This is a repository copy of *Generalised Reactive Processes in Isabelle/UTP*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/129381/

---

**Monograph:**

---

**Reuse**
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

**Takedown**
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.
Abstract
Hoare and He’s UTP theory of reactive processes provides a unifying foundation for the semantics of process calculi and reactive programming. A reactive process is a form of UTP relation which can refer to both state variables and also a trace history of events. In their original presentation, a trace was modelled solely by a discrete sequence of events. Here, we generalise the trace model using “trace algebra”, which characterises traces abstractly using cancellative monoids, and thus enables application of the theory to a wider family of computational models, including hybrid computation. We recast the reactive healthiness conditions in this setting, and prove all the associated distributivity laws. We tackle parallel composition of reactive processes using the “parallel-by-merge” scheme from UTP. We also identify the associated theory of “reactive relations”, and use it to define generic reactive laws, a Hoare logic, and a weakest precondition calculus.

Contents

1 Reactive Processes Core Definitions 2
  1.1 Alphabet and Signature 2
  1.2 Reactive Lemmas 4
  1.3 Trace contribution lens 5

2 Reactive Healthiness Conditions 6
  2.1 R1: Events cannot be undone 6
  2.2 R2: No dependence upon trace history 9
  2.3 R3: No activity while predecessor is waiting 17
  2.4 R4: The trace strictly increases 19
  2.5 R5: The trace does not increase 20
  2.6 RP laws 20
  2.7 UTP theories 21

3 Reactive Parallel-by-Merge 22

4 Reactive Relations 26
  4.1 Healthiness Conditions 26
  4.2 Reactive relational operators 27
  4.3 Unrestriction and substitution laws 28
  4.4 Closure laws 29
  4.5 Reactive relational calculus 34
  4.6 UTP theory 36
  4.7 Instantaneous Reactive Relations 37
5 Reactive Conditions
5.1 Healthiness Conditions .................................. 38
5.2 Closure laws ........................................... 39

6 Reactive Programs ........................................ 41
6.1 Stateful reactive alphabet .................................. 41
6.2 State Lifting ........................................... 43
6.3 Reactive Program Operators ............................... 44
  6.3.1 State Substitution .................................... 44
  6.3.2 Assignment .......................................... 45
  6.3.3 Conditional .......................................... 46
  6.3.4 Assumptions .......................................... 47
  6.3.5 State Abstraction .................................... 47
  6.3.6 Reactive Frames and Extensions ....................... 48
6.4 Stateful Reactive specifications ............................. 49

7 Reactive Weakest Preconditions .................................. 51

8 Reactive Hoare Logic ........................................ 54

9 Meta-theory for Generalised Reactive Processes .................. 56

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 [4], and thus provides the theory of “generalised reactive processes” [4]. The reactive process alphabet also extends the design alphabet, and thus includes the \( ok \) variable. For more information on these, see the UTP book [5], or the associated tutorial [2].

\[
\text{algebra } t::trace \text{ rp-vars } = \text{ des-vars } +
\]

\[
\begin{align*}
\text{wait} &:: \text{ bool} \\
\text{tr} &:: t
\end{align*}
\]

\[
\text{type-synonym } \left( t, 'a \right) \text{ rp } = \left( t, 'a \right) \text{ rp-vars-scheme des}
\]

\[
\text{type-synonym } \left( t,'a,'b \right) \text{ rel-rp } = \left( \left( t,'a \right) \text{ rp}, \left( t,'b \right) \text{ rp} \right) \text{ urel}
\]

\[
\text{type-synonym } \left( t,'a \right) \text{ hrel-rp } = \left( t,'a \right) \text{ rp hrel}
\]

\[
\text{translations}
\]

\[
\left( \text{type} \right) \left( t,'a \right) \text{ rp } \leq \left( \text{type} \right) \left( t,'a \right) \text{ rp-vars-scheme des}
\]

\[
\left( \text{type} \right) \left( t,'a \right) \text{ rp } \leq \left( \text{type} \right) \left( t,'a \right) \text{ rp-vars-ext des}
\]
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 \((\Sigma_r)\)

**notation** rp-vars-child-lens \((\Sigma_R)\)

**syntax**
- svid-rea-alpha :: svid \((\Sigma R)\)

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

Lens \(\Sigma_R\) exists because reactive alphabets are extensible. \(\Sigma_R\) 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 \(\lambda(\text{ok}, r). (\text{ok}, \text{wait} v r, \text{tr} v r, \text{more} r)\)
  - apply (unfold-locales)
  - apply (rule injI)
  - apply (clarsimp)
  - done

**interpretation** rp-vars-rel; lens-interp \(\lambda(\text{ok}, \text{ok}', r, r').\)
- (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]:
- \(\text{tr} \prec v \text{tr}' \prec v \text{wait} \prec v \text{wait}'\)
- \(\text{ok} \prec v \text{tr} \prec v \text{ok}' \prec v \text{tr}' \prec v \text{ok} \prec v \text{tr}' \prec v \text{ok}' \prec v \text{tr}\)
- \(\text{ok} \prec v \text{wait} \prec v \text{ok}' \prec v \text{wait}' \prec v \text{ok} \prec v \text{wait}' \prec v \text{ok}' \prec v \text{wait}\)
- \(\text{tr} \prec v \text{wait} \text{tr} \prec v \text{wait}' \prec v \text{tr} \prec v \text{wait}' \prec v \text{tr}' \prec v \text{wait}' \prec v \text{wait}\)
  - by (simp-all add: var-name-ord-def)

**abbreviation** wait-f::\((t::\text{trace}, ', \alpha, ', \beta)\) rel-rp => \((t, ', \alpha, ', \beta)\) rel-rp
where \( \text{wait-f} \ R \equiv R[\text{false}/\text{wait}] \)

abbreviation \( \text{wait-t} :: (t::\text{trace}, \alpha, \beta) \text{ rel-rp} \Rightarrow (t, \alpha, \beta) \text{ rel-rp} \)
where \( \text{wait-t} \ R \equiv R[\text{true}/\text{wait}] \)

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

translations
\[
\begin{align*}
P \ t & := \text{CONST subst (CONST subst-upd \text{CONST id} (\text{CONST ivar \text{CONST wait} \ false}))} P \\
P \ t & := \text{CONST subst (CONST subst-upd \text{CONST id} (\text{CONST ivar \text{CONST wait} \ true}))} P
\end{align*}
\]

abbreviation \( \text{lift-rea} :: - \Rightarrow -(\neg [\cdot]_R) \) where
\[
\begin{align*}
\lceil P \rceil_R & \equiv P \oplus p(\Sigma_R \times \Sigma_R)
\end{align*}
\]

abbreviation \( \text{drop-rea} :: (t::\text{trace}, \alpha, \beta) \text{ rel-rp} \Rightarrow (\alpha, \beta) \text{ urel} (\lfloor \neg [\cdot]_R \rceil) \) where
\[
\begin{align*}
\lfloor P \rceil_R & \equiv P \rhd (\Sigma_R \times \Sigma_R)
\end{align*}
\]

abbreviation \( \text{rea-pre-lift} :: - \Rightarrow -(\neg [\cdot]_{R<}) \) where
\[
\begin{align*}
\lceil n \rceil_{R<} & \equiv \lceil \lceil n \rceil < \rceil_R
\end{align*}
\]

1.2 Reactive Lemmas

lemma \( \text{unrest-ok-lift-rea \ [unrest]} : \)
\[
\begin{align*}
\$ok \n\triangleright [P]_R \ $ok' \n\triangleright [P]_R \\
\text{by} (\text{pred-auto})+
\end{align*}
\]

lemma \( \text{unrest-wait-lift-rea \ [unrest]} : \)
\[
\begin{align*}
\$wait \n\triangleright [P]_R \ $wait' \n\triangleright [P]_R \\
\text{by} (\text{pred-auto})+
\end{align*}
\]

lemma \( \text{unrest-tr-lift-rea \ [unrest]} : \)
\[
\begin{align*}
\$tr \n\triangleright [P]_R \ $tr' \n\triangleright [P]_R \\
\text{by} (\text{pred-auto})+
\end{align*}
\]

lemma \( \text{wait-tr-bij-lemma} : \text{bij-lens (wait}_a + L \text{ tr}_a + L \Sigma_r) \)
\[
\begin{align*}
\text{by} (\text{unfold-locales, auto simp add: lens-defs})
\end{align*}
\]

lemma \( \text{des-lens-equiv-wait-tr-rest} : \Sigma_D \approx_L \text{wait}_+L \text{ tr}_+L \Sigma_R \)
\[
\begin{align*}
\text{proof} & \quad \text{have} \text{ wait}_+L \text{ tr}_+L \Sigma_R = (\text{wait}_a + L \text{ tr}_a + L \Sigma_r)_+L \Sigma_D \\
\text{by} (\text{simp add: plus-lens-distr wait-def tr-def rp-vars-child-lens-def}) \\
\text{also have} & \quad \approx_L 1_L : L \Sigma_D \\
\text{using} & \quad \text{lens-equiv-via-bij wait-tr-bij-lemma by auto} \\
\text{also have} & \quad \ldots = \Sigma_D \\
\text{by} (\text{simp}) \\
\text{finally show} & \quad \text{thesis} \\
\text{using} & \quad \text{lens-equiv-sym by blast}
\end{align*}
\]

qed

lemma \( \text{rea-lens-bij} : \text{bij-lens (ok}_+L \text{ wait}_+L \text{ tr}_+L \Sigma_R) \)
\[
\begin{align*}
\text{proof} & \quad \text{have} \text{ ok}_+L \text{ wait}_+L \text{ tr}_+L \Sigma_R \approx_L \text{ok}_+L \Sigma_D \\
\text{using} & \quad \text{des-lens-equiv-wait-tr-rest des-vars-indeps lens-equiv-sym lens-plus-eq-right by blast} \\
\text{also have} & \quad \ldots \approx_L 1_L \\
\text{using} & \quad \text{bij-lens-equiv-id[of ok}_+L \Sigma_D] \text{ by (simp add: ok-des-bij-lens)}
\end{align*}
\]
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 tcontr :: \('t::trace \Rightarrow ('t, 'a) \times ('t, 'a)\) where
  [lens-defs]:
  tcontr = \( \lambda \) (get($\text{str'}$)) \_ s - (get($\text{str}$)) \_ s \)
  lens-put = \( \lambda \) (s v. put($\text{str'}$) \_ s (get($\text{str}$) \_ s + v)) \)

definition itrace :: \('t::trace \Rightarrow ('t, 'a) \times ('t, 'a)\) where
  [lens-defs]:
  itrace = \( \lambda \) (get($\text{str}$) \_ s) 
  lens-put = \( \lambda \) (s v. put($\text{str'}$) \_ s (put($\text{str}$) \_ 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 = ($\text{str'}$ - $\text{str}$) 
  by (rel-auto)

lemma tcontr-alt-def': utp-expr.var tt = ($\text{str'}$ - $\text{str}$) 
  by (rel-auto)

lemma tt-indeps [simp]:
  assumes x \approx ($\text{str}$) \_ v x \approx ($\text{str'}$) \_ v
  shows x \approx tt tt \approx 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 R1 :: \((t::trace, \alpha, \beta) \rel rp \Rightarrow (t, \alpha, \beta) \rel rp\) where
R1[upred-defs]: \(R1(P) = (P \land (\$tr \leq_u \$tr'))\)

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

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

lemma R1-mono: \(P \subseteq Q \Rightarrow R1(P) \subseteq R1(Q)\)
  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]: \([x \triangleright in-var tr; x \triangleright out-var tr; x \not \in P] \Rightarrow x \not \in R1(P)\)
  by (simp add: R1-def unrest lens-indep-sym)

lemma R1-false: \(R1(false) = false\)
  by pred-auto

lemma R1-conj: \(R1(P \land Q) = (R1(P) \land R1(Q))\)
  by pred-auto

lemma conj-R1-closed-1 [closure]: \(P \is R1 \Rightarrow (P \land Q) \is R1\)
  by (rel-blast)

lemma conj-R1-closed-2 [closure]: \(Q \is R1 \Rightarrow (P \land Q) \is R1\)
  by (rel-blast)

lemma R1-disj: \(R1(P \lor Q) = (R1(P) \lor R1(Q))\)
  by pred-auto

lemma disj-R1-closed [closure]: \(P \is R1; Q \is R1 \Rightarrow (P \lor Q) \is R1\)
  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]: \[
\bigwedge P. P \in A \Rightarrow P \text{ is } R1; A \neq \{\} \Rightarrow \bigcap A \text{ is } R1
\]
using $R1$-Continuous by (auto simp add: Continuous-def Healthy-def)

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

lemma $R1$-UINF-ind:
$R1\left(\bigcup i \cdot P(i)\right) = \bigcup i \cdot R1(P(i))$
by (rel-auto)

lemma $UINF$-R1-closed [closure]:
\[
\bigwedge i. P(i) \text{ is } R1 \Rightarrow \bigcap i \cdot P(i) \text{ is } R1
\]
by (rel-blast)

lemma $UINF$-R1-closed [closure]:
\[
\bigwedge i. P i \text{ is } R1 \Rightarrow \bigcap i \in A \cdot P i \text{ is } R1
\]
by (rel-blast)

lemma tr-ext-conj-$R1$ [closure]:
$\text{tr} \subseteq u \Rightarrow \text{tr} \underset{u}{\subseteq} e \land P \text{ is } R1$
by (rel-auto, simp add: Prefix-Order.prefix1)

lemma tr-id-conj-$R1$ [closure]:
$\text{tr} \subseteq u \Rightarrow \text{tr} \land P \text{ is } R1$
by (rel-auto)

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

lemma $R1$-extend-conj': $R1(P \land Q) = (P \land R1(Q))$
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]: $(R1 P)[true/\text{true}'] = R1(P[true/\text{true}'])$
by (rel-auto)

lemma $R1$-wait'-false [usubst]: $(R1 P)[false/\text{false}'] = R1(P[false/\text{false}'])$
by (rel-auto)
lemma R1-wait-false-closed [closure]: $P$ is $R1 \implies P[\text{false}/\text{wait}]$ is $R1$
by (rel-auto)

lemma R1-wait'-false-closed [closure]: $P$ is $R1 \implies P[\text{false}/\text{wait}']$ is $R1$
by (rel-auto)

lemma R1-skip: $R1(\text{II}) = \text{II}$
by (rel-auto)

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

lemma subst-R1: $[ [ \text{tr} \uparrow \sigma; \text{tr} \downarrow \sigma ] ] \implies \sigma \uparrow (R1 P) = R1(\sigma \uparrow P)$
by (simp add: R1-def usubst)

lemma subst-R1-closed [closure]: $[ [ \text{tr} \uparrow \sigma; \text{tr} \downarrow \sigma; \text{P} \text{ is } R1 ] ] \implies \sigma \uparrow \text{P} \text{ is } R1$
by (metis Healthy-def subst-R1)

lemma R1-by-refinement:
$P \text{ is } R1 \iff ( (\text{tr} \leq \text{u} \text{tr} \leq \text{tr} \leq \text{u} ) \subseteq P )$
by (rel-blast)

lemma R1-trace-extension [closure]:
$\text{tr} \leq \text{u} \text{tr} \leq \text{tr} \leq \text{e}$ is $R1$
by (rel-auto)

lemma tr-le-trans:
($($tr \leq \text{u} \text{tr} \leq \text{tr} \leq \text{u} ) ;; ($tr \leq \text{u} \text{tr} \leq \text{tr} \leq \text{u} )$) = ($tr \leq \text{u} \text{tr} \leq \text{tr} \leq \text{u} )$
by (rel-auto)

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

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

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

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

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

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

lemma R1-ok true: $(R1(P))[true/\text{ok}] = R1(P[true/\text{ok}])$
by pred-auto
lemma R1-ok-false: \((R1(P))[false/ok] = R1(P[false/ok])\)
by pred-auto

lemma seqr-R1-true-right: \(((P ;; R1(true)) \lor P) = (P ;; (\$tr \leq \alpha \$tr'))\)
by (rel-auto)

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

lemma R1-extend-conj-unrest: \([\$tr \not\in Q; \$tr' \not\in Q] \iff R1(P \land Q) = (R1(P) \land Q)\)
by pred-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 subst, rel-auto)

2.2 R2: No dependence upon trace history

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

definition R2a :: \((t::trace, 'a, 'beta) rel-rp \Rightarrow ('t, 'a,'beta) rel-rp\) where
\[upred-defs\]: \(R2a\ (P) = (\prod s \cdot P[<s>,<s> + (\$tr' - \$tr)/\$tr,\$tr'])\)

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

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

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

definition R2c :: \((t::trace, 'a, 'beta) rel-rp \Rightarrow ('t, 'a, 'beta) rel-rp\) where
\[upred-defs\]: \(R2c(P) = (R2s(P) \triangleleft R1(true) \triangleright P)\)

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

lemma unrest-ok-R2s [unrest]: \(\$ok \not\in P \iff \$ok \not\in R2s(P)\)
by (simp add: R2s-def unrest)
lemma unrest-ok\textsuperscript{-}R2s [unrest]: $\text{\textit{ok}} \, \sharp \, P \implies \text{\textit{ok}} \, \sharp \, R2s(P)$
by (simp add: R2s-def unrest)

lemma unrest-ok-R2c [unrest]: $\text{\textit{ok}} \, \sharp \, P \implies \text{\textit{ok}} \, \sharp \, R2c(P)$
by (simp add: R2c-def unrest)

lemma unrest-ok\textsuperscript{-}R2c [unrest]: $\text{\textit{ok}} \, \sharp \, P \implies \text{\textit{ok}} \, \sharp \, R2c(P)$
by (simp add: R2c-def unrest)

lemma R2s-unrest [unrest]: \begin{align*}
\text{\textit{vwb-lens}}\, x; x \bowtie \text{\textit{in-var tr}}; x \bowtie \text{\textit{out-var tr}}; x \, \sharp \, P & \implies x \, \sharp \, R2s(P) \\
\text{by} & (simp add: R2s-def unrest usubst lens-indep-sym)
\end{align*}

lemma R2s-subst-wait-true [usubst]:
\begin{align*}
(R2s(P))[\text{true}/\text{\textit{wait}}] & = R2s(P[\text{true}/\text{\textit{wait}}]) \\
\text{by} & (simp add: R2s-def usubst unrest)
\end{align*}

lemma R2s-subst-wait\textsuperscript{-}true [usubst]:
\begin{align*}
(R2s(P))[\text{true}/\text{\textit{wait}}] & = R2s(P[\text{true}/\text{\textit{wait}}]) \\
\text{by} & (simp add: R2s-def usubst unrest)
\end{align*}

lemma R2-subst-wait-true [usubst]:
\begin{align*}
(R2(P))[\text{true}/\text{\textit{wait}}] & = R2(P[\text{true}/\text{\textit{wait}}]) \\
\text{by} & (simp add: R2-def R1-def R2s-def usubst unrest)
\end{align*}

lemma R2-subst-wait\textsuperscript{-}true [usubst]:
\begin{align*}
(R2(P))[\text{true}/\text{\textit{wait}}] & = R2(P[\text{true}/\text{\textit{wait}}]) \\
\text{by} & (simp add: R2-def R1-def R2s-def usubst unrest)
\end{align*}

lemma R2-subst-wait-false [usubst]:
\begin{align*}
(R2(P))[\text{false}/\text{\textit{wait}}] & = R2(P[\text{false}/\text{\textit{wait}}]) \\
\text{by} & (simp add: R2-def R1-def R2s-def usubst unrest)
\end{align*}

lemma R2-subst-wait\textsuperscript{-}false [usubst]:
\begin{align*}
(R2(P))[\text{false}/\text{\textit{wait}}] & = R2(P[\text{false}/\text{\textit{wait}}]) \\
\text{by} & (simp add: R2-def R1-def R2s-def usubst unrest)
\end{align*}

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: $P \leftrightarrow P$ is $R2a \leftrightarrow P$ is $R2s$
by (metis Healthy-def\textsuperscript{-}R2s R2s-R2a)

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

lemma R2a\textsuperscript{-}idem: $R2a'(R2a'(P)) = R2a'(P)$
by (rel-auto)

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

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

lemma $R2a'$-weakening: $R2a'(P) \subseteq P$
apply (rel-simp)
apply (rename-tac ok wait tr more ok' wait' tr' more')
apply (rule-tac x=tr in exI)
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: $P \subseteq Q \implies R2(P) \subseteq R2(Q)$
by (pred-auto)

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

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

lemma $R2c$-lit: $R2c(<x>) = <x>$
by (rel-auto)

lemma tr-strict-prefix-$R2c$-closed [closure]: $\forall u \exists v$ $tr < u$ $tr$ is $R2c$
by (rel-auto)

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

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

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

lemma $R2s$-USUP:
$R2s(\bigsqcap_{i \in A} P(i)) = (\bigsqcap_{i \in A} R2s(P(i)))$
by (simp add: $R2s$-def usubst)

lemma $R2c$-USUP:
$R2c(\bigsqcap_{i \in A} P(i)) = (\bigsqcap_{i \in A} R2c(P(i)))$
by (rel-auto)

lemma $R2s$-UINF:
$R2s(\bigsqcup_{i \in A} P(i)) = (\bigsqcup_{i \in A} R2s(P(i)))$
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:** \( R2(P \lor Q) = (R2(P) \lor R2(Q)) \)
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':** \( R2(P \triangleleft b \triangleright Q) = (R2(P) \triangleleft R2s(b) \triangleright R2s(Q)) \)
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:** \( R2s(\text{true}) = \text{true} \)
by pred-auto

**Lemma R2s-false:** \( R2s(\text{false}) = \text{false} \)
by pred-auto

**Lemma true-is-R2s:**
\( \text{true is R2s} \)
by (simp add: Healthy-def R2s-true)

**Lemma R2s-lift-rea:** \( R2s([P]_R) = [P]_R \)
by (simp add: R2s-def usubst unrest)

**Lemma R2c-lift-rea:** \( R2c([P]_R) = [P]_R \)
by (simp add: R2c-def R2s-lift-rea cond-idem usubst unrest)

**Lemma R2c-true:** \( R2c(\text{true}) = \text{true} \)
by (rel-auto)

**Lemma R2c-false:** \( R2c(\text{false}) = \text{false} \)
by (rel-auto)

**Lemma R2c-and:** \( R2c(P \land Q) = (R2c(P) \land R2c(Q)) \)
by (rel-auto)
lemma `conj-R2c-closed [closure]: \[ P is R2c; Q is R2c \implies (P \land Q) is R2c \]
by (simp add: Healthy-def R2c-and)

lemma `R2c-disj: R2c(P \lor Q) = (R2c(P) \lor R2c(Q))
by (rel-auto)

lemma `R2c-inf: R2c(P \land Q) = (R2c(P) \land R2c(Q))
by (rel-auto)

lemma `R2c-not: R2c(\lnot P) = (\lnot 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]: (\exists P. R2c(P)) = (\exists P. R2c(P))
by (rel-auto)

lemma `R2c-wait'-false [usubst]: (\exists P. R2c(P)) = (\exists P. R2c(P))
by (rel-auto)

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

lemma `R2c-tr'-ge-tr: R2c(\$tr' \ge _ \$tr) = (\$tr' \ge _ \$tr)
by (rel-auto)

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

lemma `R2c-condr: R2c(P \land b \land Q) = (R2c(P) \land R2c(b) \land 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' = _ \$tr \land II_{\alpha} tr)
  by (subst skip-r-unfold[of tr], simp-all)
  also have ... = (R2c(\$tr' = _ \$tr) \land II_{\alpha} tr)
  by (simp add: R2c-and, simp add: R2c-def R2s-def usubst unrest cond-idem)
  also have ... = (\$tr' = _ \$tr \land II_{\alpha} tr)

by (simp add: R2c-tr'-minus-tr)
finally show ?thesis  
  by (subst skip-r-anfold[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($str' =_u str \land \$wait')) = ($str' =_u str \land \$wait')
  apply rel-auto using minus-zero-eq by blast

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

lemma R1-R2s-tr'-extend-tr:
  [[str z v; str' z v]] \implies R1(R2s($str' =_u str' \circ u v)) = ($str' =_u str' \circ u v)
  apply (rel-auto)
    apply (metis append-minus)
  apply (simp add: Prefix-Order-prefixI)
done

lemma R2-tr-prefix: R2($str \leq_u str') = ($str \leq_u str')
  by (pred-auto)

lemma R2-form:
  R2(P) = (\exists tt_0 \cdot P[\tt[0]/\tt[0]]\tt[tt_0]/\tt[tt_0]' \land \str' =_u \str + \tt[tt_0]]
  by (rel-auto, metis trace-class.add-diff-cancel-left trace-class.le-if-diff-add)

lemma R2-subst-tr:
  assumes P is R2
  shows [$str \circ s tr_0, \str' \circ s tr_0 + t] \vdash P = [$str \circ s 0, \str' \circ s t] \vdash P
  proof
    have [$str \circ s tr_0, \str' \circ s tr_0 + t] \vdash R2 P = [$str \circ s 0, \str' \circ s t] \vdash R2 P
    by (rel-auto)
    thus ?thesis
    by (simp add: Healthy-if assms)
  qed

lemma R2-seq-form:
  shows (R2(P) ;; R2(Q)) =
    (\exists tt_1 \cdot \exists tt_2 \cdot ((P[\tt[0]/\tt[0]]\tt[tt_1]/\tt[tt_1]'] \circ (Q[\tt[0]/\tt[0]]\tt[tt_2]/\tt[tt_2]'\circ \tt' \circ =_u \str + \tt[tt_1] + \tt[tt_2])))
  proof
    have (R2(P) ;; R2(Q)) = (\exists tt_0 \cdot ((P[\tt[0]/\tt[0]]\tt[tt_0]/\tt[tt_0]'] \circ (R2(Q)\tt[tt_0]/\tt[tt_0]')) \circ (R2(Q)\tt[tt_0]/\tt[tt_0]'))
    by (subst seqr-middle[of tr], simp-all)
    also have ... =
      (\exists tt_0 \cdot \exists tt_1 \cdot \exists tt_2 \cdot ((P[\tt[0]/\tt[0]]\tt[tt_1]/\tt[tt_1]'] \land \tt[tt_0] =_u \str + \tt[tt_1]) ;;
Lemma R2-seqr-form:

Assumes P is R2 Q is R2

Shows P :: Q =

(∃ tt1, tt2 :: ((P[0/str] tt1)/str ); (Q[0/str] tt2)) \land \n (str =_u <tt0> + <tt2>)

By (simp add: R2-form subst unrest atquant-lift, rel-blast)

Also have ...

(∃ tt0 \cdot \exists tt1 \cdot \exists tt2 \cdot ((<tt0> =_u str + <tt1> \land P[0/str] tt1); (Q[0/str] tt2)) \land 
 (str' =_u <tt0> + <tt2>)

By (simp add: conj-comm)

Also have ...

(∃ tt1, tt2, tt0 :: ((P[0/str] tt1)/str ); (Q[0/str] tt2)) \land 
 (tt0 =_u str + <tt1> \land str' =_u <tt0> + <tt2>)

By (rel-blast)

Also have ...

(∃ tt1, tt2 :: ((P[0/str] tt1)/str ); (Q[0/str] tt2)) \land 
 (tt0 =_u str + <tt1> \land str' =_u <tt0> + <tt2>)

By (rel-auto)

Also have ...

(∃ tt1, tt2 :: ((P[0/str] tt1)/str ); (Q[0/str] tt2)) \land 
 (tt0 =_u str + <tt1> \land str' =_u <tt0> + <tt2>)

By (rel-auto)

Finally show thesis.

Qed

Lemma R2-seqr-form'':

Assumes P is R2 Q is R2

Shows P :: Q =

(∃ tt1, tt2 :: ((P[0, <tt1>, str, str'] ); (Q[0, <tt2>, str'])) \land 
 (str' =_u str + <tt1> + <tt2>)

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 =

(∃ tt1, tt2 :: ((P[0, <tt1>]/str'); (Q[0, <tt2>]/str')) \land 
 (str' =_u str + <tt1> + <tt2>)

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

Lemma R2-tr-middle:

Assumes P is R2 Q is R2

Shows (∃ tr0 :: (P[<tr0>/str']); (Q[<tr0>/str]) \land <tr0> \leq_u str') = (P :: Q)

Proof –

Have (P :: Q) = (∃ tr0 :: (P[<tr0>/str']); (Q[<tr0>/str]) \land <tr0> \leq_u str') = (P :: Q)

By (simp add: seqr-middle)

Also have ...

(∃ tr0 :: ((R2 P)[<tr0>/str']); (R2 Q)[<tr0>/str]) \land <tr0> \leq_u str'

By (simp add: assms Healthy-if)

Also have ...

(∃ tr0 :: ((R2 P)[<tr0>/str']); (R2 Q)[<tr0>/str]) \land <tr0> \leq_u str'

By (rel-auto)

Also have ...

(∃ tr0 :: ((R2 P)[<tr0>/str']); (R2 Q)[<tr0>/str]) \land <tr0> \leq_u str'

By (simp add: assms Healthy-if)

Finally show thesis.

Qed

Lemma R2-seqr-distribute:

Fixes P :: (t::trace, o, b) rel-rp and Q :: (t:b,g) rel-rp

Shows R2(R2(P :: R2(Q)) = (R2(P :: R2(Q))

Proof –

Have R2(R2(P :: R2(Q)) =
((∃ tt₁ · ∃ tt₂ · (P[0/str][<tt₁>/str'] • Q[0/str][<tt₂>/str']) • ($str' − str) = u <tt₁> + <tt₂> • $str' ≥ u $str) )
by (simp add: R2-seqr-form, simp add: R2s-def usubst unrest, rel-auto)
also have ... =
((∃ tt₁ · ∃ tt₂ · (P[0/str][<tt₁>/str'] • Q[0/str][<tt₂>/str']) • ($str' − str) = u <tt₁> + <tt₂> • $str' ≥ u $str)
by (subst subst-eq-replace, simp)
also have ... =
((∃ tt₁ · ∃ tt₂ · (P[0/str][<tt₁>/str'] • Q[0/str][<tt₂>/str']) • ($str' − str) = u <tt₁> + <tt₂> • $str' ≥ u $str)
by (rel-auto)
also have ... =
((∃ tt₁ · ∃ tt₂ · (P[0/str][<tt₁>/str'] • Q[0/str][<tt₂>/str']) • ($str' − str) = u <tt₁> + <tt₂> • $str' ≥ u $str)
by (rel-auto)
also have ... =
((∃ tt₁ · ∃ tt₂ · (P[0/str][<tt₁>/str'] • Q[0/str][<tt₂>/str']) • ($str' − str) = u <tt₁> + <tt₂> • $str' ≥ u $str)
by (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 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, 'a) hrel-rp ⇒ ('t, 'a) hrel-rp where
[upred-defs]: R3(P) = (H < $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 ∨ Q) = (R3(P) ∨ R3(Q))
  by (rel-auto)

lemma R3-USUP:
  assumes A ≠ {}
  shows R3(⨝ i ∈ A · P(i)) = (∏ i ∈ A · R3(P(i)))
  using assms by (rel-auto)

lemma R3-UINF:
  assumes A ≠ {}
  shows R3(⨆ i ∈ A · P(i)) = (∪ i ∈ A · R3(P(i)))
  using assms by (rel-auto)

lemma R3-condr: R3(P ◁ b ▷ Q) = (R3(P) ◁ b ▷ R3(Q))
  by (rel-auto)

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

lemma R3-form: R3(P) = (($wait ∧ II) ∨ (¬$wait ∧ P))
  by (rel-auto)

lemma wait-R3:
  ($wait ∧ R3(P)) = (II ∧ $wait´)
  by (rel-auto)

lemma nwait-R3:
  (¬$wait ∧ R3(P)) = (¬$wait ∧ 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: $R_1(R_3(P)) = R_3(R_1(P))$
  by (rel-auto)

lemma R2-R3-commute: $R_2(R_3(P)) = R_3(R_2(P))$
  apply (rel-auto)
  using minus-zero-eq apply blast+
done

2.4 R4: The trace strictly increases

definition R4 :: $'t$::trace, $'a$, $'b$ rel-rp $\Rightarrow$ $('t$, $'a$, $'b$) rel-rp where
[upred-defs]: $R_4(P) = (P \wedge \$tr <_u \$tr')$

lemma R4-implies-R1 [closure]: $P$ is $R_4$ $\implies$ $P$ is $R_1$
  using less-iff by rel-blast

lemma R4-iff-refine:
  $P$ $\Rightarrow$ ($\$tr <_u \$tr'$) $\subset$ $P$
  by (rel-blast)

lemma R4-idem: $R_4$ ($R_4 P$) = $R_4 P$
  by (rel-auto)

lemma R4-false: $R_4$(false) = false
  by (rel-auto)

lemma R4-conj: $R_4$($P \wedge Q$) = ($R_4$($P$) $\wedge$ $R_4$($Q$))
  by (rel-auto)

lemma R4-disj: $R_4$($P \vee Q$) = ($R_4$($P$) $\vee$ $R_4$($Q$))
  by (rel-auto)

lemma R4-is-R4 [closure]: $R_4(P)$ is $R_4$
  by (rel-auto)

lemma false-R4 [closure]: false is $R_4$
  by (rel-auto)

lemma UINF-R4-closed [closure]: $\bigwedge i. \text{P i is } R_4$ $\implies$ ($\bigwedge i. \text{P i}$) is $R_4$
  by (rel-blast)

lemma conj-R4-closed [closure]: $\text{P is } R_4$; $\text{Q is } R_4$ $\implies$ ($\text{P } \wedge \text{Q}$) is $R_4$
  by (simp add: Healthy-def R4-conj)

lemma disj-R4-closed [closure]: $\text{P is } R_4$; $\text{Q is } R_4$ $\implies$ ($\text{P } \vee \text{Q}$) is $R_4$
  by (simp add: Healthy-def R4-disj)

lemma seq-R4-closed-1 [closure]: $\text{P is } R_4$; $\text{Q is } R_1$ $\implies$ ($\text{P } :: \text{Q}$) is $R_4$
  using less-le-trans by rel-blast

lemma seq-R4-closed-2 [closure]:

19
\[ P \text{ is } R1; \ Q \text{ is } R4 \implies (P \ ;; Q) \text{ is } R4 \]
using le-less-trans by rel-blast

### 2.5 R5: The trace does not increase

**definition** R5 \( \coloneqq (t::\text{trace}, \alpha, \beta) \rel\text{-rp} \Rightarrow (t', \alpha', \beta') \rel\text{-rp} \) where

\[ \upreddefs: \ R5(P) = (P \land \$tr = \_ \$tr') \]

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

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

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

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

**lemma** R4-R5: R4 (R5 P) = false
by (rel-auto)

**lemma** R5-R4: R5 (R4 P) = false
by (rel-auto)

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

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

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

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

### 2.6 RP laws

**definition** RP-def [upred-defs]: \( \text{RP}(P) = R1(R2c(R3(P))) \)

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

**lemma** RP-alt-def: \( \text{RP}(P) = R1(R2c(R3(P))) \)
by (metis R1-R2c-is-R2 R1-idem RP-def)

**lemma** RP-intro: \[ P \text{ is } R1; \ P \text{ is } R2; \ P \text{ is } R3 \implies P \text{ is } RP \]
by (simp add: Healthy-def’ RP-alt-def)
lemma \( \text{RP-idem}: \text{RP}(\text{RP}(P)) = \text{RP}(P) \)
by (simp add: R1-R2c-is-R2 R2-R3-commute R2-idem R3-idem \text{RP-def})

lemma \( \text{RP-Idempotent} \) [closure]: Idempotent \( \text{RP} \)
by (simp add: \text{Idempotent-def} \text{RP-idem})

lemma \( \text{RP-mono}: P \subseteq Q \implies \text{RP}(P) \subseteq \text{RP}(Q) \)
by (simp add: R1-R2c-is-R2 R2-mono R3-mono \text{RP-def})

lemma \( \text{RP-Monotonic} \): Monotonic \( \text{RP} \)
by (simp add: \text{mono-def} \text{RP-mono})

lemma \( \text{RP-Continuous} \): Continuous \( \text{RP} \)
by (simp add: \text{Continuous-comp} \text{R1-Continuous} \text{R2c-Continuous} \text{R3-Continuous} \text{RP-comp-def})

lemma \( \text{RP-skip} \):
\( \text{RP}(\text{II}) = \text{II} \)
by (simp add: \text{R1-skip} \text{R2c-skip-r} \text{R3-skipr} \text{RP-def})

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

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

2.7 UTP theories

typedecl \( \text{REA} \)
abbreviation \( \text{REA} \equiv \text{UTHY}(\text{REA}, (\text{'t::trace}, \text{'a}) \text{rp}) \)

overloading
\begin{align*}
\text{rea-hcond} & \equiv \text{utp-hcond} :: (\text{REA}, (\text{'t::trace}, \text{'a}) \text{rp}) \text{uthy} \Rightarrow ((\text{'t}, \text{'a}) \text{rp} \times (\text{'t}, \text{'a}) \text{rp}) \text{health} \\
\text{rea-unit} & \equiv \text{utp-unit} :: (\text{REA}, (\text{'t::trace}, \text{'a}) \text{rp}) \text{uthy} \Rightarrow (\text{'t}, \text{'a}) \text{hrel-rp} \\
\end{align*}

begin
definition \( \text{rea-hcond} :: (\text{REA}, (\text{'t::trace}, \text{'a}) \text{rp}) \text{uthy} \Rightarrow ((\text{'t}, \text{'a}) \text{rp} \times (\text{'t}, \text{'a}) \text{rp}) \text{health} \\
where [\text{apred-defs}]: \text{rea-hcond} \ T = \text{RP} \\
definition \( \text{rea-unit} :: (\text{REA}, (\text{'t::trace}, \text{'a}) \text{rp}) \text{uthy} \Rightarrow (\text{'t}, \text{'a}) \text{hrel-rp} \\
where [\text{apred-defs}]: \text{rea-unit} \ T = \text{II} \\
end

interpretation \( \text{rea-utp-theory}: \text{utp-theory} \text{UTHY}(\text{REA}, (\text{'t::trace}, \text{'a}) \text{rp}) \)
rewrites carrier (\text{uthy-order REA}) = [\text{RP}]_H
by (simp-all add: \text{rea-hcond-def} \text{utp-theory-def} \text{RP-idem})

interpretation \( \text{rea-utp-theory-mono}: \text{utp-theory-continuous} \text{UTHY}(\text{REA}, (\text{'t::trace}, \text{'a}) \text{rp}) \)
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, 'a) 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 [4]. We first define healthiness conditions for R1 and R2 merge predicates.

definition R1m ::= (t :: trace, 'a) rp merge ⇒ (t, 'a) rp merge
where [upred-defs]: R1m(M) = (M ∧ $tr < ≤_a $tr -)

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

**Lemma R2m⁻¹-form:**

\[ R2m^{-1}(M) = (\exists (t_p, t_0, t_1) \cdot M[0,tt_p\rangle,tt_0\rangle,tt_1\rangle/\langle tr, tr', 0-\langle tr, 1-\langle tr, {\langle tr, 0-\langle tr, 1-\langle tr, \langle tr \rangle \rangle \rangle \rangle \rangle \rangle \rangle] \]

\[ \wedge tr' = u \cdot \langle tr \rangle \]

\[ \wedge 0-\langle tr = u \cdot \langle tr \rangle \]

\[ \wedge 1-\langle tr = u \cdot \langle tr \rangle \]

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 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⁻¹-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 $M \vdash P$ is $R2m'$
by (metis Healthy-def $R2m$-seq-lemma assms(1) assms(2))

lemma $R1$-par-by-merge [closure]:
$M$ is $R1m \Longrightarrow (P ||_M Q)$ is $R1$
by (rel-blast)

lemma $R2$-$R2m'$-pbm: $R2(P ||_M Q) = (R2(P) ||_{R2m'}(M) R2(Q))$
proof –
  have $(R2(P) ||_{R2m'}(M) R2(Q)) = ((R2(P)) ||_M R2(Q))$ ;
    $(\exists \ (tt_p, tt_0, tt_1) \cdot M[0,\langle tt_p\rangle,\langle tt_0\rangle,\langle tt_1\rangle/\text{str}_{\text{tr}},\text{str}_{\text{tr}},\text{0-tr},\text{1-tr}]
    \land \text{str}_{\text{tr}}' = u \cdot \text{str}_{\text{tr}} + \langle tt_p\rangle
    \land \text{0-tr} = u \cdot \text{str}_{\text{tr}} + \langle tt_0\rangle
    \land \text{1-tr} = u \cdot \text{str}_{\text{tr}} + \langle tt_1\rangle))$ ;
  by (simp add: par-by-merge-def $R2m'$-form)
also have \dots = (\exists \ (tt_p, tt_0, tt_1) \cdot ((R2(P) ||_M R2(Q)) \land \text{0-tr}' = u \cdot \text{str}_{\text{tr}} + \langle tt_0\rangle \land \text{1-tr}' = u
    \text{str}_{\text{tr}} + \langle tt_1\rangle)) ;
  by (rel-blast)
also have \dots = (\exists \ (tt_p, tt_0, tt_1) \cdot ((R2(P) \land \text{str}' = u \cdot \text{str} + \langle tt_0\rangle) \land \text{0-tr}' = u
    \text{str} + \langle tt_0\rangle \land \text{1-tr}' = u
    \text{str} + \langle tt_1\rangle)) ;
  by (rel-blast)
also have \dots = (\exists \ (tt_p, tt_0, tt_1) \cdot ((P[0,\langle tt_0\rangle/\text{str},\text{str}'] \land \text{str}' = u \cdot \text{str} + \langle tt_0\rangle) \land
    \text{0-tr}' = u \cdot \text{str} + \langle tt_0\rangle \land \text{1-tr}' = u
    \text{str} + \langle tt_1\rangle)) ;
  by (rel-auto, blast, metis le-adrace-class.add-diff-cancel-left)
also have \dots = (\exists \ (tt_p, tt_0, tt_1) \cdot (Q[0,\langle tt_1\rangle/\text{str},\text{str}'] \land \text{str}' = u \cdot \text{str} + \langle tt_1\rangle) \land \text{str}' = u
    \text{str} + \langle tt_1\rangle)) ;
  by (simp add: R2-form usubst)
also have \dots = (\exists \ (tt_p, tt_0, tt_1) \cdot (Q[0,\langle tt_1\rangle/\text{str},\text{str}'] \land \text{str}' = u \cdot \text{str} + \langle tt_1\rangle)) ;
  by (rel-auto, rel-auto, blast, metis le-adrace-class.add-diff-cancel-left)
finally show \dots = (\exists \ (tt_p, tt_0, tt_1) \cdot (Q[0,\langle tt_1\rangle/\text{str},\text{str}'] \land \text{str}' = u \cdot \text{str} + \langle tt_1\rangle)) ;
  by (rel-auto, rel-auto, blast, metis le-adrace-class.add-diff-cancel-left)
qed
lemma \( R2m-R2m\'-pbm \): \( (R2(P) \parallel R2m(M) \cdot R2(Q)) = (R2(P) \parallel R2m'(M) \cdot R2(Q)) \)
    by (rel-blast)

lemma \( R2-par-by-merge \) [closure]:
    assumes \( P \) is \( R2 \) \( Q \) is \( R2 \) \( M \) is \( R2m \)
    shows \( (P \parallel M \cdot Q) \) 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 \( (P \parallel M \cdot Q) \) 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 \( R1m-disj \): \( R1m(P \lor Q) = (R1m(P) \lor R1m(Q)) \)
    by (rel-auto)

lemma \( R1m-conj \): \( R1m(P \land Q) = (R1m(P) \land R1m(Q)) \)
    by (rel-auto)

lemma \( R2m-skip-merge \): \( R2m(skip_m) = skip_m \)
    apply (rel-auto) using minus-zero-eq by blast

lemma \( R2m-disj \): \( R2m(P \lor Q) = (R2m(P) \lor R2m(Q)) \)
    by (rel-auto)

lemma \( R2m-conj \): \( R2m(P \land Q) = (R2m(P) \land R2m(Q)) \)
    by (rel-auto)

definition \( R3m \) := ('t :: trace, 'α) rp merge ⇒ ('t, 'α) rp merge where
    upred-defs: \( R3m(M) = \) skip_m @ $\text{wait} \triangleright M$

lemma \( R3-par-by-merge \):
    assumes \( P \) is \( R3 \) \( Q \) is \( R3 \) \( M \) is \( R3m \)
    shows \( (P \parallel M \cdot Q) \) is \( R3 \)
    proof
        have \( (P \parallel M \cdot Q) = ((P \parallel M \cdot Q)[true/$\text{wait}$] \triangleright $\text{wait} \triangleright (P \parallel M \cdot Q)) \)
            by (metis cond-L6 cond-var-split in-var-war wait-var-lens)
        also have \( ... = (((R3 \cdot P)[true/$\text{wait}$] \parallel (R3m \cdot M)[true/$\text{wait} \cdot _\cdot ] \cdot (R3 \cdot Q)[true/$\text{wait}$] \triangleright $\text{wait} \triangleright (P \parallel M \cdot Q)) \)
            by (simp add: subst-tac, simp add: Healthy-if assms)
        also have \( ... = (((I[true/$\text{wait}$] \parallel skip_m[true/$\text{wait} \cdot _\cdot ] \cdot II[[true/$\text{wait}$] \triangleright $\text{wait} \triangleright (P \parallel M \cdot Q)) \)
            by (simp add: R3-def R3m-def usubst)
        also have \( ... = (((I \parallel skip_m \cdot II)[true/$\text{wait}$] \triangleright $\text{wait} \triangleright (P \parallel M \cdot Q)) \)
            by (subst-tac)
        also have \( ... = (II \triangleright $\text{wait} \triangleright (P \parallel M \cdot Q)) \)
            by (simp add: cond-var-subst-left par-by-merge-skip)
        also have \( ... = R3(P \parallel M \cdot Q) \)
            by (simp add: R3-def)
        finally show ?thesis
            by (simp add: Healthy-def)
    qed
lemma SymMerge-R1-true [closure]:
M is SymMerge ⟹ 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 reactive relational calculus for $R1$-$R2$ predicates as an extension of the standard alphabetised predicate calculus. This enables us to formally characterise relational programs that refer to both state variables and a trace history. For more details on reactive relations, please see the associated journal paper [3].

4.1 Healthiness Conditions

definition $RR :: ('t::trace, 'a, 'β) rel-rp ⇒ ('t, 'a, 'β) rel-rp$ 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 $\$ok \notin P \$ok \notin P \$wait \notin P \$wait \notin P P is R1 P is R2c$
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 $\text{ok \notin} P \text{ $\text{ok} \notin P \text{ $\text{wait} \notin P \text{ $\text{wait} \notin P \text{ $ P \text{ is } R2}$ shows $P$ is RR
by (simp add: RR-def Healthy-def ex-plus, simp add: Healthy-if assms ex-unrest)

lemma RR-unrests [unrest]:
assumes $P$ is RR
shows $\text{ok \notin} P \text{ $\text{ok} \notin P \text{ $\text{wait} \notin P \text{ $\text{wait} \notin P$
proof
have $\text{ok \notin} \text{RR}(P) \text{ $\text{ok} \notin \text{RR}(P) \text{ $\text{wait} \notin \text{RR}(P) \text{ $\text{wait} \notin \text{RR}(P)$
by (simp-all add: RR-def ex-plus unrest)
thus $\text{ok \notin} P \text{ $\text{ok} \notin P \text{ $\text{wait} \notin P \text{ $\text{wait} \notin P$
by (simp-all add: assms Healthy-if)
qed

lemma RR-refine-intro:
assumes $P$ is RR $Q$ is RR
$\forall t. \ P[0,\langle t\rangle]/\text{str}$ $\subseteq Q[0,\langle t\rangle]/\text{str}$
shows $P \subseteq Q$
proof
have $\forall t. (\text{RR}(P))[0,\langle t\rangle]/\text{str} \subseteq (\text{RR}(Q))[0,\langle t\rangle]/\text{str}$
by (simp add: Healthy-if assms)
thus $\text{RR}(P) \subseteq \text{RR}(Q)$
by (rel-auto)
thus $\text{thesis}$
by (simp add: Healthy-if assms)
qed

lemma R4-RR-closed [closure]:
assumes $P$ is RR
shows $R4(P)$ is RR
proof
have $R4(\text{RR}(P))$ is RR
by (rel-blast)
thus $\text{thesis}$
by (simp add: Healthy-if assms)
qed

lemma R5-RR-closed [closure]:
assumes $P$ is RR
shows $R5(P)$ is RR
proof
have $R5(\text{RR}(P))$ is RR
using minus-zero-eq by rel-auto
thus $\text{thesis}$
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 $\equiv R1(\text{true})$

definition rea-not :: (′t::trace,′α,′β) rel-rp $\Rightarrow (′t,′α,′β) rel-rp (\lnot_r - [40] 40)$
where [upred-defs]: (\lnot_r P) $\equiv R1(\lnot P)$
4.3 Unrestriction and substitution laws

lemma rea-true-unrest [unrest]:
  \(\x \because (\text{str})_{c}; x \because (\text{str'})_{v} \quad \Rightarrow \quad x \not\because \text{true}_r\)
  by (simp add: R1-def unrest lens-indep-sym)

lemma rea-not-unrest [unrest]:
  \(\x \because (\text{str})_{c}; x \because (\text{str'})_{v}; x \not\because r \quad \Rightarrow \quad x \not\because r \quad P\)
  by (simp add: rea-not-def R1-def unrest lens-indep-sym)

lemma rea-impl-unrest [unrest]:
  \(\x \because (\text{str})_{c}; x \because (\text{str'})_{v}; x \not\because r \quad \Rightarrow \quad x \not\because r \quad (P \Rightarrow Q)\)
  by (simp add: rea-impl-def unrest)

lemma rea-true-usubst [usubst]:
  \(\text{str} \not\because \sigma; \text{str'} \not\because \sigma \quad \Rightarrow \quad \sigma \uparrow \text{true}_r = \text{true}_r\)
  by (simp add: R1-def usubst)

lemma rea-not-usubst [usubst]:
  \(\text{str} \not\because \sigma; \text{str'} \not\because \sigma \quad \Rightarrow \quad \sigma \uparrow (\neg_r \quad P) = (\neg_r \quad \sigma \uparrow r \quad P)\)
  by (simp add: rea-not-def R1-def usubst)

lemma rea-impl-usubst [usubst]:
  \(\text{str} \not\because \sigma; \text{str'} \not\because \sigma \quad \Rightarrow \quad \sigma \uparrow (P \Rightarrow Q) = (\sigma \uparrow r \quad P \Rightarrow_r \quad \sigma \uparrow r \quad Q)\)
  by (simp add: rea-impl-def usubst R1-def)

lemma rea-true-usubst-tt [usubst]:
  \(R1(\text{true})[v/\&tt] = true\)
  by (rel-simp)

lemma unrest-rea-subst [unrest]:
  \(\text{mwb-lens} \; x; x \because (\text{str})_{c}; x \because (\text{str'})_{v}; x \not\because r; x \not\because r \quad P \quad \Rightarrow \quad x \not\because r \quad P[v]_{r}\)
  by (simp add: rea-subst-def R1-def unrest lens-indep-sym)
lemma rea-substs [usubst]:
\[
\begin{align*}
\text{true} &_{r} = \text{true} \quad \text{true} v_{r} = \text{true} \\
(\neg_{r} \text{P})_{r} = (\neg_{r} \text{P} v_{r}) \\
(\text{P} \land Q)_{r} = (P v_{r} \land Q v_{r}) \\
(\text{P} \lor Q)_{r} = (P v_{r} \lor Q v_{r}) \\
(\text{P} \Rightarrow Q)_{r} = (P v_{r} \Rightarrow Q v_{r})
\end{align*}
\]
by rel-auto

lemma rea-substs-lattice [usubst]:
\[
\begin{align*}
\bigwedge_{i} i \cdot (P(i))_{r} = \bigwedge_{i} i \cdot (P(i))_{r} \\
\bigwedge_{i \in A} i \cdot (P(i))_{r} = \bigwedge_{i \in A} i \cdot (P(i))_{r} \\
\bigvee_{i} i \cdot (P(i))_{r} = \bigvee_{i} i \cdot (P(i))_{r}
\end{align*}
\]
by (rel-auto)+

lemma rea-subst-USUP-set [usubst]:
\[
A \neq \emptyset \implies \bigwedge_{i \in A} i \cdot (P(i))_{r} = \bigwedge_{i \in A} i \cdot (P(i))_{r}
\]
by (rel-auto)+

4.4 Closure laws

lemma rea-lift-R1 [closure]: \text{[P]}_r is R1
by (rel-simp)

lemma R1-rea-not: R1(\neg_{r} \text{P}) = (\neg_{r} \text{P})
by rel-auto

lemma R1-rea-not: R1(\neg_{r} \text{P}) = (\neg_{r} \text{R1}(P))
by rel-auto

lemma R2c-rea-not: R2c(\neg_{r} \text{P}) = (\neg_{r} \text{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(\text{P} \Rightarrow_{r} Q) = (\text{P} \Rightarrow_{r} \text{R1}(Q))
by (rel-auto)

lemma R1-rea-impl: R1(\text{P} \Rightarrow_{r} Q) = (\text{R1}(P) \Rightarrow_{r} \text{R1}(Q))
by (rel-auto)

lemma R2c-rea-impl: R2c(\text{P} \Rightarrow_{r} Q) = (\text{R2c}(P) \Rightarrow_{r} \text{R2c}(Q))
by (rel-auto)

lemma RR-rea-impl: RR(\text{R1}(P) \Rightarrow_{r} RR(Q)) = (\text{RR}(P) \Rightarrow_{r} \text{RR}(Q))
by (rel-auto)

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

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

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

lemma rea-not-R1 [closure]: \neg_{r} \text{P} is R1
by (rel-auto)

lemma rea-not-R2c [closure]: \( P \text{ is } R2c \implies \neg r P \text{ is } R2c \)
by (simp add: Healthy-def rea-not-def R1-R2c-commute[THEN sym] R2c-not)

lemma rea-not-R2-closed [closure]:
\( P \text{ is } R2 \implies (\neg r P) \text{ is } R2 \)
by (simp add: Healthy-def' R1-rea-not' R2-R2c-def R2c-rea-not)

lemma rea-not-R2-closed [closure]:
\( P \text{ is } R2 \implies \neg (\neg r P) \text{ is } R2 \)
by (metis Healthy-def' R2-R2c-def R2c-rea-not)

lemma rea-no-RR [closure]:
\( \{ P \text{ is } RR \} \implies (\neg r P) \text{ is } RR \)
by (metis Healthy-def' RR-rea-not)

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

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

lemma rea-impl-R2 [closure]:
\( \{ P \text{ is } R2; Q \text{ is } R2 \} \implies (P \Rightarrow r Q) \text{ is } R2 \)
by (rel-blast)

lemma rea-impl-RR [closure]:
\( \{ P \text{ is } RR; Q \text{ is } RR \} \implies (P \Rightarrow r Q) \text{ is } RR \)
by (metis Healthy-def' RR-rea-impl)

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

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

lemma USUP-mem-RR-closed [closure]:
assumes \( \bigwedge i. i \in A \implies P i \text{ is } RR A \neq \{ \} \)
shows \( \{ \bigcup i \in A \cdot P(i) \} \text{ is } RR \)
proof
  have 1:\( \{ \bigcup i \in A \cdot P(i) \} \text{ is } R1 \)
    by (unfold Healthy-def, subst R1-UINF, simp-all add: Healthy-if assms closure cong: USUP-cong)
  have 2:\( \{ \bigcup i \in A \cdot P(i) \} \text{ is } R2c \)
    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 \( \bigwedge i. P i \text{ is } RR \)
shows \( \{ \bigcup i \cdot P(i) \} \text{ is } RR \)
using \textit{USUP-mem-RR-closed}[of UNIV P] by (simp add: assms)

\textbf{lemma} \textit{UINF-mem-RR-closed} \[\text{clos}eu\]:
\begin{itemize}
\item \textbf{assumes} \(\forall i. \ P i \text{ is RR}\)
\item \textbf{shows} \((\bigwedge i \in A \cdot P(i)) \text{ is RR}\)
\end{itemize}
\textbf{proof} --
\begin{itemize}
\item \textbf{have} \(1:(\bigwedge i \in A \cdot P(i)) \text{ is R1}\)
  \textbf{by} (unfold \textit{Healthy-def}, subst \textit{R1-USUP}, simp-all add: \textit{Healthy-if} assms closure)
\item \textbf{have} \(2:(\bigwedge i \in A \cdot P(i)) \text{ is R2c}\)
  \textbf{by} (unfold \textit{Healthy-def}, subst \textit{R2c-USUP}, simp-all add: \textit{Healthy-if} assms \textit{RR-implies-R2c} closure)
\item \textbf{show} \(?\text{thesis}\)
  using \(1\ 2\) by (rule-tac \textit{RR-intro}, simp-all add: unrest assms)
\end{itemize}
\textbf{qed}

\textbf{lemma} \textit{UINF-ind-RR-closed} \[\text{clos}eu\]:
\begin{itemize}
\item \textbf{assumes} \(\forall i. \ P i \text{ is RR}\)
\item \textbf{shows} \((\bigsqcap i \in A \cdot P(i)) \text{ is RR}\)
\end{itemize}
\textbf{proof} --
\begin{itemize}
\item \textbf{have} \(1:(\bigsqcap i \in A \cdot P(i)) \text{ is R1}\)
  \textbf{by} (unfold \textit{Healthy-def}, subst \textit{R1-UINF}, simp-all add: \textit{Healthy-if} assms closure)
\item \textbf{have} \(2:(\bigsqcap i \in A \cdot P(i)) \text{ is R2c}\)
  \textbf{by} (unfold \textit{Healthy-def}, subst \textit{R2c-UINF}, simp-all add: \textit{Healthy-if} assms \textit{RR-implies-R2c} closure)
\item \textbf{show} \(?\text{thesis}\)
  using \(1\ 2\) by (rule-tac \textit{RR-intro}, simp-all add: unrest assms)
\end{itemize}
\textbf{qed}

\textbf{lemma} \textit{USUP-elem-RR} \[\text{clos}eu\]:
\begin{itemize}
\item \textbf{assumes} \(\forall i. \ P i \text{ is RR} \ A \neq \{}\)
\item \textbf{shows} \((\bigsqcup i \in A \cdot P i) \text{ is RR}\)
\end{itemize}
\textbf{proof} --
\begin{itemize}
\item \textbf{have} \(1:(\bigsqcup i \in A \cdot P(i)) \text{ is R1}\)
  \textbf{by} (unfold \textit{Healthy-def}, subst \textit{R1-USUP}, simp-all add: \textit{Healthy-if} assms closure)
\item \textbf{have} \(2:(\bigsqcup i \in A \cdot P(i)) \text{ is R2c}\)
  \textbf{by} (unfold \textit{Healthy-def}, subst \textit{R2c-USUP}, simp-all add: \textit{Healthy-if} assms \textit{RR-implies-R2c} closure)
\item \textbf{show} \(?\text{thesis}\)
  using \(1\ 2\) by (rule-tac \textit{RR-intro}, simp-all add: unrest assms)
\end{itemize}
\textbf{qed}

\textbf{lemma} \textit{seq-RR-closed} \[\text{clos}eu\]:
\begin{itemize}
\item \textbf{assumes} \(P \text{ is RR} \ Q \text{ is RR}\)
\item \textbf{shows} \(P ;; Q \text{ is RR}\)
\end{itemize}
\textbf{unfolding} \textit{Healthy-def}
\textbf{by} (simp add: \textit{RR-def} \textit{Healthy-if} assms \textit{RR-implies-R2} ex-unrest unrest)

\textbf{lemma} \textit{power-Suc-RR-closed} \[\text{clos}eu\]:
\begin{itemize}
\item \textbf{P is RR} \(\Rightarrow\) \(P ;; P^i \text{ is RR}\)
\end{itemize}
\textbf{by} (induct \(i\), simp-all add: closure upred-semiring.power-Suc)

\textbf{lemma} \textit{seqr-iter-RR-closed} \[\text{clos}eu\]:
\begin{itemize}
\item \textbf{I} \(\neq\) \(\{}\)
\item \textbf{\(\forall i. \ i \in \text{set}(I) \Rightarrow P(i) \text{ is RR} \ \Rightarrow (;; i : I \cdot P(i)) \text{ is RR}\)}
\item \textbf{apply} (induct \(I\), simp-all)
\item \textbf{apply} (rename-tac \(i \ I\)
\item \textbf{apply} (case-tac \(I\)
\item \textbf{apply} (simp-all add: \textit{seq-RR-closed})
\item \textbf{done}

\textbf{lemma} \textit{cond-tt-RR-closed} \[\text{clos}eu\]:
\begin{itemize}
\item \textbf{assumes} \(P \text{ is RR} \ Q \text{ is RR}\)
\item \textbf{shows} \(P \triangleright \ \triangleright tr \ \trianglelefteq Q \text{ is RR}\)
\item \textbf{apply} (rule \textit{RR-intro})
\item \textbf{apply} (simp-all add: unrest assms)
\item \textbf{apply} (simp-all add: \textit{Healthy-def})
\end{itemize}
apply (simp-all add: R1-cond R2c-condr Healthy-if assms RR-implies-R2c closure R2c-tr'-minus-tr)
done

lemma rea-skip-RR [closure]:
  II \_ 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 conj-tr-strict-RR-closed [closure]:
  assumes P is RR
  shows (P \land $tr <_u tr'$) is RR
proof -
  have RR(RR(P) \land $tr <_u tr'$) = (RR(P) \land $tr <_u tr'$)
    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]:
  \[ i > 0; P is RR \] \implies P ^ 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 ^ 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 ;; (\prod i \in \{0..\} \cdot Q ^ i)
    by (simp add: ustar-def)
  also have \_ = P ;; (II \cap (\prod i \in \{1..\} \cdot Q ^ i))
    by (metis One-nat-def UNF-atLeast-first upred-semiring.power-0)
  also have \_ = (P \lor P ;; (\prod i \in \{1..\} \cdot Q ^ i))
    by (simp add: disj-upred-def[THEN sym] seqr-or-distr)
  also have \_ is RR
proof -
  have (\prod i \in \{1..\} \cdot Q ^ 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 = (\bigsqcap i ∈ {0..} \cdot P ^ i) * Q
    by (simp add: ustar-def)
  also have \ldots = (P ∩ (\prod i ∈ {1..} \cdot P ^ i)) * Q
    by (metis One-nat-def UINF-atLeast-first upred-semiring.power-0)
  also have \ldots = (Q ∪ (\prod i ∈ {1..} \cdot P ^ i)) * Q
    by (simp add: disj-upred-def THEN sym seqr-or-distl)
  also have \ldots is RR
    proof –
      have (\prod i ∈ {1..} \cdot P ^ 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]:
\[
\langle \text{tr} \# \text{e} ; \text{ok} \# \text{e} ; \text{wait} \# \text{e} ; \text{out} \alpha \# \text{e} \rangle = \langle \text{tr} \hat{u} \leq \text{u} \rangle \text{ is RR}
\]
  apply (rel-auto)
  apply (metis (no-types, lifting) Prefix-Order.same-prefix-prefix less-eq-list-def prefix-concat-minus zero-list-def)
  done

lemma rea-subst-R1-closed [closure]: P[v\], 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 $P ; ; II_r = P II_r ; ; P = P$
proof
  have 1: $RR(P) ; ; II_r = RR(P)$
    by (rel-auto)
  have 2: $II_r ; ; RR(P) = RR(P)$
    by (rel-auto)
  from 1 2 show $P ; ; II_r = P II_r ; ; P = P$
    by (simp-all add: Healthy-if assms)

qed

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

lemma rea-not-not [rpred]: $P$ is R1
  implies $(\neg r \neg r P) = P$
by (simp add: rea-not-def R1-negate-R1 Healthy-if)

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

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

lemma rea-true-impl [rpred]:
  $P$ is R1
  implies $(true_r \Rightarrow r P) = P$
by (simp add: rea-impl-def R1-negate-R1 R1-false Healthy-if)

lemma rea-true-impl' [rpred]:
  $P$ is R1
  implies $(true \Rightarrow r, P) = P$
by (simp add: rea-impl-def R1-negate-R1 R1-false Healthy-if)

lemma rea-false-impl [rpred]:
  $P$ is R1
  implies $(false \Rightarrow r P) = true_r$
by (simp add: rea-impl-def rpred Healthy-if)

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

lemma rea-impl-false [simp]: $(P \Rightarrow r false) = (\neg r P)$
by (rel-simp)
lemma rea-impl-refl [rpred]: \( P \text{ is } R1 \implies (P \Rightarrow P) = true_r \)
by (rel-blast)

lemma rea-impl-conj [rpred]:
\( (P \Rightarrow Q, R) = ((P \land Q) \Rightarrow R) \)
by (rel-auto)

lemma rea-impl-mp [rpred]:
\( (P \land (P \Rightarrow Q)) = (P \Rightarrow Q \land R) \)
by (rel-auto)

lemma rea-impl-conj-combine [rpred]:
\( ((P \Rightarrow Q) \land (P \Rightarrow R)) = (P \Rightarrow Q \land R) \)
by (rel-auto)

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

lemma rea-not-true [simp]: \( \neg_r true = false \)
by (rel-auto)

lemma rea-not-demorgan1 [simp]:
\( \neg_r (P \land Q) = (\neg_r P \lor \neg_r Q) \)
by (rel-auto)

lemma rea-not-demorgan2 [simp]:
\( \neg_r (P \lor Q) = (\neg_r P \land \neg_r Q) \)
by (rel-auto)

lemma rea-not-or [rpred]:
\( P \text{ is } R1 \implies (P \lor \neg_r P) = true_r \)
by (rel-blast)

lemma rea-not-and [simp]:
\( (P \land \neg_r P) = false \)
by (rel-auto)

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

lemma rea-not-UINF [simp]:
  \( A \neq \{\} \implies (\neg_r (\prod_i \in A \cdot Q(i))) = (\bigcup_i \in A \cdot \neg_r Q(i)) \)
  by (rel-auto)

lemma USUP-mem-rea-true [simp]: \( A \neq \{\} \implies (\bigcup i \in A \cdot true_r) = true_r \)
  by (rel-auto)

lemma USUP-ind-rea-true [simp]: \( (\bigcup i \in A \cdot true_r) = true_r \)
  by (rel-auto)

lemma UINF-rea-impl: \( (\bigcap P \in A \cdot F(P) \implies G(P)) = (\bigcup P \in A \cdot F(P) \implies (\bigcap P \in A \cdot G(P))) \)
  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]: \( I \neq \{\} \implies R4(\bigcup i \in I \cdot P(i)) = (\bigcup i \in I \cdot R4(P(i))) \)
  by (rel-auto)

lemma R5-USUP [rpred]: \( I \neq \{\} \implies R5(\bigcup i \in I \cdot P(i)) = (\bigcup i \in I \cdot R5(P(i))) \)
  by (rel-auto)

lemma R4-UINF [rpred]: \( R4(\prod i \in I \cdot P(i)) = (\prod i \in I \cdot R4(P(i))) \)
  by (rel-auto)

lemma R5-UINF [rpred]: \( R5(\prod i \in I \cdot P(i)) = (\prod i \in I \cdot R5(P(i))) \)
  by (rel-auto)

4.6 UTP theory

We create a UTP theory of reactive relations which in particular provides Kleene star theorems
**4.7 Instantaneous Reactive Relations**

Instantaneous Reactive Relations, where the trace stays the same.

**Lemma** skip-rea-Instant [closure]: \( H_r \) 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, 'α, 'β) rel-rp ⇒ ('t, 'α, 'β) rel-rp where  
[upred-defs]: RC1(P) = (¬\_r (¬\_r P ;; true\_r) )

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

**lemma** RC-intro: \[ P \text{ is } RR; ((\neg \_r (\neg \_r P) ;; \text{true}_r) = P) \] ⇒ P is RC  
by (simp add: Healthy-def RC1-def RC-def)

**lemma** RC-intro': \[ P \text{ is } RR; P \text{ is } RC1 \] ⇒ P is RC  
by (simp add: Healthy-def RC1-def RC-def)

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

**lemma** RC1-mono: P \sqsubseteq Q ⇒ RC1 (P) \sqsubseteq 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 \(\neg\_r RR P \text{ is } RR \)  
  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) = 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  
sows P[0, <s>/str, \$tr′] ⊑ P[0, <t>/str, \$tr′]  
proof –  
from assms(2) have (RC P)[0, <s>/str, \$tr′] ⊑ (RC P)[0, <t>/str, \$tr′]

38
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 \implies P is RC1
by (metis Healthy-def RC-R2-def RC-implies-RR comp-eq-dest-lhs)

lemma RC1-trace-ext-prefix:
out α ♯ e \implies RC1(\neg_\tau \; \text{str} \; \tau_\epsilon \leq_\epsilon \; \text{str} \; \epsilon) = (\neg_\tau \; \text{str} \; \tau_\epsilon \leq_\epsilon \; \text{str} \; \epsilon)
by (rel-auto, blast, metis (no-types, lifting) dual-order.trans)

lemma RC1-conj: RC1(P \land Q) = (RC1(P) \land RC1(Q))
by (rel-blast)

lemma conj-RC1-closed [closure]:
[ P is RC; Q is RC ] \implies P \land Q is RC1
by (simp add: Healthy-def RC1-conj)

lemma disj-RC1-closed [closure]:
assumes P is RC Q is RC1
shows (P \lor Q) is RC1
proof -
  have 1:RC1(\neg_\tau \; \text{str} \; \tau_\epsilon \leq_\epsilon \; \text{str} \; \epsilon) = (\neg_\tau \; \text{str} \; \tau_\epsilon \leq_\epsilon \; \text{str} \; \epsilon)
  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 \land 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, 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 ] \implies (P \lor Q) is RC
by (metis Healthy-def RC-R2-def RC-implies-RR assms comp-apply disj-RC1-closed disj-RR)

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

lemma UINF-mem-RC-closed [closure]:
assumes \(\forall \ i. \ P \ i \text{ is RC}\)
shows \((\bigcap_{i \in A} \cdot P \ i)\) is RC
proof
  have \(\text{RC}(\bigcap_{i \in A} \cdot P \ i) = (\text{RC}1 \circ \text{RR})(\bigcap_{i \in A} \cdot P \ i)\)
    by (simp add: RC-def)
  also have \(...) = \(\text{RC}1(\bigcap_{i \in A} \cdot P \ i)\)
    by (rel-blast)
  also have \(...) = (\bigcap_{i \in A} \cdot \text{RC}1(P \ i))
    by (simp add: Healthy-if assms)
  also have \(...) = (\bigcap_{i \in A} \cdot P \ i)
    by (simp add: Healthy-if assms)
  finally show \(\text{thesis}\)
    by (simp add: Healthy-def)
qed

lemma UINF-ind-RC-closed [closure]:
assumes \(\forall \ i. \ P \ i \text{ is RC}\)
shows \((\bigcap_{i \in A} \cdot P \ i)\) 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 \(\forall \ i. i \in A \Rightarrow P \ i \text{ is RC1 A} \neq \{}\)
shows \((\bigsup_{i \in A} \cdot P \ i)\) is RC1
proof
  have \(\text{RC1}(\bigsup_{i \in A} \cdot P \ i) = \text{RC1}(\bigsup_{i \in A} \cdot \text{RC}1(P \ i))\)
    by (simp add: Healthy-if assms cong: USUP-cong)
  also from assms \(2\) have \(...) = (\bigsup_{i \in A} \cdot \text{RC}1(P \ i))
    using dual-order.trans by (rel-blast)
  also have \(...) = (\bigsup_{i \in A} \cdot P \ i)
    by (simp add: Healthy-if assms cong: USUP-cong)
  finally show \(\text{thesis}\)
    using Healthy-def by blast
qed

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

lemma neg-trace-ext-prefix-RC [closure]:
\([\ [tr \sharp e; \ ok \sharp e; \ wait \sharp e; \ out \alpha \sharp e \ ] \] \Rightarrow \neg \ \text{str \ ' \ u \ \ v \ \ str \ ' \ is \ RC}\)
by (rule RC-intro, simp add: closure, metis RC1-def RC1-trace-ext-prefix)

lemma RC1-unrest:
\([\ \text{mwb-lens} \ x; \ x \bowtie tr \ ] \Rightarrow \text{str} \ ' \sharp RC1(P)\)
by (simp add: RC1-def unrest)
lemmas $\text{RC-unrest-dashed \ [unrest]}$:
\[
[P \text{ is RC; } mwb-lens x; x \bowtie tr] \implies \$x' \parallel P
\]
by \text{metis Healthy-if RC1-unrest RC-implies-RC1}

lemmas $\text{RC1-RR-closed}$: $P \text{ is RR} \implies \text{RC1}(P) \text{ is RR}$
by \text{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 it is necessary to use a modification of R3 from Butterfield et al. \cite{1} that explicitly states that intermediate waiting states do not propagate final state variables. In order to do this we need an additional observational variable that captures the program state that we call st. Upon this foundation, we can define operators for reactive programs \cite{3}.

alphabet $'s \text{ rsp-vars} = 't \text{ rp-vars} +$
\text{st :: 's}

declare \text{rsp-vars.defs \ [lens-defs]}

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

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

\text{notation} \text{rsp-vars-child-lens}_a (\Sigma_a)
\text{notation} \text{rsp-vars-child-lens} (\Sigma_S)

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

\text{translations}
\text{-svid-st-alpha => CONST} \text{ rsp-vars-child-lens}

\text{lemma} \text{srea-var-ords [usubst]}
\text{ $st \prec_v \$st'$}
\text{ $ok' \prec_v \$st \$ok' \prec_v \$st' \$ok \prec_v \$st' \$ok' \prec_v \$st}
lemma st-bij-lemma: bij-lens \((st_a +_L S)\)
by (unfold-locales, auto simp add: lens-defs)

lemma rea-lens-equiv-st-rest: \(\Sigma_R \approx L st +_L \Sigma_S\)
proof –
have \(st +_L \Sigma_S = (st_a +_L S) :L \Sigma_R\)
by (simp add: plus-lens-distr st-def rsp-vars-child-lens-def)
also have \(\ldots \approx L I_L :L \Sigma_R\)
using lens-equiv-via-bij st-bij-lemma by auto
also have \(\ldots = \Sigma_R\)
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 \(\ldots \approx L I_L :L \Sigma_R\)
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]: \(x :_L \text{fst}\) \(_L :L \text{fst} \times_L \text{st} = (\$st:x)\)
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 \(\lambda(\text{ok}, \text{wait}, \text{tr}, \text{r}). (\text{ok}, \text{wait}, \text{tr}, \text{st}, \text{r}, \text{more} \text{r})\)
apply (unfold-locales)
apply (rule injI)
apply (clarsimp)
done

interpretation alphabet-state-rel: lens-interp \(\lambda(\text{ok}, \text{ok}', \text{wait}, \text{wait}', \text{tr}, \text{tr}', \text{r}, \text{r}')\).
(\(\text{ok}, \text{ok}', \text{wait}, \text{wait}', \text{tr}, \text{tr}', \text{st}, \text{r}, \text{st}, \text{r}', \text{more} \text{r}, \text{more} \text{r}')\)
apply (unfold-locales)
apply (rule injI)
apply (clarsimp)
done

lemma unrest-st’-neg-RC [unrest]:
assumes \(P \text{ is RR} \text{ P is RC}\)
shows \(\$st' \notin P\)
proof –
have \(P = (\neg_r \neg_r P)\)
by (simp add: closure rpred assms)
also have \(\ldots = (\neg_r (\neg_r P) :: \text{true}_r)\)
by (metis Healthy-if RC1-def RC-implies-RC1 assms(2) calculation)
also have \(\$st' \notin \ldots\)
by (rel-auto)
finally show thesis.
qed

lemma ex-st′-RR-closed [closure]:
assumes P is RR
shows (∃ $st′ · P) is RR
proof –
have RR (∃ $st′ · RR(P)) = (∃ $st′ · RR(P))
by (rel-auto)
thus thesis
by (metis Healthy-def assms)
qed

lemma unrest-st′-R4 [unrest]:
$st′ ≠ P ⇒ $st′ ≠ R4(P)
by (rel-auto)

lemma unrest-st′-R5 [unrest]:
$st′ ≠ P ⇒ $st′ ≠ R5(P)
by (rel-auto)

6.2 State Lifting

abbreviation lift-state-rel ([·]s)
where [P]s ≡ P ⊕ p (st × L st)

abbreviation drop-state-rel ([·]s)
where [P]s ≡ P↾e (st × L st)

abbreviation lift-state-pre ([·]s<)
where [p]s< ≡ [[p]<]s

abbreviation drop-state-pre ([·]s<)
where [p]s< ≡ [[p]<]s

abbreviation lift-state-post ([·]s>)
where [p]s> ≡ [[p]>]s

abbreviation drop-state-post ([·]s>)
where [p]s> ≡ [[p]>]s

lemma st′-unrest-st-lift-pred [unrest]:
$st′ ≠ [a]s<
by (pred-auto)

lemma out-alpha-unrest-st-lift-pre [unrest]:
out α ≠ [a]s<
by (rel-auto)

lemma R1-st′-unrest [unrest]: $st′ ≠ P ⇒ $st′ ≠ R1(P)
by (simp add: R1-def unrest)

lemma R2c-st′-unrest [unrest]: $st′ ≠ P ⇒ $st′ ≠ R2c(P)
by (simp add: R2c-def unrest)
lemma \textit{st-lift-R1-true-right}: \([b]_{S<} \Rightarrow R1(\text{true}) = [b]_{S<}\)
by (rel-auto)

lemma \textit{R2c-lift-state-pre}: \(R2c([b]_{S<}) = [b]_{S<}\)
by (rel-auto)

6.3 Reactive Program Operators

6.3.1 State Substitution

Lifting substitutions on the reactive state

definition \textit{usubst-st-lift} ::
  \(\sigma \usubst \Rightarrow (\alpha \times \beta) \usubst \ (\lfloor \cdot \rfloor_{S\sigma})\) where

[upred-defs]: \([\sigma]_{S\sigma} = [\sigma \oplus_{s} st]_{s}\)

abbreviation \textit{st-subst} :: \(\sigma \usubst \Rightarrow (\alpha \times \beta) \usubst \ (\lfloor \cdot \rfloor_{S\sigma})\) where

\(\sigma \uparrow_{S} P \equiv [\sigma]_{S\sigma} \uparrow P\)

translations

\(\sigma \uparrow_{S} P \leq [\sigma \oplus_{s} st]_{s} \uparrow P\)

\(\sigma \uparrow_{S} P \leq [\sigma]_{S\sigma} \uparrow P\)

lemma \textit{st-lift-lemma}:

\([\sigma]_{S\sigma} = \sigma \oplus_{s} (\text{fst} : L \times L)\)
by (auto simp add: upred-defs lens-defs prod_case_if)

lemma \textit{unrest-st-lift} [unrest]:

\(\textbf{fixes} \ x :: \alpha \Rightarrow (\alpha \times \beta) \usubst \ (\lfloor \cdot \rfloor_{S\sigma})\)

\(\textbf{assumes} \ x \triangleleft \triangleleft (\text{st})_{c}\)

\(\textbf{shows} \ x \uparrow [\sigma]_{S\sigma} (\text{is} \ ?P)\)
by (simp add: st-lift-lemma)

(\textit{metis \ \textbf{assms} in-var-def in-var-prod-lens \ textbf{prod-comp-left-id} \ st-vwb-lens unrest-subst-alpha-ext \ vwb-lens-wb})

lemma \textit{id-st-subst} [usubst]:

\([\text{id}]_{S\sigma} = \text{id}\)
by (pred-auto)

lemma \textit{st-subst-comp} [usubst]:

\([\sigma]_{S\sigma} \circ [g]_{S\sigma} = [\sigma \circ g]_{S\sigma}\)
by (rel-auto)

definition \textit{lift-cond-srea} ([\lfloor \cdot \rfloor_{S\sigma}) where

[upred-defs]: \([b]_{S<} = [b]_{S<}\)

lemma \textit{unrest-lift-cond-srea} [unrest]:

\(x \uparrow [b]_{S<} \Rightarrow x \uparrow [b]_{S<}\)
by (simp add: lift-cond-srea_def)

lemma \textit{st-subst-RR-closed} [closure]:

\(\textbf{assumes} \ P \text{ is RR}\)

\(\textbf{shows} \ [\sigma]_{S\sigma} \uparrow P \text{ is RR}\)

\textbf{proof} -

\(\textbf{have} \ RR([\sigma]_{S\sigma} \uparrow RR(P)) = [\sigma]_{S\sigma} \uparrow RR(P)\)
by (rel-auto)

44
thus \( \text{?thesis} \)
by (metis Healthy-def assms)
qed

lemma subst-lift-cond-srea [usubst]: \( \sigma \uparrow_S [P]_{S<} = [\sigma \uparrow P]_{S<} \)
by (rel-auto)

lemma st-subst-rea-not [usubst]: \( \sigma \uparrow_S (\neg_r P) = (\neg_r \sigma \uparrow_S P) \)
by (rel-auto)

lemma st-subst-seq [usubst]: \( \sigma \uparrow_S (P ;; Q) = \sigma \uparrow_S P ;; Q \)
by (rel-auto)

lemma st-subst-RC-closed [closure]:
assumes \( P \) is RC
shows \( \sigma \uparrow_S P \) 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 :: \( ('s :\to \) 's, 't :\to \) trace, 'a) hrel-rsp \( (\cdot r) \) where
[upred-defs]: \( (\cdot r) = (\$tr \cdot \to u \cdot \$tr \land \cdot \uparrow_S \cdot S \cdot \to u \cdot \$\Sigma_S \cdot = u \cdot \$\Sigma) \)

syntax
- assign-rea :: svids \to uexprs \to logic \( (\cdot \cdot :=) \cdot r \cdot 90 \)
- assign-rea :: svids \to uexprs \to logic \( \text{infixr} := r \cdot 90 \)

translations
- assign-rea xs vs \( \Rightarrow \) CONST rea-assigns \( (-\mk-usubst \cdot \mk-id \cdot x \cdot s) \cdot x \cdot s \)
- assign-rea x v \( \Leftarrow \) CONST rea-assigns \( \cdot \mk-subst-upd \cdot \mk-id \cdot x \cdot v \)
- assign-rea x v \( \Leftarrow \) assign-rea \( (-\mk-svar x) \cdot v \)
- x,y := u,v \( \Leftarrow \) CONST rea-assigns \( \cdot \mk-subst-upd \cdot \mk-subst-upd \cdot \mk-id \cdot \cdot \mk-svar x \cdot y \cdot u \cdot (\cdot \mk-svar y) \cdot v \)

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

lemma st-subst-assigns-rea [usubst]:
\( \sigma \uparrow_S (\cdot r) = (\cdot \uparrow_S \sigma) \)
by (rel-auto)

lemma st-subst-rea-skip [usubst]:
\( \sigma \uparrow_S I_r = (\cdot r) \)
by (rel-auto)

lemma rea-assigns-comp [rpred]:
assumes \( P \) is RR
shows \( (\cdot r) ;; P = \sigma \uparrow_S P \)
proof –
have \( (\cdot r) ;; (\cdot RR P) = \sigma \uparrow_S (\cdot RR P) \)
by (rel-auto)
thus \( \text{?thesis} \)

45
by (metis Healthy-def assms)

qed

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

qed

lemma rea-assigns-st-subst [unsubst]:
[σ ⊗ₚ st]ₚ † ⟨q⟩ₚ = ⟨σ ◦ ⟨q⟩ₚ⟩ₚ
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,′α,′β) rel-rsp ⇒
  ′s upred ⇒
  (′s,′t,′α,′β) rel-rsp ⇒
  (′s,′t,′α,′β) rel-rsp ((3- ◦ - ◦ R/ -) [52,0,53] 52) where
cond-srea P b Q ≡ P ◦ b ◦ ⊲ Q

lemma st-cond-assigns [r_pred]:
⟨σ⟩ₚ ◦ b ◦ ⊲ R ⟨̺⟩ₚ = ⟨σ ◦ b ◦ ̺⟩ₚ
by (rel-auto)

lemma cond-srea-RR-closed [closure]:
assumes P is RR Q is RR
shows P ◦ b ◦ ⊲ R Q is RR
proof –
  have RR(RR(P) ◦ b ◦ ⊲ R RR(Q)) = RR(P) ◦ b ◦ ⊲ 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 ◦ b ◦ ⊲ R Q is RC1
proof –
  have RC1(RC1(P) ◦ b ◦ ⊲ R RC1(Q)) = RC1(P) ◦ b ◦ ⊲ 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 ◦ b ◦ ⊲ R Q is RC

by (rule RC-intro', simp-all add: closure cond-srea-RC1-closed RC-implies-RC1 assms)

lemma R4-cond [rpred]: $R_4(P \triangleright b \triangleright R Q) = (R_4(P) \triangleright b \triangleright R_4(Q))$
  by (rel-auto)

lemma R5-cond [rpred]: $R_5(P \triangleright b \triangleright R Q) = (R_5(P) \triangleright b \triangleright R_5(Q))$
  by (rel-auto)

6.3.4 Assumptions

definition rea-assume :: 's upred ⇒ ('s, 't::trace, 'a) hrel-rsp ($\downarrow r$) where
  [upred-defs]: $[b]_r^r = (H_r \triangleright b \triangleright R false)$

lemma rea-assume-RR [closure]: $[b]_r^r \triangleright b \triangleright R R$ is RR
  by (simp add: rea-assume-def closure)

lemma rea-assume-false [rpred]: $[false]_r^r = false$
  by (rel-auto)

lemma rea-assume-true [rpred]: $[true]_r^r = H_r$
  by (rel-auto)

lemma rea-assume-comp [rpred]: $[b]_r^r \sqcup [c]_r^r = [b \land c]_r^r$
  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 =⇒ 'b) ⇒ ('a, 't::trace, 'α) rp ⇒ ('b, 't, 'α) rp
  where
    [lens-defs]: lmap R = lmap D ◦ lmap [rp-vars]

definition map-rsp-st :: ('s ⇒ 't) ⇒ ('s, 't::trace, 'a) rp ⇒ ('t, 'b) rp where
  [lens-defs]: map-rsp-st f = (λr. (st v = f (st v r), ... = rsp-vars.more r))

definition map-st-lens :: ('σ ⇒ 'ψ) ⇒ ('σ, 't::trace, 'a) rsp-vars-scheme ⇒ ('ψ, 't::trace, 'a) rsp-vars-scheme
  where
    [lens-defs]: map-st-lens l = map-rsp-st (get l)
      lens-get = map-rsp-st (get l) o rsp-vars.st.v

lemma map-set-vwb [simp]: vwb-lens X ⇒ vwb-lens (map-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-st L ≡ (map-st L 0 L) × L (map-st L 0 L)

abbreviation abs-st (⟨−⟩S) where
    abs-st P ≡ P [e abs-st L}

47
6.3.6 Reactive Frames and Extensions

definition rea-frame :: (′a ⇒ ′a) ⇒ (′a, ′t::trace) rdes ⇒ (′a, ′t) rdes where
[upred-defs]: rea-frame x P = frame (ok +, wait +, tr +, (x ; s)!) P

definition rea-frame-ext :: (′a ⇒ ′β) ⇒ (′a, ′t::trace) rdes ⇒ (′β, ′t) rdes where
[upred-defs]: rea-frame-ext a P = rea-frame a (rel-aext P (map-st_L a))

syntax
- rea-frame :: salpha ⇒ logic ⇒ logic (-[1] [99,0] 100)
- rea-frame-ext :: salpha ⇒ logic ⇒ logic (-[1]+ [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 => x:[P]r+ is RR
by (simp add: rea-frame-ext-def closure)

lemma rel-aext-st-Instant-closed [closure]:
P is Instant => rel-aext P (map-st_L x) is Instant
by (rel-auto)

lemma rea-frame-ext-false [frame]:
x:[false]r+ = false
by (rel-auto)

lemma rea-frame-ext-skip [frame]:
wv-b-lens x => x:[II]r+, = IIr
by (rel-auto)

lemma rea-frame-ext-assigns [frame]:
wv-b-lens x => x:[(σ) r]+ = (σ ⊕ x) r

48
by (rel-auto)

**Lemma** rea-frame-ext-cond [frame]:

\[ x : [P \triangleright b \triangleright R \downarrow Q]^+ = x : [P]_{r^+} \land (b \triangleright P) x : [Q]_{r^+} \]

by (rel-auto)

**Lemma** rea-frame-ext-seq [frame]:

\[ \text{vwb-lens } x \rightarrow x : [P ; Q]^+ = x : [P]_{r^+} ; x : [Q]_{r^+} \]

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 \( x \ni y \Sigma \notin v \) \( P \) is RR

shows \( \sigma(y \mapsto_s v) \vdash S x : [P]_{r^+} = (\sigma \vdash S x : [P]_{r^+}) \ni y :=_r v \)

proof –
from assms \( 1 - 2 \) have \( \sigma(y \mapsto_s v) \vdash S x : [R \downarrow P]_{r^+} = (\sigma \vdash S x : [R \downarrow P]_{r^+}) \ni y :=_r v \)
by (rel-auto, (metis (no-types, lifting) lens-indep-lens-put-comm lens-indep-get)+)
thus \( ? \)thesis
by (simp add: Healthy-if assms)

qed

**Lemma** rea-frame-ext-subst-within [usubst]:

assumes \( \text{vwb-lens } x \text{ vwb-lens } y \Sigma \notin v \) \( P \) is RR

shows \( \sigma(x : y \mapsto_s v) \vdash S x : [P]_{r^+} = (\sigma \vdash S x : [P]_{r^+}) \ni y :=_r (v \mapsto_s x) \ni P_{r^+} \)

proof –
from assms \( 1,3 \) have \( \sigma(x : y \mapsto_s v) \vdash S x : [R \downarrow P]_{r^+} = (\sigma \vdash S x : [R \downarrow P]_{r^+}) \ni y :=_r (v \mapsto_s x) \ni RR(P)_{r^+} \)
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 ([\cdot]_S) where
[upred-defs]: rea-st-rel b = ([\cdot]_S \land \$tr' =_u \$tr)

**Definition** rea-st-rel' :: 's hrel ⇒ ('s, 't::trace, 'α, 'β) rel-rsp ([\cdot]_S') where
[upred-defs]: rea-st-rel' b = R1([\cdot]_S)

**Definition** rea-st-cond :: 's upred ⇒ ('s, 't::trace, 'α, 'β) rel-rsp ([\cdot]_S<) where
[upred-defs]: rea-st-cond b = R1([\cdot]_S<)

**Definition** rea-st-post :: 's upred ⇒ ('s, 't::trace, 'α, 'β) rel-rsp ([\cdot]_S>) where
[upred-defs]: rea-st-post b = R1([\cdot]_S>)

**Lemma** lift-state-pre-unrest [unrest]: \( x \ni (\$st) v \implies x \ni [P]_{S<} \)
by (rel-simp, simp add: lens-indep-def)

**Lemma** rea-st-rel-unrest [unrest]:
\[ \ni x \ni (\$tr' v) ; x \ni (\$st v) ; x \ni (\$st' v) \ni x \ni [P]_{S<} \]
by (simp add: add: rea-st-cond-def R1-def unrest lens-indep-sym)

**Lemma** rea-st-cond-unrest [unrest]:
\[ \ni x \ni (\$tr' v) ; x \ni (\$st v) ; x \ni (\$st' v) \ni x \ni [P]_{S<} \]
\[ x \triangleright (\text{Str})_\triangleright; x \triangleright (\text{Str'})_\triangleright; x \triangleright (\text{St})_\triangleright \implies x \triangleright [P]_{S<} \]

by (simp add: add: rea-st-cond-def R1-def unrest lens-indep-sym)

**Lemma** subst-st-cond [usubst]: \([\sigma]_{S\sigma} \downarrow [P]_{S<} = [\sigma \downarrow P]_{S<}\)
by (rel-auto)

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

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

**Lemma** rea-st-rel-RR [closure]: \([P]_{S} \text{ is RR}\)
using minus-zero-eq by (rel-auto)

**Lemma** rea-st-rel'-RR [closure]: \([P]_{S'} \text{ is RR}\)
by (rel-auto)

**Lemma** st-subst-rel [usubst]:
\[ \sigma \downarrow S [P]_{S} = [[\sigma]_{S} \downarrow P]_{S} \]
by (rel-auto)

**Lemma** st-rel-cond [rpred]:
\[ [P \triangleright b \triangleright_{r} Q]_{S} = [P]_{S} \triangleright b \triangleright_{R} [Q]_{S} \]
by (rel-auto)

**Lemma** st-rel-false [rpred]: \([\text{false}]_{S} = \text{false}\)
by (rel-auto)

**Lemma** st-rel-skip [rpred]:
\[ ([I]_{S} = ([I]_{r} :: ('s, 't::trace) rdes) \]
by (rel-auto)

**Lemma** st-rel-seq [rpred]:
\[ [P ;; Q]_{S} = [P]_{S} ;; [Q]_{S} \]
by (rel-auto)

**Lemma** st-rel-conj [rpred]:
\[ [P \wedge Q]_{S} = ([P]_{S} \wedge [Q]_{S}) \]
by (rel-auto)

**Lemma** rea-st-cond-RR [closure]: \([b]_{S<} \text{ is RR}\)
by (rule RR-intro, simp-all add: unrest closure)

**Lemma** rea-st-cond-RC [closure]: \([b]_{S<} \text{ is RC}\)
by (rule RC-intro, simp add: closure, rel-auto)

**Lemma** rea-st-cond-true [rpred]: \([\text{true}]_{S<} = \text{true}_{r}\)
by (rel-auto)

**Lemma** rea-st-cond-false [rpred]: \([\text{false}]_{S<} = \text{false}\)
by (rel-auto)

**Lemma** st-cond-not [rpred]: \((-_{r} [P]_{S<}) = [\neg P]_{S<}\)
by (rel-auto)

50
lemma \textit{st-cond-conj} [\textit{rpred}]: ([P]_{S <} \land [Q]_{S <}) = [P \land Q]_{S <}
\begin{aligned}
\text{by (rel-auto)}
\end{aligned}

lemma \textit{st-rel-assigns} [\textit{rpred}]:
\begin{aligned}
[(\sigma)]_{S} = ((\sigma)_{r} \bowtie (\alpha', 't::trace) rdes)
\end{aligned}
\begin{aligned}
\text{by (rel-auto)}
\end{aligned}

lemma \textit{cond-st-distr} [\textit{rpred}]: (P \bowtie b \bowtie R Q) ;; R = (P ;; R \bowtie b \bowtie Q ;; R)
\begin{aligned}
\text{by (rel-auto)}
\end{aligned}

lemma \textit{cond-st-miracle} [\textit{rpred}]: P is R1 \implies P \bowtie b \bowtie R false = ([b]_{S <} \land P)
\begin{aligned}
\text{by (rel-blast)}
\end{aligned}

lemma \textit{cond-st-true} [\textit{rpred}]: P \bowtie true \bowtie R Q = P
\begin{aligned}
\text{by (rel-blast)}
\end{aligned}

lemma \textit{cond-st-false} [\textit{rpred}]: P \bowtie false \bowtie R Q = Q
\begin{aligned}
\text{by (rel-blast)}
\end{aligned}

lemma \textit{cond-st-true-or} [\textit{rpred}]: P is R1 \implies (R1 true \bowtie b \bowtie R P) = ([b]_{S <} \lor P)
\begin{aligned}
\text{by (rel-blast)}
\end{aligned}

lemma \textit{cond-st-left-impl-RC-closed} [\textit{closure}]:
P is RC \implies ([b]_{S <} \Rightarrow P) is RC
\begin{aligned}
\text{by (simp add: rea-impl-def rpred closure)}
\end{aligned}

end

7 Reactive Weakest Preconditions

theory utp-rea-wp
\begin{aligned}
\text{imports utp-rea-prog}
\end{aligned}
begin

Here, we create a weakest precondition calculus for reactive relations, using the recast boolean algebra and relational operators. Please see our journal paper [3] for more information.

definition \textit{wp-rea} ::
\begin{aligned}
('t::trace, 'a) hrel-rp \Rightarrow ('t, 'a) hrel-rp \Rightarrow ('t, 'a) hrel-rp
\text{(infix wp, 60)}
\end{aligned}
\begin{aligned}
\text{where [upred-defs]: P wp_r Q = (\neg_r P ;; (\neg_r Q))}
\end{aligned}

lemma \textit{in-var-unrest-wp-rea} [\textit{unrest}]: [ \$x \parallel P; tr \bowtie x ] \implies \$x \parallel (P wp_r Q)
\begin{aligned}
\text{by (simp add: wp-rea-def unrest R1-def rea-not-def)}
\end{aligned}

lemma \textit{out-var-unrest-wp-rea} [\textit{unrest}]: [ \$x' \parallel Q; tr \bowtie x ] \implies \$x' \parallel (P wp_r Q)
\begin{aligned}
\text{by (simp add: wp-rea-def unrest R1-def rea-not-def)}
\end{aligned}

lemma \textit{wp-rea-R1} [\textit{closure}]: P wp_r Q is R1
\begin{aligned}
\text{by (rel-auto)}
\end{aligned}

lemma \textit{wp-rea-RR-closed} [\textit{closure}]: [ P is RR; Q is RR ] \implies P wp_r Q is RR
\begin{aligned}
\text{by (simp add: wp-rea-def closure)}
\end{aligned}
lemma wp-rea-impl-lemma:
\((P \text{ wp}_r Q) \Rightarrow_r (R1(P) ;; R1(Q \Rightarrow_r R))) = ((P \text{ wp}_r Q) \Rightarrow_r (R1(P) ;; R1(R)))
by (rel-auto, blast)

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

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

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

lemma wp-rea-Inf-pre [wp]:
\(P \text{ wp}_r (\bigcap_{i \in \{0..n::\text{nat}\}} Q(i)) = (\bigcap_{i \in \{0..n\}} P \text{ wp}_r Q(i))\)
by (simp add: wp-rea-def seq-UINF-distl)

lemma wp-rea-Inf-pre [wp]:
\(P \text{ wp}_r (\bigcap_{i \in \{0..n::\text{nat}\}} Q(i)) = (\bigcap_{i \in \{0..n\}} P \text{ wp}_r Q(i))\)
by (simp add: wp-rea-def seq-UINF-distl)

lemma wp-rea-seq [wp]:
assumes \(\text{ Q is R}_1\)
shows \((P ;; Q) \text{ wp}_r R = P \text{ wp}_r (Q \text{ wp}_r R)\) (is \(?lhs = \text{ ?rhs}\))
proof –
have \(?rhs = R1 (\neg P ;; R1 (Q ;; R1 (\neg R)))\)
  by (simp add: wp-rea-def rea-not-def R1-negate-R1 assms)
also have ... = R1 (\neg P ;; (Q ;; R1 (\neg R)))
  by (metis Healthy-if R1-seqr assms)
also have ... = R1 (\neg (P ;; Q) ;; R1 (\neg R))
  by (simp add: seqr-assoc)
finally show \(?thesis\)
  by (simp add: wp-rea-def rea-not-def)
qed

lemma wp-rea-skip [wp]:
assumes \(\text{ Q is R}_1\)
shows \(\text{ II} \text{ wp}_r Q = Q\)
by (simp add: wp-rea-def rpred assms Healthy-if)

**Lemma**: \(\text{wp-rea-skip}\) [wp]:

- **assumes**: \(Q\) is RR
- **shows**: \(I_r \text{ wp}_r Q = Q\)
  - by (simp add: wp-rea-def rpred closure assms Healthy-if)

**Lemma**: \(\text{power-wp-rea-RR-closed}\) [closure]:

\[R\text{ is RR; } P\text{ is RR} \implies R^i \text{ wp}_r P\text{ is RR}\]
  - by (induct \(i\), simp-all add: wp closure)

**Lemma**: \(\text{wp-rea-assigns}\) [wp]:

- **assumes**: \(P\) is RR
  - **shows**: \(\langle \sigma \rangle_r \text{ wp}_r \sigma S \sigma \vdash P\)

**Proof**:

- **have**: \(\langle \sigma \rangle_r \text{ wp}_r (\text{RR } P) = \sigma S \sigma \vdash (\text{RR } P)\)
  - by (rel-auto)
- **thus**: \(?\text{thesis}\)
  - by (metis Healthy-def assms)

**Qed**

**Lemma**: \(\text{wp-rea-miracle}\) [wp]: false wp \(r\) Q = true \(r\)
  - by (simp add: wp-rea-def)

**Lemma**: \(\text{wp-rea-disj}\) [wp]: \((P \lor Q)\) wp \(r\) R = (\(P\) wp \(r\) R \& \(Q\) wp \(r\) R)
  - by (rel-blast)

**Lemma**: \(\text{wp-rea-UINF}\) [wp]:

- **assumes**: \(A \neq \{\}\)
  - **shows**: \(\bigcap x \in A \cdot P(x)\) wp \(r\) Q = (\(\bigcap x \in A \cdot P(x)\) wp \(r\) Q)
  - by (simp add: wp-rea-def rea-not-def seq-UINF-distr not-UINF R1-UINF assms)

**Lemma**: \(\text{wp-rea-choice}\) [wp]:

\((P \cap Q)\) wp \(r\) R = (\(P\) wp \(r\) R \& \(Q\) wp \(r\) R)
  - by (rel-blast)

**Lemma**: \(\text{wp-rea-UINF-ind}\) [wp]:

\((\bigcap i \cdot P(i))\) wp \(r\) Q = (\(\bigcap i \cdot P(i)\) wp \(r\) Q)
  - by (simp add: wp-rea-def rea-not-def seq-UINF-distr’ not-UINF-ind R1-UINF-ind)

**Lemma**: \(\text{rea-assume-wp}\) [wp]:

- **assumes**: \(P\) is RC
  - **shows**: \([b]^T_r \text{ wp}_r P\) = ([b]_S \Rightarrow_r P)

**Proof**:

- **have**: \([b]^T_r \text{ wp}_r RC P\) = ([b]_S \Rightarrow_r RC P)
  - by (rel-auto)
- **thus**: \(?\text{thesis}\)
  - by (simp add: Healthy-if assms)

**Qed**

**Lemma**: \(\text{rea-star-wp}\) [wp]:

- **assumes**: \(P\) is RR \(Q\) is RR
  - **shows**: \(P^* r\) wp \(r\) Q = (\(\bigcap i \cdot P^* r\) wp \(r\) Q)

**Proof**:

- **have**: \(P^* r\) wp \(r\) Q = (\(Q\) \& \(P^* r\) wp \(r\) Q)
by (simp add: assms rrel-thy Star-alt-def wp-rea-choice wp-rea-skip)
also have ... = (I wp_r Q ∧ (∓ i ∨ P ∨ Suc i wp_r Q))
  by (simp add: uplus-power-def wp closure assms)
also have ... = (∓ i ∨ P ∨ i wp_r Q)
proof
  have P* wp_r Q = P** 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] ⇒ P wp_r Q is R2
by (simp add: wp-rea-def closure)

lemma wp-rea-false-RC1': P is R2 ⇒ 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 ⇒ P wp_r false is RC1
by (simp add: Healthy-def wp-rea-false-RC1')

lemma wp-rea-false-RR:
  [ok ♯ P; wait ♯ P; P is R2] ⇒ P wp_r false is RR
by (rule RR-R2-intro, simp-all add: unrest closure)

lemma wp-rea-false-RC:
  [ok ♯ P; wait ♯ P; P is R2] ⇒ 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] ⇒ P wp_r Q is RC1
by (rule Healthy-intro, simp add: wp-rea-def RC1-def rpred seqr-assoc RC1-prop RC-implies-RC1)

lemma wp-rea-RC [closure]: [P is RR; Q is RC] ⇒ 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^n 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 (\{\|\} / -/ \{\|\}) where
  [upred-defs]: hoare-rp p Q r = (\{\|p| < \⇒ |r| >\} ⊆ Q)

lemma hoare-rp-conseq:
lemma hoare-rp-weaken:
[ ‘p ⇒ p’; ‘q ⇒ q’; \{p\} S \{q\} r ] \implies \{p\} S \{q\} r
by (rel-auto)

lemma hoare-rp-strengthen:
[ ‘p ⇒ p’; \{p\} S \{q\} r ] \implies \{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 ⇒ \sigma \uplus q’ \implies \{p\} \langle \sigma \rangle r \{q\} r
by rel-auto

lemma skip-hoare-rp [hoare-safe]: \{p\} II \{p\} r
by rel-auto

lemma seq-hoare-rp: \[ \{p\} Q_1 \{s\} r ; \{s\} Q_2 \{r\} r \] \implies \{p\} Q_1 ; Q_2 \{r\} r
by (rel-auto)

lemma seq-est-hoare-rp [hoare-safe]:
\[ \{true\} Q_1 \{p\} r ; \{p\} Q_2 \{p\} r \] \implies \{true\} Q_1 ; Q_2 \{p\} r
by (rel-auto)

lemma seq-inv-hoare-rp [hoare-safe]:
\[ \{p\} Q_1 \{p\} r ; \{p\} Q_2 \{p\} r \] \implies \{p\} Q_1 ; Q_2 \{p\} r
by (rel-auto)

lemma cond-hoare-rp [hoare-safe]:
\[ \{b \land p\} P \{r\} r ; \{\neg b \land p\} Q \{r\} r \] \implies \{p\} P \land b \supseteq Q \{r\} r
by (rel-auto)

lemma repeat-hoare-rp [hoare-safe]:
\{p\} Q \{p\} r \implies \{p\} Q \land \neg n \{p\} r
by (induct n, rel-auto+)

lemma UINF-ind-hoare-rp [hoare-safe]:
\[ \land i . \{p\} Q(i) \{r\} r \] \implies \{p\| i \cdot Q(i) \{r\} r
by (rel-auto)

lemma star-hoare-rp [hoare-safe]:
\{p\} Q \{p\} r \implies \{p\} Q^{*} \{p\} r
by (simp add: ustar-def hoare-safe)

lemma conj-hoare-rp [hoare-safe]:
\[ \{p_1\} Q_1 \{r_1\} r ; \{p_2\} Q_2 \{r_2\} r \] \implies \{p_1 \land p_2\} Q_1 \land Q_2 \{r_1 \land r_2\} r

55
by (rel-auto)

lemma iter-hoare-rp [hoare-safe]:
\[ \langle I \rangle P \langle I \rangle_r \Rightarrow \langle I \rangle P^r \langle I \rangle_r \]
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

throry 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


