Generalised Reactive Processes

Simon Foster Samuel Canham

April 4, 2018

Contents

1 Reactive Processes Core Definitions

theory *utp-rea-core*
imports
  UTP−Toolkit.Trace-Algebra
  UTP.utp-concurrency
  UTP−Designs.utp-designs
begin recall-syntax

1.1 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 [3], and thus provides the theory of "generalised reactive processes" [3]. The reactive process alphabet also extends the design alphabet, and thus includes the *ok* variable. For more information on these, see the UTP book [4], or the associated tutorial [2].

alphabet $'t$::*trace rp-vars* = *des-vars* +
  wait :: *bool*
  tr :: $'t$

type-synonym ($'t$, $'\alpha$) *rp* = ($'t$, $'\alpha$) *rp-vars-scheme des*

type-synonym ($'t$,$'\alpha$,$'\beta$) *rel-rp* = (($'t$,$'\alpha$) *rp*, ($'t$,$'\beta$) *rp*) *urel*
type-synonym ($'t$,$'\alpha$) *hrel-rp* = ($'t$,$'\alpha$) *rp hrel*

translations
  (*type*) ($'t$,$'\alpha$) *rp* <= (*type*) ($'t$, $'\alpha$) *rp-vars-scheme des*
  (*type*) ($'t$,$'\alpha$) *rp* <= (*type*) ($'t$, $'\alpha$) *rp-vars-ext des*
  (*type*) ($'t$,$'\alpha$,$'\beta$) *rel-rp* <= (*type*) (($'t$,$'\alpha$) *rp*, ($'\gamma$,$'\beta$) *rp*) *urel*
  (*type*) ($'t$, $'\alpha$) *hrel-rp* <= (*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-rp* and ($'t$, $'\alpha$) *hrel-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-vars-child-lens*$_a$ (Σ_r)
notation *rp-vars-child-lens* (Σ_R)

syntax
  -svid-rea-alpha :: *svid* (Σ_R)

translations

-svid-rea-alpha => *CONST rp-vars-child-lens*

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 *rp-vars.splits* [*alpha-splits*]
declare *rp-vars.defs* [*lens-defs*]
declare *zero-list-def* [*upred-defs*]
declare *plus-list-def* [*upred-defs*]
declare *prefixE* [*elim*]

The two locale interpretations below are a technicality to improve automatic proof support via the predicate and relational tactics. This is to enable the (re-)interpretation of state spaces to remove any occurrences of lens types after the proof tactics *pred-simp* and *rel-simp*, or any of their derivatives have been applied. Eventually, it would be desirable to automate both interpretations as part of a custom outer command for defining alphabets.

interpretation *rp-vars*:
  lens-interp $\lambda(ok,\ r).\ (ok,\ wait_v\ r,\ tr_v\ r,\ more\ r)$
  apply (*unfold-locales*)
  apply (*rule injI*)
  apply (*clarsimp*)
  done

interpretation *rp-vars-rel*: *lens-interp* $\lambda(ok,\ ok',\ r,\ r').$
  $(ok,\ ok',\ wait_v\ r,\ wait_v\ r',\ tr_v\ r,\ tr_v\ r',\ more\ r,\ more\ r')$
  apply (*unfold-locales*)
  apply (*rule injI*)
  apply (*clarsimp*)
  done

The following syntactic orders exist to help to order lens names when, for example, performing substitution, to achieve normalisation of terms.

lemma *rea-var-ords* [*usubst*]:
  $\$tr \prec_v \$tr\acute{}\ \ \$wait \prec_v \$wait\acute{}$
  $\$ok \prec_v \$tr\ \ \$ok\acute{} \prec_v \$tr\acute{}\ \ \$ok \prec_v \$tr\acute{}\ \ \$ok\acute{} \prec_v \$tr$
  $\$ok \prec_v \$wait\ \ \$ok\acute{} \prec_v \$wait\acute{}\ \ \$ok \prec_v \$wait\acute{}\ \ \$ok\acute{} \prec_v \$wait$
  $\$tr \prec_v \$wait\ \ \$tr\acute{} \prec_v \$wait\acute{}\ \ \$tr \prec_v \$wait\acute{}\ \ \$tr\acute{} \prec_v \$wait$
  by (*simp-all add*: *var-name-ord-def*)

abbreviation *wait-f*::('*t*::*trace*, 'α, 'β) *rel-rp* ⇒ ('*t*, 'α, 'β) *rel-rp*
where *wait-f R* ≡ $R⟦false/\$wait⟧$

abbreviation *wait-t*::('*t*::*trace*, 'α, 'β) *rel-rp* ⇒ ('*t*, 'α, 'β) *rel-rp*
where *wait-t R* ≡ $R⟦true/\$wait⟧$

syntax
  -wait-f :: *logic* ⇒ *logic* (-_f [*1000*] *1000*)
  -wait-t :: *logic* ⇒ *logic* (-_t [*1000*] *1000*)

translations
  $P\ _f$ ⇌ *CONST usubst* (*CONST subst-upd CONST id* (*CONST ivar CONST wait*) *false*) *P*
  $P\ _t$ ⇌ *CONST usubst* (*CONST subst-upd CONST id* (*CONST ivar CONST wait*) *true*) *P*

abbreviation *lift-rea* :: - ⇒ - ($\lceil\text{-}\rceil_R$) **where**
$\lceil P\rceil_R \equiv P \oplus_p (\Sigma_R \times_L \Sigma_R)$

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

abbreviation *rea-pre-lift* :: - $\Rightarrow$ - $(\lceil\text{-}\rceil_{R<})$ **where** $\lceil n\rceil_{R<} \equiv \lceil\lceil n\rceil_{<}\rceil_R$

1.2 Reactive Lemmas

lemma *unrest-ok-lift-rea* [*unrest*]:
$\$ok \sharp \lceil P\rceil_R$ $\$ok' \sharp \lceil P\rceil_R$
by (*pred-auto*)+

lemma *unrest-wait-lift-rea* [*unrest*]:
$\$wait \sharp \lceil P\rceil_R$ $\$wait' \sharp \lceil P\rceil_R$
by (*pred-auto*)+

lemma *unrest-tr-lift-rea* [*unrest*]:
$\$tr \sharp \lceil P\rceil_R$ $\$tr' \sharp \lceil P\rceil_R$
by (*pred-auto*)+

lemma *wait-tr-bij-lemma*: *bij-lens* $(wait_a +_L tr_a +_L \Sigma_r)$
by (*unfold-locales*, *auto simp add*: *lens-defs*)

lemma *des-lens-equiv-wait-tr-rest*: $\Sigma_D \approx_L wait +_L tr +_L \Sigma_R$
proof −
have $wait +_L tr +_L \Sigma_R = (wait_a +_L tr_a +_L \Sigma_r) ;_L \Sigma_D$
by (*simp add*: *plus-lens-distr wait-def tr-def rp-vars-child-lens-def*)
also have ... $\approx_L 1_L ;_L \Sigma_D$
using *lens-equiv-via-bij wait-tr-bij-lemma* **by** *auto*
also have ... $= \Sigma_D$
by (*simp*)
finally show *?thesis*
using *lens-equiv-sym* **by** *blast*
qed

lemma *rea-lens-bij*: *bij-lens* $(ok +_L wait +_L tr +_L \Sigma_R)$
proof −
have $ok +_L wait +_L tr +_L \Sigma_R \approx_L ok +_L \Sigma_D$
using *des-lens-equiv-wait-tr-rest des-vars-indeps lens-equiv-sym lens-plus-eq-right* **by** *blast*
also have ... $\approx_L 1_L$
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 *seqr-wait-true* [*usubst*]: $(P\ ;;\ Q)_{\ t} = (P_{\ t}\ ;;\ Q)$
by (*rel-auto*)

lemma *seqr-wait-false* [*usubst*]: $(P\ ;;\ Q)_{\ f} = (P_{\ f}\ ;;\ Q)$
by (*rel-auto*)

1.3 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 $\mathit{tcontr} :: {}'t{::}\mathit{trace} \Longrightarrow ('t,\ '\alpha)\ \mathit{rp} \times ('t,\ '\alpha)\ \mathit{rp}\ (\mathit{tt})$ **where**
  [*lens-defs*]:
  $\mathit{tcontr} = (\!|\ \mathit{lens\text{-}get} = (\lambda\ s.\ \mathit{get}_{(\$tr\,')_v}\ s - \mathit{get}_{(\$tr)_v}\ s)\ ,$
  $\qquad\qquad \mathit{lens\text{-}put} = (\lambda\ s\ v.\ \mathit{put}_{(\$tr\,')_v}\ s\ (\mathit{get}_{(\$tr)_v}\ s + v))\ |\!)$

definition $\mathit{itrace} :: {}'t{::}\mathit{trace} \Longrightarrow ('t,\ '\alpha)\ \mathit{rp} \times ('t,\ '\alpha)\ \mathit{rp}\ (\mathbf{it})$ **where**
  [*lens-defs*]:
  $\mathit{itrace} = (\!|\ \mathit{lens\text{-}get} = \mathit{get}_{(\$tr)_v},$
  $\qquad\qquad \mathit{lens\text{-}put} = (\lambda\ s\ v.\ \mathit{put}_{(\$tr\,')_v}\ (\mathit{put}_{(\$tr)_v}\ s\ v)\ v)\ |\!)$

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
  -svid-tcontr :: *svid* (*tt*)
  -svid-itrace :: *svid* (**it**)

translations
  -svid-tcontr == *CONST tcontr*
  -svid-itrace == *CONST itrace*

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

lemma *tcontr-alt-def'*: $\mathit{utp\text{-}expr.var}\ tt = (\$tr\,' - \$tr)$
  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*)+)

end

2 Reactive Healthiness Conditions

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

2.1 R1: Events cannot be undone

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

lemma *R1-idem*: $\mathit{R1}(\mathit{R1}(P)) = \mathit{R1}(P)$
  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*]: $[\![\ x \bowtie \mathit{in\text{-}var}\ tr;\ x \bowtie \mathit{out\text{-}var}\ tr;\ x \ \sharp\ P\ ]\!] \Longrightarrow x \ \sharp\ R1(P)$
  by (*simp add*: *R1-def unrest lens-indep-sym*)

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

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

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

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

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

lemma *disj-R1-closed* [*closure*]: $[\![\ P\ \mathit{is}\ R1;\ Q\ \mathit{is}\ R1\ ]\!] \Longrightarrow (P \vee Q)\ \mathit{is}\ R1$
  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*]: $[\![\ \bigwedge\ P.\ P \in A \Longrightarrow P\ \mathit{is}\ R1;\ A \neq \{\}\ ]\!] \Longrightarrow \bigsqcap\ A\ \mathit{is}\ R1$
  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*]:
  $[\![\ \bigwedge\ i.\ P(i)\ \mathit{is}\ R1\ ]\!] \Longrightarrow (\bigsqcap\ i \cdot P(i))\ \mathit{is}\ R1$
  by (*rel-blast*)

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

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

lemma *tr-id-conj-R1* [*closure*]:
$\$tr' =_u \$tr \wedge P\ \mathit{is}\ R1$
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*]: $P\ \mathit{is}\ R1 \Longrightarrow P\llbracket false/\$wait \rrbracket\ \mathit{is}\ R1$
by (*rel-auto*)

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

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

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

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

lemma *subst-R1-closed* [*closure*]: $\llbracket\ \$tr \sharp \sigma;\ \$tr' \sharp \sigma;\ P\ \mathit{is}\ \mathit{R1}\ \rrbracket \Longrightarrow \sigma \dagger P\ \mathit{is}\ \mathit{R1}$
by (*metis Healthy-def subst-R1*)

lemma *R1-by-refinement*:
$P\ \mathit{is}\ \mathit{R1} \longleftrightarrow ((\$tr \le_u \$tr') \sqsubseteq P)$
by (*rel-blast*)

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

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

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

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

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

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

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

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

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

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

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

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

lemma *R1-extend-conj-unrest*: $\llbracket\ \$tr \sharp Q;\ \$tr' \sharp Q\ \rrbracket \Longrightarrow \mathit{R1}(P \wedge Q) = (\mathit{R1}(P) \wedge Q)$
by *pred-auto*

lemma *R1-extend-conj-unrest'*: $\llbracket\ \$tr \sharp P;\ \$tr' \sharp P\ \rrbracket \Longrightarrow \mathit{R1}(P \wedge Q) = (P \wedge \mathit{R1}(Q))$
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*)

2.2 R2: No dependence upon trace history

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

definition $R2a$ :: $('t::trace, '\alpha, '\beta)$ *rel-rp* $\Rightarrow$ $('t,'\alpha,'\beta)$ *rel-rp* **where**
[*upred-defs*]: $R2a\ (P) = (\bigsqcap\ s \cdot P⟦«s»,«s»+(\$tr´-\$tr)/\$tr,\$tr´⟧)$

definition $R2a'$ :: $('t::trace, '\alpha, '\beta)$ *rel-rp* $\Rightarrow$ $('t,'\alpha,'\beta)$ *rel-rp* **where**
[*upred-defs*]: $R2a'\ P = (R2a(P) \triangleleft R1(true) \triangleright P)$

definition $R2s$ :: $('t::trace, '\alpha, '\beta)$ *rel-rp* $\Rightarrow$ $('t,'\alpha,'\beta)$ *rel-rp* **where**
[*upred-defs*]: $R2s\ (P) = (P⟦0/\$tr⟧⟦(\$tr´-\$tr)/\$tr´⟧)$

definition $R2$ :: $('t::trace, '\alpha, '\beta)$ *rel-rp* $\Rightarrow$ $('t, '\alpha, '\beta)$ *rel-rp* **where**
[*upred-defs*]: $R2(P) = R1(R2s(P))$

definition $R2c$ :: $('t::trace, '\alpha, '\beta)$ *rel-rp* $\Rightarrow$ $('t, '\alpha, '\beta)$ *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 [4]. An issue with these forms is that their definition depends upon *R1* also being satisfied [3], 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*]: $\$ok \sharp P \Longrightarrow \$ok \sharp R2s(P)$
by (*simp add*: *R2s-def unrest*)

lemma *unrest-ok′-R2s* [*unrest*]: $\$ok´ \sharp P \Longrightarrow \$ok´ \sharp R2s(P)$
by (*simp add*: *R2s-def unrest*)

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

lemma *unrest-ok′-R2c* [*unrest*]: $\$ok´ \sharp P \Longrightarrow \$ok´ \sharp R2c(P)$
by (*simp add*: *R2c-def unrest*)

lemma *R2s-unrest* [*unrest*]: $⟦$ *vwb-lens* x; $x \bowtie$ *in-var tr*; $x \bowtie$ *out-var tr*; $x \sharp P$ $⟧ \Longrightarrow x \sharp R2s(P)$
by (*simp add*: *R2s-def unrest usubst lens-indep-sym*)

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

```
lemma R2s-subst-wait′-true [usubst]:
  (R2s(P))⟦true/$wait´⟧ = R2s(P⟦true/$wait´⟧)
  by (simp add: R2s-def usubst unrest)

lemma R2-subst-wait-true [usubst]:
  (R2(P))⟦true/$wait⟧ = R2(P⟦true/$wait⟧)
  by (simp add: R2-def R1-def R2s-def usubst unrest)

lemma R2-subst-wait′-true [usubst]:
  (R2(P))⟦true/$wait´⟧ = R2(P⟦true/$wait´⟧)
  by (simp add: R2-def R1-def R2s-def usubst unrest)

lemma R2-subst-wait-false [usubst]:
  (R2(P))⟦false/$wait⟧ = R2(P⟦false/$wait⟧)
  by (simp add: R2-def R1-def R2s-def usubst unrest)

lemma R2-subst-wait′-false [usubst]:
  (R2(P))⟦false/$wait´⟧ = R2(P⟦false/$wait´⟧)
  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 is R2a ⟷ P 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 ⊑ Q ⟹ R2a(P) ⊑ R2a(Q)
  by (rel-blast)

lemma R2a′-mono: P ⊑ Q ⟹ R2a′(P) ⊑ R2a′(Q)
  by (rel-blast)

lemma R2a′-weakening: R2a′(P) ⊑ 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*: $P \sqsubseteq Q \Longrightarrow R2(P) \sqsubseteq R2(Q)$
by (*pred-auto*)

lemma *R2-implies-R1* [*closure*]: *P is R2* $\Longrightarrow$ *P 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*]: $\$tr <_u \$tr\acute{}$ *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*:
$R2s(\bigsqcap\ i \in A \cdot P(i)) = (\bigsqcap\ i \in A \cdot R2s(P(i)))$
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*]: $[\![\ P\ is\ R2c;\ Q\ is\ R2c\ ]\!] \Longrightarrow (P \wedge Q)\ is\ R2c$
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*]: $(R2c\ P)[\![true/\$wait']\!] = R2c(P[\![true/\$wait']\!])$
by (*rel-auto*)

lemma *R2c-wait'-false* [*usubst*]: $(R2c\ P)[\![false/\$wait']\!] = R2c(P[\![false/\$wait']\!])$
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'-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 $R2c(II) = R2c(\$tr' =_u \$tr \wedge II{\restriction_\alpha}tr)$
by (*subst skip-r-unfold*[*of tr*], *simp-all*)
also have ... $= (R2c(\$tr' =_u \$tr) \wedge II{\restriction_\alpha}tr)$
by (*simp add*: *R2c-and*, *simp add*: *R2c-def R2s-def usubst unrest cond-idem*)
also have ... $= (\$tr' =_u \$tr \wedge II{\restriction_\alpha}tr)$
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:
  ⟦ $tr ♯ v; $tr´ ♯ v ⟧ ⟹ R1 (R2s ($tr´ =_u $tr ^_u v)) = ($tr´ =_u $tr  ^_u v)
  apply (rel-auto)
   apply (metis append-minus)
  apply (simp add: Prefix-Order.prefixI)
  done

lemma R2-tr-prefix: R2($tr ≤_u $tr´) = ($tr ≤_u $tr´)
  by (pred-auto)

lemma R2-form:
  R2(P) = (∃ tt_0 · P⟦0/$tr⟧⟦«tt_0»/$tr´⟧ ∧ $tr´ =_u $tr + «tt_0»)
  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 ↦_s tr_0, $tr´ ↦_s tr_0 + t] † P = [$tr ↦_s 0, $tr´ ↦_s t] † P
proof −
  have [$tr ↦_s tr_0, $tr´ ↦_s tr_0 + t] † R2 P = [$tr ↦_s 0, $tr´ ↦_s t] † R2 P
    by (rel-auto)
  thus ?thesis
    by (simp add: Healthy-if assms)
qed

lemma R2-seqr-form:
  shows (R2(P) ;; R2(Q)) =
         (∃ tt_1 · ∃ tt_2 · ((P⟦0/$tr⟧⟦«tt_1»/$tr´⟧) ;; (Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧))
                        ∧ ($tr´ =_u $tr + «tt_1» + «tt_2»))
proof −
  have (R2(P) ;; R2(Q)) = (∃ tr_0 · (R2(P))⟦«tr_0»/$tr´⟧ ;; (R2(Q))⟦«tr_0»/$tr⟧)
    by (subst seqr-middle[of tr], simp-all)
  also have ... =
       (∃ tr_0 · ∃ tt_1 · ∃ tt_2 · ((P⟦0/$tr⟧⟦«tt_1»/$tr´⟧ ∧ «tr_0» =_u $tr + «tt_1») ;;
                                (Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧ ∧ $tr´ =_u «tr_0» + «tt_2»)))
    by (simp add: R2-form usubst unrest uquant-lift, rel-blast)
  also have ... =
       (∃ tr_0 · ∃ tt_1 · ∃ tt_2 · ((«tr_0» =_u $tr + «tt_1» ∧ P⟦0/$tr⟧⟦«tt_1»/$tr´⟧) ;;
                                (Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧ ∧ $tr´ =_u «tr_0» + «tt_2»)))
    by (simp add: conj-comm)
  also have ... =
       (∃ tt_1 · ∃ tt_2 · ∃ tr_0 · ((P⟦0/$tr⟧⟦«tt_1»/$tr´⟧) ;; (Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧))
                                ∧ «tr_0» =_u $tr + «tt_1» ∧ $tr´ =_u «tr_0» + «tt_2»)
    by (rel-blast)
  also have ... =
       (∃ tt_1 · ∃ tt_2 · ((P⟦0/$tr⟧⟦«tt_1»/$tr´⟧) ;; (Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧))
                     ∧ (∃ tr_0 · «tr_0» =_u $tr + «tt_1» ∧ $tr´ =_u «tr_0» + «tt_2»))
    by (rel-auto)
  also have ... =
       (∃ tt_1 · ∃ tt_2 · ((P⟦0/$tr⟧⟦«tt_1»/$tr´⟧) ;; (Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧))
                    ∧ ($tr´ =_u $tr + «tt_1» + «tt_2»))
    by (rel-auto)
  finally show ?thesis .
qed

lemma R2-seqr-form′:
  assumes P is R2 Q is R2
  shows P ;; Q =
        (∃ tt_1 · ∃ tt_2 · ((P⟦0/$tr⟧⟦«tt_1»/$tr´⟧) ;; (Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧))
                    ∧ ($tr´ =_u $tr + «tt_1» + «tt_2»))
  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 =
        (∃ (tt_1, tt_2) · ((P⟦0,«tt_1»/$tr,$tr´⟧) ;; (Q⟦0,«tt_2»/$tr,$tr´⟧))
                    ∧ ($tr´ =_u $tr + «tt_1» + «tt_2»))
  by (subst R2-seqr-form′, simp-all add: assms, rel-auto)

lemma R2-tr-middle:
  assumes P is R2 Q is R2
  shows (∃ tr_0 · (P⟦«tr_0»/$tr´⟧ ;; Q⟦«tr_0»/$tr⟧) ∧ «tr_0» ≤_u $tr´) = (P ;; Q)
proof −
  have (P ;; Q) = (∃ tr_0 · (P⟦«tr_0»/$tr´⟧ ;; Q⟦«tr_0»/$tr⟧))
    by (simp add: seqr-middle)
  also have ... = (∃ tr_0 · ((R2 P)⟦«tr_0»/$tr´⟧ ;; (R2 Q)⟦«tr_0»/$tr⟧))
    by (simp add: assms Healthy-if)
  also have ... = (∃ tr_0 · ((R2 P)⟦«tr_0»/$tr´⟧ ;; (R2 Q)⟦«tr_0»/$tr⟧) ∧ «tr_0» ≤_u $tr´)
    by (rel-auto)
  also have ... = (∃ tr_0 · (P⟦«tr_0»/$tr´⟧ ;; Q⟦«tr_0»/$tr⟧) ∧ «tr_0» ≤_u $tr´)
    by (simp add: assms Healthy-if)
  finally show ?thesis ..
qed

lemma R2-seqr-distribute:
  fixes P :: ('t::trace,'α,'β) rel-rp and Q :: ('t,'β,'γ) rel-rp
  shows R2(R2(P) ;; R2(Q)) = (R2(P) ;; R2(Q))
proof −
  have R2(R2(P) ;; R2(Q)) =
    ((∃ tt_1 · ∃ tt_2 · (P⟦0/$tr⟧⟦«tt_1»/$tr´⟧ ;; Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧)⟦($tr´ − $tr)/$tr´⟧
      ∧ $tr´ − $tr =_u «tt_1» + «tt_2») ∧ $tr´ ≥_u $tr)
    by (simp add: R2-seqr-form, simp add: R2s-def usubst unrest, rel-auto)
  also have ... =
    ((∃ tt_1 · ∃ tt_2 · (P⟦0/$tr⟧⟦«tt_1»/$tr´⟧ ;; Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧)⟦(«tt_1» + «tt_2»)/$tr´⟧
      ∧ $tr´ − $tr =_u «tt_1» + «tt_2») ∧ $tr´ ≥_u $tr)
      by (subst subst-eq-replace, simp)
  also have ... =
    ((∃ tt_1 · ∃ tt_2 · (P⟦0/$tr⟧⟦«tt_1»/$tr´⟧ ;; Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧)
      ∧ $tr´ − $tr =_u «tt_1» + «tt_2») ∧ $tr´ ≥_u $tr)
      by (rel-auto)
  also have ... =
    (∃ tt_1 · ∃ tt_2 · (P⟦0/$tr⟧⟦«tt_1»/$tr´⟧ ;; Q⟦0/$tr⟧⟦«tt_2»/$tr´⟧)
      ∧ ($tr´ − $tr =_u «tt_1» + «tt_2» ∧ $tr´ ≥_u $tr))
    by pred-auto
  also have ... =
    ((∃ tt₁ · ∃ tt₂ · (P⟦0/$tr⟧⟦«tt₁»/$tr´⟧ ;; Q⟦0/$tr⟧⟦«tt₂»/$tr´⟧)
      ∧ $tr´ =ᵤ $tr + «tt₁» + «tt₂»))
  proof −
    have ⋀ tt₁ tt₂. ((($tr´ − $tr =ᵤ «tt₁» + «tt₂») ∧ $tr´ ≥ᵤ $tr) :: ('t,'α,'γ) rel-rp)
           = ($tr´ =ᵤ $tr + «tt₁» + «tt₂»)
      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* ∘ *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*) *rel-rp* **and** *Q* :: (*'t*,*'β*,*'γ*) *rel-rp*
  assumes *P is R1 P is R2c Q is R1 Q 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*) *is R2*
  by (*rel-auto*)

lemma *R1-R2s-R1-true-lemma*:
  R1(*R2s*(*R1* (¬ *R2s P*) ;; *R1 true*)) = *R1*(*R2s*((¬ *P*) ;; *R1 true*))
  by (*rel-auto*)

lemma *R2c-healthy-R2s*: *P is R2c* ⟹ *R1*(*R2s*(*P*)) = *R1*(*P*)
  by (*simp add*: *Healthy-def R1-R2s-R2c*)

2.3 R3: No activity while predecessor is waiting

definition *R3* :: (*'t*::*trace*, *'α*) *hrel-rp* ⇒ (*'t*, *'α*) *hrel-rp* **where**
[*upred-defs*]: *R3*(*P*) = (*II* ◃ $*wait* ▹ *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* ⊑ *Q* ⟹ *R3*(*P*) ⊑ *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* ∧ *Q*) = (*R3*(*P*) ∧ *R3*(*Q*))
  by (*rel-auto*)

lemma *R3-disj*: $R3(P \lor Q) = (R3(P) \lor R3(Q))$
  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-UINF*:
  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*: $R3(P) = ((\$wait \land II) \lor (\neg\ \$wait \land P))$
  by (*rel-auto*)

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

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

2.4 R4: The trace strictly increases

definition *R4* :: ('*t*::*trace*, 'α, 'β) *rel-rp* ⇒ ('*t*, 'α, 'β) *rel-rp* **where**
[*upred-defs*]: $R4(P) = (P \land \$tr <_u \$tr')$

lemma *R4-implies-R1* [*closure*]: *P is R4* ⟹ *P is R1*
  using *less-iff* **by** *rel-blast*

lemma *R4-iff-refine*:
$P\ \mathit{is}\ R4 \longleftrightarrow (\$tr <_u \$tr') \sqsubseteq P$
by (*rel-blast*)

lemma *R4-idem*: $R4\ (R4\ P) = R4\ P$
by (*rel-auto*)

lemma *R4-false*: $R4(\mathit{false}) = \mathit{false}$
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*]: $R4(P)\ \mathit{is}\ R4$
by (*rel-auto*)

lemma *false-R4* [*closure*]: $\mathit{false}\ \mathit{is}\ R4$
by (*rel-auto*)

lemma *UINF-R4-closed* [*closure*]:
$[\![\ \bigwedge\ i.\ P\ i\ \mathit{is}\ R4\ ]\!] \Longrightarrow (\bigsqcap\ i \cdot P\ i)\ \mathit{is}\ R4$
by (*rel-blast*)

lemma *conj-R4-closed* [*closure*]:
$[\![\ P\ \mathit{is}\ R4;\ Q\ \mathit{is}\ R4\ ]\!] \Longrightarrow (P \wedge Q)\ \mathit{is}\ R4$
by (*simp add*: *Healthy-def R4-conj*)

lemma *disj-R4-closed* [*closure*]:
$[\![\ P\ \mathit{is}\ R4;\ Q\ \mathit{is}\ R4\ ]\!] \Longrightarrow (P \vee Q)\ \mathit{is}\ R4$
by (*simp add*: *Healthy-def R4-disj*)

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

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

2.5 R5: The trace does not increase

definition $R5 :: ('t{::}\mathit{trace}, '\alpha, '\beta)\ \mathit{rel\text{-}rp} \Rightarrow ('t, '\alpha, '\beta)\ \mathit{rel\text{-}rp}$ **where**
[*upred-defs*]: $R5(P) = (P \wedge \$tr =_u \$tr')$

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

lemma *R5-iff-refine*:
$P\ \mathit{is}\ R5 \longleftrightarrow (\$tr =_u \$tr') \sqsubseteq P$
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*: $R4\ (R5\ P) = \mathit{false}$
by (*rel-auto*)

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

lemma *UINF-R5-closed* [*closure*]:
$[\![\bigwedge i.\ P\ i\ \mathit{is}\ R5\ ]\!] \Longrightarrow (\bigsqcap\ i \cdot P\ i)\ \mathit{is}\ R5$
by (*rel-blast*)

lemma *conj-R5-closed* [*closure*]:
$[\![P\ \mathit{is}\ R5;\ Q\ \mathit{is}\ R5\ ]\!] \Longrightarrow (P \wedge Q)\ \mathit{is}\ R5$
by (*simp add*: *Healthy-def R5-conj*)

lemma *disj-R5-closed* [*closure*]:
$[\![P\ \mathit{is}\ R5;\ Q\ \mathit{is}\ R5\ ]\!] \Longrightarrow (P \vee Q)\ \mathit{is}\ R5$
by (*simp add*: *Healthy-def R5-disj*)

lemma *seq-R5-closed* [*closure*]:
$[\![P\ \mathit{is}\ R5;\ Q\ \mathit{is}\ R5\ ]\!] \Longrightarrow (P\ ;;\ Q)\ \mathit{is}\ R5$
by (*rel-auto*, *metis*)

2.6 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*: $[\![P\ \mathit{is}\ R1;\ P\ \mathit{is}\ R2;\ P\ \mathit{is}\ R3\ ]\!] \Longrightarrow P\ \mathit{is}\ RP$
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*: $P \sqsubseteq Q \Longrightarrow RP(P) \sqsubseteq RP(Q)$
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:
  II is RP
  by (simp add: Healthy-def' RP-skip)

lemma RP-seq-closure:
  assumes P is RP Q is RP
  shows (P ;; Q) is RP
proof (rule RP-intro)
  show (P ;; Q) is R1
    by (metis Healthy-def R1-seqr RP-def assms)
  thus (P ;; Q) is R2
    by (metis Healthy-def' R2-R2c-def R2c-R1-seq RP-def  assms)
  show (P ;; Q) 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
```

2.7 UTP theories

```
typedecl REA
abbreviation REA ≡ UTHY(REA, ('t::trace,'α) rp)

overloading
  rea-hcond == utp-hcond :: (REA, ('t::trace,'α) rp) uthy ⇒ (('t,'α) rp × ('t,'α) rp) health
  rea-unit == utp-unit :: (REA, ('t::trace,'α) rp) uthy ⇒ ('t,'α) hrel-rp
begin
  definition rea-hcond :: (REA, ('t::trace,'α) rp) uthy ⇒ (('t,'α) rp × ('t,'α) rp) health
  where [upred-defs]: rea-hcond T = RP
  definition rea-unit :: (REA, ('t::trace,'α) rp) uthy ⇒ ('t,'α) hrel-rp
  where [upred-defs]: rea-unit T = II
end

interpretation rea-utp-theory: utp-theory UTHY(REA, ('t::trace,'α) rp)
  rewrites carrier (uthy-order REA) = ⟦RP⟧H
  by (simp-all add: rea-hcond-def utp-theory-def RP-idem)

interpretation rea-utp-theory-mono: utp-theory-continuous UTHY(REA, ('t::trace,'α) rp)
  rewrites carrier (uthy-order REA) = ⟦RP⟧H
  by (unfold-locales, simp-all add: RP-Continuous rea-hcond-def)

interpretation rea-utp-theory-rel: utp-theory-unital UTHY(REA, ('t::trace,'α) rp)
  rewrites carrier (uthy-order REA) = ⟦RP⟧H
  by (unfold-locales, simp-all add: rea-hcond-def rea-unit-def RP-seq-closure RP-skip-closure)

lemma rea-top: ⊤REA = ($wait ∧ II)
proof −
  have ⊤REA = RP(false)
    by (simp add: rea-utp-theory-mono.healthy-top, simp add: rea-hcond-def)
  also have ... = ($wait ∧ II)
    by (rel-auto, metis minus-zero-eq)
  finally show ?thesis .
qed

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

lemma rea-bottom: ⊥_REA = R1($wait ⇒ II)
proof −
  have ⊥_REA = RP(true)
    by (simp add: rea-utp-theory-mono.healthy-bottom, simp add: rea-hcond-def)
  also have ... = R1($wait ⇒ II)
    by (rel-auto, metis minus-zero-eq)
  finally show ?thesis .
qed

end
```

3 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. We first define healthiness conditions for R1 and R2 merge predicates.

```
definition R1m :: ('t :: trace, 'α) rp merge ⇒ ('t, 'α) rp merge
  where [upred-defs]: R1m(M) = (M ∧ $tr_< ≤_u $tr´)

definition R1m' :: ('t :: trace, 'α) rp merge ⇒ ('t, 'α) rp merge
  where [upred-defs]: R1m'(M) = (M ∧ $tr_< ≤_u $tr´ ∧ $tr_< ≤_u $0−tr ∧ $tr_< ≤_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.

```
term M⟦0,x,k/y,z,a⟧

term M⟦0,$tr´ − $tr_<,$0−tr − $tr_<,$1−tr − $tr_</$tr_<,$tr´,$0−tr,$1−tr⟧

definition R2m :: ('t :: trace, 'α) rp merge ⇒ ('t, 'α) rp merge
 where [upred-defs]: R2m(M) = R1m(M⟦0,$tr´ − $tr_<,$0−tr − $tr_<,$1−tr − $tr_</$tr_<,$tr´,$0−tr,$1−tr⟧)

definition R2m' :: ('t :: trace, 'α) rp merge ⇒ ('t, 'α) rp merge
 where [upred-defs]: R2m'(M) = R1m'(M⟦0,$tr´ − $tr_<,$0−tr − $tr_<,$1−tr − $tr_</$tr_<,$tr´,$0−tr,$1−tr⟧)

definition R2cm :: ('t :: trace, 'α) rp merge ⇒ ('t, 'α) rp merge
```

```
 where [upred-defs]: R2cm(M) = M⟦0,$tr´ − $tr_<,$0−tr − $tr_<,$1−tr − $tr_</$tr_<,$tr´,$0−tr,$1−tr⟧
◁ $tr_< ≤_u $tr´ ▷ M

lemma R2m'-form:
  R2m'(M) =
  (∃ (tt_p, tt_0, tt_1) · M⟦0,«tt_p»,«tt_0»,«tt_1»/$tr_<,$tr´,$0−tr,$1−tr⟧
                  ∧ $tr´ =_u $tr_< + «tt_p»
                  ∧ $0−tr =_u $tr_< + «tt_0»
                  ∧ $1−tr =_u $tr_< + «tt_1»)
  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 ... = R1m(M) ;; R1(P)
    by (simp add: R1m-seq-lemma)
  also have ... = 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 ⟹ (P ‖_M Q) is R1
  by (rel-blast)

lemma R2-R2m'-pbm: R2(P ‖_M Q) = (R2(P) ‖_R2m'(M) R2(Q))
proof −
  have (R2(P) ‖_R2m'(M) R2(Q)) = ((R2(P) ‖_s R2(Q)) ;;
                 (∃ (tt_p, tt_0, tt_1) · M⟦0,«tt_p»,«tt_0»,«tt_1»/$tr_<,$tr´,$0−tr,$1−tr⟧
                                ∧ $tr´ =_u $tr_< + «tt_p»
                                ∧ $0−tr =_u $tr_< + «tt_0»
```

$\wedge\ \$1{-}tr =_u \$tr_< + \ll tt_1 \gg))$
by (*simp add*: *par-by-merge-def R2m′-form*)
also have ... $= (\exists\ (tt_p, tt_0, tt_1) \cdot ((R2(P) \parallel_s R2(Q)) \ ;;\ (M[\![0,\ll tt_p\gg,\ll tt_0\gg,\ll tt_1\gg/\$tr_<,\$tr´,\$0{-}tr,\$1{-}tr]\!]$
$\wedge\ \$tr´ =_u \$tr_< + \ll tt_p\gg$
$\wedge\ \$0{-}tr =_u \$tr_< + \ll tt_0\gg$
$\wedge\ \$1{-}tr =_u \$tr_< + \ll tt_1\gg)))$
by (*rel-blast*)
also have ... $= (\exists\ (tt_p, tt_0, tt_1) \cdot (((R2(P) \parallel_s R2(Q)) \wedge \$0{-}tr´ =_u \$tr_<´ + \ll tt_0\gg \wedge \$1{-}tr´ =_u \$tr_<´ + \ll tt_1\gg) \ ;;$
$(M[\![0,\ll tt_p\gg,\ll tt_0\gg,\ll tt_1\gg/\$tr_<,\$tr´,\$0{-}tr,\$1{-}tr]\!] \wedge \$tr´ =_u \$tr_< + \ll tt_p\gg)))$
by (*rel-blast*)
also have ... $= (\exists\ (tt_p, tt_0, tt_1) \cdot (((R2(P) \parallel_s R2(Q)) \wedge \$0{-}tr´ =_u \$tr_<´ + \ll tt_0\gg \wedge \$1{-}tr´ =_u \$tr_<´ + \ll tt_1\gg) \ ;;$
$(M[\![0,\ll tt_p\gg,\ll tt_0\gg,\ll tt_1\gg/\$tr_<,\$tr´,\$0{-}tr,\$1{-}tr]\!])) \wedge \$tr´ =_u \$tr + \ll tt_p\gg)$
by (*rel-blast*)
also have ... $= (\exists\ (tt_p, tt_0, tt_1) \cdot (((R2(P) \wedge \$tr´ =_u \$tr + \ll tt_0\gg) \parallel_s (R2(Q) \wedge \$tr´ =_u \$tr + \ll tt_1\gg)) \ ;;$
$(M[\![0,\ll tt_p\gg,\ll tt_0\gg,\ll tt_1\gg/\$tr_<,\$tr´,\$0{-}tr,\$1{-}tr]\!])) \wedge \$tr´ =_u \$tr + \ll tt_p\gg)$
by (*rel-auto*, *blast*, *metis le-add trace-class.add-diff-cancel-left*)
also have ... $= (\exists\ (tt_p, tt_0, tt_1) \cdot ((\ \ ((\exists\ tt_0{}' \cdot P[\![0,\ll tt_0{}'\gg/\$tr,\$tr´]\!] \wedge \$tr´ =_u \$tr + \ll tt_0{}'\gg) \wedge \$tr´ =_u \$tr + \ll tt_0\gg)$
$\parallel_s ((\exists\ tt_1{}' \cdot Q[\![0,\ll tt_1{}'\gg/\$tr,\$tr´]\!] \wedge \$tr´ =_u \$tr + \ll tt_1{}'\gg) \wedge \$tr´ =_u \$tr + \ll tt_1\gg)) \ ;;$
$(M[\![0,\ll tt_p\gg,\ll tt_0\gg,\ll tt_1\gg/\$tr_<,\$tr´,\$0{-}tr,\$1{-}tr]\!])) \wedge \$tr´ =_u \$tr + \ll tt_p\gg)$
by (*simp add*: *R2-form usubst*)
also have ... $= (\exists\ (tt_p, tt_0, tt_1) \cdot ((\ \ (P[\![0,\ll tt_0\gg/\$tr,\$tr´]\!] \ \wedge \$tr´ =_u \$tr + \ll tt_0\gg)$
$\parallel_s (Q[\![0,\ll tt_1\gg/\$tr,\$tr´]\!] \wedge \$tr´ =_u \$tr + \ll tt_1\gg)) \ ;;$
$(M[\![0,\ll tt_p\gg,\ll tt_0\gg,\ll tt_1\gg/\$tr_<,\$tr´,\$0{-}tr,\$1{-}tr]\!])) \wedge \$tr´ =_u \$tr + \ll tt_p\gg)$
by (*rel-auto*, *metis left-cancel-monoid-class.add-left-imp-eq*, *blast*)
also have ... $= R2(P \parallel_M Q)$
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* :: $('t :: trace, '\alpha)\ rp\ merge \Rightarrow ('t, '\alpha)\ rp\ merge$ **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 $(P \parallel_M Q) = ((P \parallel_M Q)[\![true/\$wait]\!] \triangleleft \$wait \triangleright (P \parallel_M Q))$
by (*metis cond-L6 cond-var-split in-var-uvar wait-vwb-lens*)
also have ... $= (((R3\ P)[\![true/\$wait]\!] \parallel_{(R3m\ M)[\![true/\$wait_<]\!]} (R3\ Q)[\![true/\$wait]\!]) \triangleleft \$wait \triangleright (P \parallel_M Q))$
by (*subst-tac*, *simp add*: *Healthy-if assms*)
also have ... $= ((II[\![true/\$wait]\!] \parallel_{skip_m[\![true/\$wait_<]\!]} II[\![true/\$wait]\!]) \triangleleft \$wait \triangleright (P \parallel_M Q))$
by (*simp add*: *R3-def R3m-def usubst*)
also have ... $= ((II \parallel_{skip_m} II)[\![true/\$wait]\!] \triangleleft \$wait \triangleright (P \parallel_M Q))$
by (*subst-tac*)
also have ... $= (II \triangleleft \$wait \triangleright (P \parallel_M Q))$
by (*simp add*: *cond-var-subst-left par-by-merge-skip*)
also have ... $= R3(P \parallel_M Q)$
by (*simp add*: *R3-def*)
finally show *?thesis*
by (*simp add*: *Healthy-def*)
qed

lemma *SymMerge-R1-true* [*closure*]:
M *is SymMerge* $\Longrightarrow M\ ;;\ R1(true)$ *is SymMerge*
by (*rel-auto*)

end

4 Reactive Relations

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

This theory defines a predicate calculus for R1-R2 predicates as an extension of the standard alphabetised predicate calculus.

4.1 Healthiness Conditions

```
definition RR :: ('t::trace, 'α, 'β) rel-rp ⇒ ('t, 'α, 'β) rel-rp where
[upred-defs]: RR(P) = (∃ {$ok,$ok´,$wait,$wait´} · 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 is RR ⟹ P is R1
  by (metis Healthy-def R1-RR)

lemma RR-implies-R2c: P is RR ⟹ P is R2c
  by (metis Healthy-def R2c-RR)

lemma RR-implies-R2 [closure]: P is RR ⟹ P 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 is R1 P is R2c
  shows P 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 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 ⋀ t. P⟦0,«t»/$tr,$tr´⟧ ⊑ Q⟦0,«t»/$tr,$tr´⟧
  shows P ⊑ Q
proof –
```

have $\bigwedge$ *t*. (*RR P*)⟦*0*,«*t*»/\$*tr*,\$*tr*´⟧ ⊑ (*RR Q*)⟦*0*,«*t*»/\$*tr*,\$*tr*´⟧
  by (*simp add*: *Healthy-if assms*)
hence *RR*(*P*) ⊑ *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

4.2 Reactive relational operators

named-theorems *rpred*

abbreviation *rea-true* :: (′*t*::*trace*,′α,′β) *rel-rp* ($true_r$) **where**
$true_r$ ≡ *R1*(*true*)

definition *rea-not* :: (′*t*::*trace*,′α,′β) *rel-rp* ⇒ (′*t*,′α,′β) *rel-rp* ($\neg_r$ - [*40*] *40*)
where [*upred-defs*]: ($\neg_r$ *P*) = *R1*(¬ *P*)

definition *rea-diff* :: (′*t*::*trace*,′α,′β) *rel-rp* ⇒ (′*t*,′α,′β) *rel-rp* ⇒ (′*t*,′α,′β) *rel-rp* (**infixl** $-_r$ *65*)
where [*upred-defs*]: *rea-diff P Q* = (*P* ∧ $\neg_r$ *Q*)

definition *rea-impl* ::
(′*t*::*trace*,′α,′β) *rel-rp* ⇒ (′*t*,′α,′β) *rel-rp* ⇒ (′*t*,′α,′β) *rel-rp* (**infixr** $\Rightarrow_r$ *25*)
where [*upred-defs*]: (*P* $\Rightarrow_r$ *Q*) = ($\neg_r$ *P* ∨ *Q*)

definition *rea-lift* :: (′*t*::*trace*,′α,′β) *rel-rp* ⇒ (′*t*,′α,′β) *rel-rp* ($[\text{-}]_r$)
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* ∧ \$$\Sigma_R$´ $=_u$ \$$\Sigma_R$)

definition *rea-assert* :: (′*t*::*trace*,′α) *hrel-rp* ⇒ (′*t*,′α) *hrel-rp* ($\{\text{-}\}_r$)
where [*upred-defs*]: $\{b\}_r$ = (II_r ∨ $\neg_r$ *b*)

Trace contribution substitution: make a substitution for the trace contribution lens *tt*, and apply *R1* to make the resulting predicate healthy again.

definition *rea-subst* :: $('t{::}trace,\ '\alpha)\ hrel\text{-}rp \Rightarrow ('t,\ ('t,\ '\alpha)\ rp)\ hexpr \Rightarrow ('t,\ '\alpha)\ hrel\text{-}rp\ (\text{-}[\![\text{-}]\!]_r\ [999,0]\ 999)$

where [*upred-defs*]: $P[\![v]\!]_r = R1(P[\![v/\&tt]\!])$

4.3 Unrestriction and substitution laws

lemma *rea-true-unrest* [*unrest*]:

$[\![\ x \bowtie (\$tr)_v;\ x \bowtie (\$tr')_v\ ]\!] \Longrightarrow x \sharp\ true_r$

by (*simp add*: *R1-def unrest lens-indep-sym*)

lemma *rea-not-unrest* [*unrest*]:

$[\![\ x \bowtie (\$tr)_v;\ x \bowtie (\$tr')_v;\ x \sharp P\ ]\!] \Longrightarrow x \sharp \neg_r P$

by (*simp add*: *rea-not-def R1-def unrest lens-indep-sym*)

lemma *rea-impl-unrest* [*unrest*]:

$[\![\ x \bowtie (\$tr)_v;\ x \bowtie (\$tr')_v;\ x \sharp P;\ x \sharp Q\ ]\!] \Longrightarrow x \sharp (P \Rightarrow_r Q)$

by (*simp add*: *rea-impl-def unrest*)

lemma *rea-true-usubst* [*usubst*]:

$[\![\ \$tr \sharp \sigma;\ \$tr' \sharp \sigma\ ]\!] \Longrightarrow \sigma \dagger true_r = true_r$

by (*simp add*: *R1-def usubst*)

lemma *rea-not-usubst* [*usubst*]:

$[\![\ \$tr \sharp \sigma;\ \$tr' \sharp \sigma\ ]\!] \Longrightarrow \sigma \dagger (\neg_r P) = (\neg_r \sigma \dagger P)$

by (*simp add*: *rea-not-def R1-def usubst*)

lemma *rea-impl-usubst* [*usubst*]:

$[\![\ \$tr \sharp \sigma;\ \$tr' \sharp \sigma\ ]\!] \Longrightarrow \sigma \dagger (P \Rightarrow_r Q) = (\sigma \dagger P \Rightarrow_r \sigma \dagger Q)$

by (*simp add*: *rea-impl-def usubst R1-def*)

lemma *rea-true-usubst-tt* [*usubst*]:

$R1(true)[\![e/\&tt]\!] = true$

by (*rel-simp*)

lemma *unrest-rea-subst* [*unrest*]:

$[\![\ mwb\text{-}lens\ x;\ x \bowtie (\$tr)_v;\ x \bowtie (\$tr')_v;\ x \sharp v;\ x \sharp P\ ]\!] \Longrightarrow\ x \sharp P[\![v]\!]_r$

by (*simp add*: *rea-subst-def R1-def unrest lens-indep-sym*)

lemma *rea-substs* [*usubst*]:

$true_r[\![v]\!]_r = true_r\ \ true[\![v]\!]_r = true_r\ \ false[\![v]\!]_r = false$

$(\neg_r P)[\![v]\!]_r = (\neg_r P[\![v]\!]_r)\ (P \wedge Q)[\![v]\!]_r = (P[\![v]\!]_r \wedge Q[\![v]\!]_r)\ (P \vee Q)[\![v]\!]_r = (P[\![v]\!]_r \vee Q[\![v]\!]_r)$

$(P \Rightarrow_r Q)[\![v]\!]_r = (P[\![v]\!]_r \Rightarrow_r Q[\![v]\!]_r)$

by *rel-auto+*

lemma *rea-substs-lattice* [*usubst*]:

$(\bigsqcap\ i \cdot P(i))[\![v]\!]_r\ \ = (\bigsqcap\ i \cdot (P(i))[\![v]\!]_r)$

$(\bigsqcap\ i{\in}A \cdot P(i))[\![v]\!]_r = (\bigsqcap\ i{\in}A \cdot (P(i))[\![v]\!]_r)$

$(\bigsqcup\ i \cdot P(i))[\![v]\!]_r\ \ = (\bigsqcup\ i \cdot (P(i))[\![v]\!]_r)$

by (*rel-auto*)+

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

$A \neq \{\} \Longrightarrow (\bigsqcup\ i{\in}A \cdot P(i))[\![v]\!]_r\ \ = (\bigsqcup\ i{\in}A \cdot (P(i))[\![v]\!]_r)$

by (*rel-auto*)+

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

lemma *rea-not-R2-closed* [*closure*]:
P is R2 $\Longrightarrow (\neg_r\ P)$ *is R2*
by (*simp add*: *Healthy-def' R1-rea-not' R2-R2c-def R2c-rea-not*)

lemma *rea-no-RR* [*closure*]:
$[\![$ *P is RR* $]\!] \Longrightarrow (\neg_r\ P)$ *is RR*
by (*metis Healthy-def' RR-rea-not*)

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

lemma *rea-impl-R2c* [*closure*]:
  $\llbracket$ *P is R2c*; *Q is R2c* $\rrbracket \Longrightarrow (P \Rightarrow_r Q)$ *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 is R2*; *Q is R2* $\rrbracket \Longrightarrow (P \Rightarrow_r Q)$ *is R2*
  by (*rel-blast*)

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

lemma *conj-RR* [*closure*]:
  $\llbracket$ *P is RR*; *Q is RR* $\rrbracket \Longrightarrow (P \wedge Q)$ *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 is RR*; *Q is RR* $\rrbracket \Longrightarrow (P \vee Q)$ *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 $\bigwedge i.\ i \in A \Longrightarrow P\ i$ *is RR* $A \neq \{\}$
  shows $(\bigsqcup\ i{\in}A \cdot P(i))$ *is RR*
proof −
  have *1*:$(\bigsqcup\ i{\in}A \cdot P(i))$ *is R1*
    by (*unfold Healthy-def*, *subst R1-UINF*, *simp-all add*: *Healthy-if assms closure cong*: *USUP-cong*)
  have *2*:$(\bigsqcup\ i{\in}A \cdot P(i))$ *is R2c*
    by (*unfold Healthy-def*, *subst R2c-UINF*, *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 $\bigwedge i.\ P\ i$ *is RR*
  shows $(\bigsqcup\ i \cdot P(i))$ *is RR*
  using *USUP-mem-RR-closed*[*of UNIV P*] **by** (*simp add*: *assms*)

lemma *UINF-mem-RR-closed* [*closure*]:
  assumes $\bigwedge i.\ P\ i$ *is RR*
  shows $(\bigsqcap\ i{\in}A \cdot P(i))$ *is RR*
proof −
  have *1*:$(\bigsqcap\ i{\in}A \cdot P(i))$ *is R1*
    by (*unfold Healthy-def*, *subst R1-USUP*, *simp-all add*: *Healthy-if assms closure*)
  have *2*:$(\bigsqcap\ i{\in}A \cdot P(i))$ *is R2c*
    by (*unfold Healthy-def*, *subst R2c-USUP*, *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 *UINF-ind-RR-closed* [*closure*]:
  assumes $\bigwedge i.\ P\ i$ *is RR*
  shows $(\bigsqcap\ i \cdot P(i))$ *is RR*

```
  using UINF-mem-RR-closed[of P UNIV] by (simp add: assms)

lemma USUP-elem-RR [closure]:
  assumes ⋀ i. P i is RR A ≠ {}
  shows (⨆ i ∈ A · P i) is RR
proof –
  have 1:(⨆ i∈A · P(i)) is R1
    by (unfold Healthy-def, subst R1-UINF, simp-all add: Healthy-if assms closure)
  have 2:(⨆ i∈A · P(i)) 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 is RR Q is RR
  shows P ;; Q 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 is RR ⟹ P ;; P ˆ i is RR
  by (induct i, simp-all add: closure upred-semiring.power-Suc)

lemma seqr-iter-RR-closed [closure]:
  ⟦ I ≠ []; ⋀ i. i ∈ set(I) ⟹ P(i) is RR ⟧ ⟹ (;; i : I · P(i)) 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 is RR Q is RR
  shows P ◁ $tr´ =ᵤ $tr ▷ Q 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]:
  IIᵣ is RR
  apply (rel-auto) using minus-zero-eq by blast

lemma tr'-eq-tr-RR-closed [closure]: $tr´ =ᵤ $tr is RR
  apply (rel-auto) using minus-zero-eq by auto

lemma inf-RR-closed [closure]:
  ⟦ P is RR; Q is RR ⟧ ⟹ P ⊓ Q is RR
  by (simp add: disj-RR uinf-or)

lemma conj-tr-strict-RR-closed [closure]:
  assumes P is RR
  shows (P ∧ $tr <ᵤ $tr´) is RR
```

```
proof −
  have RR(RR(P) ∧ $tr <u $tr´) = (RR(P) ∧ $tr <u $tr´)
    by (rel-auto)
  thus ?thesis
    by (metis Healthy-def assms)
qed

lemma rea-assert-RR-closed [closure]:
  assumes b is RR
  shows {b}r is RR
  by (simp add: closure assms rea-assert-def)

lemma upower-RR-closed [closure]:
  ⟦ i > 0; P is RR ⟧ ⟹ P ˆ i 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 is RR Q is RR
  shows (P ˆ i) ;; Q 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 is RR Q is RR
  shows P ;; Q⋆ is RR
proof −
  have P ;; Q⋆ = P ;; (⨅ i ∈ {0..} · Q ˆ i)
    by (simp add: ustar-def)
  also have ... = P ;; (II ⊓ (⨅ i ∈ {1..} · Q ˆ i))
    by (metis One-nat-def UINF-atLeast-first upred-semiring.power-0)
  also have ... = (P ∨ P ;; (⨅ i ∈ {1..} · Q ˆ i))
    by (simp add: disj-upred-def[THEN sym] seqr-or-distr)
  also have ... is RR
  proof −
    have (⨅ i ∈ {1..} · Q ˆ 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 ustar-left-RR-closed [closure]:
  assumes P is RR Q is RR
  shows P⋆ ;; Q is RR
proof −
  have P⋆ ;; Q = (⨅ i ∈ {0..} · P ˆ i) ;; Q
    by (simp add: ustar-def)
  also have ... = (II ⊓ (⨅ i ∈ {1..} · P ˆ i)) ;; Q
    by (metis One-nat-def UINF-atLeast-first upred-semiring.power-0)
  also have ... = (Q ∨ (⨅ i ∈ {1..} · P ˆ i) ;; Q)
    by (simp add: disj-upred-def[THEN sym] seqr-or-distl)
  also have ... is RR
  proof −
    have (⨅ i ∈ {1..} · P ˆ 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 ⟹ P⁺ is RR
  by (simp add: uplus-def ustar-right-RR-closed)

lemma trace-ext-prefix-RR [closure]:
  ⟦ $tr ♯ e; $ok ♯ e; $wait ♯ e; outα ♯ e ⟧ ⟹ $tr ˆu e ≤u $tr´ 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-refl)
done

lemma rea-subst-R1-closed [closure]: P⟦v⟧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
```

4.5 Reactive relational calculus

```
lemma rea-skip-unit [rpred]:
  assumes P is RR
  shows P ;; IIr = P IIr ;; P = P
proof −
  have 1: RR(P) ;; IIr = RR(P)
    by (rel-auto)
  have 2: IIr ;; RR(P) = RR(P)
    by (rel-auto)
  from 1 2 show P ;; IIr = P IIr ;; 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 *is R1* $\Longrightarrow (\neg_r \neg_r P) = P$
by (*simp add*: *rea-not-def R1-negate-R1 Healthy-if*)

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

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

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

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

lemma *rea-false-impl* [*rpred*]:
P *is R1* $\Longrightarrow (\mathit{false} \Rightarrow_r P) = true_r$
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*]: $(P \Rightarrow_r \mathit{false}) = (\neg_r\ P)$
by (*rel-simp*)

lemma *rea-imp-refl* [*rpred*]: P *is R1* $\Longrightarrow (P \Rightarrow_r P) = true_r$
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 *is R1*
shows $(P \Rightarrow_r Q) = \mathit{R1}(P \Rightarrow Q)$
proof –
have $(P \Rightarrow_r \mathit{R1}(Q)) = \mathit{R1}(P \Rightarrow Q)$
by (*rel-auto*)
thus *?thesis*
by (*simp add*: *assms Healthy-if*)
qed

lemma *rea-not-true* [*simp*]: $(\neg_r\ \mathit{true}) = \mathit{false}$
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 *is R1* $\Longrightarrow (P \vee \neg_r\ P) = \mathit{true}_r$
by (*rel-blast*)

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

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

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

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

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

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

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

lemma *UINF-ind-rea-true* [*rpred*]: $A \neq \{\} \Longrightarrow (\bigsqcap\ i{\in}A \cdot \mathit{true}_r) = \mathit{true}_r$
by (*rel-auto*)

lemma *UINF-rea-impl*: $(\bigsqcap\ P{\in}A \cdot F(P) \Rightarrow_r G(P)) = ((\bigsqcup\ P{\in}A \cdot F(P)) \Rightarrow_r (\bigsqcap\ P{\in}A \cdot G(P)))$
by (*rel-auto*)

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

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

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

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*]: $I \neq \{\} \Longrightarrow R4(\bigsqcup \; i{\in}I \cdot P(i)) = (\bigsqcup \; i{\in}I \cdot R4(P(i)))$
by (*rel-auto*)

lemma *R5-USUP* [*rpred*]: $I \neq \{\} \Longrightarrow R5(\bigsqcup \; i{\in}I \cdot P(i)) = (\bigsqcup \; i{\in}I \cdot R5(P(i)))$
by (*rel-auto*)

lemma *R4-UINF* [*rpred*]: $R4(\bigsqcap \; i{\in}I \cdot P(i)) = (\bigsqcap \; i{\in}I \cdot R4(P(i)))$
by (*rel-auto*)

lemma *R5-UINF* [*rpred*]: $R5(\bigsqcap \; i{\in}I \cdot P(i)) = (\bigsqcap \; i{\in}I \cdot R5(P(i)))$
by (*rel-auto*)

4.6 UTP theory

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

typedecl *RREL*

abbreviation $RREL \equiv UTHY(RREL, \; ('t{::}trace,'\alpha) \; rp)$

overloading
rrel-hcond == *utp-hcond* :: $(RREL, \; ('t{::}trace,'\alpha) \; rp) \; uthy \Rightarrow (('t,'\alpha) \; rp \times ('t,'\alpha) \; rp) \; health$
rrel-unit == *utp-unit* :: $(RREL, \; ('t{::}trace,'\alpha) \; rp) \; uthy \Rightarrow ('t, \; '\alpha) \; hrel\text{-}rp$
begin
definition *rrel-hcond* :: $(RREL, \; ('t{::}trace,'\alpha) \; rp) \; uthy \Rightarrow (('t,'\alpha) \; rp \times ('t,'\alpha) \; rp) \; health$ **where**
[*upred-defs*]: $rrel\text{-}hcond \; T = RR$
definition *rrel-unit* :: $(RREL, \; ('t{::}trace,'\alpha) \; rp) \; uthy \Rightarrow ('t,'\alpha) \; hrel\text{-}rp$ **where**
[*upred-defs*]: $rrel\text{-}unit \; T = II_r$
end

interpretation *rrel-thy*: *utp-theory-kleene* $UTHY(RREL, \; ('t{::}trace,'\alpha) \; rp)$
rewrites $\bigwedge \; P. \; P \in carrier \; (uthy\text{-}order \; RREL) \longleftrightarrow P \; is \; RR$
and $P \; is \; \mathcal{H}_{RREL} \longleftrightarrow P \; is \; RR$

and $\mathit{carrier}\ (\mathit{uthy\text{-}order}\ \mathit{RREL}) \rightarrow \mathit{carrier}\ (\mathit{uthy\text{-}order}\ \mathit{RREL}) \equiv [\![\mathit{RR}]\!]_H \rightarrow [\![\mathit{RR}]\!]_H$
and $[\![\mathcal{H}_{\mathit{RREL}}]\!]_H \rightarrow [\![\mathcal{H}_{\mathit{RREL}}]\!]_H \equiv [\![\mathit{RR}]\!]_H \rightarrow [\![\mathit{RR}]\!]_H$
and $\top_{\mathit{RREL}} = \mathit{false}$
and $\mathcal{II}_{\mathit{RREL}} = \mathit{II}_r$
and $\mathit{le}\ (\mathit{uthy\text{-}order}\ \mathit{RREL}) = \mathit{op}\ \sqsubseteq$
proof −
interpret *lat*: *utp-theory-continuous UTHY*(*RREL*, ($'t$::*trace*,$'\alpha$) *rp*)
by (*unfold-locales*, *simp-all add*: *rrel-hcond-def rrel-unit-def closure Healthy-if rpred*)
show *1*: $\top_{\mathit{RREL}} = (\mathit{false} :: ('t,'\alpha)\ \mathit{hrel\text{-}rp})$
by (*metis Healthy-if lat.healthy-top rea-no-RR rea-not-rea-true rea-true-RR rrel-hcond-def*)
thus *utp-theory-kleene UTHY*(*RREL*, ($'t$,$'\alpha$) *rp*)
by (*unfold-locales*, *simp-all add*: *rrel-hcond-def rrel-unit-def closure Healthy-if rpred*)
qed (*simp-all add*: *rrel-hcond-def rrel-unit-def closure Healthy-if rpred*)

declare *rrel-thy.top-healthy* [*simp del*]
declare *rrel-thy.bottom-healthy* [*simp del*]

abbreviation *rea-star* :: - ⇒ - ($\text{-}^{\star r}$ [*999*] *999*) **where**
$P^{\star r} \equiv P\star_{\mathit{RREL}}$

4.7 Instantaneous Reactive Relations

Instantaneous Reactive Relations, where the trace stays the same.

abbreviation *Instant* :: ($'t$::*trace*, $'\alpha$) *hrel-rp* ⇒ ($'t$, $'\alpha$) *hrel-rp* **where**
Instant(*P*) ≡ *RID*(*tr*)(*P*)

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

end

5 Reactive Conditions

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

5.1 Healthiness Conditions

definition *RC1* :: ($'t$::*trace*, $'\alpha$, $'\beta$) *rel-rp* ⇒ ($'t$, $'\alpha$, $'\beta$) *rel-rp* **where**
[*upred-defs*]: $\mathit{RC1}(P) = (\neg_r\ (\neg_r\ P)\ ;;\ \mathit{true}_r)$

definition *RC* :: ($'t$::*trace*, $'\alpha$, $'\beta$) *rel-rp* ⇒ ($'t$, $'\alpha$, $'\beta$) *rel-rp* **where**
[*upred-defs*]: $\mathit{RC} = \mathit{RC1} \circ \mathit{RR}$

lemma *RC-intro*: $[\![\ P\ \mathit{is}\ \mathit{RR};\ ((\neg_r\ (\neg_r\ P)\ ;;\ \mathit{true}_r) = P)\ ]\!] \Longrightarrow P\ \mathit{is}\ \mathit{RC}$
by (*simp add*: *Healthy-def RC1-def RC-def*)

lemma *RC-intro'*: $[\![\ P\ \mathit{is}\ \mathit{RR};\ P\ \mathit{is}\ \mathit{RC1}\ ]\!] \Longrightarrow P\ \mathit{is}\ \mathit{RC}$
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*: $P \sqsubseteq Q \Longrightarrow \mathit{RC1}(P) \sqsubseteq \mathit{RC1}(Q)$

```
  by (rel-blast)

lemma RC1-prop:
  assumes P is RC1
  shows (¬r P) ;; R1 true = (¬r P)
proof −
  have (¬r P) = (¬r (RC1 P))
    by (simp add: Healthy-if assms)
  also have ... = (¬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 ¬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 ∘ RR
  by (auto simp add: RC-def fun-eq-iff R1-R2c-commute[THEN sym] R1-R2c-is-R2)

lemma RC-implies-R2: P is RC ⟹ P is R2
  by (metis Healthy-def ′ R2-RC)

lemma RC-ex-ok-wait: (∃ {$ok, $ok´, $wait, $wait´} · 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 is RC s ≤ t
  shows P⟦0,«s»/$tr,$tr´⟧ ⊑ P⟦0,«t»/$tr,$tr´⟧
proof −
  from assms(2) have (RC P)⟦0,«s»/$tr,$tr´⟧ ⊑ (RC P)⟦0,«t»/$tr,$tr´⟧
    apply (rel-auto)
    using dual-order.trans apply blast
    done
  thus ?thesis
    by (simp only: assms(1) Healthy-if)
qed
```

5.2 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 ⟹ P is RC1
  by (metis Healthy-def RC-R2-def RC-implies-RR comp-eq-dest-lhs)
```

lemma *RC1-trace-ext-prefix*:
$out\alpha \sharp e \Longrightarrow RC1(\neg_r \$tr\ \hat{}_u\ e \leq_u \$tr') = (\neg_r \$tr\ \hat{}_u\ e \leq_u \$tr')$
by (*rel-auto*, *blast*, *metis* (*no-types*, *lifting*) *dual-order.trans*)

lemma *RC1-conj*: $RC1(P \wedge Q) = (RC1(P) \wedge RC1(Q))$
by (*rel-blast*)

lemma *conj-RC1-closed* [*closure*]:
$[\![P\ is\ RC1;\ Q\ is\ RC1]\!] \Longrightarrow P \wedge Q\ is\ RC1$
by (*simp add*: *Healthy-def RC1-conj*)

lemma *disj-RC1-closed* [*closure*]:
assumes *P is RC1 Q is RC1*
shows $(P \vee Q)$ *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 is RC Q is RC*
shows $(P \wedge Q)$ *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*]: $true_r$ *is RC*
by (*rel-auto*)

lemma *false-RC* [*closure*]: *false is RC*
by (*rel-auto*)

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

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

lemma *UINF-mem-RC-closed* [*closure*]:
assumes $\bigwedge i.\ P\ i\ is\ RC$
shows $(\bigsqcap\ i{\in}A \cdot P\ i)\ is\ RC$
proof −
have $RC(\bigsqcap\ i{\in}A \cdot P\ i) = (RC1 \circ RR)(\bigsqcap\ i{\in}A \cdot P\ i)$
by (*simp add*: *RC-def*)
also have ... $= RC1(\bigsqcap\ i{\in}A \cdot RR(P\ i))$
by (*rel-blast*)
also have ... $= RC1(\bigsqcap\ i{\in}A \cdot RC1(P\ i))$
by (*simp add*: *Healthy-if RC-implies-RR RC-implies-RC1 assms*)

```
  also have ... = (⨅ i∈A · RC1(P i))
    by (rel-auto, meson order.trans)
  also have ... = (⨅ i∈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 ⋀ i. P i is RC
  shows (⨅ i · P i) 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 ⋀ i. i ∈ A ⟹ P i is RC1 A ≠ {}
  shows (⨆ i∈A · P i) is RC1
proof –
  have RC1(⨆ i∈A · P i) = RC1(⨆ i∈A · RC1(P i))
    by (simp add: Healthy-if assms(1) cong: USUP-cong)
  also from assms(2) have ... = (⨆ i∈A · RC1(P i))
    using dual-order.trans by (rel-blast)
  also have ... = (⨆ i∈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 ⋀ i. i ∈ A ⟹ P i is RC A ≠ {}
  shows (⨆ i∈A · P i) is RC
  by (rule RC-intro', simp-all add: closure assms RC-implies-RC1)

lemma neg-trace-ext-prefix-RC [closure]:
  ⟦ $tr ♯ e; $ok ♯ e; $wait ♯ e; outα ♯ e ⟧ ⟹ ¬ᵣ $tr ^ᵤ e ≤ᵤ $tr´ is RC
  by (rule RC-intro, simp add: closure, metis RC1-def RC1-trace-ext-prefix)

lemma RC1-unrest:
  ⟦ mwb-lens x; x ⋈ tr ⟧ ⟹ $x´ ♯ RC1(P)
  by (simp add: RC1-def unrest)

lemma RC-unrest-dashed [unrest]:
  ⟦ P is RC; mwb-lens x; x ⋈ tr ⟧ ⟹ $x´ ♯ P
  by (metis Healthy-if RC1-unrest RC-implies-RC1)

lemma RC1-RR-closed: P is RR ⟹ RC1(P) is RR
  by (simp add: RC1-def closure)

end
```

6 Reactive Programs

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

6.1 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 is is necessary to use a modification of *R3* from Butterfield et al. [1] that explicitly states that intermediate waiting states do not propogate final state variables. In order to do this we need an additional observational variable that capture the program state that we call *st*. Upon this foundation, we can define operators for reactive programs.

```
alphabet 's rsp-vars = 't rp-vars +
  st :: 's

declare rsp-vars.defs [lens-defs]

type-synonym ('s,'t,'α) rsp = ('t, ('s, 'α) rsp-vars-scheme) rp
type-synonym ('s,'t,'α,'β) rel-rsp  = (('s,'t,'α) rsp, ('s,'t,'β) rsp) urel
type-synonym ('s,'t,'α) hrel-rsp  = ('s,'t,'α) rsp hrel
type-synonym ('s,'t) rdes = ('s,'t,unit) hrel-rsp

translations
  (type) ('s,'t,'α) rsp <= (type) ('t, ('s, 'α) rsp-vars-ext) rp
  (type) ('s,'t,'α) rsp <= (type) ('t, ('s, 'α) rsp-vars-scheme) rp
  (type) ('s,'t,unit) rsp <= (type) ('t, 's rsp-vars) rp
  (type) ('s,'t,'α,'β) rel-rsp <= (type) (('s,'t,'α) rsp, ('s1,'t1,'β) rsp) urel
  (type) ('s,'t,'α) hrel-rsp  <= (type) ('s, 't, 'α) rsp hrel
  (type) ('s,'t) rdes <= (type) ('s, 't, unit) hrel-rsp

notation rsp-vars-child-lens_a (Σ_s)
notation rsp-vars-child-lens (Σ_S)

syntax
  -svid-st-alpha  :: svid (Σ_S)

translations
  -svid-st-alpha => CONST rsp-vars-child-lens

lemma srea-var-ords [usubst]:
  $st ≺_v $st´
  $ok ≺_v $st $ok´ ≺_v $st´ $ok ≺_v $st´ $ok´ ≺_v $st
  $st ≺_v $wait $st´ ≺_v $wait´ $st ≺_v $wait´ $st´ ≺_v $wait
  $st ≺_v $tr $st´ ≺_v $tr´ $st ≺_v $tr´ $st´ ≺_v $tr
  by (simp-all add: var-name-ord-def)

lemma st-bij-lemma: bij-lens (st_a +_L Σ_s)
  by (unfold-locales, auto simp add: lens-defs)

lemma rea-lens-equiv-st-rest: Σ_R ≈_L st +_L Σ_S
proof −
  have st +_L Σ_S = (st_a +_L Σ_s) ;_L Σ_R
    by (simp add: plus-lens-distr st-def rsp-vars-child-lens-def)
  also have ... ≈_L 1_L ;_L Σ_R
    using lens-equiv-via-bij st-bij-lemma by auto
  also have ... = Σ_R
    by (simp)
  finally show ?thesis
    using lens-equiv-sym by blast
```

qed

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

lemma *st-qual-alpha* [*alpha*]: $x ;_L fst_L ;_L st \times_L st = (\$st{:}x)_v$
  by (*metis* (*no-types*, *hide-lams*) *in-var-def in-var-prod-lens lens-comp-assoc st-vwb-lens vwb-lens-wb*)

interpretation *alphabet-state*:
  lens-interp $\lambda(ok, wait, tr, r).\ (ok, wait, tr, st_v\ r, more\ r)$
  apply (*unfold-locales*)
  apply (*rule injI*)
  apply (*clarsimp*)
  done

interpretation *alphabet-state-rel*: *lens-interp* $\lambda(ok, ok', wait, wait', tr, tr', r, r').$
  $(ok, ok', wait, wait', tr, tr', st_v\ r, st_v\ r', more\ r, more\ r')$
  apply (*unfold-locales*)
  apply (*rule injI*)
  apply (*clarsimp*)
  done

lemma *unrest-st′-neg-RC* [*unrest*]:
  assumes *P is RR P is RC*
  shows $\$st' \sharp P$
proof –
  have $P = (\neg_r \neg_r P)$
    by (*simp add*: *closure rpred assms*)
  also have ... $= (\neg_r (\neg_r P) ;; true_r)$
    by (*metis Healthy-if RC1-def RC-implies-RC1 assms*(*2*) *calculation*)
  also have $\$st' \sharp$ ...
    by (*rel-auto*)
  finally show *?thesis* **.**
qed

lemma *ex-st′-RR-closed* [*closure*]:
  assumes *P is RR*
  shows $(\exists\ \$st' \cdot P)$ *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*]:
  $\$st' \sharp P \Longrightarrow \$st' \sharp R4(P)$
  by (*rel-auto*)

lemma *unrest-st′-R5* [*unrest*]:
$\$st\acute{} \sharp P \Longrightarrow \$st\acute{} \sharp R5(P)$
by (*rel-auto*)

6.2 State Lifting

abbreviation *lift-state-rel* ($\lceil \text{-} \rceil_S$)
where $\lceil P \rceil_S \equiv P \oplus_p (st \times_L st)$

abbreviation *drop-state-rel* ($\lfloor \text{-} \rfloor_S$)
where $\lfloor P \rfloor_S \equiv P \upharpoonright_e (st \times_L st)$

abbreviation *lift-state-pre* ($\lceil \text{-} \rceil_{S<}$)
where $\lceil p \rceil_{S<} \equiv \lceil \lceil p \rceil_< \rceil_S$

abbreviation *drop-state-pre* ($\lfloor \text{-} \rfloor_{S<}$)
where $\lfloor p \rfloor_{S<} \equiv \lfloor \lfloor p \rfloor_S \rfloor_<$

abbreviation *lift-state-post* ($\lceil \text{-} \rceil_{S>}$)
where $\lceil p \rceil_{S>} \equiv \lceil \lceil p \rceil_> \rceil_S$

abbreviation *drop-state-post* ($\lfloor \text{-} \rfloor_{S>}$)
where $\lfloor p \rfloor_{S>} \equiv \lfloor \lfloor p \rfloor_S \rfloor_>$

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

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

lemma *R1-st′-unrest* [*unrest*]: $\$st\acute{} \sharp P \Longrightarrow \$st\acute{} \sharp R1(P)$
by (*simp add*: *R1-def unrest*)

lemma *R2c-st′-unrest* [*unrest*]: $\$st\acute{} \sharp P \Longrightarrow \$st\acute{} \sharp R2c(P)$
by (*simp add*: *R2c-def unrest*)

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*)

6.3 Reactive Program Operators

6.3.1 State Substitution

Lifting substitutions on the reactive state

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

abbreviation *st-subst* :: $'s\ usubst \Rightarrow ('s,'t{::}trace,'\alpha,'\beta)\ rel\text{-}rsp \Rightarrow ('s, 't, '\alpha, '\beta)\ rel\text{-}rsp$ (**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 (*auto simp add*: *upred-defs lens-defs prod.case-eq-if*)

lemma *unrest-st-lift* [*unrest*]:
  fixes $x :: 'a \Longrightarrow ('s, 't{::}trace, '\alpha)\ rsp \times ('s, 't, '\alpha)\ rsp$
  assumes $x \bowtie (\$st)_v$
  shows $x \sharp \lceil\sigma\rceil_{S\sigma}$ (**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*]:
  $\lceil id\rceil_{S\sigma} = id$
  by (*pred-auto*)

lemma *st-subst-comp* [*usubst*]:
  $\lceil\sigma\rceil_{S\sigma} \circ \lceil\varrho\rceil_{S\sigma} = \lceil\sigma \circ \varrho\rceil_{S\sigma}$
  by (*rel-auto*)

definition *lift-cond-srea* ($\lceil$-$\rceil_{S\leftarrow}$) **where**
[*upred-defs*]: $\lceil b\rceil_{S\leftarrow} = \lceil b\rceil_{S<}$

lemma *unrest-lift-cond-srea* [*unrest*]:
  $x \sharp \lceil b\rceil_{S<} \Longrightarrow x \sharp \lceil b\rceil_{S\leftarrow}$
  by (*simp add*: *lift-cond-srea-def*)

lemma *st-subst-RR-closed* [*closure*]:
  assumes *P is RR*
  shows $\lceil\sigma\rceil_{S\sigma} \dagger P$ *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 is RC*
  shows $\sigma \dagger_S P$ *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
```

6.3.2 Assignment

```
definition rea-assigns :: ('s ⇒ 's) ⇒ ('s, 't::trace, 'α) hrel-rsp (⟨-⟩ᵣ) where
[upred-defs]: ⟨σ⟩ᵣ = ($tr´ =ᵤ $tr ∧ ⌈⟨σ⟩ₐ⌉S ∧ $ΣS´ =ᵤ $ΣS)

syntax
  -assign-rea :: svids ⇒ uexprs ⇒ logic  ('(-') :=ᵣ '(-'))
  -assign-rea :: svids ⇒ uexprs ⇒ logic  (infixr :=ᵣ 90)

translations
  -assign-rea xs vs => CONST rea-assigns (-mk-usubst (CONST id) xs vs)
  -assign-rea x v <= CONST rea-assigns (CONST subst-upd (CONST id) 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 (CONST id) (CONST
svar x) u) (CONST svar y) v)

lemma rea-assigns-RR-closed [closure]:
  ⟨σ⟩ᵣ is RR
  apply (rel-auto) using minus-zero-eq by auto

lemma st-subst-assigns-rea [usubst]:
  σ †S ⟨ϱ⟩ᵣ = ⟨ϱ ∘ σ⟩ᵣ
  by (rel-auto)

lemma st-subst-rea-skip [usubst]:
  σ †S IIᵣ = ⟨σ⟩ᵣ
  by (rel-auto)

lemma rea-assigns-comp [rpred]:
  assumes P is RR
  shows ⟨σ⟩ᵣ ;; P = σ †S P
proof –
  have ⟨σ⟩ᵣ ;; (RR P) = σ †S (RR P)
    by (rel-auto)
  thus ?thesis
    by (metis Healthy-def assms)
qed

lemma st-subst-RR [closure]:
  assumes P is RR
  shows (σ †S P) is RR
proof –
  have (σ †S RR(P)) is RR
    by (rel-auto)
  thus ?thesis
    by (simp add: Healthy-if assms)
qed

lemma rea-assigns-st-subst [usubst]:
  ⌈σ ⊕ₛ st⌉ₛ † ⟨ϱ⟩ᵣ = ⟨ϱ ∘ σ⟩ᵣ
  by (rel-auto)
```

6.3.3 Conditional

We guard the reactive conditional condition so that it can't be simplified by alphabet laws unless explicitly simplified.

abbreviation *cond-srea* ::
  $('s,'t::trace,'\alpha,'\beta)$ *rel-rsp* $\Rightarrow$
  $'s$ *upred* $\Rightarrow$
  $('s,'t,'\alpha,'\beta)$ *rel-rsp* $\Rightarrow$
  $('s,'t,'\alpha,'\beta)$ *rel-rsp* $((3\text{-} \lhd \text{-} \rhd_R/ \text{-})\ [52,0,53]\ 52)$ **where**
cond-srea $P\ b\ Q \equiv P \lhd \lceil b \rceil_{S\leftarrow} \rhd Q$

lemma *st-cond-assigns* [*rpred*]:
  $\langle\sigma\rangle_r \lhd b \rhd_R \langle\varrho\rangle_r = \langle\sigma \lhd b \rhd_s \varrho\rangle_r$
  by (*rel-auto*)

lemma *cond-srea-RR-closed* [*closure*]:
  assumes *P is RR Q is RR*
  shows $P \lhd b \rhd_R Q$ *is RR*
proof –
  have $RR(RR(P) \lhd b \rhd_R RR(Q)) = RR(P) \lhd b \rhd_R RR(Q)$
    by (*rel-auto*)
  thus *?thesis*
    by (*metis Healthy-def′ assms*(*1*) *assms*(*2*))
qed

lemma *cond-srea-RC1-closed*:
  assumes *P is RC1 Q is RC1*
  shows $P \lhd b \rhd_R Q$ *is RC1*
proof –
  have $RC1(RC1(P) \lhd b \rhd_R RC1(Q)) = RC1(P) \lhd b \rhd_R RC1(Q)$
    using *dual-order.trans* **by** (*rel-blast*)
  thus *?thesis*
    by (*metis Healthy-def′ assms*)
qed

lemma *cond-srea-RC-closed* [*closure*]:
  assumes *P is RC Q is RC*
  shows $P \lhd b \rhd_R Q$ *is RC*
  by (*rule RC-intro′*, *simp-all add*: *closure cond-srea-RC1-closed RC-implies-RC1 assms*)

lemma *R4-cond* [*rpred*]: $R4(P \lhd b \rhd_R Q) = (R4(P) \lhd b \rhd_R R4(Q))$
  by (*rel-auto*)

lemma *R5-cond* [*rpred*]: $R5(P \lhd b \rhd_R Q) = (R5(P) \lhd b \rhd_R R5(Q))$
  by (*rel-auto*)

6.3.4 Assumptions

definition *rea-assume* :: $'s$ *upred* $\Rightarrow$ $('s,\ 't::trace,\ '\alpha)$ *hrel-rsp* $([\text{-}]^{\top}{}_r)$ **where**
[*upred-defs*]: $[b]^{\top}{}_r = (II_r \lhd b \rhd_R false)$

lemma *rea-assume-RR* [*closure*]: $[b]^{\top}{}_r$ *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*)

6.3.5 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 $lmap_R$ :: $('a \Longrightarrow 'b) \Rightarrow ('t{::}trace,\ 'a)\ rp \Longrightarrow ('t,\ 'b)\ rp$ **where**
[*lens-defs*]: $lmap_R = lmap_D \circ lmap[rp\text{-}vars]$

definition *map-rsp-st* ::
$('\sigma \Rightarrow '\tau) \Rightarrow$
$('\sigma,\ '\alpha)$ *rsp-vars-scheme* $\Rightarrow$ $('\tau,\ '\alpha)$ *rsp-vars-scheme* **where**
[*lens-defs*]: *map-rsp-st f* $= (\lambda r.\ (\!|st_v = f\ (st_v\ r),\ \ldots = rsp\text{-}vars.more\ r|\!))$

definition *map-st-lens* ::
$('\sigma \Longrightarrow '\psi) \Rightarrow$
$(('\sigma,\ '\tau{::}trace,\ '\alpha)\ rsp \Longrightarrow ('\psi,\ '\tau{::}trace,\ '\alpha)\ rsp)$ $(map'\text{-}st_L)$ **where**
[*lens-defs*]:
map-st-lens l $= lmap_R\ (\!|$
lens-get = *map-rsp-st* (get_l),
lens-put = *map-rsp-st o* (put_l) *o rsp-vars.*$st_v|\!)$

lemma *map-set-vwb* [*simp*]: *vwb-lens X* $\Longrightarrow$ *vwb-lens* $(map\text{-}st_L\ X)$
apply (*unfold-locales*, *simp-all add*: *lens-defs*)
apply (*metis des-vars.surjective rp-vars.surjective rsp-vars.surjective*)+
done

abbreviation $abs\text{-}st_L \equiv (map\text{-}st_L\ 0_L) \times_L (map\text{-}st_L\ 0_L)$

abbreviation *abs-st* $(\langle\text{-}\rangle_S)$ **where**
$abs\text{-}st\ P \equiv P \upharpoonright_e abs\text{-}st_L$

6.3.6 Reactive Frames and Extensions

definition *rea-frame* :: $('a \Longrightarrow '\alpha) \Rightarrow ('\alpha,\ 't{::}trace)\ rdes \Rightarrow ('\alpha,\ 't)\ rdes$ **where**
[*upred-defs*]: *rea-frame x P* = *frame* $(ok +_L wait +_L tr +_L (x\ ;_L\ st))\ P$

definition *rea-frame-ext* :: $('\alpha \Longrightarrow '\beta) \Rightarrow ('\alpha,\ 't{::}trace)\ rdes \Rightarrow ('\beta,\ 't)\ rdes$ **where**
[*upred-defs*]: *rea-frame-ext a P* = *rea-frame a* (*rel-aext P* $(map\text{-}st_L\ a)$)

syntax
-rea-frame :: *salpha* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $(\text{-}{:}[\text{-}]_r\ [99,0]\ 100)$
-rea-frame-ext :: *salpha* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $(\text{-}{:}[\text{-}]_r{}^{+}\ [99,0]\ 100)$

translations
-rea-frame x P => *CONST rea-frame x P*
-rea-frame (*-salphaset* (*-salphamk x*)) *P* <= *CONST rea-frame x P*
-rea-frame-ext x P => *CONST rea-frame-ext x P*
-rea-frame-ext (*-salphaset* (*-salphamk x*)) *P* <= *CONST rea-frame-ext x P*

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-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 *rea-frame-ext-RR-closed* [*closure*]:
  P *is RR* $\Longrightarrow x{:}[P]_r{}^+$ *is RR*
  by (*simp add*: *rea-frame-ext-def closure*)

lemma *rel-aext-st-Instant-closed* [*closure*]:
  P *is Instant* $\Longrightarrow$ *rel-aext P* (*map-st*$_L$ x) *is Instant*
  by (*rel-auto*)

lemma *rea-frame-ext-false* [*frame*]:
  $x{:}[\mathit{false}]_r{}^+ = \mathit{false}$
  by (*rel-auto*)

lemma *rea-frame-ext-skip* [*frame*]:
  vwb-lens $x \Longrightarrow x{:}[II_r]_r{}^+ = II_r$
  by (*rel-auto*)

lemma *rea-frame-ext-assigns* [*frame*]:
  vwb-lens $x \Longrightarrow x{:}[\langle\sigma\rangle_r]_r{}^+ = \langle\sigma \oplus_s x\rangle_r$
  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*]:
  vwb-lens $x \Longrightarrow x{:}[P\ ;;\ Q]_r{}^+ = x{:}[P]_r{}^+\ ;;\ x{:}[Q]_r{}^+$
  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 $x \bowtie y$ $\Sigma \sharp v$ P *is RR*

shows $\sigma(y \mapsto_s v) \dagger_S x{:}[P]_r{}^+ = (\sigma \dagger_S x{:}[P]_r{}^+) \;;\; y :=_r v$
proof $-$
from *assms(1−2)* **have** $\sigma(y \mapsto_s v) \dagger_S x{:}[RR\ P]_r{}^+ = (\sigma \dagger_S x{:}[RR\ P]_r{}^+) \;;\; y :=_r v$
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 *vwb-lens x vwb-lens y* $\Sigma \sharp v$ *P is RR*
shows $\sigma(x{:}y \mapsto_s v) \dagger_S x{:}[P]_r{}^+ = (\sigma \dagger_S x{:}[y :=_r (v \upharpoonright_e x) \;;\; P]_r{}^+)$
proof $-$
from *assms(1,3)* **have** $\sigma(x{:}y \mapsto_s v) \dagger_S x{:}[RR\ P]_r{}^+ = (\sigma \dagger_S x{:}[y :=_r (v \upharpoonright_e x) \;;\; RR(P)]_r{}^+)$
by (*rel-auto*, *metis*+)
thus *?thesis*
by (*simp add*: *assms Healthy-if*)
qed

6.4 Stateful Reactive specifications

definition *rea-st-rel* :: *'s hrel* ⇒ (*'s*, *'t::trace*, *'α*, *'β*) *rel-rsp* ($[\text{-}]_S$) **where**
[*upred-defs*]: *rea-st-rel b* = $(\lceil b \rceil_S \wedge \$tr' =_u \$tr)$

definition *rea-st-rel'* :: *'s hrel* ⇒ (*'s*, *'t::trace*, *'α*, *'β*) *rel-rsp* ($[\text{-}]_S{}'$) **where**
[*upred-defs*]: *rea-st-rel' b* = $R1(\lceil b \rceil_S)$

definition *rea-st-cond* :: *'s upred* ⇒ (*'s*, *'t::trace*, *'α*, *'β*) *rel-rsp* ($[\text{-}]_{S<}$) **where**
[*upred-defs*]: *rea-st-cond b* = $R1(\lceil b \rceil_{S<})$

definition *rea-st-post* :: *'s upred* ⇒ (*'s*, *'t::trace*, *'α*, *'β*) *rel-rsp* ($[\text{-}]_{S>}$) **where**
[*upred-defs*]: *rea-st-post b* = $R1(\lceil b \rceil_{S>})$

lemma *lift-state-pre-unrest* [*unrest*]: $x \bowtie (\$st)_v \Longrightarrow x \sharp \lceil P \rceil_{S<}$
by (*rel-simp*, *simp add*: *lens-indep-def*)

lemma *rea-st-rel-unrest* [*unrest*]:
$[\![\ x \bowtie (\$tr)_v;\ x \bowtie (\$tr')_v;\ x \bowtie (\$st)_v;\ x \bowtie (\$st')_v\ ]\!] \Longrightarrow x \sharp [P]_{S<}$
by (*simp add*: *add*: *rea-st-cond-def R1-def unrest lens-indep-sym*)

lemma *rea-st-cond-unrest* [*unrest*]:
$[\![\ x \bowtie (\$tr)_v;\ x \bowtie (\$tr')_v;\ x \bowtie (\$st)_v\ ]\!] \Longrightarrow x \sharp [P]_{S<}$
by (*simp add*: *add*: *rea-st-cond-def R1-def unrest lens-indep-sym*)

lemma *subst-st-cond* [*usubst*]: $\lceil \sigma \rceil_{S\sigma} \dagger [P]_{S<} = [\sigma \dagger P]_{S<}$
by (*rel-auto*)

lemma *rea-st-cond-R1* [*closure*]: $[b]_{S<}$ *is R1*
by (*rel-auto*)

lemma *rea-st-cond-R2c* [*closure*]: $[b]_{S<}$ *is R2c*
by (*rel-auto*)

lemma *rea-st-rel-RR* [*closure*]: $[P]_S$ *is RR*
using *minus-zero-eq* **by** (*rel-auto*)

lemma *rea-st-rel'-RR* [*closure*]: $[P]_S{}'$ *is RR*

by (*rel-auto*)

lemma *st-subst-rel* [*usubst*]:
$\sigma \dagger_S [P]_S = [\lceil \sigma \rceil_s \dagger P]_S$
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*]: $[\mathit{false}]_S = \mathit{false}$
by (*rel-auto*)

lemma *st-rel-skip* [*rpred*]:
$[II]_S = (II_r :: ('s, 't{::}\mathit{trace})\ \mathit{rdes})$
by (*rel-auto*)

lemma *st-rel-seq* [*rpred*]:
$[P \;;;\; Q]_S = [P]_S \;;;\; [Q]_S$
by (*rel-auto*)

lemma *st-rel-conj* [*rpred*]:
$[P \wedge Q]_S = ([P]_S \wedge [Q]_S)$
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*]: $[\mathit{true}]_{S<} = \mathit{true}_r$
by (*rel-auto*)

lemma *rea-st-cond-false* [*rpred*]: $[\mathit{false}]_{S<} = \mathit{false}$
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*]:
$[\langle \sigma \rangle_a]_S = (\langle \sigma \rangle_r :: ('\alpha, 't{::}\mathit{trace})\ \mathit{rdes})$
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 *is R1* $\Longrightarrow P \triangleleft b \triangleright_R \mathit{false} = ([b]_{S<} \wedge P)$
by (*rel-blast*)

lemma *cond-st-true* [*rpred*]: $P \triangleleft \mathit{true} \triangleright_R Q = P$
by (*rel-blast*)

lemma *cond-st-false* [*rpred*]: $P \triangleleft \mathit{false} \triangleright_R Q = Q$
by (*rel-blast*)

lemma *st-cond-true-or* [*rpred*]: $P\ \mathit{is}\ R1 \Longrightarrow (R1\ \mathit{true} \triangleleft b \triangleright_R P) = ([b]_{S<} \vee P)$
by (*rel-blast*)

lemma *st-cond-left-impl-RC-closed* [*closure*]:
$P\ \mathit{is}\ RC \Longrightarrow ([b]_{S<} \Rightarrow_r P)\ \mathit{is}\ RC$
by (*simp add*: *rea-impl-def rpred closure*)

end

7 Reactive Weakest Preconditions

theory *utp-rea-wp*
imports *utp-rea-prog*
begin

definition *wp-rea* ::
$('t::\mathit{trace}, '\alpha)\ \mathit{hrel\text{-}rp} \Rightarrow$
$('t, '\alpha)\ \mathit{hrel\text{-}rp} \Rightarrow$
$('t, '\alpha)\ \mathit{hrel\text{-}rp}$ (**infix** wp_r *60*)
where [*upred-defs*]: $P\ \mathit{wp}_r\ Q = (\neg_r\ P\ ;;\ (\neg_r\ Q))$

lemma *in-var-unrest-wp-rea* [*unrest*]: $[\![\ \$x \sharp P;\ tr \bowtie x\ ]\!] \Longrightarrow \$x \sharp (P\ \mathit{wp}_r\ Q)$
by (*simp add*: *wp-rea-def unrest R1-def rea-not-def*)

lemma *out-var-unrest-wp-rea* [*unrest*]: $[\![\ \$x' \sharp Q;\ tr \bowtie x\ ]\!] \Longrightarrow \$x' \sharp (P\ \mathit{wp}_r\ Q)$
by (*simp add*: *wp-rea-def unrest R1-def rea-not-def*)

lemma *wp-rea-R1* [*closure*]: $P\ \mathit{wp}_r\ Q\ \mathit{is}\ R1$
by (*rel-auto*)

lemma *wp-rea-RR-closed* [*closure*]: $[\![\ P\ \mathit{is}\ RR;\ Q\ \mathit{is}\ RR\ ]\!] \Longrightarrow P\ \mathit{wp}_r\ Q\ \mathit{is}\ RR$
by (*simp add*: *wp-rea-def closure*)

lemma *wp-rea-impl-lemma*:
$((P\ \mathit{wp}_r\ Q) \Rightarrow_r (R1(P)\ ;;\ R1(Q \Rightarrow_r R))) = ((P\ \mathit{wp}_r\ Q) \Rightarrow_r (R1(P)\ ;;\ R1(R)))$
by (*rel-auto*, *blast*)

lemma *wpR-R1-right* [*wp*]:
$P\ \mathit{wp}_r\ R1(Q) = P\ \mathit{wp}_r\ Q$
by (*rel-auto*)

lemma *wp-rea-true* [*wp*]: $P\ \mathit{wp}_r\ \mathit{true} = \mathit{true}_r$
by (*rel-auto*)

lemma *wp-rea-conj* [*wp*]: $P\ \mathit{wp}_r\ (Q \wedge R) = (P\ \mathit{wp}_r\ Q \wedge P\ \mathit{wp}_r\ R)$
by (*simp add*: *wp-rea-def seqr-or-distr*)

lemma *wp-rea-USUP-mem* [*wp*]:
$A \neq \{\} \Longrightarrow P\ \mathit{wp}_r\ (\bigsqcup\ i{\in}A \cdot Q(i)) = (\bigsqcup\ i{\in}A \cdot P\ \mathit{wp}_r\ Q(i))$
by (*simp add*: *wp-rea-def seq-UINF-distl*)

```
lemma wp-rea-Inf-pre [wp]:
  P wpᵣ (⨆ i∈{0..n::nat}. Q(i)) = (⨆ i∈{0..n}. P wpᵣ Q(i))
  by (simp add: wp-rea-def seq-SUP-distl)

lemma wp-rea-div [wp]:
  (¬ᵣ P ;; trueᵣ) = trueᵣ ⟹ trueᵣ wpᵣ P = false
  by (simp add: wp-rea-def rpred, rel-blast)

lemma wp-rea-st-cond-div [wp]:
  P ≠ true ⟹ trueᵣ wpᵣ [P]S< = false
  by (rel-auto)

lemma wp-rea-cond [wp]:
  outα ♯ b ⟹ (P ◃ b ▹ Q) wpᵣ R = P wpᵣ R ◃ b ▹ Q wpᵣ R
  by (simp add: wp-rea-def cond-seq-left-distr, rel-auto)

lemma wp-rea-RC-false [wp]:
  P is RC ⟹ (¬ᵣ P) wpᵣ false = P
  by (metis Healthy-if RC1-def RC-implies-RC1 rea-not-false wp-rea-def)

lemma wp-rea-seq [wp]:
  assumes Q is R1
  shows (P ;; Q) wpᵣ R = P wpᵣ (Q wpᵣ R) (is ?lhs = ?rhs)
proof −
  have ?rhs = R1 (¬ P ;; R1 (Q ;; R1 (¬ R)))
    by (simp add: wp-rea-def rea-not-def R1-negate-R1 assms)
  also have ... = R1 (¬ P ;; (Q ;; R1 (¬ R)))
    by (metis Healthy-if R1-seqr assms)
  also have ... = R1 (¬ (P ;; Q) ;; R1 (¬ 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 is R1
  shows II wpᵣ Q = Q
  by (simp add: wp-rea-def rpred assms Healthy-if)

lemma wp-rea-rea-skip [wp]:
  assumes Q is RR
  shows IIᵣ wpᵣ Q = Q
  by (simp add: wp-rea-def rpred closure assms Healthy-if)

lemma power-wp-rea-RR-closed [closure]:
  ⟦ R is RR; P is RR ⟧ ⟹ R ^ i wpᵣ P is RR
  by (induct i, simp-all add: wp closure)

lemma wp-rea-rea-assigns [wp]:
  assumes P is RR
  shows ⟨σ⟩ᵣ wpᵣ P = ⌈σ⌉Sσ † P
proof −
  have ⟨σ⟩ᵣ wpᵣ (RR P) = ⌈σ⌉Sσ † (RR P)
    by (rel-auto)
  thus ?thesis
```

by (*metis Healthy-def assms*)
qed

lemma *wp-rea-miracle* [*wp*]: $\mathit{false}\ wp_r\ Q = \mathit{true}_r$
by (*simp add*: *wp-rea-def*)

lemma *wp-rea-disj* [*wp*]: $(P \lor Q)\ wp_r\ R = (P\ wp_r\ R \land Q\ wp_r\ R)$
by (*rel-blast*)

lemma *wp-rea-UINF* [*wp*]:
assumes $A \neq \{\}$
shows $(\bigsqcap\ x{\in}A \cdot P(x))\ wp_r\ Q = (\bigsqcup\ x{\in}A \cdot P(x)\ wp_r\ Q)$
by (*simp add*: *wp-rea-def rea-not-def seq-UINF-distr not-UINF R1-UINF assms*)

lemma *wp-rea-choice* [*wp*]:
$(P \sqcap Q)\ wp_r\ R = (P\ wp_r\ R \land Q\ wp_r\ R)$
by (*rel-blast*)

lemma *wp-rea-UINF-ind* [*wp*]:
$(\bigsqcap\ i \cdot P(i))\ wp_r\ Q = (\bigsqcup\ i \cdot P(i)\ wp_r\ Q)$
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 $([b]^{\top}{}_r\ wp_r\ P) = ([b]_{S<} \Rightarrow_r P)$
proof −
have $([b]^{\top}{}_r\ wp_r\ RC\ P) = ([b]_{S<} \Rightarrow_r RC\ P)$
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 $P^{\star r}\ wp_r\ Q = (\bigsqcup\ i \cdot P\ \hat{}\ i\ wp_r\ Q)$
proof −
have $P^{\star r}\ wp_r\ Q = (Q \land P^{+}\ wp_r\ Q)$
by (*simp add*: *assms rrel-thy.Star-alt-def wp-rea-choice wp-rea-rea-skip*)
also have ... $= (II\ wp_r\ Q \land (\bigsqcup\ i \cdot P\ \hat{}\ Suc\ i\ wp_r\ Q))$
by (*simp add*: *uplus-power-def wp closure assms*)
also have ... $= (\bigsqcup\ i \cdot P\ \hat{}\ i\ wp_r\ Q)$
proof −
have $P^{\star}\ wp_r\ Q = P^{\star r}\ wp_r\ Q$
by (*metis* (*no-types*) *RA1 assms*(*2*) *rea-no-RR rea-skip-unit*(*2*) *rrel-thy.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*]:
$[\![\ P\ is\ R2;\ Q\ is\ R2\ ]\!] \Longrightarrow P\ wp_r\ Q\ is\ R2$
by (*simp add*: *wp-rea-def closure*)

lemma *wp-rea-false-RC1′*: $P\ is\ R2 \Longrightarrow RC1(P\ wp_r\ false) = P\ wp_r\ false$

  by (*simp add*: *wp-rea-def RC1-def closure rpred seqr-assoc*)

lemma *wp-rea-false-RC1*: *P is R2* $\Longrightarrow$ *P* wp_r *false is RC1*
  by (*simp add* :*Healthy-def wp-rea-false-RC1'*)

lemma *wp-rea-false-RR*:
  $[\![$ *\$ok* $\sharp$ *P*; *\$wait* $\sharp$ *P*; *P is R2* $]\!]$ $\Longrightarrow$ *P* wp_r *false is RR*
  by (*rule RR-R2-intro*, *simp-all add*: *unrest closure*)

lemma *wp-rea-false-RC*:
  $[\![$ *\$ok* $\sharp$ *P*; *\$wait* $\sharp$ *P*; *P is R2* $]\!]$ $\Longrightarrow$ *P* wp_r *false is RC*
  by (*rule RC-intro'*, *simp-all add*: *wp-rea-false-RC1 wp-rea-false-RR*)

lemma *wp-rea-RC1*: $[\![$ *P is RR*; *Q is RC* $]\!]$ $\Longrightarrow$ *P* wp_r *Q is RC1*
 by (*rule Healthy-intro*, *simp add*: *wp-rea-def RC1-def rpred closure seqr-assoc RC1-prop RC-implies-RC1*)

lemma *wp-rea-RC* [*closure*]: $[\![$ *P is RR*; *Q is RC* $]\!]$ $\Longrightarrow$ *P* wp_r *Q is RC*
  by (*rule RC-intro'*, *simp-all add*: *wp-rea-RC1 closure*)

lemma *wpR-power-RC-closed* [*closure*]:
  assumes *P is RR Q is RC*
  shows *P* ^ *i* wp_r *Q 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

8 Reactive Hoare Logic

theory *utp-rea-hoare*
  imports *utp-rea-prog*
begin

definition *hoare-rp* :: *'α upred* ⇒ (*'α*, *real pos*) *rdes* ⇒ *'α upred* ⇒ *bool* ($\{\!|$-$|\!\}$/ -/ $\{\!|$-$|\!\}_r$) **where**
[*upred-defs*]: *hoare-rp p Q r* = (($\lceil p\rceil_{S<} \Rightarrow \lceil r\rceil_{S>}$) $\sqsubseteq$ *Q*)

lemma *hoare-rp-conseq*:
  $[\![$ ‘*p* ⇒ *p'*‘; ‘*q'* ⇒ *q*‘; $\{\!|p'|\!\}S\{\!|q'|\!\}_r$ $]\!]$ $\Longrightarrow$ $\{\!|p|\!\}S\{\!|q|\!\}_r$
  by (*rel-auto*)

lemma *hoare-rp-weaken*:
  $[\![$ ‘*p* ⇒ *p'*‘; $\{\!|p'|\!\}S\{\!|q|\!\}_r$ $]\!]$ $\Longrightarrow$ $\{\!|p|\!\}S\{\!|q|\!\}_r$
  by (*rel-auto*)

lemma *hoare-rp-strengthen*:
  $[\![$ ‘*q'* ⇒ *q*‘; $\{\!|p|\!\}S\{\!|q'|\!\}_r$ $]\!]$ $\Longrightarrow$ $\{\!|p|\!\}S\{\!|q|\!\}_r$
  by (*rel-auto*)

lemma *false-pre-hoare-rp* [*hoare-safe*]: $\{\!|false|\!\}P\{\!|q|\!\}_r$
  by (*rel-auto*)

lemma *true-post-hoare-rp* [*hoare-safe*]: $\{\!|p|\!\}Q\{\!|true|\!\}_r$
  by (*rel-auto*)

lemma *miracle-hoare-rp* [*hoare-safe*]: $\{\!|p|\!\}false\{\!|q|\!\}_r$

by (*rel-auto*)

lemma *assigns-hoare-rp* [*hoare-safe*]: $`p \Rightarrow \sigma \dagger q` \Longrightarrow \{\!|p|\!\}\langle\sigma\rangle_r\{\!|q|\!\}_r$
by *rel-auto*

lemma *skip-hoare-rp* [*hoare-safe*]: $\{\!|p|\!\}II_r\{\!|p|\!\}_r$
by *rel-auto*

lemma *seq-hoare-rp*: $[\![\ \{\!|p|\!\}Q_1\{\!|s|\!\}_r\ ;\ \{\!|s|\!\}Q_2\{\!|r|\!\}_r\ ]\!] \Longrightarrow \{\!|p|\!\}Q_1\ ;;\ Q_2\{\!|r|\!\}_r$
by (*rel-auto*)

lemma *seq-est-hoare-rp* [*hoare-safe*]:
$[\![\ \{\!|true|\!\}Q_1\{\!|p|\!\}_r\ ;\ \{\!|p|\!\}Q_2\{\!|p|\!\}_r\ ]\!] \Longrightarrow \{\!|true|\!\}Q_1\ ;;\ Q_2\{\!|p|\!\}_r$
by (*rel-auto*)

lemma *seq-inv-hoare-rp* [*hoare-safe*]:
$[\![\ \{\!|p|\!\}Q_1\{\!|p|\!\}_r\ ;\ \{\!|p|\!\}Q_2\{\!|p|\!\}_r\ ]\!] \Longrightarrow \{\!|p|\!\}Q_1\ ;;\ Q_2\{\!|p|\!\}_r$
by (*rel-auto*)

lemma *cond-hoare-rp* [*hoare-safe*]:
$[\![\ \{\!|b \wedge p|\!\}P\{\!|r|\!\}_r;\ \{\!|\neg b \wedge p|\!\}Q\{\!|r|\!\}_r\ ]\!] \Longrightarrow \{\!|p|\!\}P \triangleleft b \triangleright_R Q\{\!|r|\!\}_r$
by (*rel-auto*)

lemma *repeat-hoare-rp* [*hoare-safe*]:
$\{\!|p|\!\}Q\{\!|p|\!\}_r \Longrightarrow \{\!|p|\!\}Q\ \hat{}\ n\{\!|p|\!\}_r$
by (*induct n*, *rel-auto*+)

lemma *UINF-ind-hoare-rp* [*hoare-safe*]:
$[\![\ \bigwedge i.\ \{\!|p|\!\}Q(i)\{\!|r|\!\}_r\ ]\!] \Longrightarrow \{\!|p|\!\}\bigsqcap\ i \cdot Q(i)\{\!|r|\!\}_r$
by (*rel-auto*)

lemma *star-hoare-rp* [*hoare-safe*]:
$\{\!|p|\!\}Q\{\!|p|\!\}_r \Longrightarrow \{\!|p|\!\}Q^{\star}\{\!|p|\!\}_r$
by (*simp add*: *ustar-def hoare-safe*)

lemma *conj-hoare-rp* [*hoare-safe*]:
$[\![\ \{\!|p_1|\!\}Q_1\{\!|r_1|\!\}_r;\ \{\!|p_2|\!\}Q_2\{\!|r_2|\!\}_r\ ]\!] \Longrightarrow \{\!|p_1 \wedge p_2|\!\}Q_1 \wedge Q_2\{\!|r_1 \wedge r_2|\!\}_r$
by (*rel-auto*)

lemma *iter-hoare-rp* [*hoare-safe*]:
$\{\!|I|\!\}\ P\ \{\!|I|\!\}_r \Longrightarrow \{\!|I|\!\}\ P^{\star r}\ \{\!|I|\!\}_r$
by (*simp add*: *star-hoare-rp utp-star-def rrel-unit-def seq-inv-hoare-rp skip-hoare-rp*)

end

9 Meta-theory for Generalised Reactive Processes

theory *utp-reactive*
imports
utp-rea-core
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, J. Woodcock, and F. Zeyda. Unifying theories of time with generalised reactive processes. *Accepted for Information Processing Letters*, Dec 2017. Preprint: https://arxiv.org/abs/1712.10213.

[4] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.