra

UTP.utp-concurrency

UTP-Designs.utp-designs

begin recall-syntax

2.1 Alphabet and Signature

The alphabet of reactive processes contains a boolean variable *wait*, which denotes whether a process is exhibiting an intermediate observation. It also has the variable *tr* which denotes

the trace history of a process. The type parameter $'t$ represents the trace model being used, which must form a trace algebra [4], and thus provides the theory of “generalised reactive processes” [4]. The reactive process alphabet also extends the design alphabet, and thus includes the ok variable. For more information on these, see the UTP book [5], or the associated tutorial [2].

alphabet $'t::trace$ $rp\text{-}vars = des\text{-}vars +$
 $wait :: bool$
 $tr :: 't$

type-synonym $('t, ' \alpha) rp = ('t, ' \alpha) rp\text{-}vars\text{-}scheme$

type-synonym $('t, ' \alpha, ' \beta) rel\text{-}rp = (('t, ' \alpha) rp, ('t, ' \beta) rp) urel$

type-synonym $('t, ' \alpha) hrel\text{-}rp = ('t, ' \alpha) rp hrel$

translations

$(type) ('t, ' \alpha) rp \leq (type) ('t, ' \alpha) rp\text{-}vars\text{-}scheme$
 $(type) ('t, ' \alpha) rp \leq (type) ('t, ' \alpha) rp\text{-}vars\text{-}ext$
 $(type) ('t, ' \alpha, ' \beta) rel\text{-}rp \leq (type) (('t, ' \alpha) rp, (' \gamma, ' \beta) rp) urel$
 $(type) ('t, ' \alpha) hrel\text{-}rp \leq (type) ('t, ' \alpha) rp hrel$

As for designs, we set up various types to represent reactive processes. The main types to be used are $('t, ' \alpha, ' \beta) rel\text{-}rp$ and $('t, ' \alpha) hrel\text{-}rp$, which correspond to heterogeneous/homogeneous reactive processes whose trace model is $'t$ and alphabet types are $' \alpha$ and $' \beta$. We also set up some useful syntax translations for these.

notation $rp\text{-}vars.more_L (\Sigma_R)$

syntax

$-svid\text{-}rea\text{-}\alpha :: svid (\Sigma_R)$

translations

$-svid\text{-}rea\text{-}\alpha \Rightarrow CONST rp\text{-}vars.more_L$

Lens Σ_R exists because reactive alphabets are extensible. Σ_R points to the portion of the alphabet / state space that is neither ok , $wait$, or tr .

declare $des\text{-}vars.splits [alpha\text{-}splits del]$
declare $des\text{-}vars.splits [alpha\text{-}splits]$
declare $zero\text{-}list\text{-}def [upred\text{-}defs]$
declare $plus\text{-}list\text{-}def [upred\text{-}defs]$
declare $prefixE [elim]$

abbreviation $wait\text{-}f::('t::trace, ' \alpha, ' \beta) rel\text{-}rp \Rightarrow ('t, ' \alpha, ' \beta) rel\text{-}rp$
where $wait\text{-}f R \equiv R \llbracket false / \$wait \rrbracket$

abbreviation $wait\text{-}t::('t::trace, ' \alpha, ' \beta) rel\text{-}rp \Rightarrow ('t, ' \alpha, ' \beta) rel\text{-}rp$
where $wait\text{-}t R \equiv R \llbracket true / \$wait \rrbracket$

syntax

$-wait\text{-}f :: logic \Rightarrow logic (-_f [1000] 1000)$
 $-wait\text{-}t :: logic \Rightarrow logic (-_t [1000] 1000)$

translations

$P_f \Leftarrow CONST usubst (CONST subst\text{-}upd id_s (CONST in\text{-}var CONST wait) false) P$

$P \text{ }_t \Rightarrow \text{CONST usubst } (\text{CONST subst-upd id}_s (\text{CONST in-var CONST wait}) \text{ true}) P$

abbreviation *lift-rea* :: $- \Rightarrow - ([\cdot]_R)$ **where**
 $[P]_R \equiv P \oplus_p (\Sigma_R \times_L \Sigma_R)$

abbreviation *drop-rea* :: $('t::\text{trace}, ' \alpha, ' \beta) \text{ rel-rp} \Rightarrow (' \alpha, ' \beta) \text{ urel } ([\cdot]_R)$ **where**
 $[P]_R \equiv P \upharpoonright_e (\Sigma_R \times_L \Sigma_R)$

abbreviation *rea-pre-lift* :: $- \Rightarrow - ([\cdot]_{R<})$ **where** $[n]_{R<} \equiv [[n]_{<}]_R$

2.2 Reactive Lemmas

lemma *unrest-ok-lift-rea* [*unrest*]:

$\$ok \# [P]_R \$ok' \# [P]_R$
by (*pred-auto*) $+$

lemma *unrest-wait-lift-rea* [*unrest*]:

$\$wait \# [P]_R \$wait' \# [P]_R$
by (*pred-auto*) $+$

lemma *unrest-tr-lift-rea* [*unrest*]:

$\$tr \# [P]_R \$tr' \# [P]_R$
by (*pred-auto*) $+$

lemma *des-lens-equiv-wait-tr-rest*: $\Sigma_D \approx_L \text{wait} +_L \text{tr} +_L \Sigma_R$

by *simp*

lemma *rea-lens-bij*: *bij-lens* ($ok +_L \text{wait} +_L \text{tr} +_L \Sigma_R$)

proof $-$

have $ok +_L \text{wait} +_L \text{tr} +_L \Sigma_R \approx_L ok +_L \Sigma_D$

using *des-lens-equiv-wait-tr-rest des-vars.indeps lens-equiv-sym lens-plus-eq-right* **by** *blast*

also have $\dots \approx_L 1_L$

using *bij-lens-equiv-id*[*of* $ok +_L \Sigma_D$] **by** (*simp add: ok-des-bij-lens*)

finally show *?thesis*

by (*simp add: bij-lens-equiv-id*)

qed

lemma *segr-wait-true* [*usubst*]: $(P ;; Q) \text{ }_t = (P \text{ }_t ;; Q)$

by (*rel-auto*)

lemma *segr-wait-false* [*usubst*]: $(P ;; Q) \text{ }_f = (P \text{ }_f ;; Q)$

by (*rel-auto*)

2.3 Trace contribution lens

The following lens represents the portion of the state-space that is the difference between *tr'* and *tr*, that is the contribution that a process is making to the trace history.

definition *tcontr* :: $'t::\text{trace} \Rightarrow ('t, ' \alpha) \text{ rp} \times ('t, ' \alpha) \text{ rp}$ (*tt*) **where**

[*lens-defs*]:

$tcontr = (\text{ lens-get } = (\lambda s. \text{get}(\$tr')_v s - \text{get}(\$tr)_v s) ,$
 $\text{ lens-put } = (\lambda s v. \text{put}(\$tr')_v s (\text{get}(\$tr)_v s + v)))$

definition *itrace* :: $'t::\text{trace} \Rightarrow ('t, ' \alpha) \text{ rp} \times ('t, ' \alpha) \text{ rp}$ (*it*) **where**

[*lens-defs*]:

$itrace = (\text{ lens-get } = \text{get}(\$tr)_v ,$

$$lens\text{-}put = (\lambda s v. put_{(\$tr')_v} (put_{(\$tr)_v} s v) v) \mid$$

lemma *tcontr-mwb-lens* [*simp*]: *mwb-lens tt*
by (*unfold-locales, simp-all add: lens-defs prod.case-eq-if*)

lemma *itrace-mwb-lens* [*simp*]: *mwb-lens* **it**
by (*unfold-locales*, *simp-all add: lens-defs prod.case-eq-if*)

syntax

$$\begin{aligned} \text{-svid-contr} &:: \text{svid } (tt) \\ \text{-svid-itrace} &:: \text{svid } (\mathbf{it}) \\ \text{-utr-iter} &:: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic } (\text{iter}[-])'(-) \end{aligned}$$

translations

$$\begin{aligned} \text{-svid-tcontr} &== \text{CONST } t\text{contr} \\ \text{-svid-itrace} &== \text{CONST } i\text{trace} \\ \text{iter}[n](P) &== \text{CONST } u\text{op } (\text{CONST } tr\text{-iter } n) P \end{aligned}$$

lemma *tcontr-alt-def*: $\&tt = (\$str' - \$tr)$
by (*rel-auto*)

lemma *tcontr-alt-def'*: *utp-expr.var tt* = (*\$tr'* - *\$tr*)
by (*rel-auto*)

```

lemma tt-indeps [simp]:
  assumes  $x \bowtie (\$tr)_v \ x \bowtie (\$tr')_v$ 
  shows  $x \bowtie tt \ tt \bowtie x$ 
  using assms
  by (unfold lens-indep-def, safe, simp-all add: tcontr-def, (metis lens-indep-get var-update-out)+)

```

```

lemma itrace-indeps [simp]:
  assumes  $x \bowtie (\$tr)_v \ x \bowtie (\$tr')_v$ 
  shows  $x \bowtie \mathbf{it} \ \mathbf{it} \bowtie x$ 
  using assms by (unfold lens-indep-def, safe, simp-all add: lens-defs)

```

We lift a few trace properties from the trace class using *transfer*.

```

lemma ueexpr-diff-zero [simp]:
  fixes a :: ('α::trace, 'a) ueexpr
  shows a - 0 = a
  by (simp add: minus-ueexpr-def zero-ueexpr-def, transfer, auto)

```

lemma *ueexpr-add-diff-cancel-left* [*simp*]:
fixes $a\ b :: ('a :: \text{trace}, 'a)\ \text{ueexpr}$
shows $(a + b) - a = b$
by (*simp add: minus-ueexpr-def plus-ueexpr-def, transfer, auto*)

lemma *iter-0* [simp]: $iter[0](t) = U(\square)$
by (*transfer, simp add: zero-list-def*)

end

3 Events for Reactive Processes

```
theory utp-rea-event
imports UTP.utp
```

begin

3.1 Events

Events of some type $'\vartheta$ are just the elements of that type.

type-synonym $'\vartheta \text{ event} = '\vartheta$

3.2 Channels

Typed channels are modelled as functions. Below, $'a$ determines the channel type and $'\vartheta$ the underlying event type. As with values, it is difficult to introduce channels as monomorphic types due to the fact that they can have arbitrary parametrisations in term of $'a$. Applying a channel to an element of its type yields an event, as we may expect. Though this is not formalised here, we may also sensibly assume that all channel- representing functions are injective. Note: is there benefit in formalising this here?

type-synonym $('a, '\vartheta) \text{ chan} = 'a \Rightarrow '\vartheta \text{ event}$

A downside of the approach is that the event type $'\vartheta$ must be able to encode *all* events of a process model, and hence cannot be fixed upfront for a single channel or channel set. To do so, we actually require a notion of ‘extensible’ datatypes, in analogy to extensible record types. Another solution is to encode a notion of channel scoping that namely uses *sum* types to lift channel types into extensible ones, that is using channel-set specific scoping operators. This is a current work in progress.

3.2.1 Operators

The Z type of a channel corresponds to the entire carrier of the underlying HOL type of that channel.

definition $\text{chan-type} :: ('a, '\vartheta) \text{ chan} \Rightarrow 'a \text{ set } (\delta_u)$ **where**
 $[\text{upred-defs}]: \delta_u \text{ c} = \text{UNIV}$

The next lifted function creates an expression that yields a channel event, from an expression on the channel type $'a$.

definition $\text{chan-apply} ::$
 $('a, '\vartheta) \text{ chan} \Rightarrow ('a, '\alpha) \text{ uexpr} \Rightarrow (''\vartheta \text{ event}, '\alpha) \text{ uexpr } ((' \cdot / - ')_u)$ **where**
 $[\text{upred-defs}]: (c \cdot e)_u = \text{uop } c \ e$

no-utp-lift $\text{chan-apply } (0)$

lemma $\text{unrest-chan-apply } [\text{unrest}]: x \# e \Longrightarrow x \# (c \cdot e)_u$
by (rel-auto)

lemma $\text{usubst-chan-apply } [\text{usubst}]: \sigma \dagger (c \cdot v)_u = (c \cdot \sigma \dagger v)_u$
by (rel-auto)

lemma $\text{msubst-event } [\text{usubst}]:$
 $(c \cdot v \ x)_u \llbracket x \rightarrow u \rrbracket = (c \cdot (v \ x) \llbracket x \rightarrow u \rrbracket)_u$
by (pred-simp)

lemma $\text{msubst-event-2 } [\text{usubst}]:$
 $(c \cdot v \ x \ y)_u \llbracket (x, y) \rightarrow u \rrbracket = (c \cdot (v \ x \ y) \llbracket (x, y) \rightarrow u \rrbracket)_u$
by $(\text{pred-simp})+$

lemma *aext-event* [*alpha*]: $(c \cdot v)_u \oplus_p a = (c \cdot v \oplus_p a)_u$
by (*pred-auto*)

end

4 Reactive Healthiness Conditions

theory *utp-rea-healths*
imports *utp-rea-core*
begin

4.1 R1: Events cannot be undone

definition *R1* :: $('t::\text{trace}, ' \alpha, ' \beta) \text{ rel-rp} \Rightarrow ('t, ' \alpha, ' \beta) \text{ rel-rp}$ **where**
R1-def [*upred-defs*]: $R1(P) = (P \wedge (\$tr \leq_u \$tr'))$

utp-const *R1*

lemma *R1-idem*: $R1(R1(P)) = R1(P)$
by *pred-auto*

lemma *R1-Idempotent* [*closure*]: *Idempotent R1*
by (*simp add: Idempotent-def R1-idem*)

lemma *R1-mono*: $P \sqsubseteq Q \Longrightarrow R1(P) \sqsubseteq R1(Q)$
by *pred-auto*

lemma *R1-Monotonic*: *Monotonic R1*
by (*simp add: mono-def R1-mono*)

lemma *R1-Continuous*: *Continuous R1*
by (*auto simp add: Continuous-def, rel-auto*)

lemma *R1-unrest* [*unrest*]: $\llbracket x \bowtie \text{in-var } tr; x \bowtie \text{out-var } tr; x \# P \rrbracket \Longrightarrow x \# R1(P)$
by (*simp add: R1-def unrest lens-indep-sym*)

lemma *R1-false*: $R1(\text{false}) = \text{false}$
by *pred-auto*

lemma *R1-conj*: $R1(P \wedge Q) = (R1(P) \wedge R1(Q))$
by *pred-auto*

lemma *conj-R1-closed-1* [*closure*]: $P \text{ is } R1 \Longrightarrow (P \wedge Q) \text{ is } R1$
by (*rel-blast*)

lemma *conj-R1-closed-2* [*closure*]: $Q \text{ is } R1 \Longrightarrow (P \wedge Q) \text{ is } R1$
by (*rel-blast*)

lemma *R1-disj*: $R1(P \vee Q) = (R1(P) \vee R1(Q))$
by *pred-auto*

lemma *disj-R1-closed* [*closure*]: $\llbracket P \text{ is } R1; Q \text{ is } R1 \rrbracket \Longrightarrow (P \vee Q) \text{ is } R1$
by (*simp add: Healthy-def R1-def utp-pred-laws.inf-sup-distrib2*)

lemma *R1-impl*: $R1(P \Rightarrow Q) = ((\neg R1(\neg P)) \Rightarrow R1(Q))$
by (*rel-auto*)

lemma *R1-inf*: $R1(P \sqcap Q) = (R1(P) \sqcap R1(Q))$
by *pred-auto*

lemma *R1-USUP*:
 $R1(\bigsqcap i \in A \cdot P(i)) = (\bigsqcap i \in A \cdot R1(P(i)))$
by (*rel-auto*)

lemma *R1-Sup [closure]*: $\llbracket \bigwedge P. P \in A \implies P \text{ is } R1; A \neq \{\} \rrbracket \implies \bigsqcap A \text{ is } R1$
using *R1-Continuous* **by** (*auto simp add: Continuous-def Healthy-def*)

lemma *R1-UINF*:
assumes $A \neq \{\}$
shows $R1(\bigsqcup i \in A \cdot P(i)) = (\bigsqcup i \in A \cdot R1(P(i)))$
using *assms* **by** (*rel-auto*)

lemma *R1-UINF-ind*:
 $R1(\bigsqcup i \cdot P(i)) = (\bigsqcup i \cdot R1(P(i)))$
by (*rel-auto*)

lemma *UINF-ind-R1-closed [closure]*:
 $\llbracket \bigwedge i. P(i) \text{ is } R1 \rrbracket \implies (\bigsqcap i \cdot P(i)) \text{ is } R1$
by (*rel-blast*)

lemma *UINF-R1-closed [closure]*:
 $\llbracket \bigwedge i. P \ i \text{ is } R1 \rrbracket \implies (\bigsqcap i \in A \cdot P \ i) \text{ is } R1$
by (*rel-blast*)

lemma *tr-ext-conj-R1 [closure]*:
 $\$tr' =_u \$tr \hat{\ }_u e \wedge P \text{ is } R1$
by (*rel-auto, simp add: Prefix-Order.prefixI*)

lemma *tr-id-conj-R1 [closure]*:
 $\$tr' =_u \$tr \wedge P \text{ is } R1$
by (*rel-auto*)

lemma *R1-extend-conj*: $R1(P \wedge Q) = (R1(P) \wedge Q)$
by *pred-auto*

lemma *R1-extend-conj'*: $R1(P \wedge Q) = (P \wedge R1(Q))$
by *pred-auto*

lemma *R1-cond*: $R1(P \triangleleft b \triangleright Q) = (R1(P) \triangleleft b \triangleright R1(Q))$
by (*rel-auto*)

lemma *R1-cond'*: $R1(P \triangleleft b \triangleright Q) = (R1(P) \triangleleft R1(b) \triangleright R1(Q))$
by (*rel-auto*)

lemma *R1-negate-R1*: $R1(\neg R1(P)) = R1(\neg P)$
by *pred-auto*

lemma *R1-wait-true [usubst]*: $(R1 \ P)_t = R1(P)_t$
by *pred-auto*

lemma *R1-wait-false* [usubst]: $(R1\ P)_f = R1(P)_f$
by *pred-auto*

lemma *R1-wait'-true* [usubst]: $(R1\ P)\llbracket true/\$wait' \rrbracket = R1(P\llbracket true/\$wait' \rrbracket)$
by *(rel-auto)*

lemma *R1-wait'-false* [usubst]: $(R1\ P)\llbracket false/\$wait' \rrbracket = R1(P\llbracket false/\$wait' \rrbracket)$
by *(rel-auto)*

lemma *R1-wait-false-closed* [closure]: $P\ is\ R1 \implies P\llbracket false/\$wait \rrbracket\ is\ R1$
by *(rel-auto)*

lemma *R1-wait'-false-closed* [closure]: $P\ is\ R1 \implies P\llbracket false/\$wait' \rrbracket\ is\ R1$
by *(rel-auto)*

lemma *R1-skip*: $R1(II) = II$
by *(rel-auto)*

lemma *skip-is-R1* [closure]: $II\ is\ R1$
by *(rel-auto)*

lemma *subst-R1*: $\llbracket \$tr\ \#_s\ \sigma; \$tr'\ \#_s\ \sigma \rrbracket \implies \sigma \dagger (R1\ P) = R1(\sigma \dagger P)$
by *(simp add: R1-def usubst)*

lemma *subst-R1-closed* [closure]: $\llbracket \$tr\ \#_s\ \sigma; \$tr'\ \#_s\ \sigma; P\ is\ R1 \rrbracket \implies \sigma \dagger P\ is\ R1$
by *(metis Healthy-def subst-R1)*

lemma *R1-by-refinement*:
 $P\ is\ R1 \iff ((\$tr \leq_u \$tr') \sqsubseteq P)$
by *(rel-blast)*

lemma *R1-trace-extension* [closure]:
 $\$tr' \geq_u \$tr \hat{^}_u e\ is\ R1$
by *(rel-auto)*

lemma *tr-le-trans*:
 $((\$tr \leq_u \$tr') ;; (\$tr \leq_u \$tr')) = (\$tr \leq_u \$tr')$
by *(rel-auto)*

lemma *R1-seqr*:
 $R1(R1(P) ;; R1(Q)) = (R1(P) ;; R1(Q))$
by *(rel-auto)*

lemma *R1-seqr-closure* [closure]:
assumes $P\ is\ R1\ Q\ is\ R1$
shows $(P ;; Q)\ is\ R1$
using *assms unfolding R1-by-refinement*
by *(metis seqr-mono tr-le-trans)*

lemma *R1-power* [closure]: $P\ is\ R1 \implies P^{\wedge} n\ is\ R1$
by *(induct n, simp-all add: upred-semiring.power-Suc closure)*

lemma *R1-true-comp* [simp]: $(R1(true) ;; R1(true)) = R1(true)$
by *(rel-auto)*

lemma *R1-ok'-true*: $(R1(P))^t = R1(P^t)$
by *pred-auto*

lemma *R1-ok'-false*: $(R1(P))^f = R1(P^f)$
by *pred-auto*

lemma *R1-ok-true*: $(R1(P))\llbracket true/\$ok \rrbracket = R1(P\llbracket true/\$ok \rrbracket)$
by *pred-auto*

lemma *R1-ok-false*: $(R1(P))\llbracket false/\$ok \rrbracket = R1(P\llbracket false/\$ok \rrbracket)$
by *pred-auto*

lemma *seqr-R1-true-right*: $((P ;; R1(true)) \vee P) = (P ;; (\$tr \leq_u \$tr'))$
by (*rel-auto*)

lemma *conj-R1-true-right*: $(P ;; R1(true) \wedge Q ;; R1(true)) ;; R1(true) = (P ;; R1(true) \wedge Q ;; R1(true))$
apply (*rel-auto*) **using** *dual-order.trans* **by** *blast+*

lemma *R1-extend-conj-unrest*: $\llbracket \$tr \# Q; \$tr' \# Q \rrbracket \implies R1(P \wedge Q) = (R1(P) \wedge Q)$
by *pred-auto*

lemma *R1-extend-conj-unrest'*: $\llbracket \$tr \# P; \$tr' \# P \rrbracket \implies R1(P \wedge Q) = (P \wedge R1(Q))$
by *pred-auto*

lemma *R1-tr'-eq-tr*: $R1(\$tr' =_u \$tr) = (\$tr' =_u \$tr)$
by (*rel-auto*)

lemma *R1-tr-less-tr'*: $R1(\$tr <_u \$tr') = (\$tr <_u \$tr')$
by (*rel-auto*)

lemma *tr-strict-prefix-R1-closed* [*closure*]: $\$tr <_u \tr' is *R1*
by (*rel-auto*)

lemma *R1-H2-commute*: $R1(H2(P)) = H2(R1(P))$
by (*simp add: H2-split R1-def usubst, rel-auto*)

4.2 R2: No dependence upon trace history

There are various ways of expressing *R2*, which are enumerated below.

definition *R2a* :: $(t::trace, 'a, 'b) \text{ rel-rp} \Rightarrow (t, 'a, 'b) \text{ rel-rp}$ **where**
[*upred-defs*]: $R2a(P) = (\bigcap s \cdot P\llbracket \llbracket s \rrbracket, (\llbracket s \rrbracket + (\$tr' - \$tr)) / \$tr, \$tr' \rrbracket)$

definition *R2a'* :: $(t::trace, 'a, 'b) \text{ rel-rp} \Rightarrow (t, 'a, 'b) \text{ rel-rp}$ **where**
[*upred-defs*]: $R2a'(P) = (R2a(P) \triangleleft R1(true) \triangleright P)$

definition *R2s* :: $(t::trace, 'a, 'b) \text{ rel-rp} \Rightarrow (t, 'a, 'b) \text{ rel-rp}$ **where**
[*upred-defs*]: $R2s(P) = (P\llbracket 0/\$tr \rrbracket\llbracket (\$tr' - \$tr) / \$tr' \rrbracket)$

definition *R2* :: $(t::trace, 'a, 'b) \text{ rel-rp} \Rightarrow (t, 'a, 'b) \text{ rel-rp}$ **where**
[*upred-defs*]: $R2(P) = R1(R2s(P))$

definition *R2c* :: $(t::trace, 'a, 'b) \text{ rel-rp} \Rightarrow (t, 'a, 'b) \text{ rel-rp}$ **where**
[*upred-defs*]: $R2c(P) = (R2s(P) \triangleleft R1(true) \triangleright P)$

$R2a$ and $R2s$ are the standard definitions from the UTP book [5]. An issue with these forms is that their definition depends upon $R1$ also being satisfied [4], since otherwise the trace minus operator is not well defined. We overcome this with our own version, $R2c$, which applies $R2s$ if $R1$ holds, and otherwise has no effect. This latter healthiness condition can therefore be reasoned about independently of $R1$, which is useful in some circumstances.

lemma *unrest-ok-R2s* [unrest]: $\$ok \# P \implies \$ok \# R2s(P)$
by (*simp add: R2s-def unrest*)

lemma *unrest-ok'-R2s* [unrest]: $\$ok' \# P \implies \$ok' \# R2s(P)$
by (*simp add: R2s-def unrest*)

lemma *unrest-ok-R2c* [unrest]: $\$ok \# P \implies \$ok \# R2c(P)$
by (*simp add: R2c-def unrest*)

lemma *unrest-ok'-R2c* [unrest]: $\$ok' \# P \implies \$ok' \# R2c(P)$
by (*simp add: R2c-def unrest*)

lemma *R2s-unrest* [unrest]: $\llbracket vwb\text{-}lens\ x; x \bowtie in\text{-}var\ tr; x \bowtie out\text{-}var\ tr; x \# P \rrbracket \implies x \# R2s(P)$
by (*simp add: R2s-def unrest usubst lens-indep-sym*)

lemma *R2s-subst-wait-true* [usubst]:
 $(R2s(P))\llbracket true/\$wait \rrbracket = R2s(P\llbracket true/\$wait \rrbracket)$
by (*simp add: R2s-def usubst unrest*)

lemma *R2s-subst-wait'-true* [usubst]:
 $(R2s(P))\llbracket true/\$wait' \rrbracket = R2s(P\llbracket true/\$wait' \rrbracket)$
by (*simp add: R2s-def usubst unrest*)

lemma *R2-subst-wait-true* [usubst]:
 $(R2(P))\llbracket true/\$wait \rrbracket = R2(P\llbracket true/\$wait \rrbracket)$
by (*simp add: R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait'-true* [usubst]:
 $(R2(P))\llbracket true/\$wait' \rrbracket = R2(P\llbracket true/\$wait' \rrbracket)$
by (*simp add: R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait-false* [usubst]:
 $(R2(P))\llbracket false/\$wait \rrbracket = R2(P\llbracket false/\$wait \rrbracket)$
by (*simp add: R2-def R1-def R2s-def usubst unrest*)

lemma *R2-subst-wait'-false* [usubst]:
 $(R2(P))\llbracket false/\$wait' \rrbracket = R2(P\llbracket false/\$wait' \rrbracket)$
by (*simp add: R2-def R1-def R2s-def usubst unrest*)

lemma *R2c-R2s-absorb*: $R2c(R2s(P)) = R2s(P)$
by (*rel-auto*)

lemma *R2a-R2s*: $R2a(R2s(P)) = R2s(P)$
by (*rel-auto*)

lemma *R2s-R2a*: $R2s(R2a(P)) = R2a(P)$
by (*rel-auto*)

lemma *R2a-equiv-R2s*: $P \text{ is } R2a \longleftrightarrow P \text{ is } R2s$
by (*metis Healthy-def' R2a-R2s R2s-R2a*)

lemma *R2a-idem*: $R2a(R2a(P)) = R2a(P)$
by (*rel-auto*)

lemma *R2a'-idem*: $R2a'(R2a'(P)) = R2a'(P)$
by (*rel-auto*)

lemma *R2a-mono*: $P \sqsubseteq Q \implies R2a(P) \sqsubseteq R2a(Q)$
by (*rel-blast*)

lemma *R2a'-mono*: $P \sqsubseteq Q \implies R2a'(P) \sqsubseteq R2a'(Q)$
by (*rel-blast*)

lemma *R2a'-weakening*: $R2a'(P) \sqsubseteq P$
apply (*rel-simp*)
apply (*rename-tac ok wait tr more ok' wait' tr' more'*)
apply (*rule-tac x=tr in exI*)
apply (*simp add: diff-add-cancel-left'*)
done

lemma *R2s-idem*: $R2s(R2s(P)) = R2s(P)$
by (*pred-auto*)

lemma *R2-idem*: $R2(R2(P)) = R2(P)$
by (*pred-auto*)

lemma *R2-mono*: $P \sqsubseteq Q \implies R2(P) \sqsubseteq R2(Q)$
by (*pred-auto*)

lemma *R2-implies-R1 [closure]*: $P \text{ is } R2 \implies P \text{ is } R1$
by (*rel-blast*)

lemma *R2c-Continuous*: *Continuous* $R2c$
by (*rel-simp*)

lemma *R2c-lit*: $R2c(\ll x \gg) = \ll x \gg$
by (*rel-auto*)

lemma *tr-strict-prefix-R2c-closed [closure]*: $\$tr <_u \$tr' \text{ is } R2c$
by (*rel-auto*)

lemma *R2s-conj*: $R2s(P \wedge Q) = (R2s(P) \wedge R2s(Q))$
by (*pred-auto*)

lemma *R2-conj*: $R2(P \wedge Q) = (R2(P) \wedge R2(Q))$
by (*pred-auto*)

lemma *R2s-disj*: $R2s(P \vee Q) = (R2s(P) \vee R2s(Q))$
by *pred-auto*

lemma *R2s-USUP*:
 $R2s(\bigcap i \in A \cdot P(i)) = (\bigcap i \in A \cdot R2s(P(i)))$
by (*simp add: R2s-def usubst*)

lemma *R2c-USUP*:

$R2c(\bigsqcap i \in A \cdot P(i)) = (\bigsqcap i \in A \cdot R2c(P(i)))$
by (*rel-auto*)

lemma *R2s-UINF*:

$R2s(\bigsqcup i \in A \cdot P(i)) = (\bigsqcup i \in A \cdot R2s(P(i)))$
by (*simp add: R2s-def usubst*)

lemma *R2c-UINF*:

$R2c(\bigsqcup i \in A \cdot P(i)) = (\bigsqcup i \in A \cdot R2c(P(i)))$
by (*rel-auto*)

lemma *R2-disj*: $R2(P \vee Q) = (R2(P) \vee R2(Q))$

by (*pred-auto*)

lemma *R2s-not*: $R2s(\neg P) = (\neg R2s(P))$

by *pred-auto*

lemma *R2s-condr*: $R2s(P \triangleleft b \triangleright Q) = (R2s(P) \triangleleft R2s(b) \triangleright R2s(Q))$

by (*rel-auto*)

lemma *R2-condr*: $R2(P \triangleleft b \triangleright Q) = (R2(P) \triangleleft R2(b) \triangleright R2(Q))$

by (*rel-auto*)

lemma *R2-condr'*: $R2(P \triangleleft b \triangleright Q) = (R2(P) \triangleleft R2s(b) \triangleright R2(Q))$

by (*rel-auto*)

lemma *R2s-ok*: $R2s(\$ok) = \ok

by (*rel-auto*)

lemma *R2s-ok'*: $R2s(\$ok') = \ok'

by (*rel-auto*)

lemma *R2s-wait*: $R2s(\$wait) = \$wait$

by (*rel-auto*)

lemma *R2s-wait'*: $R2s(\$wait') = \$wait'$

by (*rel-auto*)

lemma *R2s-true*: $R2s(true) = true$

by *pred-auto*

lemma *R2s-false*: $R2s(false) = false$

by *pred-auto*

lemma *true-is-R2s*:

true is R2s

by (*simp add: Healthy-def R2s-true*)

lemma *R2s-lift-rea*: $R2s(\lceil P \rceil_R) = \lceil P \rceil_R$

by (*simp add: R2s-def usubst unrest*)

lemma *R2c-lift-rea*: $R2c(\lceil P \rceil_R) = \lceil P \rceil_R$

by (*simp add: R2c-def R2s-lift-rea cond-idem usubst unrest*)

lemma *R2c-true*: $R2c(true) = true$

by (rel-auto)

lemma *R2c-false*: $R2c(false) = false$
by (rel-auto)

lemma *R2c-and*: $R2c(P \wedge Q) = (R2c(P) \wedge R2c(Q))$
by (rel-auto)

lemma *conj-R2c-closed* [closure]: $\llbracket P \text{ is } R2c; Q \text{ is } R2c \rrbracket \implies (P \wedge Q) \text{ is } R2c$
by (simp add: Healthy-def R2c-and)

lemma *R2c-disj*: $R2c(P \vee Q) = (R2c(P) \vee R2c(Q))$
by (rel-auto)

lemma *R2c-inf*: $R2c(P \sqcap Q) = (R2c(P) \sqcap R2c(Q))$
by (rel-auto)

lemma *R2c-not*: $R2c(\neg P) = (\neg R2c(P))$
by (rel-auto)

lemma *R2c-ok*: $R2c(\$ok) = (\$ok)$
by (rel-auto)

lemma *R2c-ok'*: $R2c(\$ok') = (\$ok')$
by (rel-auto)

lemma *R2c-wait*: $R2c(\$wait) = \$wait$
by (rel-auto)

lemma *R2c-wait'*: $R2c(\$wait') = \$wait'$
by (rel-auto)

lemma *R2c-wait'-true* [usubst]: $(R2c\ P) \llbracket true/\$wait' \rrbracket = R2c(P \llbracket true/\$wait' \rrbracket)$
by (rel-auto)

lemma *R2c-wait'-false* [usubst]: $(R2c\ P) \llbracket false/\$wait' \rrbracket = R2c(P \llbracket false/\$wait' \rrbracket)$
by (rel-auto)

lemma *R2c-tr'-minus-tr*: $R2c(\$tr' =_u \$tr) = (\$tr' =_u \$tr)$
apply (rel-auto) using minus-zero-eq by blast

lemma *R2c-tr-le-tr'*: $R2c(\$tr \leq_u \$tr') = (\$tr \leq_u \$tr')$
by (rel-auto)

lemma *R2c-tr'-ge-tr*: $R2c(\$tr' \geq_u \$tr) = (\$tr' \geq_u \$tr)$
by (rel-auto)

lemma *R2c-tr-less-tr'*: $R2c(\$tr <_u \$tr') = (\$tr <_u \$tr')$
by (rel-auto)

lemma *R2c-condr*: $R2c(P \triangleleft b \triangleright Q) = (R2c(P) \triangleleft R2c(b) \triangleright R2c(Q))$
by (rel-auto)

lemma *R2c-shAll*: $R2c(\forall x \cdot P\ x) = (\forall x \cdot R2c(P\ x))$
by (rel-auto)

lemma *R2c-impl*: $R2c(P \Rightarrow Q) = (R2c(P) \Rightarrow R2c(Q))$
 by (metis (no-types, lifting) R2c-and R2c-not double-negation impl-alt-def not-conj-deMorgans)

lemma *R2c-skip-r*: $R2c(II) = II$

proof –

have $R2c(II) = R2c(\$tr' =_u \$tr \wedge II \upharpoonright_{\alpha} tr)$
 by (subst skip-r-unfold[of tr], simp-all)
 also have $\dots = (R2c(\$tr' =_u \$tr) \wedge II \upharpoonright_{\alpha} tr)$
 by (simp add: R2c-and, simp add: R2c-def R2s-def usubst unrest cond-idem)
 also have $\dots = (\$tr' =_u \$tr \wedge II \upharpoonright_{\alpha} tr)$
 by (simp add: R2c-tr'-minus-tr)
 finally show ?thesis
 by (subst skip-r-unfold[of tr], simp-all)

qed

lemma *R1-R2c-commute*: $R1(R2c(P)) = R2c(R1(P))$

by (rel-auto)

lemma *R1-R2c-is-R2*: $R1(R2c(P)) = R2(P)$

by (rel-auto)

lemma *R1-R2s-R2c*: $R1(R2s(P)) = R1(R2c(P))$

by (rel-auto)

lemma *R1-R2s-tr-wait*:

$R1(R2s(\$tr' =_u \$tr \wedge \$wait')) = (\$tr' =_u \$tr \wedge \$wait')$
 apply rel-auto using minus-zero-eq by blast

lemma *R1-R2s-tr'-eq-tr*:

$R1(R2s(\$tr' =_u \$tr)) = (\$tr' =_u \$tr)$
 apply (rel-auto) using minus-zero-eq by blast

lemma *R1-R2s-tr'-extend-tr*:

$\llbracket \$tr \# v; \$tr' \# v \rrbracket \Longrightarrow R1(R2s(\$tr' =_u \$tr \hat{^}_u v)) = (\$tr' =_u \$tr \hat{^}_u v)$
 apply (rel-auto)
 apply (metis append-minus)
 apply (simp add: Prefix-Order.prefixI)
 done

lemma *R2-tr-prefix*: $R2(\$tr \leq_u \$tr') = (\$tr \leq_u \$tr')$

by (pred-auto)

lemma *R2-form*:

$R2(P) = (\exists tt_0 \cdot P[\llbracket 0/\$tr \rrbracket \llbracket \ll tt_0 \gg / \$tr' \rrbracket \wedge \$tr' =_u \$tr + \ll tt_0 \gg])$
 by (rel-auto, metis trace-class.add-diff-cancel-left trace-class.le-iff-add)

lemma *R2-subst-tr*:

assumes P is $R2$
 shows $[\$tr \mapsto_s tr_0, \$tr' \mapsto_s tr_0 + t] \vdash P = [\$tr \mapsto_s 0, \$tr' \mapsto_s t] \vdash P$

proof –

have $[\$tr \mapsto_s tr_0, \$tr' \mapsto_s tr_0 + t] \vdash R2 P = [\$tr \mapsto_s 0, \$tr' \mapsto_s t] \vdash R2 P$
 by (rel-auto)

thus ?thesis

by (simp add: Healthy-if assms)

qed

lemma *R2-seqr-form*:

shows $(R2(P) ;; R2(Q)) =$
 $(\exists \ tt_1 \cdot \exists \ tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr']) ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr']))$
 $\wedge (\$tr' =_u \$tr + \langle tt_1 \rangle + \langle tt_2 \rangle))$

proof –

have $(R2(P) ;; R2(Q)) = (\exists \ tr_0 \cdot (R2(P)[\langle tr_0 \rangle / \$tr'] ;; (R2(Q)[\langle tr_0 \rangle / \$tr]))$
by (*subst seqr-middle*[of *tr*], *simp-all*)

also have ... =

$(\exists \ tr_0 \cdot \exists \ tt_1 \cdot \exists \ tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr'] \wedge \langle tr_0 \rangle =_u \$tr + \langle tt_1 \rangle) ;;$
 $(Q[0/\$tr][\langle tt_2 \rangle / \$tr'] \wedge \$tr' =_u \langle tr_0 \rangle + \langle tt_2 \rangle)))$

by (*simp add: R2-form usubst unrest uquant-lift, rel-blast*)

also have ... =

$(\exists \ tr_0 \cdot \exists \ tt_1 \cdot \exists \ tt_2 \cdot ((\langle tr_0 \rangle =_u \$tr + \langle tt_1 \rangle \wedge P[0/\$tr][\langle tt_1 \rangle / \$tr']) ;;$
 $(Q[0/\$tr][\langle tt_2 \rangle / \$tr'] \wedge \$tr' =_u \langle tr_0 \rangle + \langle tt_2 \rangle)))$

by (*simp add: conj-comm*)

also have ... =

$(\exists \ tt_1 \cdot \exists \ tt_2 \cdot \exists \ tr_0 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr']) ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr']))$
 $\wedge \langle tr_0 \rangle =_u \$tr + \langle tt_1 \rangle \wedge \$tr' =_u \langle tr_0 \rangle + \langle tt_2 \rangle)$

by (*rel-blast*)

also have ... =

$(\exists \ tt_1 \cdot \exists \ tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr']) ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr']))$
 $\wedge (\exists \ tr_0 \cdot \langle tr_0 \rangle =_u \$tr + \langle tt_1 \rangle \wedge \$tr' =_u \langle tr_0 \rangle + \langle tt_2 \rangle))$

by (*rel-auto*)

also have ... =

$(\exists \ tt_1 \cdot \exists \ tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr']) ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr']))$
 $\wedge (\$tr' =_u \$tr + \langle tt_1 \rangle + \langle tt_2 \rangle))$

by (*rel-auto*)

finally show *?thesis* .

qed

lemma *R2-seqr-form'*:

assumes *P is R2 Q is R2*

shows $P ;; Q =$

$(\exists \ tt_1 \cdot \exists \ tt_2 \cdot ((P[0/\$tr][\langle tt_1 \rangle / \$tr']) ;; (Q[0/\$tr][\langle tt_2 \rangle / \$tr']))$
 $\wedge (\$tr' =_u \$tr + \langle tt_1 \rangle + \langle tt_2 \rangle))$

using *R2-seqr-form*[of *P Q*] **by** (*simp add: Healthy-if assms*)

lemma *R2-seqr-form''*:

assumes *P is R2 Q is R2*

shows $P ;; Q =$

$(\exists \ (tt_1, tt_2) \cdot ((P[0, \langle tt_1 \rangle / \$tr, \$tr']) ;; (Q[0, \langle tt_2 \rangle / \$tr, \$tr']))$
 $\wedge (\$tr' =_u \$tr + \langle tt_1 \rangle + \langle tt_2 \rangle))$

by (*subst R2-seqr-form', simp-all add: assms, rel-auto*)

lemma *R2-tr-middle*:

assumes *P is R2 Q is R2*

shows $(\exists \ tr_0 \cdot (P[\langle tr_0 \rangle / \$tr'] ;; Q[\langle tr_0 \rangle / \$tr]) \wedge \langle tr_0 \rangle \leq_u \$tr') = (P ;; Q)$

proof –

have $(P ;; Q) = (\exists \ tr_0 \cdot (P[\langle tr_0 \rangle / \$tr'] ;; Q[\langle tr_0 \rangle / \$tr]))$

by (*simp add: seqr-middle*)

also have ... = $(\exists \ tr_0 \cdot ((R2 \ P)[\langle tr_0 \rangle / \$tr'] ;; (R2 \ Q)[\langle tr_0 \rangle / \$tr]))$

by (*simp add: assms Healthy-if*)

also have ... = $(\exists \ tr_0 \cdot ((R2 \ P)[\langle tr_0 \rangle / \$tr'] ;; (R2 \ Q)[\langle tr_0 \rangle / \$tr]) \wedge \langle tr_0 \rangle \leq_u \$tr')$

by (*rel-auto*)
 also have ... = $(\exists tr_0 \cdot (P[\ll tr_0 \gg / \$tr'] ;; Q[\ll tr_0 \gg / \$tr]) \wedge \ll tr_0 \gg \leq_u \$tr')$
 by (*simp add: assms Healthy-if*)
 finally show ?thesis ..
 qed

lemma *R2-seqr-distribute*:

fixes $P :: ('t::trace, 'α, 'β) \text{ rel-rp}$ and $Q :: ('t, 'β, 'γ) \text{ rel-rp}$

shows $R2(R2(P) ;; R2(Q)) = (R2(P) ;; R2(Q))$

proof –

have $R2(R2(P) ;; R2(Q)) =$

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\ll tt_1 \gg / \$tr'] ;; Q[0/\$tr][\ll tt_2 \gg / \$tr'])(\$tr' - \$tr)/\$tr') \wedge \$tr' - \$tr =_u \ll tt_1 \gg + \ll tt_2 \gg) \wedge \$tr' \geq_u \$tr)$

by (*simp add: R2-seqr-form, simp add: R2s-def usubst unrest, rel-auto*)

also have ... =

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\ll tt_1 \gg / \$tr'] ;; Q[0/\$tr][\ll tt_2 \gg / \$tr'])(\ll tt_1 \gg + \ll tt_2 \gg)/\$tr') \wedge \$tr' - \$tr =_u \ll tt_1 \gg + \ll tt_2 \gg) \wedge \$tr' \geq_u \$tr)$

by (*subst subst-eq-replace, simp*)

also have ... =

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\ll tt_1 \gg / \$tr'] ;; Q[0/\$tr][\ll tt_2 \gg / \$tr'])(\ll tt_1 \gg + \ll tt_2 \gg) \wedge \$tr' \geq_u \$tr)$

by (*rel-auto*)

also have ... =

$(\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\ll tt_1 \gg / \$tr'] ;; Q[0/\$tr][\ll tt_2 \gg / \$tr'])(\$tr' - \$tr =_u \ll tt_1 \gg + \ll tt_2 \gg \wedge \$tr' \geq_u \$tr))$

by (*pred-auto*)

also have ... =

$((\exists tt_1 \cdot \exists tt_2 \cdot (P[0/\$tr][\ll tt_1 \gg / \$tr'] ;; Q[0/\$tr][\ll tt_2 \gg / \$tr'])(\$tr' =_u \$tr + \ll tt_1 \gg + \ll tt_2 \gg))$

proof –

have $\bigwedge tt_1 tt_2. (((\$tr' - \$tr =_u \ll tt_1 \gg + \ll tt_2 \gg) \wedge \$tr' \geq_u \$tr) :: ('t, 'α, 'γ) \text{ rel-rp})$
 $= (\$tr' =_u \$tr + \ll tt_1 \gg + \ll tt_2 \gg)$

apply (*rel-auto*)

apply (*metis add.assoc diff-add-cancel-left*)

apply (*simp add: add.assoc*)

apply (*meson le-add order-trans*)

done

thus ?thesis by *simp*

qed

also have ... = $(R2(P) ;; R2(Q))$

by (*simp add: R2-seqr-form*)

finally show ?thesis .

qed

lemma *R2-seqr-closure* [closure]:

assumes P is $R2$ Q is $R2$

shows $(P ;; Q)$ is $R2$

by (*metis Healthy-def' R2-seqr-distribute assms(1) assms(2)*)

lemma *false-R2* [closure]: *false* is $R2$

by (*rel-auto*)

lemma *R1-R2-commute*:

$R1(R2(P)) = R2(R1(P))$

by (*pred-auto*)

lemma *R2-R1-form*: $R2(R1(P)) = R1(R2s(P))$
by (*rel-auto*)

lemma *R2s-H1-commute*:
 $R2s(H1(P)) = H1(R2s(P))$
by (*rel-auto*)

lemma *R2s-H2-commute*:
 $R2s(H2(P)) = H2(R2s(P))$
by (*simp add: H2-split R2s-def usubst*)

lemma *R2-R1-seq-drop-left*:
 $R2(R1(P) ;; R1(Q)) = R2(P ;; R1(Q))$
by (*rel-auto*)

lemma *R2c-idem*: $R2c(R2c(P)) = R2c(P)$
by (*rel-auto*)

lemma *R2c-Idempotent [closure]*: *Idempotent R2c*
by (*simp add: Idempotent-def R2c-idem*)

lemma *R2c-Monotonic [closure]*: *Monotonic R2c*
by (*rel-auto*)

lemma *R2c-H2-commute*: $R2c(H2(P)) = H2(R2c(P))$
by (*simp add: H2-split R2c-disj R2c-def R2s-def usubst, rel-auto*)

lemma *R2c-seq*: $R2c(R2(P) ;; R2(Q)) = (R2(P) ;; R2(Q))$
by (*metis (no-types, lifting) R1-R2c-commute R1-R2c-is-R2 R2-seqr-distribute R2c-idem*)

lemma *R2-R2c-def*: $R2(P) = R1(R2c(P))$
by (*rel-auto*)

lemma *R2-comp-def*: $R2 = R1 \circ R2c$
by (*auto simp add: R2-R2c-def*)

lemma *R2c-R1-seq*: $R2c(R1(R2c(P)) ;; R1(R2c(Q))) = (R1(R2c(P)) ;; R1(R2c(Q)))$
using *R2c-seq[of P Q]* **by** (*simp add: R2-R2c-def*)

lemma *R1-R2c-seqr-distribute*:
fixes $P :: ('t::trace, 'α, 'β) \text{rel-rp}$ **and** $Q :: ('t, 'β, 'γ) \text{rel-rp}$
assumes $P \text{ is } R1 \ P \text{ is } R2c \ Q \text{ is } R1 \ Q \text{ is } R2c$
shows $R1(R2c(P ;; Q)) = P ;; Q$
by (*metis Healthy-if R1-seqr R2c-R1-seq assms*)

lemma *R2-R1-true*:
 $R2(R1(true)) = R1(true)$
by (*simp add: R2-R1-form R2s-true*)

lemma *R1-true-R2 [closure]*: $R1(true) \text{ is } R2$
by (*rel-auto*)

lemma *R1-R2s-R1-true-lemma*:
 $R1(R2s(R1(\neg R2s P) ;; R1 true)) = R1(R2s((\neg P) ;; R1 true))$

by (rel-auto)

lemma *R2c-healthy-R2s*: $P \text{ is } R2c \implies R1(R2s(P)) = R1(P)$
 by (simp add: Healthy-def R1-R2s-R2c)

4.3 R3: No activity while predecessor is waiting

definition $R3 :: ('t::trace, 'a) \text{ hrel-rp} \Rightarrow ('t, 'a) \text{ hrel-rp}$ **where**
 $[upred-defs]: R3(P) = (II \triangleleft \$wait \triangleright P)$

lemma *R3-idem*: $R3(R3(P)) = R3(P)$
 by (rel-auto)

lemma *R3-Idempotent [closure]*: *Idempotent R3*
 by (simp add: Idempotent-def R3-idem)

lemma *R3-mono*: $P \sqsubseteq Q \implies R3(P) \sqsubseteq R3(Q)$
 by (rel-auto)

lemma *R3-Monotonic*: *Monotonic R3*
 by (simp add: mono-def R3-mono)

lemma *R3-Continuous*: *Continuous R3*
 by (rel-auto)

lemma *R3-conj*: $R3(P \wedge Q) = (R3(P) \wedge R3(Q))$
 by (rel-auto)

lemma *R3-disj*: $R3(P \vee Q) = (R3(P) \vee R3(Q))$
 by (rel-auto)

lemma *R3-USUP*:
 assumes $A \neq \{\}$
 shows $R3(\bigsqcap i \in A \cdot P(i)) = (\bigsqcap i \in A \cdot R3(P(i)))$
 using assms by (rel-auto)

lemma *R3-UIINF*:
 assumes $A \neq \{\}$
 shows $R3(\bigsqcup i \in A \cdot P(i)) = (\bigsqcup i \in A \cdot R3(P(i)))$
 using assms by (rel-auto)

lemma *R3-condr*: $R3(P \triangleleft b \triangleright Q) = (R3(P) \triangleleft b \triangleright R3(Q))$
 by (rel-auto)

lemma *R3-skipr*: $R3(II) = II$
 by (rel-auto)

lemma *R3-form*: $R3(P) = ((\$wait \wedge II) \vee (\neg \$wait \wedge P))$
 by (rel-auto)

lemma *wait-R3*:
 $(\$wait \wedge R3(P)) = (II \wedge \$wait')$
 by (rel-auto)

lemma *nwait-R3*:
 $(\neg \$wait \wedge R3(P)) = (\neg \$wait \wedge P)$

by (rel-auto)

lemma *R3-semir-form*:

$(R3(P) ;; R3(Q)) = R3(P ;; R3(Q))$

by (rel-auto)

lemma *R3-semir-closure*:

assumes *P is R3 Q is R3*

shows $(P ;; Q)$ is *R3*

using *assms*

by (metis *Healthy-def' R3-semir-form*)

lemma *R1-R3-commute*: $R1(R3(P)) = R3(R1(P))$

by (rel-auto)

lemma *R2-R3-commute*: $R2(R3(P)) = R3(R2(P))$

apply (rel-auto)

using *minus-zero-eq* apply blast+

done

4.4 R4: The trace strictly increases

definition $R4 :: ('t::trace, 'α, 'β) \text{rel-rp} \Rightarrow ('t, 'α, 'β) \text{rel-rp}$ where
[*upred-defs*]: $R4(P) = (P \wedge \$tr <_u \$tr')$

lemma *R4-implies-R1* [*closure*]: $P \text{ is } R4 \implies P \text{ is } R1$

using *less-iff* by rel-blast

lemma *R4-iff-refine*:

$P \text{ is } R4 \iff (\$tr <_u \$tr') \sqsubseteq P$

by (rel-blast)

lemma *R4-idem*: $R4(R4 P) = R4 P$

by (rel-auto)

lemma *R4-false*: $R4(\text{false}) = \text{false}$

by (rel-auto)

lemma *R4-conj*: $R4(P \wedge Q) = (R4(P) \wedge R4(Q))$

by (rel-auto)

lemma *R4-disj*: $R4(P \vee Q) = (R4(P) \vee R4(Q))$

by (rel-auto)

lemma *R4-is-R4* [*closure*]: $R4(P)$ is *R4*

by (rel-auto)

lemma *false-R4* [*closure*]: *false* is *R4*

by (rel-auto)

lemma *UINF-R4-closed* [*closure*]:

$\llbracket \bigwedge i. P \text{ is } R4 \rrbracket \implies (\bigcap i. P \text{ is } R4)$

by (rel-blast)

lemma *conj-R4-closed* [*closure*]:

$\llbracket P \text{ is } R4; Q \text{ is } R4 \rrbracket \implies (P \wedge Q) \text{ is } R4$

by (*simp add: Healthy-def R4-conj*)

lemma *disj-R4-closed* [*closure*]:
 $\llbracket P \text{ is } R4; Q \text{ is } R4 \rrbracket \implies (P \vee Q) \text{ is } R4$
by (*simp add: Healthy-def R4-disj*)

lemma *seq-R4-closed-1* [*closure*]:
 $\llbracket P \text{ is } R4; Q \text{ is } R1 \rrbracket \implies (P ;; Q) \text{ is } R4$
using *less-le-trans* **by** *rel-blast*

lemma *seq-R4-closed-2* [*closure*]:
 $\llbracket P \text{ is } R1; Q \text{ is } R4 \rrbracket \implies (P ;; Q) \text{ is } R4$
using *le-less-trans* **by** *rel-blast*

4.5 R5: The trace does not increase

definition *R5* :: (*t::trace*, *'α*, *'β*) *rel-rp* \Rightarrow (*t*, *'α*, *'β*) *rel-rp* **where**
[upred-defs]: $R5(P) = (P \wedge \$tr =_u \$tr')$

lemma *R5-implies-R1* [*closure*]: $P \text{ is } R5 \implies P \text{ is } R1$
using *eq-iff* **by** *rel-blast*

lemma *R5-iff-refine*:
 $P \text{ is } R5 \iff (\$tr =_u \$tr') \sqsubseteq P$
by (*rel-blast*)

lemma *R5-conj*: $R5(P \wedge Q) = (R5(P) \wedge R5(Q))$
by (*rel-auto*)

lemma *R5-disj*: $R5(P \vee Q) = (R5(P) \vee R5(Q))$
by (*rel-auto*)

lemma *R4-R5*: $R4 (R5 P) = \text{false}$
by (*rel-auto*)

lemma *R5-R4*: $R5 (R4 P) = \text{false}$
by (*rel-auto*)

lemma *UINF-R5-closed* [*closure*]:
 $\llbracket \bigwedge i. P \ i \text{ is } R5 \rrbracket \implies (\bigcap i \cdot P \ i) \text{ is } R5$
by (*rel-blast*)

lemma *conj-R5-closed* [*closure*]:
 $\llbracket P \text{ is } R5; Q \text{ is } R5 \rrbracket \implies (P \wedge Q) \text{ is } R5$
by (*simp add: Healthy-def R5-conj*)

lemma *disj-R5-closed* [*closure*]:
 $\llbracket P \text{ is } R5; Q \text{ is } R5 \rrbracket \implies (P \vee Q) \text{ is } R5$
by (*simp add: Healthy-def R5-disj*)

lemma *seq-R5-closed* [*closure*]:
 $\llbracket P \text{ is } R5; Q \text{ is } R5 \rrbracket \implies (P ;; Q) \text{ is } R5$
by (*rel-auto, metis*)

4.6 RP laws

definition *RP-def* [*upred-defs*]: $RP(P) = R1(R2c(R3(P)))$

lemma *RP-comp-def*: $RP = R1 \circ R2c \circ R3$
by (*auto simp add: RP-def*)

lemma *RP-alt-def*: $RP(P) = R1(R2(R3(P)))$
by (*metis R1-R2c-is-R2 R1-idem RP-def*)

lemma *RP-intro*: $\llbracket P \text{ is } R1; P \text{ is } R2; P \text{ is } R3 \rrbracket \implies P \text{ is } RP$
by (*simp add: Healthy-def' RP-alt-def*)

lemma *RP-idem*: $RP(RP(P)) = RP(P)$
by (*simp add: R1-R2c-is-R2 R2-R3-commute R2-idem R3-idem RP-def*)

lemma *RP-Idempotent* [*closure*]: *Idempotent RP*
by (*simp add: Idempotent-def RP-idem*)

lemma *RP-mono*: $P \sqsubseteq Q \implies RP(P) \sqsubseteq RP(Q)$
by (*simp add: R1-R2c-is-R2 R2-mono R3-mono RP-def*)

lemma *RP-Monotonic*: *Monotonic RP*
by (*simp add: mono-def RP-mono*)

lemma *RP-Continuous*: *Continuous RP*
by (*simp add: Continuous-comp R1-Continuous R2c-Continuous R3-Continuous RP-comp-def*)

lemma *RP-skip*:
 $RP(II) = II$
by (*simp add: R1-skip R2c-skip-r R3-skipr RP-def*)

lemma *RP-skip-closure* [*closure*]:
 $II \text{ is } RP$
by (*simp add: Healthy-def' RP-skip*)

lemma *RP-seq-closure* [*closure*]:
assumes $P \text{ is } RP \ Q \text{ is } RP$
shows $(P ;; Q) \text{ is } RP$
proof (*rule RP-intro*)
show $(P ;; Q) \text{ is } R1$
by (*metis Healthy-def R1-seqr RP-def assms*)
thus $(P ;; Q) \text{ is } R2$
by (*metis Healthy-def' R2-R2c-def R2c-R1-seq RP-def assms*)
show $(P ;; Q) \text{ is } R3$
by (*metis (no-types, lifting) Healthy-def' R1-R2c-is-R2 R2-R3-commute R3-idem R3-semir-form RP-def assms*)
qed

4.7 UTP theories

interpretation *rea-theory*: *utp-theory-continuous RP*
rewrites $P \in \text{carrier } \text{rea-theory.thy-order} \longleftrightarrow P \text{ is } RP$
and $\text{le } \text{des-theory.thy-order} = (\sqsubseteq)$
and $\text{eq } \text{des-theory.thy-order} = (=)$
proof —

```

show utp-theory-continuous RP
  by (unfold-locales, simp-all add: RP-idem RP-Continuous)
qed (simp-all)

```

```

notation rea-theory.utp-top ( $\top_r$ )
notation rea-theory.utp-bottom ( $\perp_r$ )

```

```

interpretation rea-theory-rel: utp-theory-unital RP skip-r
  by (unfold-locales, simp-all add: closure)

```

```

lemma rea-top:  $\top_r = (\$wait \wedge II)$ 
proof –
  have  $\top_r = RP(false)$ 
    by (simp add: rea-theory.healthy-top)
  also have  $\dots = (\$wait \wedge II)$ 
    by (rel-auto, metis minus-zero-eq)
  finally show ?thesis .
qed

```

```

lemma rea-top-left-zero:
  assumes P is RP
  shows  $(\top_r ;; P) = \top_r$ 
proof –
  have  $(\top_r ;; P) = ((\$wait \wedge II) ;; R3(P))$ 
    by (metis (no-types, lifting) Healthy-def R1-R2c-is-R2 R2-R3-commute R3-idem RP-def assms
rea-top)
  also have  $\dots = (\$wait \wedge R3(P))$ 
    by (rel-auto)
  also have  $\dots = (\$wait \wedge II)$ 
    by (metis R3-skipr wait-R3)
  also have  $\dots = \top_r$ 
    by (simp add: rea-top)
  finally show ?thesis .
qed

```

```

lemma rea-bottom:  $\perp_r = R1(\$wait \Rightarrow II)$ 
proof –
  have  $\perp_r = RP(true)$ 
    by (simp add: rea-theory.healthy-bottom)
  also have  $\dots = R1(\$wait \Rightarrow II)$ 
    by (rel-auto, metis minus-zero-eq)
  finally show ?thesis .
qed

```

end

5 Reactive Parallel-by-Merge

```

theory utp-rea-parallel
  imports utp-rea-healths
begin

```

We show closure of parallel by merge under the reactive healthiness conditions by means of suitable restrictions on the merge predicate [4]. We first define healthiness conditions for *R1* and *R2* merge predicates.

definition $R1m :: ('t :: trace, 'α) rp merge \Rightarrow ('t, 'α) rp merge$
where $[upred-defs]: R1m(M) = (M \wedge \$<:tr \leq_u \$tr')$

definition $R1m' :: ('t :: trace, 'α) rp merge \Rightarrow ('t, 'α) rp merge$
where $[upred-defs]: R1m'(M) = (M \wedge \$<:tr \leq_u \$tr' \wedge \$<:tr \leq_u \$0:tr \wedge \$<:tr \leq_u \$1:tr)$

A merge predicate can access the history through tr , as usual, but also through $0.tr$ and $1.tr$.
Thus we have to remove the latter two histories as well to satisfy R2 for the overall construction.

definition $R2m :: ('t :: trace, 'α) rp merge \Rightarrow ('t, 'α) rp merge$
where $[upred-defs]: R2m(M) = R1m(M \llbracket 0, (\$tr' - \$<:tr), (\$0:tr - \$<:tr), (\$1:tr - \$<:tr) / \$<:tr, \$tr', \$0:tr, \$1:tr \rrbracket)$

definition $R2m' :: ('t :: trace, 'α) rp merge \Rightarrow ('t, 'α) rp merge$
where $[upred-defs]: R2m'(M) = R1m'(M \llbracket 0, (\$tr' - \$<:tr), (\$0:tr - \$<:tr), (\$1:tr - \$<:tr) / \$<:tr, \$tr', \$0:tr, \$1:tr \rrbracket)$

definition $R2cm :: ('t :: trace, 'α) rp merge \Rightarrow ('t, 'α) rp merge$
where $[upred-defs]: R2cm(M) = M \llbracket 0, (\$tr' - \$<:tr), (\$0:tr - \$<:tr), (\$1:tr - \$<:tr) / \$<:tr, \$tr', \$0:tr, \$1:tr \rrbracket$
 $\triangleleft \$<:tr \leq_u \$tr' \triangleright M$

lemma $R2m'$ -form:
 $R2m'(M) =$
 $(\exists (tt_p, tt_0, tt_1) \cdot M \llbracket 0, \llbracket tt_p \rrbracket, \llbracket tt_0 \rrbracket, \llbracket tt_1 \rrbracket / \$<:tr, \$tr', \$0:tr, \$1:tr \rrbracket$
 $\wedge \$tr' =_u \$<:tr + \llbracket tt_p \rrbracket$
 $\wedge \$0:tr =_u \$<:tr + \llbracket tt_0 \rrbracket$
 $\wedge \$1:tr =_u \$<:tr + \llbracket tt_1 \rrbracket)$
by (rel-auto, metis diff-add-cancel-left')

lemma $R1m$ -idem: $R1m(R1m(P)) = R1m(P)$
by (rel-auto)

lemma $R1m$ -seq-lemma: $R1m(R1m(M) ;; R1(P)) = R1m(M) ;; R1(P)$
by (rel-auto)

lemma $R1m$ -seq [closure]:
assumes M is $R1m$ P is $R1$
shows $M ;; P$ is $R1m$

proof –
from *assms* **have** $R1m(M ;; P) = R1m(R1m(M) ;; R1(P))$
by (simp add: Healthy-if)
also have $\dots = R1m(M) ;; R1(P)$
by (simp add: R1m-seq-lemma)
also have $\dots = M ;; P$
by (simp add: Healthy-if *assms*)
finally show ?thesis
by (simp add: Healthy-def)
qed

lemma $R2m$ -idem: $R2m(R2m(P)) = R2m(P)$
by (rel-auto)

lemma $R2m$ -seq-lemma: $R2m'(R2m'(M) ;; R2(P)) = R2m'(M) ;; R2(P)$
apply (simp add: R2m'-form R2-form)
apply (rel-auto)
apply (metis (no-types, lifting) add.assoc)+
done

lemma *R2m'-seq [closure]*:

assumes *M is R2m' P is R2*

shows *M ;; P is R2m'*

by (*metis Healthy-def' R2m-seq-lemma assms(1) assms(2)*)

lemma *R1-par-by-merge [closure]*:

M is R1m \implies (P \parallel_M Q) is R1

by (*rel-blast*)

lemma *R2-R2m'-pbm*: $R2(P \parallel_M Q) = (R2(P) \parallel_{R2m'(M)} R2(Q))$

proof –

have $(R2(P) \parallel_{R2m'(M)} R2(Q)) = ((R2(P) \parallel_s R2(Q)) ;;$

$$\begin{aligned} & (\exists (tt_p, tt_0, tt_1) \cdot M \llbracket 0, \langle tt_p \rangle, \langle tt_0 \rangle, \langle tt_1 \rangle \rrbracket / \$\langle :tr, \$tr', \$0:tr, \$1:tr \rrbracket \\ & \quad \wedge \$tr' =_u \$\langle :tr + \langle tt_p \rangle \\ & \quad \wedge \$0:tr =_u \$\langle :tr + \langle tt_0 \rangle \\ & \quad \wedge \$1:tr =_u \$\langle :tr + \langle tt_1 \rangle \rangle)) \end{aligned}$$

by (*simp add: par-by-merge-def R2m'-form*)

also have $\dots = (\exists (tt_p, tt_0, tt_1) \cdot ((R2(P) \parallel_s R2(Q)) ;; (M \llbracket 0, \langle tt_p \rangle, \langle tt_0 \rangle, \langle tt_1 \rangle \rrbracket / \$\langle :tr, \$tr', \$0:tr, \$1:tr \rrbracket$

$$\begin{aligned} & \quad \wedge \$tr' =_u \$\langle :tr + \langle tt_p \rangle \\ & \quad \wedge \$0:tr =_u \$\langle :tr + \langle tt_0 \rangle \\ & \quad \wedge \$1:tr =_u \$\langle :tr + \langle tt_1 \rangle \rangle)) \end{aligned}$$

by (*rel-blast*)

also have $\dots = (\exists (tt_p, tt_0, tt_1) \cdot (((R2(P) \parallel_s R2(Q)) \wedge \$0:tr' =_u \$\langle :tr' + \langle tt_0 \rangle \wedge \$1:tr' =_u$

$\$ \langle :tr' + \langle tt_1 \rangle \rangle ;;$

$$(M \llbracket 0, \langle tt_p \rangle, \langle tt_0 \rangle, \langle tt_1 \rangle \rrbracket / \$\langle :tr, \$tr', \$0:tr, \$1:tr \rrbracket \wedge \$tr' =_u \$\langle :tr +$$

$\langle \langle tt_p \rangle \rangle))$

by (*rel-blast*)

also have $\dots = (\exists (tt_p, tt_0, tt_1) \cdot (((R2(P) \parallel_s R2(Q)) \wedge \$0:tr' =_u \$\langle :tr' + \langle tt_0 \rangle \wedge \$1:tr' =_u$

$\$ \langle :tr' + \langle tt_1 \rangle \rangle ;;$

$$(M \llbracket 0, \langle tt_p \rangle, \langle tt_0 \rangle, \langle tt_1 \rangle \rrbracket / \$\langle :tr, \$tr', \$0:tr, \$1:tr \rrbracket)) \wedge \$tr' =_u \$tr +$$

$\langle \langle tt_p \rangle \rangle$

by (*rel-blast*)

also have $\dots = (\exists (tt_p, tt_0, tt_1) \cdot (((R2(P) \wedge \$tr' =_u \$tr + \langle \langle tt_0 \rangle \rangle \parallel_s (R2(Q) \wedge \$tr' =_u \$tr +$

$\langle \langle tt_1 \rangle \rangle)) ;;$

$$(M \llbracket 0, \langle tt_p \rangle, \langle tt_0 \rangle, \langle tt_1 \rangle \rrbracket / \$\langle :tr, \$tr', \$0:tr, \$1:tr \rrbracket)) \wedge \$tr' =_u \$tr +$$

$\langle \langle tt_p \rangle \rangle$

by (*rel-auto, blast, metis le-add trace-class.add-diff-cancel-left*)

also have $\dots = (\exists (tt_p, tt_0, tt_1) \cdot (((\exists tt_0' \cdot P \llbracket 0, \langle tt_0' \rangle \rrbracket / \$tr, \$tr' \rrbracket \wedge \$tr' =_u \$tr + \langle \langle tt_0' \rangle \rangle) \wedge$

$\$tr' =_u \$tr + \langle \langle tt_0 \rangle \rangle$

$$\parallel_s ((\exists tt_1' \cdot Q \llbracket 0, \langle tt_1' \rangle \rrbracket / \$tr, \$tr' \rrbracket \wedge \$tr' =_u \$tr + \langle \langle tt_1' \rangle \rangle) \wedge \$tr' =_u$$

$\$tr + \langle \langle tt_1 \rangle \rangle)) ;;$

$$(M \llbracket 0, \langle tt_p \rangle, \langle tt_0 \rangle, \langle tt_1 \rangle \rrbracket / \$\langle :tr, \$tr', \$0:tr, \$1:tr \rrbracket)) \wedge \$tr' =_u \$tr +$$

$\langle \langle tt_p \rangle \rangle$

by (*simp add: R2-form usubst*)

also have $\dots = (\exists (tt_p, tt_0, tt_1) \cdot (((P \llbracket 0, \langle tt_0 \rangle \rrbracket / \$tr, \$tr' \rrbracket \wedge \$tr' =_u \$tr + \langle \langle tt_0 \rangle \rangle$

$$\parallel_s (Q \llbracket 0, \langle tt_1 \rangle \rrbracket / \$tr, \$tr' \rrbracket \wedge \$tr' =_u \$tr + \langle \langle tt_1 \rangle \rangle)) ;;$$

$$(M \llbracket 0, \langle tt_p \rangle, \langle tt_0 \rangle, \langle tt_1 \rangle \rrbracket / \$\langle :tr, \$tr', \$0:tr, \$1:tr \rrbracket)) \wedge \$tr' =_u \$tr +$$

$\langle \langle tt_p \rangle \rangle$

by (*rel-auto, metis left-cancel-monoid-class.add-left-imp-eq, blast*)

also have $\dots = R2(P \parallel_M Q)$

by (*rel-auto, blast, metis diff-add-cancel-left'*)

finally show *?thesis ..*

qed

lemma *R2m-R2m'-pbm*: $(R2(P) \parallel_{R2m(M)} R2(Q)) = (R2(P) \parallel_{R2m'(M)} R2(Q))$

by (rel-blast)

lemma *R2-par-by-merge* [closure]:

assumes *P* is *R2* *Q* is *R2* *M* is *R2m*

shows $(P \parallel_M Q)$ is *R2*

by (metis *Healthy-def'* *R2-R2m'-pbm* *R2m-R2m'-pbm* *assms*(1) *assms*(2) *assms*(3))

lemma *R2-par-by-merge'* [closure]:

assumes *P* is *R2* *Q* is *R2* *M* is *R2m'*

shows $(P \parallel_M Q)$ is *R2*

by (metis *Healthy-def'* *R2-R2m'-pbm* *assms*(1) *assms*(2) *assms*(3))

lemma *R1m-skip-merge*: $R1m(skip_m) = skip_m$

by (rel-auto)

lemma *R1m-disj*: $R1m(P \vee Q) = (R1m(P) \vee R1m(Q))$

by (rel-auto)

lemma *R1m-conj*: $R1m(P \wedge Q) = (R1m(P) \wedge R1m(Q))$

by (rel-auto)

lemma *R2m-skip-merge*: $R2m(skip_m) = skip_m$

apply (rel-auto) using *minus-zero-eq* by blast

lemma *R2m-disj*: $R2m(P \vee Q) = (R2m(P) \vee R2m(Q))$

by (rel-auto)

lemma *R2m-conj*: $R2m(P \wedge Q) = (R2m(P) \wedge R2m(Q))$

by (rel-auto)

definition *R3m* :: $(t :: trace, 'a) \text{ rp merge} \Rightarrow (t, 'a) \text{ rp merge}$ **where**

[upred-defs]: $R3m(M) = skip_m \triangleleft \$<:wait \triangleright M$

lemma *R3-par-by-merge*:

assumes

P is *R3* *Q* is *R3* *M* is *R3m*

shows $(P \parallel_M Q)$ is *R3*

proof –

have $(P \parallel_M Q) = ((P \parallel_M Q) \llbracket true/\$wait \rrbracket \triangleleft \$wait \triangleright (P \parallel_M Q))$

by (metis *cond-L6* *cond-var-split* *in-var-uvar* *wait-vwb-lens*)

also have $\dots = (((R3 P) \llbracket true/\$wait \rrbracket \parallel_{(R3m M)} \llbracket true/\$<:wait \rrbracket (R3 Q) \llbracket true/\$wait \rrbracket) \triangleleft \$wait \triangleright (P \parallel_M Q))$

by (subst-tac, simp add: *Healthy-if* *assms*)

also have $\dots = ((II \llbracket true/\$wait \rrbracket \parallel_{skip_m} \llbracket true/\$<:wait \rrbracket II \llbracket true/\$wait \rrbracket) \triangleleft \$wait \triangleright (P \parallel_M Q))$

by (simp add: *R3-def* *R3m-def* *usubst*)

also have $\dots = ((II \parallel_{skip_m} II) \llbracket true/\$wait \rrbracket \triangleleft \$wait \triangleright (P \parallel_M Q))$

by (subst-tac)

also have $\dots = (II \triangleleft \$wait \triangleright (P \parallel_M Q))$

by (simp add: *cond-var-subst-left* *par-by-merge-skip*)

also have $\dots = R3(P \parallel_M Q)$

by (simp add: *R3-def*)

finally show *?thesis*

by (simp add: *Healthy-def*)

qed

lemma *SymMerge-R1-true* [closure]:
 $M \text{ is SymMerge} \implies M \mathrel{;;} R1(\text{true}) \text{ is SymMerge}$
by (*rel-auto*)

end

6 Reactive Relations

theory *utp-rea-rel*
imports
utp-rea-healths
UTP-KAT.utp-kleene
begin

This theory defines a reactive relational calculus for *R1-R2* predicates as an extension of the standard alphabetised predicate calculus. This enables us to formally characterise relational programs that refer to both state variables and a trace history. For more details on reactive relations, please see the associated journal paper [3].

6.1 Healthiness Conditions

definition *RR* :: $(t::\text{trace}, \alpha, \beta) \text{ rel-rp} \Rightarrow (t, \alpha, \beta) \text{ rel-rp}$ **where**
[upred-defs]: $RR(P) = (\exists \{ \$ok, \$ok', \$wait, \$wait' \} \cdot R2(P))$

lemma *RR-idem*: $RR(RR(P)) = RR(P)$
by (*rel-auto*)

lemma *RR-Idempotent* [closure]: *Idempotent RR*
by (*simp add: Idempotent-def RR-idem*)

lemma *RR-Continuous* [closure]: *Continuous RR*
by (*rel-blast*)

lemma *R1-RR*: $R1(RR(P)) = RR(P)$
by (*rel-auto*)

lemma *R2c-RR*: $R2c(RR(P)) = RR(P)$
by (*rel-auto*)

lemma *RR-implies-R1* [closure]: $P \text{ is } RR \implies P \text{ is } R1$
by (*metis Healthy-def R1-RR*)

lemma *RR-implies-R2c*: $P \text{ is } RR \implies P \text{ is } R2c$
by (*metis Healthy-def R2c-RR*)

lemma *RR-implies-R2* [closure]: $P \text{ is } RR \implies P \text{ is } R2$
by (*metis Healthy-def R1-RR R2-R2c-def R2c-RR*)

lemma *RR-intro*:
assumes $\$ok \# P \ \$ok' \# P \ \$wait \# P \ \$wait' \# P \ P \text{ is } R1 \ P \text{ is } R2c$
shows $P \text{ is } RR$
by (*simp add: RR-def Healthy-def ex-plus R2-R2c-def, simp add: Healthy-if assms ex-unrest*)

lemma *RR-R2-intro*:
assumes $\$ok \# P \ \$ok' \# P \ \$wait \# P \ \$wait' \# P \ P \text{ is } R2$

shows P is RR
by (*simp add: RR-def Healthy-def ex-plus, simp add: Healthy-if assms ex-unrest*)

lemma *RR-unrests* [*unrest*]:

assumes P is RR
shows $\$ok \# P \$ok' \# P \$wait \# P \$wait' \# P$

proof –

have $\$ok \# RR(P) \$ok' \# RR(P) \$wait \# RR(P) \$wait' \# RR(P)$

by (*simp-all add: RR-def ex-plus unrest*)

thus $\$ok \# P \$ok' \# P \$wait \# P \$wait' \# P$

by (*simp-all add: assms Healthy-if*)

qed

lemma *RR-refine-intro*:

assumes P is RR Q is $RR \wedge t. P \llbracket 0, \langle t \rangle / \$tr, \$tr' \rrbracket \sqsubseteq Q \llbracket 0, \langle t \rangle / \$tr, \$tr' \rrbracket$

shows $P \sqsubseteq Q$

proof –

have $\bigwedge t. (RR\ P) \llbracket 0, \langle t \rangle / \$tr, \$tr' \rrbracket \sqsubseteq (RR\ Q) \llbracket 0, \langle t \rangle / \$tr, \$tr' \rrbracket$

by (*simp add: Healthy-if assms*)

hence $RR(P) \sqsubseteq RR(Q)$

by (*rel-auto*)

thus *?thesis*

by (*simp add: Healthy-if assms*)

qed

lemma *R4-RR-closed* [*closure*]:

assumes P is RR

shows $R4(P)$ is RR

proof –

have $R4(RR(P))$ is RR

by (*rel-blast*)

thus *?thesis*

by (*simp add: Healthy-if assms*)

qed

lemma *R5-RR-closed* [*closure*]:

assumes P is RR

shows $R5(P)$ is RR

proof –

have $R5(RR(P))$ is RR

using *minus-zero-eq* **by** *rel-auto*

thus *?thesis*

by (*simp add: Healthy-if assms*)

qed

6.2 Reactive relational operators

named-theorems *rpred*

abbreviation *rea-true* :: $(t::trace, 'α, 'β)$ *rel-rp* (*true_r*) **where**

true_r $\equiv R1(true)$

definition *rea-not* :: $(t::trace, 'α, 'β)$ *rel-rp* $\Rightarrow (t, 'α, 'β)$ *rel-rp* $(\neg_r - [40] 40)$

where [*upred-defs*]: $(\neg_r P) = R1(\neg P)$

no-utp-lift *rea-not*

definition *rea-diff* :: (*t*::*trace*,*'α*,*'β*) *rel-rp* \Rightarrow (*t*,*'α*,*'β*) *rel-rp* \Rightarrow (*t*,*'α*,*'β*) *rel-rp* (**infixl** \neg_r 65)
where [*upred-defs*]: *rea-diff* *P Q* = (*P* \wedge \neg_r *Q*)

definition *rea-impl* ::
(*t*::*trace*,*'α*,*'β*) *rel-rp* \Rightarrow (*t*,*'α*,*'β*) *rel-rp* \Rightarrow (*t*,*'α*,*'β*) *rel-rp* (**infixr** \Rightarrow_r 25)
where [*upred-defs*]: (*P* \Rightarrow_r *Q*) = (\neg_r *P* \vee *Q*)

no-utp-lift *rea-impl*

definition *rea-lift* :: (*t*::*trace*,*'α*,*'β*) *rel-rp* \Rightarrow (*t*,*'α*,*'β*) *rel-rp* ($[-]_r$)
where [*upred-defs*]: $[P]_r = R1(P)$

definition *rea-skip* :: (*t*::*trace*,*'α*) *hrel-rp* (*II*_{*r*})
where [*upred-defs*]: *II*_{*r*} = ($\$tr' =_u \$tr \wedge \$\Sigma_R' =_u \Σ_R)

definition *rea-assert* :: (*t*::*trace*,*'α*) *hrel-rp* \Rightarrow (*t*,*'α*) *hrel-rp* ($\{-\}_r$)
where [*upred-defs*]: $\{b\}_r = (II_r \vee \neg_r b)$

Convert from one trace algebra to another using renamer functions, which are a kind of monoid homomorphism.

locale *renamer* =
fixes *f* :: *'a*::*trace* \Rightarrow *'b*::*trace*
assumes
injective: *inj f* **and**
add: *f* (*x* + *y*) = *f x* + *f y*
begin
lemma *zero*: *f 0* = 0
by (*metis add add.right-neutral add-monoid-diff-cancel-left*)

lemma *monotonic*: *mono f*
by (*metis add monoI trace-class.le-iff-add*)

lemma *minus*: *x* \leq *y* \implies *f* (*y* - *x*) = *f*(*y*) - *f*(*x*)
by (*metis add diff-add-cancel-left' trace-class.add-diff-cancel-left*)
end

declare *renamer.add* [*simp*]
declare *renamer.zero* [*simp*]
declare *renamer.minus* [*simp*]

lemma *renamer-id*: *renamer id*
by (*unfold-locales, simp-all*)

lemma *renamer-comp*: $\llbracket \text{renamer } f; \text{renamer } g \rrbracket \implies \text{renamer } (f \circ g)$
by (*unfold-locales, simp-all add: inj-compose renamer.injective*)

lemma *renamer-map*: *inj f* \implies *renamer* (*map f*)
by (*unfold-locales, simp-all add: plus-list-def*)

definition *rea-rename* :: (*t*₁::*trace*,*'α*) *hrel-rp* \Rightarrow (*t*₁ \Rightarrow *t*₂) \Rightarrow (*t*₂::*trace*,*'α*) *hrel-rp* ($([-]_r [999, 0] 999)$ **where**
[*upred-defs*]: *rea-rename* *P f* = $R2((\$tr' =_u 0 \wedge \$\Sigma_R' =_u \$\Sigma_R) ;; P ;; (\$tr' =_u \text{uop } f \$tr \wedge \$\Sigma_R' =_u \$\Sigma_R))$

Trace contribution substitution: make a substitution for the trace contribution lens *tt*, and apply

$R1$ to make the resulting predicate healthy again.

definition $rea\text{-}subst :: ('t::trace, 'a) hrel\text{-}rp \Rightarrow ('t, ('t, 'a) rp) hexpr \Rightarrow ('t, 'a) hrel\text{-}rp \ (-\llbracket - \rrbracket_r \ [999,0] \ 999)$
where $[upred\text{-}defs]: P\llbracket v \rrbracket_r = R1(P\llbracket v/\&tt \rrbracket)$

6.3 Unrestriction and substitution laws

lemma $rea\text{-}true\text{-}unrest \ [unrest]:$

$\llbracket x \bowtie (\$tr)_v; x \bowtie (\$tr')_v \rrbracket \Longrightarrow x \# true_r$
by ($simp \ add: R1\text{-}def \ unrest \ lens\text{-}indep\text{-}sym$)

lemma $rea\text{-}not\text{-}unrest \ [unrest]:$

$\llbracket x \bowtie (\$tr)_v; x \bowtie (\$tr')_v; x \# P \rrbracket \Longrightarrow x \# \neg_r P$
by ($simp \ add: rea\text{-}not\text{-}def \ R1\text{-}def \ unrest \ lens\text{-}indep\text{-}sym$)

lemma $rea\text{-}impl\text{-}unrest \ [unrest]:$

$\llbracket x \bowtie (\$tr)_v; x \bowtie (\$tr')_v; x \# P; x \# Q \rrbracket \Longrightarrow x \# (P \Rightarrow_r Q)$
by ($simp \ add: rea\text{-}impl\text{-}def \ unrest$)

lemma $rea\text{-}true\text{-}usubst \ [usubst]:$

$\llbracket \$tr \#_s \sigma; \$tr' \#_s \sigma \rrbracket \Longrightarrow \sigma \dagger true_r = true_r$
by ($simp \ add: R1\text{-}def \ usubst$)

lemma $rea\text{-}not\text{-}usubst \ [usubst]:$

$\llbracket \$tr \#_s \sigma; \$tr' \#_s \sigma \rrbracket \Longrightarrow \sigma \dagger (\neg_r P) = (\neg_r \sigma \dagger P)$
by ($simp \ add: rea\text{-}not\text{-}def \ R1\text{-}def \ usubst$)

lemma $rea\text{-}impl\text{-}usubst \ [usubst]:$

$\llbracket \$tr \#_s \sigma; \$tr' \#_s \sigma \rrbracket \Longrightarrow \sigma \dagger (P \Rightarrow_r Q) = (\sigma \dagger P \Rightarrow_r \sigma \dagger Q)$
by ($simp \ add: rea\text{-}impl\text{-}def \ usubst \ R1\text{-}def$)

lemma $rea\text{-}true\text{-}usubst\text{-}tt \ [usubst]:$

$R1(true)\llbracket e/\&tt \rrbracket = true$
by ($rel\text{-}simp$)

lemma $unrests\text{-}rea\text{-}rename \ [unrest]:$

$\$ok \# P \Longrightarrow \$ok \# P\langle f \rangle_r$
 $\$ok' \# P \Longrightarrow \$ok' \# P\langle f \rangle_r$
 $\$wait \# P \Longrightarrow \$wait \# P\langle f \rangle_r$
 $\$wait' \# P \Longrightarrow \$wait' \# P\langle f \rangle_r$
by ($simp\text{-}all \ add: rea\text{-}rename\text{-}def \ R2\text{-}def \ unrest$)

lemma $unrest\text{-}rea\text{-}subst \ [unrest]:$

$\llbracket mwb\text{-}lens \ x; x \bowtie (\$tr)_v; x \bowtie (\$tr')_v; x \# v; x \# P \rrbracket \Longrightarrow x \# P\llbracket v \rrbracket_r$
by ($simp \ add: rea\text{-}subst\text{-}def \ R1\text{-}def \ unrest \ lens\text{-}indep\text{-}sym$)

lemma $rea\text{-}substs \ [usubst]:$

$true_r\llbracket v \rrbracket_r = true_r \ true\llbracket v \rrbracket_r = true_r \ false\llbracket v \rrbracket_r = false$
 $(\neg_r P)\llbracket v \rrbracket_r = \neg_r P\llbracket v \rrbracket_r \ (P \wedge Q)\llbracket v \rrbracket_r = (P\llbracket v \rrbracket_r \wedge Q\llbracket v \rrbracket_r) \ (P \vee Q)\llbracket v \rrbracket_r = (P\llbracket v \rrbracket_r \vee Q\llbracket v \rrbracket_r)$
 $(P \Rightarrow_r Q)\llbracket v \rrbracket_r = (P\llbracket v \rrbracket_r \Rightarrow_r Q\llbracket v \rrbracket_r)$
by $rel\text{-}auto+$

lemma $rea\text{-}substs\text{-}lattice \ [usubst]:$

$(\bigcap i \cdot P(i))\llbracket v \rrbracket_r = (\bigcap i \cdot (P(i))\llbracket v \rrbracket_r)$
 $(\bigcap i \in A \cdot P(i))\llbracket v \rrbracket_r = (\bigcap i \in A \cdot (P(i))\llbracket v \rrbracket_r)$

$$(\bigsqcup i \cdot P(i))\llbracket v \rrbracket_r = (\bigsqcup i \cdot (P(i))\llbracket v \rrbracket_r) \\ \text{by } (rel-auto)+$$

lemma *rea-subst-USUP-set* [*usubst*]:

$$A \neq \{\} \implies (\bigsqcup i \in A \cdot P(i))\llbracket v \rrbracket_r = (\bigsqcup i \in A \cdot (P(i))\llbracket v \rrbracket_r) \\ \text{by } (rel-auto)+$$

6.4 Closure laws

lemma *rea-lift-R1* [*closure*]: $[P]_r$ is *R1*
by (*rel-simp*)

lemma *R1-rea-not*: $R1(\neg_r P) = (\neg_r P)$
by *rel-auto*

lemma *R1-rea-not'*: $R1(\neg_r P) = (\neg_r R1(P))$
by *rel-auto*

lemma *R2c-rea-not*: $R2c(\neg_r P) = (\neg_r R2c(P))$
by *rel-auto*

lemma *RR-rea-not*: $RR(\neg_r RR(P)) = (\neg_r RR(P))$
by (*rel-auto*)

lemma *R1-rea-impl*: $R1(P \Rightarrow_r Q) = (P \Rightarrow_r R1(Q))$
by (*rel-auto*)

lemma *R1-rea-impl'*: $R1(P \Rightarrow_r Q) = (R1(P) \Rightarrow_r R1(Q))$
by (*rel-auto*)

lemma *R2c-rea-impl*: $R2c(P \Rightarrow_r Q) = (R2c(P) \Rightarrow_r R2c(Q))$
by (*rel-auto*)

lemma *RR-rea-impl*: $RR(RR(P) \Rightarrow_r RR(Q)) = (RR(P) \Rightarrow_r RR(Q))$
by (*rel-auto*)

lemma *rea-true-R1* [*closure*]: $true_r$ is *R1*
by (*rel-auto*)

lemma *rea-true-R2c* [*closure*]: $true_r$ is *R2c*
by (*rel-auto*)

lemma *rea-true-RR* [*closure*]: $true_r$ is *RR*
by (*rel-auto*)

lemma *rea-not-R1* [*closure*]: $\neg_r P$ is *R1*
by (*rel-auto*)

lemma *rea-not-R2c* [*closure*]: P is *R2c* $\implies \neg_r P$ is *R2c*
by (*simp add: Healthy-def rea-not-def R1-R2c-commute* [*THEN sym*] *R2c-not*)

lemma *rea-not-R2-closed* [*closure*]:

$$P \text{ is } R2 \implies (\neg_r P) \text{ is } R2$$

by (*simp add: Healthy-def' R1-rea-not' R2-R2c-def R2c-rea-not*)

lemma *rea-no-RR* [*closure*]:

$\llbracket P \text{ is } RR \rrbracket \implies (\neg_r P) \text{ is } RR$
by (*metis Healthy-def' RR-rea-not*)

lemma *rea-impl-R1* [*closure*]:
 $Q \text{ is } R1 \implies (P \Rightarrow_r Q) \text{ is } R1$
by (*rel-blast*)

lemma *rea-impl-R2c* [*closure*]:
 $\llbracket P \text{ is } R2c; Q \text{ is } R2c \rrbracket \implies (P \Rightarrow_r Q) \text{ is } R2c$
by (*simp add: rea-impl-def Healthy-def rea-not-def R1-R2c-commute [THEN sym] R2c-not R2c-disj*)

lemma *rea-impl-R2* [*closure*]:
 $\llbracket P \text{ is } R2; Q \text{ is } R2 \rrbracket \implies (P \Rightarrow_r Q) \text{ is } R2$
by (*rel-blast*)

lemma *rea-impl-RR* [*closure*]:
 $\llbracket P \text{ is } RR; Q \text{ is } RR \rrbracket \implies (P \Rightarrow_r Q) \text{ is } RR$
by (*metis Healthy-def' RR-rea-impl*)

lemma *conj-RR* [*closure*]:
 $\llbracket P \text{ is } RR; Q \text{ is } RR \rrbracket \implies (P \wedge Q) \text{ is } RR$
by (*meson RR-implies-R1 RR-implies-R2c RR-intro RR-unrests(1-4) conj-R1-closed-1 conj-R2c-closed unrest-conj*)

lemma *disj-RR* [*closure*]:
 $\llbracket P \text{ is } RR; Q \text{ is } RR \rrbracket \implies (P \vee Q) \text{ is } RR$
by (*metis Healthy-def' R1-RR R1-idem R1-rea-not' RR-rea-impl RR-rea-not disj-comm double-negation rea-impl-def rea-not-def*)

lemma *USUP-mem-RR-closed* [*closure*]:
assumes $\bigwedge i. i \in A \implies P \ i \text{ is } RR$
shows $(\bigsqcup_{i \in A} P(i)) \text{ is } RR$
proof –
have 1: $(\bigsqcup_{i \in A} P(i)) \text{ is } R1$
by (*unfold Healthy-def, subst R1-UNIF, simp-all add: Healthy-if assms closure cong: USUP-cong*)
have 2: $(\bigsqcup_{i \in A} P(i)) \text{ is } R2c$
by (*unfold Healthy-def, subst R2c-UNIF, simp-all add: Healthy-if assms RR-implies-R2c closure cong: USUP-cong*)
show ?thesis
using 1 2 **by** (*rule-tac RR-intro, simp-all add: unrest assms*)
qed

lemma *USUP-ind-RR-closed* [*closure*]:
assumes $\bigwedge i. P \ i \text{ is } RR$
shows $(\bigsqcup i \cdot P(i)) \text{ is } RR$
using *USUP-mem-RR-closed* [*of UNIV P*] **by** (*simp add: assms*)

lemma *UNIF-mem-RR-closed* [*closure*]:
assumes $\bigwedge i. i \in A \implies P \ i \text{ is } RR$
shows $(\bigsqcap_{i \in A} P(i)) \text{ is } RR$
proof –
have 1: $(\bigsqcap_{i \in A} P(i)) \text{ is } R1$
by (*unfold Healthy-def, subst R1-USUP, simp add: Healthy-if RR-implies-R1 assms cong: UNIF-cong*)
have 2: $(\bigsqcap_{i \in A} P(i)) \text{ is } R2c$
by (*unfold Healthy-def, subst R2c-USUP, simp add: Healthy-if RR-implies-R2c assms cong: UNIF-cong*)

show ?thesis
 using 1 2 by (rule-tac RR-intro, simp-all add: unrest assms)
 qed

lemma UINF-ind-RR-closed [closure]:
 assumes $\bigwedge i. P\ i\ \text{is}\ RR$
 shows $(\bigcap i. P(i))\ \text{is}\ RR$
 by (simp add: assms closure)

lemma USUP-elim-RR [closure]:
 assumes $\bigwedge i. P\ i\ \text{is}\ RR\ A \neq \{\}$
 shows $(\bigsqcup i \in A. P\ i)\ \text{is}\ RR$

proof -

have 1: $(\bigsqcup i \in A. P(i))\ \text{is}\ R1$
 by (unfold Healthy-def, subst R1-UINF, simp-all add: Healthy-if assms closure)
 have 2: $(\bigsqcup i \in A. P(i))\ \text{is}\ R2c$
 by (unfold Healthy-def, subst R2c-UINF, simp-all add: Healthy-if assms RR-implies-R2c closure)
 show ?thesis
 using 1 2 by (rule-tac RR-intro, simp-all add: unrest assms)

qed

lemma seq-RR-closed [closure]:
 assumes $P\ \text{is}\ RR\ Q\ \text{is}\ RR$
 shows $P\ ;;\ Q\ \text{is}\ RR$
 unfolding Healthy-def
 by (simp add: RR-def Healthy-if assms closure RR-implies-R2 ex-unrest unrest)

lemma power-Suc-RR-closed [closure]:
 $P\ \text{is}\ RR \implies P\ ;;\ P\ ^\wedge\ i\ \text{is}\ RR$
 by (induct i, simp-all add: closure upred-semiring.power-Suc)

lemma seqr-iter-RR-closed [closure]:
 $\llbracket I \neq []; \bigwedge i. i \in \text{set}(I) \implies P(i)\ \text{is}\ RR \rrbracket \implies (;\ i : I. P(i))\ \text{is}\ RR$
 apply (induct I, simp-all)
 apply (rename-tac i I)
 apply (case-tac I)
 apply (simp-all add: seq-RR-closed)

done

lemma cond-tt-RR-closed [closure]:
 assumes $P\ \text{is}\ RR\ Q\ \text{is}\ RR$
 shows $P \triangleleft \$tr' =_u \$tr \triangleright Q\ \text{is}\ RR$
 apply (rule RR-intro)
 apply (simp-all add: unrest assms)
 apply (simp-all add: Healthy-def)
 apply (simp-all add: R1-cond R2c-condr Healthy-if assms RR-implies-R2c closure R2c-tr'-minus-tr)

done

lemma rea-skip-RR [closure]:
 $II_r\ \text{is}\ RR$
 apply (rel-auto) using minus-zero-eq by blast

lemma tr'-eq-tr-RR-closed [closure]: $\$tr' =_u \$tr\ \text{is}\ RR$
 apply (rel-auto) using minus-zero-eq by auto

lemma *inf-RR-closed* [closure]:
 $\llbracket P \text{ is } RR; Q \text{ is } RR \rrbracket \implies P \sqcap Q \text{ is } RR$
 by (*simp add: disj-RR uinf-or*)

lemma *conj-tr-strict-RR-closed* [closure]:

assumes $P \text{ is } RR$
 shows $(P \wedge \$tr <_u \$tr') \text{ is } RR$

proof –

have $RR(RR(P) \wedge \$tr <_u \$tr') = (RR(P) \wedge \$tr <_u \$tr')$
 by (*rel-auto*)
 thus ?thesis
 by (*metis Healthy-def assms*)

qed

lemma *rea-assert-RR-closed* [closure]:

assumes $b \text{ is } RR$
 shows $\{b\}_r \text{ is } RR$
 by (*simp add: closure assms rea-assert-def*)

lemma *upower-RR-closed* [closure]:

$\llbracket i > 0; P \text{ is } RR \rrbracket \implies P \wedge i \text{ is } RR$
 apply (*induct i, simp-all*)
 apply (*rename-tac i*)
 apply (*case-tac i = 0*)
 apply (*simp-all add: closure upred-semiring.power-Suc*)
 done

lemma *seq-power-RR-closed* [closure]:

assumes $P \text{ is } RR \ Q \text{ is } RR$
 shows $(P \wedge i) ;; Q \text{ is } RR$
 by (*metis assms neq0-conv seq-RR-closed seqr-left-unit upower-RR-closed upred-semiring.power-0*)

lemma *ustar-right-RR-closed* [closure]:

assumes $P \text{ is } RR \ Q \text{ is } RR$
 shows $P ;; Q^* \text{ is } RR$

proof –

have $P ;; Q^* = P ;; (\bigsqcap i \in \{0..\} \cdot Q \wedge i)$
 by (*simp add: ustar-def*)
 also have $\dots = P ;; (II \sqcap (\bigsqcap i \in \{1..\} \cdot Q \wedge i))$
 by (*metis One-nat-def UINF-atLeast-first upred-semiring.power-0*)
 also have $\dots = (P \vee P ;; (\bigsqcap i \in \{1..\} \cdot Q \wedge i))$
 by (*simp add: disj-upred-def[THEN sym] seqr-or-distr*)
 also have $\dots \text{ is } RR$

proof –

have $(\bigsqcap i \in \{1..\} \cdot Q \wedge i) \text{ is } RR$
 by (*rule UINF-mem-Continuous-closed, simp-all add: assms closure*)
 thus ?thesis
 by (*simp add: assms closure*)

qed

finally show ?thesis .

qed

lemma *ustar-left-RR-closed* [closure]:

assumes $P \text{ is } RR \ Q \text{ is } RR$
 shows $P^* ;; Q \text{ is } RR$


```

proof –
  have  $P^* \;; Q = (\bigcap i \in \{0..\} \cdot P \wedge i) \;; Q$ 
    by (simp add: ustar-def)
  also have  $\dots = (II \sqcap (\bigcap i \in \{1..\} \cdot P \wedge i)) \;; Q$ 
    by (metis One-nat-def UINF-atLeast-first upred-semiring.power-0)
  also have  $\dots = (Q \vee (\bigcap i \in \{1..\} \cdot P \wedge i) \;; Q)$ 
    by (simp add: disj-upred-def[THEN sym] segr-or-distl)
  also have  $\dots$  is  $RR$ 
proof –
  have  $(\bigcap i \in \{1..\} \cdot P \wedge i)$  is  $RR$ 
    by (rule UINF-mem-Continuous-closed, simp-all add: assms closure)
  thus ?thesis
    by (simp add: assms closure)
qed
finally show ?thesis .
qed

lemma uplus-RR-closed [closure]:  $P$  is  $RR \implies P^+$  is  $RR$ 
  by (simp add: uplus-def ustar-right-RR-closed)

lemma trace-ext-prefix-RR [closure]:
   $\llbracket \$tr \# e; \$ok \# e; \$wait \# e; out\alpha \# e \rrbracket \implies \$tr \wedge_u e \leq_u \$tr' \text{ is } RR$ 
  apply (rel-auto)
  apply (metis (no-types, lifting) Prefix-Order.same-prefix-prefix less-eq-list-def prefix-concat-minus zero-list-def)
  apply (metis append-minus list-append-prefixD minus-cancel-le order-reft)
done

lemma rea-subst-R1-closed [closure]:  $P\llbracket v \rrbracket_r$  is  $R1$ 
  by (rel-auto)

lemma R5-comp [rpred]:
  assumes  $P$  is  $RR$   $Q$  is  $RR$ 
  shows  $R5(P \;; Q) = R5(P) \;; R5(Q)$ 
proof –
  have  $R5(RR(P) \;; RR(Q)) = R5(RR(P)) \;; R5(RR(Q))$ 
    by (rel-auto; force)
  thus ?thesis
    by (simp add: Healthy-if assms)
qed

lemma R4-comp [rpred]:
  assumes  $P$  is  $R4$   $Q$  is  $RR$ 
  shows  $R4(P \;; Q) = P \;; Q$ 
proof –
  have  $R4(R4(P) \;; RR(Q)) = R4(P) \;; RR(Q)$ 
    by (rel-auto, blast)
  thus ?thesis
    by (simp add: Healthy-if assms)
qed

lemma rea-rename-RR-closed [closure]:
  assumes  $P$  is  $RR$ 
  shows  $P\llbracket f \rrbracket_r$  is  $RR$ 
proof –
  have  $(RR\ P)\llbracket f \rrbracket_r$  is  $RR$ 

```

by (*rel-auto*)
 thus ?thesis
 by (*simp add: Healthy-if assms*)
 qed

6.5 Reactive relational calculus

lemma *rea-skip-unit* [*rpred*]:
 assumes *P is RR*
 shows $P ;; II_r = P II_r ;; P = P$
proof –
 have 1: $RR(P) ;; II_r = RR(P)$
 by (*rel-auto*)
 have 2: $II_r ;; RR(P) = RR(P)$
 by (*rel-auto*)
 from 1 2 show $P ;; II_r = P II_r ;; P = P$
 by (*simp-all add: Healthy-if assms*)
 qed

lemma *rea-true-conj* [*rpred*]:
 assumes *P is R1*
 shows $(true_r \wedge P) = P (P \wedge true_r) = P$
 using *assms*
 by (*simp-all add: Healthy-def R1-def utp-pred-laws.inf-commute*)

lemma *rea-true-disj* [*rpred*]:
 assumes *P is R1*
 shows $(true_r \vee P) = true_r (P \vee true_r) = true_r$
 using *assms* by (*metis Healthy-def R1-disj disj-comm true-disj-zero*)

lemma *rea-not-not* [*rpred*]: $P \text{ is } R1 \implies (\neg_r \neg_r P) = P$
 by (*simp add: rea-not-def R1-negate-R1 Healthy-if*)

lemma *rea-not-rea-true* [*simp*]: $(\neg_r true_r) = false$
 by (*simp add: rea-not-def R1-negate-R1 R1-false*)

lemma *rea-not-false* [*simp*]: $(\neg_r false) = true_r$
 by (*simp add: rea-not-def*)

lemma *rea-true-impl* [*rpred*]:
 $P \text{ is } R1 \implies (true_r \Rightarrow_r P) = P$
 by (*simp add: rea-not-def rea-impl-def R1-negate-R1 R1-false Healthy-if*)

lemma *rea-true-impl'* [*rpred*]:
 $P \text{ is } R1 \implies (true \Rightarrow_r P) = P$
 by (*simp add: rea-not-def rea-impl-def R1-negate-R1 R1-false Healthy-if*)

lemma *rea-false-impl* [*rpred*]:
 $P \text{ is } R1 \implies (false \Rightarrow_r P) = true_r$
 by (*simp add: rea-impl-def rpred Healthy-if*)

lemma *rea-impl-true* [*simp*]: $(P \Rightarrow_r true_r) = true_r$
 by (*rel-auto*)

lemma *rea-impl-false* [*simp*]: $(P \Rightarrow_r false) = (\neg_r P)$
 by (*rel-simp*)

lemma *rea-imp-refl* [*rpred*]: $P \text{ is } R1 \implies (P \Rightarrow_r P) = \text{true}_r$
by (*rel-blast*)

lemma *rea-impl-conj* [*rpred*]:
 $(P \Rightarrow_r Q \Rightarrow_r R) = ((P \wedge Q) \Rightarrow_r R)$
by (*rel-auto*)

lemma *rea-impl-mp* [*rpred*]:
 $(P \wedge (P \Rightarrow_r Q)) = (P \wedge Q)$
by (*rel-auto*)

lemma *rea-impl-conj-combine* [*rpred*]:
 $((P \Rightarrow_r Q) \wedge (P \Rightarrow_r R)) = (P \Rightarrow_r Q \wedge R)$
by (*rel-auto*)

lemma *rea-impl-alt-def*:
assumes $Q \text{ is } R1$
shows $(P \Rightarrow_r Q) = R1(P \Rightarrow Q)$
proof –
have $(P \Rightarrow_r R1(Q)) = R1(P \Rightarrow Q)$
by (*rel-auto*)
thus *?thesis*
by (*simp add: assms Healthy-if*)
qed

lemma *rea-impl-disj*:
 $(P \Rightarrow_r Q \vee R) = (Q \vee (P \Rightarrow_r R))$
by (*rel-auto*)

lemma *rea-not-true* [*simp*]: $(\neg_r \text{true}) = \text{false}$
by (*rel-auto*)

lemma *rea-not-demorgan1* [*simp*]:
 $(\neg_r (P \wedge Q)) = (\neg_r P \vee \neg_r Q)$
by (*rel-auto*)

lemma *rea-not-demorgan2* [*simp*]:
 $(\neg_r (P \vee Q)) = (\neg_r P \wedge \neg_r Q)$
by (*rel-auto*)

lemma *rea-not-or* [*rpred*]:
 $P \text{ is } R1 \implies (P \vee \neg_r P) = \text{true}_r$
by (*rel-blast*)

lemma *rea-not-and* [*simp*]:
 $(P \wedge \neg_r P) = \text{false}$
by (*rel-auto*)

lemma *truer-bottom-rpred* [*rpred*]: $P \text{ is } RR \implies R1(\text{true}) \sqsubseteq P$
by (*metis Healthy-def R1-RR R1-mono utp-pred-laws.top-greatest*)

lemma *ext-close-weakening*: $P ;; \text{true}_r \sqsubseteq P$
by (*rel-auto*)

lemma *rea-not-INFIMUM* [simp]:
 $(\neg_r (\bigsqcup i \in A. Q(i))) = (\bigcap i \in A. \neg_r Q(i))$
by (*rel-auto*)

lemma *rea-not-USUP* [simp]:
 $(\neg_r (\bigsqcup i \in A \cdot Q(i))) = (\bigcap i \in A \cdot \neg_r Q(i))$
by (*rel-auto*)

lemma *rea-not-SUPREMUM* [simp]:
 $A \neq \{\} \implies (\neg_r (\bigcap i \in A. Q(i))) = (\bigsqcup i \in A. \neg_r Q(i))$
by (*rel-auto*)

lemma *rea-not-UNIF* [simp]:
 $A \neq \{\} \implies (\neg_r (\bigcap i \in A \cdot Q(i))) = (\bigsqcup i \in A \cdot \neg_r Q(i))$
by (*rel-auto*)

lemma *USUP-mem-rea-true* [simp]: $A \neq \{\} \implies (\bigsqcup i \in A \cdot \text{true}_r) = \text{true}_r$
by (*rel-auto*)

lemma *USUP-ind-rea-true* [simp]: $(\bigsqcup i \cdot \text{true}_r) = \text{true}_r$
by (*rel-auto*)

lemma *UNIF-ind-rea-true* [rpred]: $A \neq \{\} \implies (\bigcap i \in A \cdot \text{true}_r) = \text{true}_r$
by (*rel-auto*)

lemma *UNIF-rea-impl*: $(\bigcap P \in A \cdot F(P) \Rightarrow_r G(P)) = ((\bigsqcup P \in A \cdot F(P)) \Rightarrow_r (\bigcap P \in A \cdot G(P)))$
by (*rel-auto*)

lemma *rea-not-shEx* [rpred]: $(\neg_r \text{shEx } P) = (\text{shAll } (\lambda x. \neg_r P x))$
by (*rel-auto*)

lemma *rea-assert-true*:
 $\{\text{true}_r\}_r = \text{II}_r$
by (*rel-auto*)

lemma *rea-false-true*:
 $\{\text{false}\}_r = \text{true}_r$
by (*rel-auto*)

lemma *rea-rename-id* [rpred]:
assumes P *is* RR
shows $P(\text{id})_r = P$
proof –
have $(RR P)(\text{id})_r = RR P$
by (*rel-auto*)
thus *?thesis* **by** (*simp add: Healthy-if assms*)
qed

lemma *rea-rename-comp* [rpred]:
assumes *renamer* f *renamer* g P *is* RR
shows $P(g \circ f)_r = P(g)_r(f)_r$
oops

lemma *rea-rename-false* [rpred]: $\text{false}(f)_r = \text{false}$
by (*rel-auto*)

lemma *rea-rename-disj* [*rpred*]:
 $(P \vee Q)\langle f \rangle_r = (P\langle f \rangle_r \vee Q\langle f \rangle_r)$
by (*rel-blast*)

lemma *rea-rename-UINF-ind* [*rpred*]:
 $(\prod i \cdot P i)\langle f \rangle_r = (\prod i \cdot (P i)\langle f \rangle_r)$
by (*rel-blast*)

lemma *rea-rename-UINF-mem* [*rpred*]:
 $(\prod i \in A \cdot P i)\langle f \rangle_r = (\prod i \in A \cdot (P i)\langle f \rangle_r)$
by (*rel-blast*)

lemma *rea-rename-conj* [*rpred*]:
assumes *renamer f P is RR Q is RR*
shows $(P \wedge Q)\langle f \rangle_r = (P\langle f \rangle_r \wedge Q\langle f \rangle_r)$
proof –
interpret *ren: renamer f* **by** (*simp add: assms*)
have $(RR P \wedge RR Q)\langle f \rangle_r = ((RR P)\langle f \rangle_r \wedge (RR Q)\langle f \rangle_r)$
using *injD[OF ren.injective]*
by (*rel-auto; blast*)
thus *?thesis* **by** (*simp add: Healthy-if assms*)
qed

lemma *rea-rename-USUP-ind* [*rpred*]:
assumes *renamer f $\bigwedge i. P i$ is RR*
shows $(\bigsqcup i \cdot P i)\langle f \rangle_r = (\bigsqcup i \cdot (P i)\langle f \rangle_r)$
proof –
interpret *ren: renamer f* **by** (*simp add: assms*)
have $(\bigsqcup i \cdot RR(P i))\langle f \rangle_r = (\bigsqcup i \cdot (RR(P i))\langle f \rangle_r)$
using *injD[OF ren.injective]*
by (*rel-auto, blast, metis (mono-tags, hide-lams)*)
thus *?thesis*
by (*simp add: Healthy-if assms cong: USUP-all-cong*)
qed

lemma *rea-rename-USUP-mem* [*rpred*]:
assumes *renamer f $A \neq \{\}$ $\bigwedge i. i \in A \implies P i$ is RR*
shows $(\bigsqcup i \in A \cdot P i)\langle f \rangle_r = (\bigsqcup i \in A \cdot (P i)\langle f \rangle_r)$
proof –
interpret *ren: renamer f* **by** (*simp add: assms*)
have $(\bigsqcup i \in A \cdot RR(P i))\langle f \rangle_r = (\bigsqcup i \in A \cdot (RR(P i))\langle f \rangle_r)$
using *injD[OF ren.injective] assms(2)*
by (*rel-auto, blast, metis (no-types, hide-lams)*)
thus *?thesis*
by (*simp add: Healthy-if assms cong: USUP-cong*)
qed

lemma *rea-rename-skip-rea* [*rpred*]: *renamer f $\implies II_r\langle f \rangle_r = II_r$*
using *minus-zero-eq* **by** (*rel-auto*)

lemma *rea-rename-seq* [*rpred*]:
assumes *renamer f P is RR Q is RR*
shows $(P ;; Q)\langle f \rangle_r = P\langle f \rangle_r ;; Q\langle f \rangle_r$
proof –

```

interpret ren: renamer f by (simp add: assms)
from assms(1) have (RR(P) ;; RR(Q))(f)r = (RR P)(f)r ;; (RR Q)(f)r
by (rel-auto)
  (metis (no-types, lifting) diff-add-cancel-left' le-add minus-assoc mono-def ren.minus ren.monotonic
trace-class.add-diff-cancel-left trace-class.add-left-mono)+
thus ?thesis
by (simp add: Healthy-if assms)
qed

```

```

declare R4-idem [rpred]
declare R4-false [rpred]
declare R4-conj [rpred]
declare R4-disj [rpred]

```

```

declare R4-R5 [rpred]
declare R5-R4 [rpred]

```

```

declare R5-conj [rpred]
declare R5-disj [rpred]

```

```

lemma R4-USUP [rpred]: I ≠ {} ⇒ R4(⋒ i ∈ I • P(i)) = (⋒ i ∈ I • R4(P(i)))
by (rel-auto)

```

```

lemma R5-USUP [rpred]: I ≠ {} ⇒ R5(⋒ i ∈ I • P(i)) = (⋒ i ∈ I • R5(P(i)))
by (rel-auto)

```

```

lemma R4-UINF [rpred]: R4(⋓ i ∈ I • P(i)) = (⋓ i ∈ I • R4(P(i)))
by (rel-auto)

```

```

lemma R5-UINF [rpred]: R5(⋓ i ∈ I • P(i)) = (⋓ i ∈ I • R5(P(i)))
by (rel-auto)

```

6.6 UTP theory

We create a UTP theory of reactive relations which in particular provides Kleene star theorems

```

interpretation rrel-theory: utp-theory-kleene RR IIr
rewrites P ∈ carrier rrel-theory.thy-order ⇔ P is RR
and le rrel-theory.thy-order = (⊆)
and eq rrel-theory.thy-order = (=)
and rrel-top: rrel-theory.utp-top = false
and rrel-bottom: rrel-theory.utp-bottom = truer
proof –
interpret utp-theory-continuous RR
by (unfold-locales, simp-all add: add: RR-idem RR-Continuous)
show top:utp-top = false
by (simp add: healthy-top, rel-auto)
show bot:utp-bottom = truer
by (simp add: healthy-bottom, rel-auto)
show utp-theory-kleene RR IIr
by (unfold-locales, simp-all add: closure rpred top)
qed (simp-all)

```

```

abbreviation rea-star :: - ⇒ - (-*r [999] 999) where
P*r ≡ rrel-theory.utp-star P

```

The supernova tactic explodes conjectures using the Kleene star laws and relational calculus

```
method supernova = ((safe intro!: rrel-theory.Star-inductr rrel-theory.Star-inductl, simp-all add: closure) ; rel-auto)[1]
```

6.7 Instantaneous Reactive Relations

Instantaneous Reactive Relations, where the trace stays the same.

abbreviation *Instant* :: (*t*::trace, *'α*) *hrel-rp* \Rightarrow (*t*, *'α*) *hrel-rp* **where**
Instant(*P*) \equiv *tr*:: $\llbracket P \rrbracket$

lemma *skip-rea-Instant* [*closure*]: *II_r* is *Instant*
by (*rel-auto*)

end

7 Reactive Conditions

```
theory utp-rea-cond
imports utp-rea-rel
begin
```

7.1 Healthiness Conditions

definition *RC1* :: (*t*::trace, *'α*, *'β*) *rel-rp* \Rightarrow (*t*, *'α*, *'β*) *rel-rp* **where**
[*upred-defs*]: *RC1*(*P*) = (\neg_r (\neg_r *P*) ;; *true_r*)

definition *RC* :: (*t*::trace, *'α*, *'β*) *rel-rp* \Rightarrow (*t*, *'α*, *'β*) *rel-rp* **where**
[*upred-defs*]: *RC* = *RC1* \circ *RR*

lemma *RC-intro*: $\llbracket P \text{ is } RR; ((\neg_r (\neg_r P) ;; \text{true}_r) = P) \rrbracket \Longrightarrow P \text{ is } RC$
by (*simp add: Healthy-def RC1-def RC-def*)

lemma *RC-intro'*: $\llbracket P \text{ is } RR; P \text{ is } RC1 \rrbracket \Longrightarrow P \text{ is } RC$
by (*simp add: Healthy-def RC1-def RC-def*)

lemma *RC1-idem*: *RC1*(*RC1*(*P*)) = *RC1*(*P*)
by (*rel-auto*, (*blast intro: dual-order.trans*)+)

lemma *RC1-mono*: *P* \sqsubseteq *Q* \Longrightarrow *RC1*(*P*) \sqsubseteq *RC1*(*Q*)
by (*rel-blast*)

lemma *RC1-prop*:
assumes *P* is *RC1*
shows (\neg_r *P*) ;; *R1 true* = (\neg_r *P*)
proof –
have (\neg_r *P*) = (\neg_r (*RC1 P*))
by (*simp add: Healthy-if assms*)
also have ... = (\neg_r *P*) ;; *R1 true*
by (*simp add: RC1-def rpred closure*)
finally show ?thesis ..
qed

lemma *R2-RC*: *R2* (*RC P*) = *RC P*
proof –
have \neg_r *RR P* is *RR*

```

  by (metis (no-types) Healthy-Idempotent RR-Idempotent RR-rea-not)
then show ?thesis
  by (metis (no-types) Healthy-def' R1-R2c-seqr-distribute R2-R2c-def RC1-def RC-def RR-implies-R1
RR-implies-R2c comp-apply rea-not-R2-closed rea-true-R1 rea-true-R2c)
qed

```

```

lemma RC-R2-def:  $RC = RC1 \circ RR$ 
  by (auto simp add: RC-def fun-eq-iff R1-R2c-commute[THEN sym] R1-R2c-is-R2)

```

```

lemma RC-implies-R2:  $P \text{ is } RC \implies P \text{ is } R2$ 
  by (metis Healthy-def' R2-RC)

```

```

lemma RC-ex-ok-wait:  $(\exists \{ \$ok, \$ok', \$wait, \$wait' \} \cdot RC\ P) = RC\ P$ 
  by (rel-auto)

```

An important property of reactive conditions is they are monotonic with respect to the trace. That is, P with a shorter trace is refined by P with a longer trace.

```

lemma RC-prefix-refine:
  assumes  $P \text{ is } RC\ s \leq t$ 
  shows  $P \llbracket 0, \llbracket s \rrbracket / \$tr, \$tr' \rrbracket \sqsubseteq P \llbracket 0, \llbracket t \rrbracket / \$tr, \$tr' \rrbracket$ 
proof -
  from assms(2) have  $(RC\ P) \llbracket 0, \llbracket s \rrbracket / \$tr, \$tr' \rrbracket \sqsubseteq (RC\ P) \llbracket 0, \llbracket t \rrbracket / \$tr, \$tr' \rrbracket$ 
  apply (rel-auto)
  using dual-order.trans apply blast
  done
  thus ?thesis
  by (simp only: assms(1) Healthy-if)
qed

```

The RC healthy relations can also be defined in terms of prefix closure, which is characterised by the healthiness condition below.

definition $RC2 :: ('t::trace, 'α, 'β) \text{ rel-rp} \Rightarrow ('t, 'α, 'β) \text{ rel-rp}$ **where**
 $[upred-defs]: RC2(P) = R1(P ;; (\$tr' \leq_u \$tr))$

```

lemma RC2-RR-commute:
   $RC2(RR(P)) = RR(RC2(P))$ 
  apply (rel-auto)
  using minus-cancel-le apply blast
  apply (metis diff-add-cancel-left' le-add trace-class.add-diff-cancel-left trace-class.add-left-mono)
  done

```

Intuitive meaning of $RC2$

```

lemma RC2-form-1:
  assumes  $P \text{ is } RR$ 
  shows  $RC2(P) = (\exists tr_0 \cdot (\exists \$\Sigma_R' \cdot P) \llbracket \llbracket tr_0 \rrbracket / \$tr' \rrbracket \wedge \$tr' \leq_u \llbracket tr_0 \rrbracket \wedge \$tr \leq_u \$tr')$ 
proof -
  have  $RC2(RR(P)) = (\exists tr_0 \cdot (\exists \$\Sigma_R' \cdot RR\ P) \llbracket \llbracket tr_0 \rrbracket / \$tr' \rrbracket \wedge \$tr' \leq_u \llbracket tr_0 \rrbracket \wedge \$tr \leq_u \$tr')$ 
  by (rel-blast)
  thus ?thesis
  by (metis (mono-tags, lifting) Healthy-if assms shEx-cong)
qed

```

```

lemma RC2-form-2:
  assumes  $P \text{ is } RR$ 

```


shows $RC2(P) = (\exists (t_0, t_1) \cdot (\exists \$\Sigma_{R'} \cdot P) \llbracket 0, \langle t_1 \rangle / \$tr, \$tr' \rrbracket \wedge \langle t_0 \rangle \leq_u \langle t_1 \rangle \wedge \$tr' =_u \$tr + \langle t_0 \rangle)$
proof –
have $RC2(RR(P)) = (\exists (t_0, t_1) \cdot (\exists \$\Sigma_{R'} \cdot RR(P)) \llbracket 0, \langle t_1 \rangle / \$tr, \$tr' \rrbracket \wedge \langle t_0 \rangle \leq_u \langle t_1 \rangle \wedge \$tr' =_u \$tr + \langle t_0 \rangle)$
apply (*rel-auto*)
apply (*metis diff-add-cancel-left' trace-class.add-le-imp-le-left*)
apply (*metis le-add trace-class.add-diff-cancel-left trace-class.add-left-mono*)
done
thus *?thesis*
by (*simp add: Healthy-if assms*)
qed

Every reactive condition is prefix closed

lemma *RC-prefix-closed*:

assumes P is *RC*
shows P is *RC2*
proof –
have $RC2(RC(P)) = RC(P)$
apply (*rel-auto*) **using** *dual-order.trans* **by** *blast*
thus *?thesis*
by (*metis Healthy-def assms*)
qed

lemma *RC2-RR-is-RC1*:

assumes P is *RR* P is *RC2*
shows P is *RC1*
proof –
have $RC1(RC2(RR(P))) = RC2(RR(P))$
apply (*rel-auto*) **using** *dual-order.trans* **by** *blast*
thus *?thesis*
by (*metis Healthy-def assms(1) assms(2)*)
qed

RC closure can be demonstrated in terms of prefix closure.

lemma *RC-intro-prefix-closed*:

assumes P is *RR* P is *RC2*
shows P is *RC*
by (*simp add: RC2-RR-is-RC1 RC-intro' assms*)

7.2 Closure laws

lemma *RC-implies-RR [closure]*:

assumes P is *RC*
shows P is *RR*
by (*metis Healthy-def RC-ex-ok-wait RC-implies-R2 RR-def assms*)

lemma *RC-implies-RC1*: P is *RC* $\implies P$ is *RC1*

by (*metis Healthy-def RC-R2-def RC-implies-RR comp-eq-dest-lhs*)

lemma *RC1-trace-ext-prefix*:

$out\alpha \# e \implies RC1(\neg_r \$tr \hat{^}_u e \leq_u \$tr') = (\neg_r \$tr \hat{^}_u e \leq_u \$tr')$
by (*rel-auto, blast, metis (no-types, lifting) dual-order.trans*)

lemma *RC1-conj [rpred]*: $RC1(P \wedge Q) = (RC1(P) \wedge RC1(Q))$

by (*rel-blast*)

lemma *conj-RC1-closed* [*closure*]:
 $\llbracket P \text{ is } RC1; Q \text{ is } RC1 \rrbracket \implies P \wedge Q \text{ is } RC1$
 by (*simp add: Healthy-def RC1-conj*)

lemma *disj-RC1-closed* [*closure*]:
 assumes $P \text{ is } RC1$ $Q \text{ is } RC1$
 shows $(P \vee Q) \text{ is } RC1$
proof –
 have $1: RC1(RC1(P) \vee RC1(Q)) = (RC1(P) \vee RC1(Q))$
 apply (*rel-auto*) using *dual-order.trans* by *blast+*
 show ?thesis
 by (*metis (no-types) Healthy-def 1 assms*)
qed

lemma *conj-RC-closed* [*closure*]:
 assumes $P \text{ is } RC$ $Q \text{ is } RC$
 shows $(P \wedge Q) \text{ is } RC$
 by (*metis Healthy-def RC-R2-def RC-implies-RR assms comp-apply conj-RC1-closed conj-RR*)

lemma *rea-true-RC* [*closure*]: $true_r \text{ is } RC$
 by (*rel-auto*)

lemma *false-RC* [*closure*]: $false \text{ is } RC$
 by (*rel-auto*)

lemma *disj-RC-closed* [*closure*]: $\llbracket P \text{ is } RC; Q \text{ is } RC \rrbracket \implies (P \vee Q) \text{ is } RC$
 by (*metis Healthy-def RC-R2-def RC-implies-RR comp-apply disj-RC1-closed disj-RR*)

lemma *UINF-mem-RC1-closed* [*closure*]:
 assumes $\bigwedge i. P \ i \text{ is } RC1$
 shows $(\bigcap i \in A. P \ i) \text{ is } RC1$
proof –
 have $1: RC1(\bigcap i \in A. RC1(P \ i)) = (\bigcap i \in A. RC1(P \ i))$
 by (*rel-auto, meson order.trans*)
 show ?thesis
 by (*metis (mono-tags, lifting) 1 Healthy-def UINF-cong assms*)
qed

lemma *UINF-mem-RC-closed* [*closure*]:
 assumes $\bigwedge i. P \ i \text{ is } RC$
 shows $(\bigcap i \in A. P \ i) \text{ is } RC$
proof –
 have $RC(\bigcap i \in A. P \ i) = (RC1 \circ RR)(\bigcap i \in A. P \ i)$
 by (*simp add: RC-def*)
 also have $\dots = RC1(\bigcap i \in A. RR(P \ i))$
 by (*rel-blast*)
 also have $\dots = RC1(\bigcap i \in A. RC1(P \ i))$
 by (*simp add: Healthy-if RC-implies-RR RC-implies-RC1 assms*)
 also have $\dots = (\bigcap i \in A. RC1(P \ i))$
 by (*rel-auto, meson order.trans*)
 also have $\dots = (\bigcap i \in A. P \ i)$
 by (*simp add: Healthy-if RC-implies-RC1 assms*)
 finally show ?thesis

```

    by (simp add: Healthy-def)
qed

lemma UINF-ind-RC-closed [closure]:
  assumes  $\bigwedge i. P\ i\ \text{is}\ RC$ 
  shows  $(\bigcap i. P\ i)\ \text{is}\ RC$ 
  by (metis (no-types) UINF-as-Sup-collect' UINF-as-Sup-image UINF-mem-RC-closed assms)

lemma USUP-mem-RC1-closed [closure]:
  assumes  $\bigwedge i. i \in A \implies P\ i\ \text{is}\ RC1\ A \neq \{\}$ 
  shows  $(\bigsqcup_{i \in A} P\ i)\ \text{is}\ RC1$ 
proof -
  have  $RC1(\bigsqcup_{i \in A} P\ i) = RC1(\bigsqcup_{i \in A} RC1(P\ i))$ 
    by (simp add: Healthy-if assms(1) cong: USUP-cong)
  also from assms(2) have  $\dots = (\bigsqcup_{i \in A} RC1(P\ i))$ 
    using dual-order.trans by (rel-blast)
  also have  $\dots = (\bigsqcup_{i \in A} P\ i)$ 
    by (simp add: Healthy-if assms(1) cong: USUP-cong)
  finally show ?thesis
    using Healthy-def by blast
qed

lemma USUP-mem-RC-closed [closure]:
  assumes  $\bigwedge i. i \in A \implies P\ i\ \text{is}\ RC\ A \neq \{\}$ 
  shows  $(\bigsqcup_{i \in A} P\ i)\ \text{is}\ RC$ 
  by (rule RC-intro', simp-all add: closure assms RC-implies-RC1)

lemma USUP-ind-RC-closed [closure]:
   $\llbracket \bigwedge i. P\ i\ \text{is}\ RC \rrbracket \implies (\bigcap i. P\ i)\ \text{is}\ RC$ 
  by (metis UNIV-not-empty USUP-mem-RC-closed)

lemma neg-trace-ext-prefix-RC [closure]:
   $\llbracket \$tr \# e; \$ok \# e; \$wait \# e; out\alpha \# e \rrbracket \implies \neg_r \$tr \hat{^}_u e \leq_u \$tr' \text{ is } RC$ 
  by (rule RC-intro, simp add: closure, metis RC1-def RC1-trace-ext-prefix)

lemma RC1-unrest:
   $\llbracket mwb\text{-}lens\ x; x \bowtie tr \rrbracket \implies \$x' \# RC1(P)$ 
  by (simp add: RC1-def unrest)

lemma RC-unrest-dashed [unrest]:
   $\llbracket P\ \text{is}\ RC; mwb\text{-}lens\ x; x \bowtie tr \rrbracket \implies \$x' \# P$ 
  by (metis Healthy-if RC1-unrest RC-implies-RC1)

lemma RC1-RR-closed [closure]:  $P\ \text{is}\ RR \implies RC1(P)\ \text{is}\ RR$ 
  by (simp add: RC1-def closure)

end

```

8 Reactive Programs

```

theory utp-rea-prog
  imports utp-rea-cond
begin

```

8.1 Stateful reactive alphabet

$R3$ as presented in the UTP book and related publications is not sensitive to state, although reactive programs often need this property. Thus it is necessary to use a modification of $R3$ from Butterfield et al. [1] that explicitly states that intermediate waiting states do not propagate final state variables. In order to do this we need an additional observational variable that captures the program state that we call st . Upon this foundation, we can define operators for reactive programs [3].

alphabet $(t, s) \text{ rsp-vars} = t::\text{trace } rp\text{-vars} +$
 $st :: s$

print-theorems

type-synonym $(s, t, \alpha) \text{ rsp} = (t, s, \alpha) \text{ rsp-vars-scheme}$
type-synonym $(s, t, \alpha, \beta) \text{ rel-rsp} = ((s, t, \alpha) \text{ rsp}, (s, t, \beta) \text{ rsp}) \text{ urel}$
type-synonym $(s, t, \alpha) \text{ hrel-rsp} = (s, t, \alpha) \text{ rsp hrel}$
type-synonym $(s, t) \text{ rdes} = (s, t, \text{unit}) \text{ hrel-rsp}$

translations

$(\text{type}) (s, t, \alpha) \text{ rsp} \leq (\text{type}) (t, (s, \alpha) \text{ rsp-vars-ext}) \text{ rp}$
 $(\text{type}) (s, t, \alpha) \text{ rsp} \leq (\text{type}) (t, (s, \alpha) \text{ rsp-vars-scheme}) \text{ rp}$
 $(\text{type}) (s, t, \text{unit}) \text{ rsp} \leq (\text{type}) (t, s \text{ rsp-vars}) \text{ rp}$
 $(\text{type}) (s, t, \alpha, \beta) \text{ rel-rsp} \leq (\text{type}) ((s, t, \alpha) \text{ rsp}, (s, t, \beta) \text{ rsp}) \text{ urel}$
 $(\text{type}) (s, t, \alpha) \text{ hrel-rsp} \leq (\text{type}) (s, t, \alpha) \text{ rsp hrel}$
 $(\text{type}) (s, t) \text{ rdes} \leq (\text{type}) (s, t, \text{unit}) \text{ hrel-rsp}$

notation $\text{rsp-vars.more}_L (\Sigma_S)$

syntax

$\text{-svid-st-alpha} :: \text{svid } (\Sigma_S)$

translations

$\text{-svid-st-alpha} \Rightarrow \text{CONST } \text{rsp-vars.more}_L$

lemma $\text{rea-lens-equiv-st-rest}: \Sigma_R \approx_L st +_L \Sigma_S$

by simp

lemma $\text{srea-lens-bij}: \text{bij-lens } (ok +_L \text{wait} +_L \text{tr} +_L st +_L \Sigma_S)$

proof –

have $ok +_L \text{wait} +_L \text{tr} +_L st +_L \Sigma_S \approx_L ok +_L \text{wait} +_L \text{tr} +_L \Sigma_R$
by $(\text{auto intro!::lens-plus-cong, rule lens-equiv-sym, simp add: rea-lens-equiv-st-rest})$
also have $\dots \approx_L 1_L$
using $\text{bij-lens-equiv-id}[of ok +_L \text{wait} +_L \text{tr} +_L \Sigma_R]$ **by** $(\text{simp add: rea-lens-bij})$
finally show $?thesis$
by $(\text{simp add: bij-lens-equiv-id})$

qed

lemma $\text{st-qual-alpha } [\alpha]: x ;_L \text{fst}_L ;_L st \times_L st = (\$st:x)_v$

by $(\text{metis (no-types, hide-lams) in-var-def in-var-prod-lens lens-comp-assoc st-vwb-lens vwb-lens-wb})$

declare $\text{des-vars.splits } [\alpha\text{-splits del}]$

declare $\text{rp-vars.splits } [\alpha\text{-splits del}]$

declare $\text{rp-vars.splits } [\alpha\text{-splits}]$

declare *des-vars.splits* [*alpha-splits*]

lemma *unrest-st'-neg-RC* [*unrest*]:

assumes *P is RR P is RC*

shows $\$st' \# P$

proof –

have $P = (\neg_r \neg_r P)$

by (*simp add: closure rpred assms*)

also have $\dots = (\neg_r (\neg_r P) ;; true_r)$

by (*metis Healthy-if RC1-def RC-implies-RC1 assms(2) calculation*)

also have $\$st' \# \dots$

by (*rel-auto*)

finally show *?thesis* .

qed

lemma *ex-st'-RR-closed* [*closure*]:

assumes *P is RR*

shows $(\exists \$st' \cdot P) \text{ is } RR$

proof –

have $RR (\exists \$st' \cdot RR(P)) = (\exists \$st' \cdot RR(P))$

by (*rel-auto*)

thus *?thesis*

by (*metis Healthy-def assms*)

qed

lemma *unrest-st'-R4* [*unrest*]:

$\$st' \# P \implies \$st' \# R4(P)$

by (*rel-auto*)

lemma *unrest-st'-R5* [*unrest*]:

$\$st' \# P \implies \$st' \# R5(P)$

by (*rel-auto*)

8.2 State Lifting

abbreviation *lift-state-rel* ($\lceil \cdot \rceil_S$)

where $\lceil P \rceil_S \equiv P \oplus_p (st \times_L st)$

abbreviation *drop-state-rel* ($\lfloor \cdot \rfloor_S$)

where $\lfloor P \rfloor_S \equiv P \upharpoonright_e (st \times_L st)$

abbreviation *lift-state-pre* ($\lceil \cdot \rceil_{S<}$)

where $\lceil p \rceil_{S<} \equiv \lceil \lceil p \rceil_{<} \rceil_S$

no-utp-lift *lift-state-pre*

abbreviation *drop-state-pre* ($\lfloor \cdot \rfloor_{S<}$)

where $\lfloor p \rfloor_{S<} \equiv \lfloor \lfloor p \rfloor_S \rfloor_{<}$

no-utp-lift *drop-state-pre*

abbreviation *lift-state-post* ($\lceil \cdot \rceil_{S>}$)

where $\lceil p \rceil_{S>} \equiv \lceil \lceil p \rceil_{>} \rceil_S$

no-utp-lift *lift-state-post*

abbreviation *drop-state-post* ($\lfloor \cdot \rfloor_{S>}$)

where $\lfloor p \rfloor_{S>} \equiv \lfloor \lfloor p \rfloor_S \rfloor_{>}$

no-utp-lift *lift-state-post*

lemma *st-unrest-state-pre* [*unrest*]: $\&\mathbf{v} \# s \implies \$st \# \lceil s \rceil_{S<}$
by (*rel-auto*)

lemma *st'-unrest-st-lift-pred* [*unrest*]:
 $\$st' \# \lceil a \rceil_{S<}$
by (*pred-auto*)

lemma *out-alpha-unrest-st-lift-pre* [*unrest*]:
 $out\alpha \# \lceil a \rceil_{S<}$
by (*rel-auto*)

lemma *R1-st'-unrest* [*unrest*]: $\$st' \# P \implies \$st' \# R1(P)$
by (*simp add: R1-def unrest*)

lemma *R2c-st'-unrest* [*unrest*]: $\$st' \# P \implies \$st' \# R2c(P)$
by (*simp add: R2c-def unrest*)

lemma *unrest-st-rea-rename* [*unrest*]:
 $\$st \# P \implies \$st \# P(\lfloor f \rfloor)_r$
 $\$st' \# P \implies \$st' \# P(\lfloor f \rfloor)_r$
by (*rel-blast*)⁺

lemma *st-lift-R1-true-right*: $\lceil b \rceil_{S<} ;; R1(true) = \lceil b \rceil_{S<}$
by (*rel-auto*)

lemma *R2c-lift-state-pre*: $R2c(\lceil b \rceil_{S<}) = \lceil b \rceil_{S<}$
by (*rel-auto*)

8.3 Reactive Program Operators

8.3.1 State Substitution

Lifting substitutions on the reactive state

definition *usubst-st-lift* ::

$'s \text{ usubst} \Rightarrow ((s', t :: \text{trace}, ' \alpha) \text{ rsp} \times (s', t, ' \beta) \text{ rsp}) \text{ usubst } (\lceil \cdot \rceil_{S\sigma})$ **where**
 $[upred-defs]: \lceil \sigma \rceil_{S\sigma} = \lceil \sigma \oplus_s st \rceil_s$

abbreviation *st-subst* :: $'s \text{ usubst} \Rightarrow (s', t :: \text{trace}, ' \alpha, ' \beta) \text{ rel-rsp} \Rightarrow (s', t, ' \alpha, ' \beta) \text{ rel-rsp}$ (**infixr** \dagger_S 80)
where

$\sigma \dagger_S P \equiv \lceil \sigma \rceil_{S\sigma} \dagger P$

translations

$\sigma \dagger_S P <= \lceil \sigma \oplus_s st \rceil_s \dagger P$
 $\sigma \dagger_S P <= \lceil \sigma \rceil_{S\sigma} \dagger P$

lemma *st-lift-lemma*:
 $\lceil \sigma \rceil_{S\sigma} = \sigma \oplus_s (fst_L ;_L (st \times_L st))$
by (*rel-auto*)

lemma *unrest-st-lift* [*unrest*]:

fixes $x :: 'a \Rightarrow ('s, 't::trace, 'α) \text{rsp} \times ('s, 't, 'α) \text{rsp}$

assumes $x \bowtie (\$st)_v$

shows $x \#_s \lceil \sigma \rceil_{S\sigma} \text{ (is } ?P)$

by (*simp add: st-lift-lemma*)

(*metis assms in-var-def in-var-prod-lens lens-comp-left-id st-vwb-lens unrest-subst-alpha-ext vwb-lens-wb*)

lemma *id-st-subst* [*usubst*]:

$\lceil id_s \rceil_{S\sigma} = id_s$

by (*pred-auto*)

lemma *st-subst-comp* [*usubst*]:

$\lceil \sigma \rceil_{S\sigma} \circ_s \lceil \varrho \rceil_{S\sigma} = \lceil \sigma \circ_s \varrho \rceil_{S\sigma}$

by (*rel-auto*)

definition *lift-cond-srea* ($\lceil \cdot \rceil_{S\leftarrow}$) **where**

[*upred-defs*]: $\lceil b \rceil_{S\leftarrow} = \lceil b \rceil_{S<}$

lemma *unrest-lift-cond-srea* [*unrest*]:

$x \# \lceil b \rceil_{S<} \Rightarrow x \# \lceil b \rceil_{S\leftarrow}$

by (*simp add: lift-cond-srea-def*)

lemma *st-subst-RR-closed* [*closure*]:

assumes $P \text{ is } RR$

shows $\lceil \sigma \rceil_{S\sigma} \dagger P \text{ is } RR$

proof –

have $RR(\lceil \sigma \rceil_{S\sigma} \dagger RR(P)) = \lceil \sigma \rceil_{S\sigma} \dagger RR(P)$

by (*rel-auto*)

thus *?thesis*

by (*metis Healthy-def assms*)

qed

lemma *subst-lift-cond-srea* [*usubst*]: $\sigma \dagger_S \lceil P \rceil_{S\leftarrow} = \lceil \sigma \dagger P \rceil_{S\leftarrow}$

by (*rel-auto*)

lemma *st-subst-rea-not* [*usubst*]: $\sigma \dagger_S (\neg_r P) = (\neg_r \sigma \dagger_S P)$

by (*rel-auto*)

lemma *st-subst-seq* [*usubst*]: $\sigma \dagger_S (P ;; Q) = \sigma \dagger_S P ;; Q$

by (*rel-auto*)

lemma *st-subst-RC-closed* [*closure*]:

assumes $P \text{ is } RC$

shows $\sigma \dagger_S P \text{ is } RC$

apply (*rule RC-intro, simp add: closure assms*)

apply (*simp add: st-subst-rea-not[THEN sym] st-subst-seq[THEN sym]*)

apply (*metis Healthy-if RC1-def RC-implies-RC1 assms*)

done

8.3.2 Assignment

definition *rea-assigns* :: $('s \text{ usubst}) \Rightarrow ('s, 't::trace, 'α) \text{ hrel-rsp } (\langle \cdot \rangle_r)$ **where**

[*upred-defs*]: $\langle \sigma \rangle_r = (\$tr' =_u \$tr \wedge \lceil \langle \sigma \rangle_a \rceil_S \wedge \$\Sigma_{S'} =_u \$\Sigma_S)$

syntax

-assign-rea :: svids \Rightarrow uexprs \Rightarrow logic ($'(-)' :=_r '(-)'$)
 -assign-rea :: svids \Rightarrow uexprs \Rightarrow logic (**infixr** $:=_r$ 62)

translations

-assign-rea xs vs => CONST rea-assigns (-mk-usubst (id_s) xs vs)
 -assign-rea x v <= CONST rea-assigns (CONST subst-upd (id_s) x v)
 -assign-rea x v <= -assign-rea (-spvar x) v
 x,y :=_r u,v <= CONST rea-assigns (CONST subst-upd (CONST subst-upd (id_s) (CONST pr-var x) u) (CONST pr-var y) v)

lemma rea-assigns-RR-closed [closure]:

$\langle \sigma \rangle_r$ is RR
apply (rel-auto) **using** minus-zero-eq **by** auto

lemma st-subst-assigns-rea [usubst]:

$\sigma \uparrow_S \langle \varrho \rangle_r = \langle \varrho \circ_s \sigma \rangle_r$
by (rel-auto)

lemma st-subst-rea-skip [usubst]:

$\sigma \uparrow_S II_r = \langle \sigma \rangle_r$
by (rel-auto)

lemma rea-assigns-comp [rpred]:

assumes P is RR
shows $\langle \sigma \rangle_r ;; P = \sigma \uparrow_S P$

proof –

have $\langle \sigma \rangle_r ;; (RR P) = \sigma \uparrow_S (RR P)$
by (rel-auto)

thus ?thesis

by (metis Healthy-def assms)

qed

lemma rea-assigns-rename [rpred]:

renamer f $\Longrightarrow \langle \sigma \rangle_r \llbracket f \rrbracket_r = \langle \sigma \rangle_r$
using minus-zero-eq **by** rel-auto

lemma st-subst-RR [closure]:

assumes P is RR
shows $(\sigma \uparrow_S P)$ is RR

proof –

have $(\sigma \uparrow_S RR(P))$ is RR
by (rel-auto)

thus ?thesis

by (simp add: Healthy-if assms)

qed

lemma rea-assigns-st-subst [usubst]:

$\lceil \sigma \oplus_s st \rceil_s \uparrow \langle \varrho \rangle_r = \langle \varrho \circ_s \sigma \rangle_r$
by (rel-auto)

8.3.3 Conditional

We guard the reactive conditional condition so that it can't be simplified by alphabet laws unless explicitly simplified.

abbreviation cond-srea ::

$(\text{'s}, \text{'t}::\text{trace}, \text{'}\alpha, \text{'}\beta) \text{ rel-rsp} \Rightarrow$
 $\text{'s upred} \Rightarrow$
 $(\text{'s}, \text{'t}, \text{'}\alpha, \text{'}\beta) \text{ rel-rsp} \Rightarrow$
 $(\text{'s}, \text{'t}, \text{'}\alpha, \text{'}\beta) \text{ rel-rsp where}$
 $\text{cond-srea } P \text{ b } Q \equiv P \triangleleft [b]_{S \leftarrow} \triangleright Q$

syntax

$\text{-cond-srea} :: \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} \Rightarrow \text{logic} ((\exists - \triangleleft - \triangleright_R / -) [52, 0, 53] \ 52)$

translations

$\text{-cond-srea } P \text{ b } Q == \text{CONST cond-srea } P \text{ b } Q$

lemma *st-cond-assigns* [*rpred*]:

$\langle \sigma \rangle_r \triangleleft b \triangleright_R \langle \varrho \rangle_r = \langle \sigma \triangleleft b \triangleright \varrho \rangle_r$
by (*rel-auto*)

lemma *cond-srea-RR-closed* [*closure*]:

assumes $P \text{ is } RR \ Q \text{ is } RR$
shows $P \triangleleft b \triangleright_R Q \text{ is } RR$

proof –

have $RR(RR(P) \triangleleft b \triangleright_R RR(Q)) = RR(P) \triangleleft b \triangleright_R RR(Q)$
by (*rel-auto*)
thus *?thesis*
by (*metis Healthy-def' assms(1) assms(2)*)

qed

lemma *cond-srea-RC1-closed*:

assumes $P \text{ is } RC1 \ Q \text{ is } RC1$
shows $P \triangleleft b \triangleright_R Q \text{ is } RC1$

proof –

have $RC1(RC1(P) \triangleleft b \triangleright_R RC1(Q)) = RC1(P) \triangleleft b \triangleright_R RC1(Q)$
using *dual-order.trans* **by** (*rel-blast*)
thus *?thesis*
by (*metis Healthy-def' assms*)

qed

lemma *cond-srea-RC-closed* [*closure*]:

assumes $P \text{ is } RC \ Q \text{ is } RC$
shows $P \triangleleft b \triangleright_R Q \text{ is } RC$
by (*rule RC-intro', simp-all add: closure cond-srea-RC1-closed RC-implies-RC1 assms*)

lemma *R4-cond* [*rpred*]: $R4(P \triangleleft b \triangleright_R Q) = (R4(P) \triangleleft b \triangleright_R R4(Q))$

by (*rel-auto*)

lemma *R5-cond* [*rpred*]: $R5(P \triangleleft b \triangleright_R Q) = (R5(P) \triangleleft b \triangleright_R R5(Q))$

by (*rel-auto*)

lemma *rea-rename-cond* [*rpred*]: $(P \triangleleft b \triangleright_R Q)(\llbracket f \rrbracket)_r = P(\llbracket f \rrbracket)_r \triangleleft b \triangleright_R Q(\llbracket f \rrbracket)_r$

by (*rel-auto*)

8.3.4 Assumptions

definition *rea-assume* :: $\text{'s upred} \Rightarrow (\text{'s}, \text{'t}::\text{trace}, \text{'}\alpha) \text{ hrel-rsp } ([\cdot]^\top_r) \text{ where}$
 $[\text{upred-defs}]: [b]^\top_r = (H_r \triangleleft b \triangleright_R \text{false})$

lemma *rea-assume-RR* [*closure*]: $[b]^\top_r \text{ is } RR$

by (simp add: rea-assume-def closure)

lemma *rea-assume-false* [rpred]: $[false]^\top_r = false$
by (rel-auto)

lemma *rea-assume-true* [rpred]: $[true]^\top_r = II_r$
by (rel-auto)

lemma *rea-assume-comp* [rpred]: $[b]^\top_r ;; [c]^\top_r = [b \wedge c]^\top_r$
by (rel-auto)

8.3.5 State Abstraction

We introduce state abstraction by creating some lens functors that allow us to lift a lens on the state-space to one on the whole stateful reactive alphabet.

definition *lmap_R* :: $('a \implies 'b) \Rightarrow ('t::trace, 'a) \text{ rp} \implies ('t, 'b) \text{ rp}$ **where**
[lens-defs]: *lmap_R* = *lmap*[*rp-vars*]

This construction lens is useful for conversion between a record and its product representation; it would be helpful if this could be automatically generated.

definition *rsp-make-lens* :: $('σ, 'τ::trace, 'α) \text{ rsp} \implies bool \times bool \times 'τ \times 'σ \times 'α$ **where**
[lens-defs]: *rsp-make-lens* = $\langle \text{lens-get} = \lambda (ok, wait, tr, st, more). \langle ok_v = ok, wait_v = wait, tr_v = tr, st_v = st, \dots = more \rangle$
 $, \text{lens-put} = (\lambda s v. (ok_v v, wait_v v, tr_v v, st_v v, more v)) \rangle$

lemma *rsp-make-lens-alt*: *rsp-make-lens* = *inv_L* (*ok* +_L *wait* +_L *tr* +_L *st* +_L *rsp-vars.more_L*)
by (auto simp add: lens-defs)

lemma *make-lens-bij* [simp]: *bij-lens* *rsp-make-lens*
by (unfold-locales, simp-all add: lens-defs prod.case-eq-if)

The following is an intuitive definition of the *st* functorial lens, which frames all the state space excluding *st*, to which another lens *l* is applied. We do this by splitting the state space into a product, including the application of *l* to *st*, and then invert the product creation lens to reconstruct the reactive state space.

definition *map-st-lens* ::
 $('σ \implies 'ψ) \Rightarrow$
 $((('σ, 'τ::trace, 'α) \text{ rsp} \implies ('ψ, 'τ::trace, 'α) \text{ rsp}) (\text{map}'\text{-st}_L))$ **where**
map-st-lens *l* = *inv_L* (*ok* +_L *wait* +_L *tr* +_L *st* +_L *rsp-vars.more_L*) ;_L
 $(ok +_L wait +_L tr +_L (l ;_L st) +_L \text{rsp-vars.more}_L)$

The above definition is intuitive, but unhelpful in proof automaton. Consequently, we the following optimised definition below.

lemma *map-st-lens-alt-def* [lens-defs]:
map-st-lens *l* = $\langle \text{lens-get} = \lambda s. \langle ok_v = ok_v s, wait_v = wait_v s, tr_v = tr_v s, st_v = get_l(st_v s), \dots = more s \rangle$
 $, \text{lens-put} = \lambda s v. \langle ok_v = ok_v v, wait_v = wait_v v, tr_v = tr_v v, st_v = put_l(st_v s)(st_v v), \dots = more v \rangle \rangle$
by (auto simp add: map-st-lens-def lens-defs fun-eq-iff)

lemma *map-st-vwb* [simp]: *vwb-lens* *X* \implies *vwb-lens* (*map-st_L* *X*)
by (simp add: map-st-lens-def rsp-make-lens-alt[THEN sym])

lemma *map-st-lens-indep-st* [*simp*]:
 $a \bowtie x \implies \text{map-st}_L a \bowtie x ;_L st$
by (*rule* *lens-indep.intro*, *simp-all add: lens-defs lens-indep-comm lens-indep.lens-put-irr2*)

lemma *map-st-lens-indep-st'* [*simp*]:
 $x \bowtie a \implies \text{map-st}_L a \bowtie x ;_L st$
by (*rule* *lens-indep.intro*, *simp-all add: lens-defs lens-indep-comm lens-indep.lens-put-irr2*)

syntax
 $\text{-map-st-lens} :: \text{logic} \Rightarrow \text{salpha} (\text{map}'\text{-st}_L[-])$

translations
 $\text{-map-st-lens } a \Rightarrow \text{CONST map-st-lens } a$

abbreviation $\text{abs-st}_L \equiv (\text{map-st}_L 0_L) \times_L (\text{map-st}_L 0_L)$

abbreviation $\text{abs-st } (\langle \cdot \rangle_S)$ **where**
 $\text{abs-st } P \equiv P \downarrow_e \text{abs-st}_L$

lemma *rea-impl-aext-st* [*alpha*]:
 $(P \Rightarrow_r Q) \oplus_r \text{map-st}_L[a] = (P \oplus_r \text{map-st}_L[a] \Rightarrow_r Q \oplus_r \text{map-st}_L[a])$
by (*rel-auto*)

lemma *rea-true-ext-st* [*alpha*]:
 $\text{true}_r \oplus_p \text{abs-st}_L = \text{true}_r$
by (*rel-auto*)

8.3.6 Reactive Frames and Extensions

definition *rea-frame* :: $(' \alpha \implies ' \beta) \Rightarrow (' \beta, 't::\text{trace}, 'r) \text{hrel-rsp} \Rightarrow (' \beta, 't, 'r) \text{hrel-rsp}$ **where**
 $[\text{upred-defs}]: \text{rea-frame } x P = \text{frame } (\text{ok} +_L \text{wait} +_L \text{tr} +_L (x ;_L st) +_L \Sigma_S) P$

definition *rea-frame-ext* :: $(' \alpha \implies ' \beta) \Rightarrow (' \alpha, 't::\text{trace}, 'r) \text{hrel-rsp} \Rightarrow (' \beta, 't, 'r) \text{hrel-rsp}$ **where**
 $[\text{upred-defs}]: \text{rea-frame-ext } a P = \text{rea-frame } a (P \oplus_r \text{map-st}_L[a])$

syntax
 $\text{-rea-frame} \quad :: \text{salpha} \Rightarrow \text{logic} \Rightarrow \text{logic } (-: [-]_r [99, 0] 100)$
 $\text{-rea-frame-ext} :: \text{salpha} \Rightarrow \text{logic} \Rightarrow \text{logic } (-: [-]_r^+ [99, 0] 100)$

translations
 $\text{-rea-frame } x P \Rightarrow \text{CONST rea-frame } x P$
 $\text{-rea-frame } (-\text{salphaset } (-\text{salphamk } x)) P \leq \text{CONST rea-frame } x P$
 $\text{-rea-frame-ext } x P \Rightarrow \text{CONST rea-frame-ext } x P$
 $\text{-rea-frame-ext } (-\text{salphaset } (-\text{salphamk } x)) P \leq \text{CONST rea-frame-ext } x P$

lemma *rea-frame-R1-closed* [*closure*]:
assumes $P \text{ is } R1$
shows $x:[P]_r \text{ is } R1$
proof –
have $R1(x:[R1 P]_r) = x:[R1 P]_r$
by (*rel-auto*)
thus *?thesis*
by (*metis Healthy-if Healthy-intro assms*)
qed

lemma *rea-frame-R2-closed* [*closure*]:

assumes P is $R2$
shows $x:[P]_r$ is $R2$
proof –
have $R2(x:[R2\ P]_r) = x:[R2\ P]_r$
by (*rel-auto*)
thus ?thesis
by (*metis Healthy-if Healthy-intro assms*)
qed

lemma *rea-frame-RR-closed* [*closure*]:
assumes P is RR
shows $x:[P]_r$ is RR
proof –
have $RR(x:[RR\ P]_r) = x:[RR\ P]_r$
by (*rel-auto*)
thus ?thesis
by (*metis Healthy-if Healthy-intro assms*)
qed

lemma *rea-aext-R1* [*closure*]:
assumes P is $R1$
shows *rel-aext* P (*map-st_L* x) is $R1$
proof –
have *rel-aext* ($R1\ P$) (*map-st_L* x) is $R1$
by (*rel-auto*)
thus ?thesis
by (*simp add: Healthy-if assms*)
qed

lemma *rea-aext-R2* [*closure*]:
assumes P is $R2$
shows *rel-aext* P (*map-st_L* x) is $R2$
proof –
have *rel-aext* ($R2\ P$) (*map-st_L* x) is $R2$
by (*rel-auto*)
thus ?thesis
by (*simp add: Healthy-if assms*)
qed

lemma *rea-aext-RR* [*closure*]:
assumes P is RR
shows *rel-aext* P (*map-st_L* x) is RR
proof –
have *rel-aext* ($RR\ P$) (*map-st_L* x) is RR
by (*rel-auto*)
thus ?thesis
by (*simp add: Healthy-if assms*)
qed

lemma *true-rea-map-st* [*alpha*]: ($R1\ true \oplus_r\ map-st_L[a]$) = $R1\ true$
by (*rel-auto*)

lemma *rea-frame-ext-R1-closed* [*closure*]:
 P is $R1 \implies x:[P]_r^+$ is $R1$
by (*simp add: rea-frame-ext-def closure*)

lemma *rea-frame-ext-R2-closed* [closure]:

$P \text{ is } R2 \implies x:[P]_r^+ \text{ is } R2$

by (simp add: rea-frame-ext-def closure)

lemma *rea-frame-ext-RR-closed* [closure]:

$P \text{ is } RR \implies x:[P]_r^+ \text{ is } RR$

by (simp add: rea-frame-ext-def closure)

lemma *rel-aext-st-Instant-closed* [closure]:

$P \text{ is } Instant \implies \text{rel-aext } P \text{ (map-st}_L x) \text{ is } Instant$

by (rel-auto)

lemma *rea-frame-ext-false* [frame]:

$x:[false]_r^+ = false$

by (rel-auto)

lemma *rea-frame-ext-skip* [frame]:

$\text{vwb-lens } x \implies x:[II]_r^+ = II_r$

by (rel-auto)

lemma *rea-frame-ext-assigns* [frame]:

$\text{vwb-lens } x \implies x:[\langle \sigma \rangle_r]_r^+ = \langle \sigma \oplus_s x \rangle_r$

by (rel-auto)

lemma *rea-frame-ext-cond* [frame]:

$x:[P \triangleleft b \triangleright_R Q]_r^+ = x:[P]_r^+ \triangleleft (b \oplus_p x) \triangleright_R x:[Q]_r^+$

by (rel-auto)

lemma *rea-frame-ext-seq* [frame]:

$\text{vwb-lens } x \implies x:[P ;; Q]_r^+ = x:[P]_r^+ ;; x:[Q]_r^+$

apply (simp add: rea-frame-ext-def rea-frame-def alpha frame)

apply (subst frame-seq)

apply (simp-all add: plus-vwb-lens closure)

apply (rel-auto)+

done

lemma *rea-frame-ext-subst-indep* [usubst]:

assumes $x \bowtie y \Sigma \# v P \text{ is } RR$

shows $\sigma(y \mapsto_s v) \uparrow_S x:[P]_r^+ = (\sigma \uparrow_S x:[P]_r^+) ;; y :=_r v$

proof –

from *assms*(1–2) have $\sigma(y \mapsto_s v) \uparrow_S x:[RR P]_r^+ = (\sigma \uparrow_S x:[RR P]_r^+) ;; y :=_r v$

by (rel-auto, (metis (no-types, lifting) lens-indep.lens-put-comm lens-indep-get)+)

thus ?thesis

by (simp add: Healthy-if *assms*)

qed

lemma *rea-frame-ext-subst-within* [usubst]:

assumes $\text{vwb-lens } x \text{ vwb-lens } y \Sigma \# v P \text{ is } RR$

shows $\sigma(x:y \mapsto_s v) \uparrow_S x:[P]_r^+ = (\sigma \uparrow_S x:[y :=_r (v \downarrow_e x) ;; P]_r^+)$

proof –

from *assms*(1,3) have $\sigma(x:y \mapsto_s v) \uparrow_S x:[RR P]_r^+ = (\sigma \uparrow_S x:[y :=_r (v \downarrow_e x) ;; RR(P)]_r^+)$

by (rel-auto, metis+)

thus ?thesis

by (simp add: *assms* Healthy-if)

qed

lemma *rea-frame-ext-UINF-ind* [frame]:
 $a: [\bigwedge x \cdot P x]_r^+ = (\bigwedge x \cdot a:[P x]_r^+)$
by (*rel-auto*)

lemma *rea-frame-ext-UINF-mem* [frame]:
 $a: [\bigwedge x \in A \cdot P x]_r^+ = (\bigwedge x \in A \cdot a:[P x]_r^+)$
by (*rel-auto*)

8.4 Stateful Reactive specifications

definition *rea-st-rel* :: '*s hrel* \Rightarrow ('*s*, '*t*::*trace*, ' α , ' β) *rel-rsp* ($[-]_S$) **where**
[upred-defs]: *rea-st-rel* *b* = ($\lceil b \rceil_S \wedge \$tr' =_u \$tr$)

definition *rea-st-rel'* :: '*s hrel* \Rightarrow ('*s*, '*t*::*trace*, ' α , ' β) *rel-rsp* ($[-]_{S'}$) **where**
[upred-defs]: *rea-st-rel'* *b* = *R1*($\lceil b \rceil_S$)

definition *rea-st-cond* :: '*s upred* \Rightarrow ('*s*, '*t*::*trace*, ' α , ' β) *rel-rsp* ($[-]_{S<}$) **where**
[upred-defs]: *rea-st-cond* *b* = *R1*($\lceil b \rceil_{S<}$)

definition *rea-st-post* :: '*s upred* \Rightarrow ('*s*, '*t*::*trace*, ' α , ' β) *rel-rsp* ($[-]_{S>}$) **where**
[upred-defs]: *rea-st-post* *b* = *R1*($\lceil b \rceil_{S>}$)

lemma *lift-state-pre-unrest* [*unrest*]: $x \bowtie (\$st)_v \Longrightarrow x \# \lceil P \rceil_{S<}$
by (*rel-simp*, *simp add: lens-indep-def*)

lemma *rea-st-rel-unrest* [*unrest*]:
 $\llbracket x \bowtie (\$tr)_v; x \bowtie (\$tr')_v; x \bowtie (\$st)_v; x \bowtie (\$st')_v \rrbracket \Longrightarrow x \# \lceil P \rceil_{S<}$
by (*simp add: add: rea-st-cond-def R1-def unrest lens-indep-sym*)

lemma *rea-st-cond-unrest* [*unrest*]:
 $\llbracket x \bowtie (\$tr)_v; x \bowtie (\$tr')_v; x \bowtie (\$st)_v \rrbracket \Longrightarrow x \# \lceil P \rceil_{S<}$
by (*simp add: add: rea-st-cond-def R1-def unrest lens-indep-sym*)

lemma *subst-st-cond* [*usubst*]: $\lceil \sigma \rceil_{S\sigma} \dagger \lceil P \rceil_{S<} = \lceil \sigma \dagger P \rceil_{S<}$
by (*rel-auto*)

lemma *rea-st-cond-R1* [*closure*]: $\lceil b \rceil_{S<} \text{ is } R1$
by (*rel-auto*)

lemma *rea-st-cond-R2c* [*closure*]: $\lceil b \rceil_{S<} \text{ is } R2c$
by (*rel-auto*)

lemma *rea-st-rel-RR* [*closure*]: $\lceil P \rceil_S \text{ is } RR$
using *minus-zero-eq* **by** (*rel-auto*)

lemma *rea-st-rel'-RR* [*closure*]: $\lceil P \rceil_{S'} \text{ is } RR$
by (*rel-auto*)

lemma *rea-st-post-RR* [*closure*]: $\lceil b \rceil_{S>} \text{ is } RR$
by (*rel-auto*)

lemma *st-subst-rel* [*usubst*]:
 $\sigma \dagger_S \lceil P \rceil_S = \lceil \lceil \sigma \rceil_s \dagger P \rceil_S$
by (*rel-auto*)

lemma *st-rel-cond* [*rpred*]:
 $[P \triangleleft b \triangleright_r Q]_S = [P]_S \triangleleft b \triangleright_R [Q]_S$
by (*rel-auto*)

lemma *st-rel-false* [*rpred*]: $[false]_S = false$
by (*rel-auto*)

lemma *st-rel-skip* [*rpred*]:
 $[II]_S = (II_r :: ('s, 't::trace) rdes)$
by (*rel-auto*)

lemma *st-rel-seq* [*rpred*]:
 $[P ;; Q]_S = [P]_S ;; [Q]_S$
by (*rel-auto*)

lemma *st-rel-conj* [*rpred*]:
 $([P]_S \wedge [Q]_S) = [P \wedge Q]_S$
by (*rel-auto*)

lemma *st-cond-disj* [*rpred*]:
 $([P]_{S<} \vee [Q]_{S<}) = [P \vee Q]_{S<}$
by (*rel-auto*)

lemma *rea-st-cond-RR* [*closure*]: $[b]_{S<}$ is *RR*
by (*rule RR-intro, simp-all add: unrest closure*)

lemma *rea-st-cond-RC* [*closure*]: $[b]_{S<}$ is *RC*
by (*rule RC-intro, simp add: closure, rel-auto*)

lemma *rea-st-cond-true* [*rpred*]: $[true]_{S<} = true_r$
by (*rel-auto*)

lemma *rea-st-cond-false* [*rpred*]: $[false]_{S<} = false$
by (*rel-auto*)

lemma *st-cond-not* [*rpred*]: $(\neg_r [P]_{S<}) = [\neg P]_{S<}$
by (*rel-auto*)

lemma *st-cond-conj* [*rpred*]: $([P]_{S<} \wedge [Q]_{S<}) = [P \wedge Q]_{S<}$
by (*rel-auto*)

lemma *st-rel-assigns* [*rpred*]:
 $[(\sigma)_a]_S = (\langle \sigma \rangle_r :: ('a, 't::trace) rdes)$
by (*rel-auto*)

lemma *cond-st-distr*: $(P \triangleleft b \triangleright_R Q) ;; R = (P ;; R \triangleleft b \triangleright_R Q ;; R)$
by (*rel-auto*)

lemma *cond-st-miracle* [*rpred*]: $P \text{ is } R1 \implies P \triangleleft b \triangleright_R false = ([b]_{S<} \wedge P)$
by (*rel-blast*)

lemma *cond-st-true* [*rpred*]: $P \triangleleft true \triangleright_R Q = P$
by (*rel-blast*)

lemma *cond-st-false* [rpred]: $P \triangleleft \text{false} \triangleright_R Q = Q$
by (*rel-blast*)

lemma *st-cond-true-or* [rpred]: $P \text{ is } R1 \implies (R1 \text{ true} \triangleleft b \triangleright_R P) = ([b]_{S<} \vee P)$
by (*rel-blast*)

lemma *st-cond-left-impl-RC-closed* [closure]:
 $P \text{ is } RC \implies ([b]_{S<} \Rightarrow_r P) \text{ is } RC$
by (*simp add: rea-impl-def rpred closure*)

end

9 Reactive Weakest Preconditions

theory *utp-rea-wp*
imports *utp-rea-prog*
begin

Here, we create a weakest precondition calculus for reactive relations, using the recast boolean algebra and relational operators. Please see our journal paper [3] for more information.

definition *wp-rea* ::
 $(t::\text{trace}, 'a) \text{ hrel-rp} \Rightarrow$
 $(t, 'a) \text{ hrel-rp} \Rightarrow$
 $(t, 'a) \text{ hrel-rp} \text{ (infix wp}_r \text{ 60)}$
where [*upred-defs*]: $P \text{ wp}_r Q = (\neg_r P ;; (\neg_r Q))$

lemma *in-var-unrest-wp-rea* [*unrest*]: $\llbracket \$x \# P; tr \bowtie x \rrbracket \implies \$x \# (P \text{ wp}_r Q)$
by (*simp add: wp-rea-def unrest R1-def rea-not-def*)

lemma *out-var-unrest-wp-rea* [*unrest*]: $\llbracket \$x' \# Q; tr \bowtie x \rrbracket \implies \$x' \# (P \text{ wp}_r Q)$
by (*simp add: wp-rea-def unrest R1-def rea-not-def*)

lemma *wp-rea-R1* [closure]: $P \text{ wp}_r Q \text{ is } R1$
by (*rel-auto*)

lemma *wp-rea-RR-closed* [closure]: $\llbracket P \text{ is } RR; Q \text{ is } RR \rrbracket \implies P \text{ wp}_r Q \text{ is } RR$
by (*simp add: wp-rea-def closure*)

lemma *wp-rea-impl-lemma*:
 $((P \text{ wp}_r Q) \Rightarrow_r (R1(P) ;; R1(Q \Rightarrow_r R))) = ((P \text{ wp}_r Q) \Rightarrow_r (R1(P) ;; R1(R)))$
by (*rel-auto, blast*)

lemma *wpR-impl-post-spec*:
assumes $P \text{ is } RR$
shows $(P \text{ wp}_r Q_1 \Rightarrow_r (P ;; (Q_1 \Rightarrow_r Q_2))) = (P ;; (Q_1 \Rightarrow_r Q_2))$
by (*simp add: R1-seqr-closure RR-implies-R1 assms rea-impl-def rea-not-R1 rea-not-not seqr-or-distr wp-rea-def*)

lemma *wpR-R1-right* [*wp*]:
 $P \text{ wp}_r R1(Q) = P \text{ wp}_r Q$
by (*rel-auto*)

lemma *wp-rea-true* [*wp*]: $P \text{ wp}_r \text{true} = \text{true}_r$
by (*rel-auto*)

lemma *wp-rea-conj* [wp]: $P \text{ wp}_r (Q \wedge R) = (P \text{ wp}_r Q \wedge P \text{ wp}_r R)$
by (*simp add: wp-rea-def seqr-or-distr*)

lemma *wp-rea-USUP-mem* [wp]:
 $A \neq \{\} \implies P \text{ wp}_r (\bigsqcup i \in A \cdot Q(i)) = (\bigsqcup i \in A \cdot P \text{ wp}_r Q(i))$
by (*simp add: wp-rea-def seq-UINF-distl*)

lemma *wp-rea-Inf-pre* [wp]:
 $P \text{ wp}_r (\bigsqcup i \in \{0..n::\text{nat}\}. Q(i)) = (\bigsqcup i \in \{0..n\}. P \text{ wp}_r Q(i))$
by (*simp add: wp-rea-def seq-SUP-distl*)

lemma *wp-rea-div* [wp]:
 $(\neg_r P ;; \text{true}_r) = \text{true}_r \implies \text{true}_r \text{ wp}_r P = \text{false}$
by (*simp add: wp-rea-def rpred, rel-blast*)

lemma *wp-rea-st-cond-div* [wp]:
 $P \neq \text{true} \implies \text{true}_r \text{ wp}_r [P]_{S<} = \text{false}$
by (*rel-auto*)

lemma *wp-rea-cond* [wp]:
 $\text{out}_\alpha \# b \implies (P \triangleleft b \triangleright Q) \text{ wp}_r R = (P \text{ wp}_r R) \triangleleft b \triangleright (Q \text{ wp}_r R)$
by (*simp add: wp-rea-def cond-seq-left-distr, rel-auto*)

lemma *wp-rea-RC-false* [wp]:
 $P \text{ is } RC \implies (\neg_r P) \text{ wp}_r \text{false} = P$
by (*metis Healthy-if RC1-def RC-implies-RC1 rea-not-false wp-rea-def*)

lemma *wp-rea-seq* [wp]:
assumes $Q \text{ is } R1$
shows $(P ;; Q) \text{ wp}_r R = P \text{ wp}_r (Q \text{ wp}_r R)$ (**is** ?lhs = ?rhs)
proof –
have ?rhs = $R1 (\neg P ;; R1 (Q ;; R1 (\neg R)))$
by (*simp add: wp-rea-def rea-not-def R1-negate-R1 assms*)
also have ... = $R1 (\neg P ;; (Q ;; R1 (\neg R)))$
by (*metis Healthy-if R1-seqr assms*)
also have ... = $R1 (\neg (P ;; Q) ;; R1 (\neg R))$
by (*simp add: seqr-assoc*)
finally show ?thesis
by (*simp add: wp-rea-def rea-not-def*)
qed

lemma *wp-rea-skip* [wp]:
assumes $Q \text{ is } R1$
shows $II \text{ wp}_r Q = Q$
by (*simp add: wp-rea-def rpred assms Healthy-if*)

lemma *wp-rea-rea-skip* [wp]:
assumes $Q \text{ is } RR$
shows $II_r \text{ wp}_r Q = Q$
by (*simp add: wp-rea-def rpred closure assms Healthy-if*)

lemma *power-wp-rea-RR-closed* [closure]:
 $\llbracket R \text{ is } RR; P \text{ is } RR \rrbracket \implies R \hat{=} i \text{ wp}_r P \text{ is } RR$
by (*induct i, simp-all add: wp closure*)

lemma *wp-rea-rea-assigns* [wp]:

assumes P is RR

shows $\langle \sigma \rangle_r \text{wp}_r P = \lceil \sigma \rceil_{S\sigma} \dagger P$

proof –

have $\langle \sigma \rangle_r \text{wp}_r (RR P) = \lceil \sigma \rceil_{S\sigma} \dagger (RR P)$

by (*rel-auto*)

thus *?thesis*

by (*metis Healthy-def assms*)

qed

lemma *wp-rea-miracle* [wp]: $\text{false wp}_r Q = \text{true}_r$

by (*simp add: wp-rea-def*)

lemma *wp-rea-disj* [wp]: $(P \vee Q) \text{wp}_r R = (P \text{wp}_r R \wedge Q \text{wp}_r R)$

by (*rel-blast*)

lemma *wp-rea-UINF* [wp]:

assumes $A \neq \{\}$

shows $(\bigcap x \in A \cdot P(x)) \text{wp}_r Q = (\bigcup x \in A \cdot P(x) \text{wp}_r Q)$

by (*simp add: wp-rea-def rea-not-def seq-UINF-distr not-UINF R1-UINF assms*)

lemma *wp-rea-choice* [wp]:

$(P \sqcap Q) \text{wp}_r R = (P \text{wp}_r R \wedge Q \text{wp}_r R)$

by (*rel-blast*)

lemma *wp-rea-UINF-ind* [wp]:

$(\bigcap i \cdot P(i)) \text{wp}_r Q = (\bigcup i \cdot P(i) \text{wp}_r Q)$

by (*simp add: wp-rea-def rea-not-def seq-UINF-distr' not-UINF-ind R1-UINF-ind*)

lemma *rea-assume-wp* [wp]:

assumes P is RC

shows $([b]^\top_r \text{wp}_r P) = ([b]_{S<} \Rightarrow_r P)$

proof –

have $([b]^\top_r \text{wp}_r RC P) = ([b]_{S<} \Rightarrow_r RC P)$

by (*rel-auto*)

thus *?thesis*

by (*simp add: Healthy-if assms*)

qed

lemma *rea-star-wp* [wp]:

assumes P is RR Q is RR

shows $P^{*r} \text{wp}_r Q = (\bigcup i \cdot P \wedge i \text{wp}_r Q)$

proof –

have $P^{*r} \text{wp}_r Q = (Q \wedge P^+ \text{wp}_r Q)$

by (*simp add: assms rrel-theory.Star-alt-def wp-rea-choice wp-rea-rea-skip*)

also have $\dots = (II \text{wp}_r Q \wedge (\bigcup i \cdot P \wedge \text{Suc } i \text{wp}_r Q))$

by (*simp add: uplus-power-def wp closure assms*)

also have $\dots = (\bigcup i \cdot P \wedge i \text{wp}_r Q)$

proof –

have $P^* \text{wp}_r Q = P^{*r} \text{wp}_r Q$

by (*metis (no-types) RA1 assms(2) rea-no-RR rea-skip-unit(2) rrel-theory.Star-def wp-rea-def*)

then show *?thesis*

by (*simp add: calculation ustar-def wp-rea-UINF-ind*)

qed

finally show *?thesis* .

qed

lemma *wp-rea-R2-closed* [closure]:

$\llbracket P \text{ is } R2; Q \text{ is } R2 \rrbracket \implies P \text{ wp}_r Q \text{ is } R2$
by (*simp add: wp-rea-def closure*)

lemma *wp-rea-false-RC1'*: $P \text{ is } R2 \implies RC1(P \text{ wp}_r \text{ false}) = P \text{ wp}_r \text{ false}$

by (*simp add: wp-rea-def RC1-def closure rpred segr-assoc*)

lemma *wp-rea-false-RC1*: $P \text{ is } R2 \implies P \text{ wp}_r \text{ false is } RC1$

by (*simp add :Healthy-def wp-rea-false-RC1'*)

lemma *wp-rea-false-RR*:

$\llbracket \$ok \# P; \$wait \# P; P \text{ is } R2 \rrbracket \implies P \text{ wp}_r \text{ false is } RR$
by (*rule RR-R2-intro, simp-all add: unrest closure*)

lemma *wp-rea-false-RC*:

$\llbracket \$ok \# P; \$wait \# P; P \text{ is } R2 \rrbracket \implies P \text{ wp}_r \text{ false is } RC$
by (*rule RC-intro', simp-all add: wp-rea-false-RC1 wp-rea-false-RR*)

lemma *wp-rea-RC1*: $\llbracket P \text{ is } RR; Q \text{ is } RC \rrbracket \implies P \text{ wp}_r Q \text{ is } RC1$

by (*rule Healthy-intro, simp add: wp-rea-def RC1-def rpred closure segr-assoc RC1-prop RC-implies-RC1*)

lemma *wp-rea-RC* [closure]: $\llbracket P \text{ is } RR; Q \text{ is } RC \rrbracket \implies P \text{ wp}_r Q \text{ is } RC$

by (*rule RC-intro', simp-all add: wp-rea-RC1 closure*)

lemma *wpR-power-RC-closed* [closure]:

assumes $P \text{ is } RR \ Q \text{ is } RC$

shows $P \wedge i \text{ wp}_r Q \text{ is } RC$

by (*metis RC-implies-RR RR-implies-R1 assms power.power-eq-if power-Suc-RR-closed wp-rea-RC wp-rea-skip*)

end

10 Reactive Hoare Logic

theory *utp-rea-hoare*

imports *utp-rea-prog*

begin

definition *hoare-rp* :: $'\alpha \text{ upred} \Rightarrow (' \alpha, \text{ real pos}) \text{ rdes} \Rightarrow ' \alpha \text{ upred} \Rightarrow \text{bool} (\llbracket - \rrbracket / - / \llbracket - \rrbracket_r)$ **where**
[upred-defs]: $\text{hoare-rp } p \ Q \ r = ((\llbracket p \rrbracket_{S<} \Rightarrow \llbracket r \rrbracket_{S>}) \sqsubseteq Q)$

lemma *hoare-rp-conseq*:

$\llbracket 'p \Rightarrow p'; 'q' \Rightarrow q'; \llbracket p' \rrbracket S \llbracket q' \rrbracket_r \rrbracket \implies \llbracket p \rrbracket S \llbracket q \rrbracket_r$
by (*rel-auto*)

lemma *hoare-rp-weaken*:

$\llbracket 'p \Rightarrow p'; \llbracket p' \rrbracket S \llbracket q \rrbracket_r \rrbracket \implies \llbracket p \rrbracket S \llbracket q \rrbracket_r$
by (*rel-auto*)

lemma *hoare-rp-strengthen*:

$\llbracket 'q' \Rightarrow q'; \llbracket p \rrbracket S \llbracket q' \rrbracket_r \rrbracket \implies \llbracket p \rrbracket S \llbracket q \rrbracket_r$
by (*rel-auto*)

lemma *false-pre-hoare-rp* [*hoare-safe*]: $\llbracket \text{false} \rrbracket P \llbracket q \rrbracket_r$
by (*rel-auto*)

lemma *true-post-hoare-rp* [*hoare-safe*]: $\llbracket p \rrbracket Q \llbracket \text{true} \rrbracket_r$
by (*rel-auto*)

lemma *miracle-hoare-rp* [*hoare-safe*]: $\llbracket p \rrbracket \text{false} \llbracket q \rrbracket_r$
by (*rel-auto*)

lemma *assigns-hoare-rp* [*hoare-safe*]: $'p \Rightarrow \sigma \uparrow q' \Longrightarrow \llbracket p \rrbracket \langle \sigma \rangle_r \llbracket q \rrbracket_r$
by *rel-auto*

lemma *skip-hoare-rp* [*hoare-safe*]: $\llbracket p \rrbracket II_r \llbracket p \rrbracket_r$
by *rel-auto*

lemma *seq-hoare-rp*: $\llbracket \llbracket p \rrbracket Q_1 \llbracket s \rrbracket_r ; \llbracket s \rrbracket Q_2 \llbracket r \rrbracket_r \rrbracket \Longrightarrow \llbracket p \rrbracket Q_1 ;; Q_2 \llbracket r \rrbracket_r$
by (*rel-auto*)

lemma *seq-est-hoare-rp* [*hoare-safe*]:
 $\llbracket \llbracket \text{true} \rrbracket Q_1 \llbracket p \rrbracket_r ; \llbracket p \rrbracket Q_2 \llbracket p \rrbracket_r \rrbracket \Longrightarrow \llbracket \text{true} \rrbracket Q_1 ;; Q_2 \llbracket p \rrbracket_r$
by (*rel-auto*)

lemma *seq-inv-hoare-rp* [*hoare-safe*]:
 $\llbracket \llbracket p \rrbracket Q_1 \llbracket p \rrbracket_r ; \llbracket p \rrbracket Q_2 \llbracket p \rrbracket_r \rrbracket \Longrightarrow \llbracket p \rrbracket Q_1 ;; Q_2 \llbracket p \rrbracket_r$
by (*rel-auto*)

lemma *cond-hoare-rp* [*hoare-safe*]:
 $\llbracket \llbracket b \wedge p \rrbracket P \llbracket r \rrbracket_r ; \llbracket \neg b \wedge p \rrbracket Q \llbracket r \rrbracket_r \rrbracket \Longrightarrow \llbracket p \rrbracket P \triangleleft b \triangleright_R Q \llbracket r \rrbracket_r$
by (*rel-auto*)

lemma *repeat-hoare-rp* [*hoare-safe*]:
 $\llbracket p \rrbracket Q \llbracket p \rrbracket_r \Longrightarrow \llbracket p \rrbracket Q \hat{\ }^n \llbracket p \rrbracket_r$
by (*induct n, rel-auto+*)

lemma *UINF-ind-hoare-rp* [*hoare-safe*]:
 $\llbracket \bigwedge i. \llbracket p \rrbracket Q(i) \llbracket r \rrbracket_r \rrbracket \Longrightarrow \llbracket p \rrbracket \bigcap i \cdot Q(i) \llbracket r \rrbracket_r$
by (*rel-auto*)

lemma *star-hoare-rp* [*hoare-safe*]:
 $\llbracket p \rrbracket Q \llbracket p \rrbracket_r \Longrightarrow \llbracket p \rrbracket Q^* \llbracket p \rrbracket_r$
by (*simp add: ustar-def hoare-safe*)

lemma *conj-hoare-rp* [*hoare-safe*]:
 $\llbracket \llbracket p_1 \rrbracket Q_1 \llbracket r_1 \rrbracket_r ; \llbracket p_2 \rrbracket Q_2 \llbracket r_2 \rrbracket_r \rrbracket \Longrightarrow \llbracket p_1 \wedge p_2 \rrbracket Q_1 \wedge Q_2 \llbracket r_1 \wedge r_2 \rrbracket_r$
by (*rel-auto*)

lemma *iter-hoare-rp* [*hoare-safe*]:
 $\llbracket I \rrbracket P \llbracket I \rrbracket_r \Longrightarrow \llbracket I \rrbracket P^{\star r} \llbracket I \rrbracket_r$
by (*metis rrel-theory.utp-star-def seq-hoare-rp skip-hoare-rp star-hoare-rp*)

end

11 Meta-theory for Generalised Reactive Processes

theory *utp-reactive*

```

imports
  utp-rea-core
  utp-rea-event
  utp-rea-healths
  utp-rea-parallel
  utp-rea-rel
  utp-rea-cond
  utp-rea-prog
  utp-rea-wp
  utp-rea-hoare
begin end

```

References

- [1] A. Butterfield, P. Gancarski, and J. Woodcock. State visibility and communication in unifying theories of programming. *Theoretical Aspects of Software Engineering*, 0:47–54, 2009.
- [2] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
- [3] S. Foster, A. Cavalcanti, S. Canham, J. Woodcock, and F. Zeyda. Unifying theories of reactive design contracts. *Submitted to Theoretical Computer Science*, Dec 2017. Preprint: <https://arxiv.org/abs/1712.10233>.
- [4] S. Foster, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying theories of time with generalised reactive processes. *Information Processing Letters*, 135:47–52, July 2018. Preprint: <https://arxiv.org/abs/1712.10213>.
- [5] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.