

Formalization of Refinement Calculus for Reactive Systems

Viorel Preoteasa
Aalto University, Finland

October 11, 2017

Abstract

We present a formalization of refinement calculus for reactive systems. Refinement calculus is based on monotonic predicate transformers (monotonic functions from sets of post-states to sets of pre-states), and it is a powerful formalism for reasoning about imperative programs. We model reactive systems as monotonic property transformers that transform sets of output infinite sequences into sets of input infinite sequences. Within this semantics we can model refinement of reactive systems, (unbounded) angelic and demonic nondeterminism, sequential composition, and other semantic properties. We can model systems that may fail for some inputs, and we can model compatibility of systems. We can specify systems that have liveness properties using linear temporal logic, and we can refine system specifications into systems based on symbolic transitions systems, suitable for implementations.

Contents

1	Introduction	2
2	Linear Temporal Logic	2
3	Monotonic Predicate Transformers	8
3.1	Basic predicate transformers	8
3.2	Conjunctive predicate transformers	10
3.3	Fusion of predicate transformers	14
4	Reactive Systems	16
4.1	Symbolic transition systems	18
4.2	Example: COUNTER	32
4.3	Example: LIVE	34

1 Introduction

This is a formalization of refinement calculus for reactive systems that is presented in [6].

Refinement calculus [1, 3] has been developed originally for input output imperative programs, and is based on a predicate transformer semantics of programs with a weakest precondition interpretation.

We extend the standard refinement calculus to reactive systems [4]. Within our framework a reactive system is seen as a system that accepts as input an infinite sequence of values and produces as output an infinite sequence of values. The semantics of these systems is given as *monotonic property transformers*. These are monotonic functions which maps sets of output sequences (output properties) into sets of input sequences (input properties). For a set of output sequences q , the monotonic property transformer S applied to q returns all input sequences from which the computation of S always produces a sequence from q .

Our work extends also the relational interfaces framework of [7] which can handle only finite safety properties to infinite properties and liveness.

This formalization is organized in three sections. Section 2 presents an algebraic formalization of linear temporal logic. Section 3 introduces basic constructs from refinement calculus, and finally Section 4 applies the refinement calculus to reactive systems.

```
theory Temporal imports Main  
begin
```

2 Linear Temporal Logic

In this section we introduce an algebraic axiomatization of Linear Temporal Logic (LTL). We model LTL formulas semantically as predicates on traces. For example the LTL formula $\alpha = \Box \Diamond (x = 1)$ is modeled as a predicate $\alpha : (nat \Rightarrow nat) \Rightarrow bool$, where $\alpha x = True$ if $x\ i = 1$ for infinitely many $i : nat$. In this formula \Box and \Diamond denote the always and eventually operators, respectively. Formulas with multiple variables are modeled similarly. For example a formula α in two variables is modeled as $\alpha : (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b) \Rightarrow bool$, and for example $(\Box \alpha) x\ y$ is defined as $(\forall i. \alpha\ x[i..] y[i..])$, where $x[i..] j = x\ (i + j)$. We would like to construct an algebraic structure (Isabelle class) which has the temporal operators as operations, and which has instantiations to $(nat \Rightarrow 'a) \Rightarrow bool$, $(nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b) \Rightarrow bool$, and so on. Ideally our structure should be such that if we have this structure on a type $'a :: temporal$, then we could extend it to $(nat \Rightarrow 'b) \Rightarrow 'a$ in a way similar to the way Boolean algebras are extended from a type $'a :: boolean_algebra$ to $'b \Rightarrow 'a$. Unfortunately, if we use for example \Box as primitive operation on our temporal structure, then we cannot extend \Box

from $'a :: \text{temporal}$ to $(\text{nat} \Rightarrow 'b) \Rightarrow 'a$. A possible extension of \square could be

$$(\square \alpha) x = \bigwedge_{i:\text{nat}} \square(\alpha x[i..]) \text{ and } \square b = b$$

where $\alpha : (\text{nat} \Rightarrow 'b) \Rightarrow 'a$ and $b : \text{bool}$. However, if we apply this definition to $\alpha : (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'b) \Rightarrow \text{bool}$, then we get

$$(\square \alpha) x y = (\forall i j. \alpha x[i..] y[j..])$$

which is not correct.

To overcome this problem we introduce as a primitive operation $!! : 'a \Rightarrow \text{nat} \Rightarrow 'a$, where $'a$ is the type of temporal formulas, and $\alpha!!i$ is the formula α at time point i . If α is a formula in two variables as before, then

$$(\alpha!!i) x y = \alpha x[i..] y[i..].$$

and we define for example the the operator always by

$$\square \alpha = \bigwedge_{i:\text{nat}} \alpha!!i$$

notation

bot (\perp) **and**
top (\top) **and**
inf (**infixl** \sqcap 70)
and *sup* (**infixl** \sqcup 65)

class *temporal* = *complete-boolean-algebra* +

fixes *at* :: $'a \Rightarrow \text{nat} \Rightarrow 'a$ (**infixl** $!!$ 150)

assumes [*simp*]: $a !! i !! j = a !! (i + j)$

assumes [*simp*]: $a !! 0 = a$

assumes [*simp*]: $\top !! i = \top$

assumes [*simp*]: $\neg(a !! i) = (\neg a) !! i$

assumes [*simp*]: $(a \sqcap b) !! i = (a !! i) \sqcap (b !! i)$

begin

definition *always* :: $'a \Rightarrow 'a$ (\square (-) [900] 900) **where**

$\square p = (\text{INF } i . p !! i)$

definition *eventually* :: $'a \Rightarrow 'a$ (\diamond (-) [900] 900) **where**

$\diamond p = (\text{SUP } i . p !! i)$

definition *next* :: $'a \Rightarrow 'a$ (\circ (-) [900] 900) **where**

$\circ p = p !! (\text{Suc } 0)$

definition *until* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infix** *until* 65) **where**

$(p \text{ until } q) = (\text{SUP } n . (\text{INFIMUM } \{i . i < n\} (at p))) \sqcap (q !! n)$

end

Next lemma, in the context of complete boolean algebras, will be used to prove $-(p \text{ until } -p) = \square p$.

```

context complete-boolean-algebra
begin
  lemma until-always:  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup ((p :: \text{nat} \Rightarrow 'a) n)) \leq p n$ 
    proof -
      have  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq (\text{INF } i : \{i. i \leq n\}. p i)$ 
        proof (induction n)
          have  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq (\text{SUP } i : \{i. i < 0\}. - p i) \sqcup p 0$ 
            by (rule INF-lower, simp)
          also have  $\dots \leq (\text{INF } i : \{i. i \leq 0\}. p i)$ 
            by simp
          finally show  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq (\text{INF } i : \{i. i \leq 0\}. p i)$ 
            by simp
        next
          fix  $n :: \text{nat}$  assume  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq (\text{INF } i : \{i. i \leq n\}. p i)$ 
            also have  $\bigwedge i. i \leq n \implies \dots \leq p i$  by (rule INF-lower, simp)
            finally have  $[\text{simp}]: \bigwedge i. i \leq n \implies (\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq p i$ 
              by simp
            show  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq (\text{INF } i : \{i. i \leq \text{Suc } n\}. p i)$ 
              proof (rule INF-greatest, safe, cases)
                fix  $i :: \text{nat}$ 
                  assume  $i \leq n$  from this show  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq p i$  by simp
                next
                  fix  $i :: \text{nat}$ 
                    have  $A: \{i. i \leq n\} = \{i. i < \text{Suc } n\}$  by auto
                    have  $B: (\text{SUP } i : \{i. i \leq n\}. - p i) \leq -(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n)$ 
                      by (metis (lifting, mono-tags)  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq (\text{INF } i : \{i. i \leq n\}. p i)$  compl-mono uminus-INF)
                    assume  $i \leq \text{Suc } n$  and  $\neg i \leq n$ 
                    from this have  $[\text{simp}]: i = \text{Suc } n$  by simp
                    have  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq (\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \sqcap ((\text{SUP } i : \{i. i \leq n\}. - p i) \sqcup p (\text{Suc } n))$ 
                      by (simp add: A, rule INF-lower, simp)
                    also have  $\dots \leq ((\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \sqcap ((-\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n)) \sqcup p (\text{Suc } n))$ 
                      by (rule inf-mono, simp-all, rule-tac  $y = -(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n)$  in order-trans, simp-all add: B)
                    also have  $\dots \leq p i$ 
                      by (simp add: inf-sup-distrib1 inf-compl-bot)
                    finally show  $(\text{INF } n. (\text{SUP } i : \{i. i < n\}. - p i) \sqcup p n) \leq p i$  by

```

```

simp
  qed
  qed
  also have (INF i:{i. i ≤ n}. p i) ≤ p n by (rule INF-lower, auto)
  finally show (INF n. (SUP i : {i. i < n} . - p i) ⊔ ((p :: nat ⇒ 'a) n))
≤ p n by simp
  qed

end

```

We prove now a number of results of the temporal class.

```

context temporal
begin
lemma [simp]: (a ⊔ b) !! i = (a !! i) ⊔ (b !! i)
  by (subst compl-eq-compl-iff [THEN sym], simp)

lemma always-less [simp]: □ p ≤ p
  proof -
  have □ p ≤ p !! 0
    by (unfold always-def, rule INF-lower, simp)
  also have p !! 0 = p by simp
  finally show □ p ≤ p by simp
  qed

lemma always-and: □ (p ∧ q) = (□ p) ∧ (□ q)
  by (simp add: always-def INF-inf-distrib)

lemma eventually-or: ◇ (p ⊔ q) = (◇ p) ⊔ (◇ q)
  by (simp add: eventually-def SUP-sup-distrib)

lemma neg-until-always: -(p until -p) = □ p
  proof (rule antisym)
  show -(p until -p) ≤ □ p
    by (simp add: until-def always-def uminus-SUP uminus-INF, rule
INF-greatest, cut-tac p = λ n . p !! n in until-always, simp)
  next
  have ∧ n . □ p ≤ p !! n
    by (simp add: always-def INF-lower)
  also have ∧ n . p !! n ≤ (SUP x:{i. i < n}. (- p) !! x) ⊔ p !! n
    by simp
  finally show □ p ≤ -(p until -p)
    apply (simp add: until-def uminus-SUP uminus-INF)
    by (rule INF-greatest, simp)
  qed

lemma neg-always-eventually: □ p = - ◇ (- p)
  by (simp add: fun-eq-iff always-def eventually-def until-def uminus-SUP)

lemma neg-true-until-always: -(⊤ until -p) = □ p

```

by (*simp add: fun-eq-iff always-def until-def uminus-SUP uminus-INF*)

lemma true-until-eventually: $(\top \text{ until } p) = \diamond p$

by (*cut-tac p = -p in neg-always-eventually, cut-tac p = -p in neg-true-until-always, simp*)

end

Boolean algebras with $b!!i = b$ form a temporal class.

instantiation *bool* :: *temporal*

begin

definition *at-bool-def* [*simp*]: $(p::\text{bool}) !! i = p$

instance proof

qed *auto*

end

type-synonym *'a trace* = *nat* \Rightarrow *'a*

Asuming that *'a* :: *temporal* is a type of class *temporal*, and *'b* is an arbitrary type, we would like to create the instantiation of *'b trace* \Rightarrow *'a* as a temporal class. However Isabelle allows only instatiations of functions from a class to another class. To solve this problem we introduce a new class called *trace* with an operation *suffix* :: *'a* \Rightarrow *nat* \Rightarrow *'a* where *suffix a i j* = $(a[i..]) j = a (i + j)$ when *a* is a trace with elements of some type *'b* (*'a* = *nat* \Rightarrow *'b*).

class *trace* =

fixes *suffix* :: *'a* \Rightarrow *nat* \Rightarrow *'a* (*-[- ..]* [80, 15] 80)

assumes [*simp*]: $a[i..][j..] = a[i + j..]$

assumes [*simp*]: $a[0..] = a$

begin

definition *next-trace* :: *'a* \Rightarrow *'a* (\odot (-) [900] 900) **where**

$\odot p = p[\text{Suc } 0..]$

end

instantiation *fun* :: (*trace, temporal*) *temporal*

begin

definition *at-fun-def*: $(P:: 'a \Rightarrow 'b) !! i = (\lambda x . (P (x[i..])) !! i)$

instance proof qed (*simp-all add: at-fun-def add.commute fun-eq-iff le-fun-def*)

end

In the last part of our formalization, we need to instantiate the functions from *nat* to some arbitrary type *'a* as a trace class. However, this again is not possible using the instatiation mechanism of Isabelle. We solve this problem by creating another class called *nat*, and then we instatiate the functions from *'a* :: *nat* to *'b* as traces. The class *nat* is defined such that if we have a type *'a* :: *nat*, then *'a* is isomorphic to the type *nat*.

class *nat* = *zero* + *plus* + *minus* +

```

fixes RepNat :: 'a ⇒ nat
fixes AbsNat :: nat ⇒ 'a
assumes [simp]: RepNat (AbsNat n) = n
and [simp]: AbsNat (RepNat x) = x
and zero-Nat-def: 0 = AbsNat 0
and plus-Nat-def: a + b = AbsNat (RepNat a + RepNat b)
and minus-Nat-def: a - b = AbsNat (RepNat a - RepNat b)
begin
lemma AbsNat-plus: AbsNat (i + j) = AbsNat i + AbsNat j
  by (simp add: plus-Nat-def)
lemma AbsNat-zero [simp]: AbsNat 0 + i = i
  by (simp add: plus-Nat-def)

subclass comm-monoid-diff
apply (unfold-locales)
  apply (simp-all add: plus-Nat-def zero-Nat-def minus-Nat-def add.assoc)
  by (simp add: add.commute)
end

```

The type natural numbers is an instantiation of the class *nat*.

```

instantiation nat :: nat
begin
  definition RepNat-nat-def [simp]: (RepNat:: nat ⇒ nat) = id
  definition AbsNat-nat-def [simp]: (AbsNat:: nat ⇒ nat) = id
  instance proof
    qed auto
end

```

Finally, functions from $'a :: nat$ to some arbitrary type $'b$ are instantiated as a trace class.

```

instantiation fun :: (nat, type) trace
begin
  definition at-trace-def [simp]: ((t :: 'a ⇒ 'b)[i..]) j = (t (AbsNat i + j))
  instance proof
    qed (simp-all add: fun-eq-iff AbsNat-plus add.assoc)
end

```

By putting together all class definitions and instantiations introduced so far, we obtain the temporal class structure for predicates on traces with arbitrary number of parameters.

For example in the next lemma r and r' are predicate relations, and the operator OO always is available for them as a consequence of the above construction.

```

lemma ( $\Box r$ ) OO ( $\Box r'$ ) ≤ ( $\Box$  (r OO r'))
  by (simp add: le-fun-def always-def at-fun-def, auto)

```

```

end
theory Refinement imports Main

```

begin

3 Monotonic Predicate Transformers

In this section we introduce the basics of refinement calculus [3]. Part of this theory is a reformulation of some definitions from [5], but here they are given for predicates, while [5] uses sets.

notation

bot (\perp) and
top (\top) and
inf (`infixl \sqcap 70`)
and *sup* (`infixl \sqcup 65`)

3.1 Basic predicate transformers

definition

demonic :: ($'a \Rightarrow 'b :: \text{lattice}$) $\Rightarrow 'b \Rightarrow 'a \Rightarrow \text{bool}$ (`[: - :] [0] 1000`) **where**
`[:Q:] p s = (Q s \leq p)`

definition

assert :: ($'a :: \text{semilattice-inf}$) $\Rightarrow 'a \Rightarrow 'a$ (`{. . .} [0] 1000`) **where**
`{.p.} q \equiv p \sqcap q`

definition

assume :: ($'a :: \text{boolean-algebra}$) $\Rightarrow 'a \Rightarrow 'a$ (`[. . .] [0] 1000`) **where**
`[.p.] q \equiv (\neg p \sqcup q)`

definition

angelic :: ($'a \Rightarrow 'b :: \{\text{semilattice-inf, order-bot}\}$) $\Rightarrow 'b \Rightarrow 'a \Rightarrow \text{bool}$ (`{: - :} [0] 1000`) **where**
`{:Q:} p s = (Q s \sqcap p \neq \perp)`

syntax

`-assert :: patterns \Rightarrow logic \Rightarrow logic` (`((1{./ .})`)

translations

`-assert (-patterns x xs) P == CONST assert (CONST id (-abs (-pattern x xs) P))`
`-assert x P == CONST assert (CONST id (-abs x P))`

term `{.x,z . P x y.}`

syntax `-update :: patterns \Rightarrow patterns \Rightarrow logic \Rightarrow logic` (`(- \rightsquigarrow . . - 0)`)

translations

`-update (-patterns x xs) (-patterns y ys) t == CONST id (-abs (-pattern x xs) (CONST id (-abs (-pattern y ys) t)))`
`-update x y t == CONST id (-abs x (CONST id (-abs y t)))`

term `[: x \rightsquigarrow z . (P :: 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow bool) x y y' z :]`

term $[: x, y \rightsquigarrow y', z . (P::'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow \text{bool}) x y y' z :]$
term $\{ : x, y \rightsquigarrow y', z . (P::'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow \text{bool}) x y y' z : \}$

lemma *demonic-demonic*: $[:r :] \circ [:r' :] = [:r \text{ OO } r' :]$
by (*simp add: fun-eq-iff le-fun-def demonic-def, auto*)

lemma *assert-demonic-comp*: $\{ .p. \} \circ [:r :] \circ \{ .p'. \} \circ [:r' :] =$
 $\{ .x . p x \wedge (\forall y . r x y \longrightarrow p' y). \} \circ [:r \text{ OO } r' :]$
by (*auto simp add: fun-eq-iff le-fun-def assert-def demonic-def*)

lemma *demonic-assert-comp*: $[:r :] \circ \{ .p. \} = \{ .x. (\forall y . r x y \longrightarrow p y). \} \circ [:r :]$
by (*auto simp add: fun-eq-iff le-fun-def assert-def demonic-def*)

lemma *assert-assert-comp*: $\{ .p::'a::\text{lattice}. \} \circ \{ .p'. \} = \{ .p \sqcap p'. \}$
by (*simp add: fun-eq-iff le-fun-def assert-def demonic-def inf-assoc*)

definition *inpt* $r x = (\exists y . r x y)$

definition *trs* $r = \{ . \text{ inpt } r. \} \circ [:r :]$

lemma *trs* $(\lambda x y . x = y) q x = q x$
by (*simp add: trs-def fun-eq-iff assert-def demonic-def inpt-def bot-fun-def le-fun-def*)

lemma *assert-demonic-prop*: $\{ .p. \} \circ [:r :] = \{ .p. \} \circ [:(\lambda x y . p x) \sqcap r :]$
by (*auto simp add: fun-eq-iff assert-def demonic-def*)

lemma *relcompp-exists*: $\text{relcompp } r r' x y = (\exists u . r x u \wedge r' u y)$
by *auto*

lemma *trs-trs*: $(\text{trs } r) \circ (\text{trs } r') = \text{trs } ((\lambda s t. (\forall s' . r s s' \longrightarrow (\text{inpt } r' s')) \sqcap (r \text{ OO } r')) \text{ (is } ?S = ?T))$

proof –

have [*simp*]: $(\lambda x. \text{ inpt } r x \wedge (\forall y. r x y \longrightarrow \text{ inpt } r' y)) = \text{ inpt } ((\lambda s t. \forall s'. r s s' \longrightarrow \text{ inpt } r' s') \sqcap r \text{ OO } r')$

by (*auto simp add: fun-eq-iff inpt-def relcompp-exists, blast*)

have [*simp*]: $(\lambda x y. \text{ inpt } r x \wedge (\forall y. r x y \longrightarrow \text{ inpt } r' y)) \sqcap r \text{ OO } r' = (\lambda s t. \forall s'. r s s' \longrightarrow \text{ inpt } r' s') \sqcap r \text{ OO } r'$

by (*auto simp add: fun-eq-iff inpt-def*)

have $?S = \{ . \text{ inpt } r . \} \circ [:r :] \circ \{ . \text{ inpt } r' . \} \circ [:r' :]$

by (*simp add: trs-def comp-assoc[symmetric]*)

also have $\dots = \{ . \lambda x. \text{ inpt } r x \wedge (\forall y. r x y \longrightarrow \text{ inpt } r' y) . \} \circ [:r \text{ OO } r' :]$

by (*simp add: assert-demonic-comp*)

also have $\dots = \{ . \lambda x. \text{ inpt } r x \wedge (\forall y. r x y \longrightarrow \text{ inpt } r' y) . \} \circ [:(\lambda x y . \text{ inpt } r x \wedge (\forall y. r x y \longrightarrow \text{ inpt } r' y)) \sqcap (r \text{ OO } r') :]$

by (*subst assert-demonic-prop, simp*)

also have $\dots = ?T$

by (*simp add: trs-def*)

finally show *?thesis* **by** *simp*

qed

lemma *assert-demonic-refinement*: $(\{.p.\} o [:r:] \leq \{.p'.\} o [:r':]) = (p \leq p' \wedge (\forall x . p x \longrightarrow r' x \leq r x))$
by (*auto simp add: le-fun-def assert-def demonic-def*)

lemma *trs-refinement*: $(trs r \leq trs r') = ((\forall x . inpt r x \longrightarrow inpt r' x) \wedge (\forall x . inpt r x \longrightarrow r' x \leq r x))$
by (*simp add: trs-def assert-demonic-refinement, simp add: le-fun-def*)

lemma *trs* $(\lambda x y . x \geq 0) \leq trs (\lambda x y . x = y)$
by (*simp add: trs-def le-fun-def assert-def demonic-def inpt-def*)

lemma *trs* $(\lambda x y . x \geq 0) q x = (if q = \top \text{ then } x \geq 0 \text{ else } False)$
by (*auto simp add: trs-def fun-eq-iff assert-def demonic-def inpt-def bot-fun-def*)

lemma $[:r:] \sqcap [:r':] = [:r \sqcup r':]$
by (*simp add: fun-eq-iff demonic-def*)

lemma *spec-demonic-choice*: $(\{.p.\} o [:r:]) \sqcap (\{.p'.\} o [:r':]) = (\{.p \sqcap p'.\} o [:r \sqcup r':])$
by (*auto simp add: fun-eq-iff demonic-def assert-def*)

lemma *trs-demonic-choice*: $trs r \sqcap trs r' = trs ((\lambda x y . inpt r x \wedge inpt r' x) \sqcap (r \sqcup r'))$

proof –

have $[simp]: inpt ((\lambda x y . inpt r x \wedge inpt r' x) \sqcap (r \sqcup r')) = inpt r \sqcap inpt r'$
by (*auto simp add: fun-eq-iff inpt-def*)
have $trs r \sqcap trs r' = \{. inpt r \sqcap inpt r' .\} o [: r \sqcup r' :]$
by (*simp add: trs-def spec-demonic-choice*)
also have $\dots = \{. inpt r \sqcap inpt r' .\} o [:(\lambda x y . inpt r x \wedge inpt r' x) \sqcap (r \sqcup r') :]$
by (*subst assert-demonic-prop, simp*)
also have $\dots = trs ((\lambda x y . inpt r x \wedge inpt r' x) \sqcap (r \sqcup r'))$
by (*simp add: trs-def*)
finally show *?thesis* **by** *simp*
qed

lemma $p \sqcap p' = \perp \implies (\{.p.\} o [:r:]) \sqcup (\{.p'.\} o [:r':]) = \{.p \sqcup p'.\} o [:(\lambda x y . p x \longrightarrow r x y) \sqcap ((\lambda x y . p' x \longrightarrow r' x y)):]$
by (*simp add: fun-eq-iff assert-def demonic-def, auto*)

3.2 Conjunctive predicate transformers

definition *conjunctive* $(S::'a::complete-lattice \Rightarrow 'b::complete-lattice) = (\forall Q . S (Inf Q) = INFIMUM Q S)$

definition *sconjunctive* $(S::'a::complete-lattice \Rightarrow 'b::complete-lattice) = (\forall Q . (\exists x . x \in Q) \longrightarrow S (Inf Q) = INFIMUM Q S)$

lemma [simp]: conjunctive $S \implies$ sconjunctive S
 by (simp add: conjunctive-def sconjunctive-def)

lemma [simp]: conjunctive \top
 by (simp add: conjunctive-def)

lemma conjunctive-demonic [simp]: conjunctive [:r:]
proof (auto simp add: conjunctive-def fun-eq-iff demonic-def)
 fix $Q :: 'a$ set fix $y :: 'a$ fix $x :: 'b$
 assume $A: y \in Q$
 assume $r\ x \leq \text{Inf } Q$
 also from A have $\text{Inf } Q \leq y$
 by (rule Inf-lower)
 finally show $r\ x \leq y$ by simp
next
 fix $Q :: 'a$ set fix $x :: 'b$
 assume $A : \forall y \in Q. r\ x \leq y$
 show $r\ x \leq \text{Inf } Q$
 by (rule Inf-greatest, simp add: A)
qed

lemma sconjunctive-assert [simp]: sconjunctive $\{.p.\}$
proof (auto simp add: sconjunctive-def assert-def image-def, rule antisym,
 auto)
 fix $Q :: 'a$ set
 have [simp]: $\bigwedge x. x \in Q \implies \text{Inf } Q \leq x$
 by (rule Inf-lower, simp)
 have $A: \bigwedge x. x \in Q \implies p \sqcap \text{Inf } Q \leq p \sqcap x$
 by (simp, rule-tac $y = \text{Inf } Q$ in order-trans, simp-all)
 show $p \sqcap \text{Inf } Q \leq (\text{INF } x:Q. p \sqcap x)$
 by (rule Inf-greatest, safe, rule A, simp)
next
 fix $Q :: 'a$ set
 fix $x :: 'a$
 assume $A: x \in Q$
 have $\text{Inf } \{y. \exists x \in Q. y = p \sqcap x\} \leq p \sqcap x$
 by (rule Inf-lower, cut-tac A, auto)
 also have $\dots \leq p$
 by simp
 finally show $(\text{INF } x:Q. p \sqcap x) \leq p$
 by (simp only: image-def)
next
 fix $Q :: 'a$ set
 show $(\text{INF } x:Q. p \sqcap x) \leq \text{Inf } Q$
proof (rule Inf-greatest)
 fix $x :: 'a$
 assume $A: x \in Q$
 have $\text{Inf } \{y. \exists x \in Q. y = p \sqcap x\} \leq p \sqcap x$
 by (cut-tac A, rule Inf-lower, blast)

also have $\dots \leq x$
by *simp*
finally show $(\text{INF } x:Q. p \sqcap x) \leq x$
by (*simp only: image-def*)
qed
qed

lemma *sconjunctive-simp*: $x \in Q \implies \text{sconjunctive } S \implies S (\text{Inf } Q) = \text{INFIMUM } Q S$
by (*auto simp add: sconjunctive-def*)

lemma *sconjunctive-INF-simp*: $x \in X \implies \text{sconjunctive } S \implies S (\text{INFIMUM } X Q) = \text{INFIMUM } (Q'X) S$
by (*cut-tac x = Q x and Q = Q' X in sconjunctive-simp, auto*)

lemma *demonic-comp [simp]*: $\text{sconjunctive } S \implies \text{sconjunctive } S' \implies \text{sconjunctive } (S \circ S')$
proof (*subst sconjunctive-def, safe*)
fix $X :: 'c \text{ set}$
fix $a :: 'c$
assume [*simp*]: *sconjunctive S*
assume [*simp*]: *sconjunctive S'*
assume [*simp*]: $a \in X$
have $A: S' (\text{Inf } X) = \text{INFIMUM } X S'$
by (*rule-tac x = a in sconjunctive-simp, auto*)
also have $B: S (\text{INFIMUM } X S') = \text{INFIMUM } (S' ' X) S$
by (*rule-tac x = S' a in sconjunctive-simp, auto*)
finally show $(S \circ S') (\text{Inf } X) = \text{INFIMUM } X (S \circ S')$ **by** *simp*
qed

lemma [*simp*]: $\text{conjunctive } S \implies S (\text{INFIMUM } X Q) = (\text{INFIMUM } X (S \circ Q))$
by (*metis INF-image conjunctive-def*)

lemma *conjunctive-simp*: $\text{conjunctive } S \implies S (\text{Inf } Q) = \text{INFIMUM } Q S$
by (*metis conjunctive-def*)

lemma *conjunctive-monotonic*: $\text{sconjunctive } S \implies \text{mono } S$
proof (*rule monoI*)
fix $a b :: 'a$
assume [*simp*]: $a \leq b$
assume [*simp*]: *sconjunctive S*
have [*simp*]: $a \sqcap b = a$
by (*rule antisym, auto*)
have $A: S a = S a \sqcap S b$
by (*cut-tac S = S and x = a and Q = {a, b} in sconjunctive-simp, auto*)
show $S a \leq S b$
by (*subst A, simp*)
qed

definition $grd\ S = -\ S \perp$

lemma $grd\ [:r:] = inpt\ r$
by (simp add: fun-eq-iff grd-def demonic-def le-fun-def inpt-def)

definition $fail\ S = -(S\ \top)$

definition $term\ S = (S\ \top)$

lemma $fail\ (\{.p.\} o [:r :: 'a \Rightarrow 'b \Rightarrow bool:]) = -p$
by (simp add: fail-def fun-eq-iff assert-def demonic-def le-fun-def top-fun-def)

definition $Fail = \perp$

lemma $mono\ (S :: 'a :: boolean-algebra \Rightarrow 'b :: boolean-algebra) \Longrightarrow (S = Fail) =$
 $(fail\ S = \top)$

proof auto

show $fail\ (Fail :: 'a \Rightarrow 'b) = \top$

by (metis Fail-def bot-apply compl-bot-eq fail-def)

next

assume $A: mono\ S$

assume $B: fail\ S = \top$

show $S = Fail$

proof (rule antisym)

show $S \leq Fail$

by (metis (hide-lams, no-types) A B bot.extremum-unique compl-le-compl-iff
fail-def le-fun-def monoD top-greatest)

next

show $Fail \leq S$

by (metis Fail-def bot.extremum)

qed

qed

definition $Skip = id$

lemma $[simp]: \{.\top :: 'a :: bounded-lattice.\} = Skip$
by (simp add: fun-eq-iff assert-def Skip-def)

lemma $[simp]: Skip\ o\ S = S$
by (simp add: fun-eq-iff assert-def Skip-def)

lemma $[simp]: S\ o\ Skip = S$
by (simp add: fun-eq-iff assert-def Skip-def)

lemma $[simp]: mono\ S \Longrightarrow mono\ S' \Longrightarrow mono\ (S\ o\ S')$
by (simp add: mono-def)

lemma $[simp]: mono\ \{.p.:('a \Rightarrow bool).\}$
by (simp add: conjunctive-monotonic)

lemma [simp]: $\text{mono } [:r::('a \Rightarrow 'b \Rightarrow \text{bool}):]$
by (simp add: conjunctive-monotonic)

lemma [simp]: $\{\lambda x . \text{True } .\} = \text{Skip}$
by (simp add: fun-eq-iff assert-def Skip-def)

lemma [simp]: $\perp \circ S = \perp$
by (simp add: fun-eq-iff)

lemma [simp]: $\{\perp::'a::\text{boolean-algebra.}\} = \perp$
by (simp add: fun-eq-iff assert-def)

lemma [simp]: $\top \circ S = \top$
by (simp add: fun-eq-iff)

lemma left-comp: $T \circ U = T' \circ U' \Longrightarrow S \circ T \circ U = S \circ T' \circ U'$
by (simp add: comp-assoc)

lemma assert-demonic: $\{\cdot.p.\} \circ [:r:] = \{\cdot.p.\} \circ [\lambda x y . p x \wedge r x y:]$
by (auto simp add: fun-eq-iff assert-def demonic-def le-fun-def)

lemma trs r \sqcap trs r' = trs $(\lambda x y . \text{inpt } r x \wedge \text{inpt } r' x \wedge (r x y \vee r' x y))$
by (auto simp add: fun-eq-iff trs-def assert-def demonic-def inpt-def)

3.3 Fusion of predicate transformers

In this section we define the fusion operator from [2]. The fusion of two programs S and T is intuitively equivalent with the parallel execution of the two programs. If S and T assign nondeterministically some value to some program variable x , then the fusion of S and T will assign a value to x which can be assigned by both S and T .

definition fusion :: $(('a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow \text{bool})) \Rightarrow (('a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow \text{bool}))$
 $\Rightarrow (('a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow \text{bool}))$ (infixl \parallel 70) **where**
 $(S \parallel S') q x = (\exists (p::'a \Rightarrow \text{bool}) p' . p \sqcap p' \leq q \wedge S p x \wedge S' p' x)$

lemma fusion-spec: $(\{\cdot.p.\} \circ [:r:]) \parallel (\{\cdot.p'.\} \circ [:r':]) = (\{\cdot.p \sqcap p'.\} \circ [:r \sqcap r':])$
by (auto simp add: fun-eq-iff fusion-def assert-def demonic-def le-fun-def)

lemma fusion-assoc: $S \parallel (T \parallel U) = (S \parallel T) \parallel U$

proof (rule antisym, auto simp add: fusion-def)

fix $p p' q s s' :: 'a \Rightarrow \text{bool}$

fix a

assume $A: p \sqcap p' \leq q$ **and** $B: s \sqcap s' \leq p'$

assume $C: S p a$ **and** $D: T s a$ **and** $E: U s' a$

from A **and** B **have** $F: (p \sqcap s) \sqcap s' \leq q$

by (simp add: le-fun-def)

have $(\exists v v' . v \sqcap v' \leq (p \sqcap s) \wedge S v a \wedge T v' a)$

by (metis C D order-refl)

show $\exists u u'. u \sqcap u' \leq q \wedge (\exists v v'. v \sqcap v' \leq u \wedge S v a \wedge T v' a) \wedge U u' a$
by (*metis F C D E order-refl*)
next
fix $p p' q s s' :: 'a \Rightarrow \text{bool}$
fix a
assume $A: p \sqcap p' \leq q$ **and** $B: s \sqcap s' \leq p$
assume $C: S s a$ **and** $D: T s' a$ **and** $E: U p' a$
from A **and** B **have** $F: s \sqcap (s' \sqcap p') \leq q$
by (*simp add: le-fun-def*)
have $(\exists v v'. v \sqcap v' \leq s' \sqcap p' \wedge T v a \wedge U v' a)$
by (*metis D E eq-iff*)
show $\exists u u'. u \sqcap u' \leq q \wedge S u a \wedge (\exists v v'. v \sqcap v' \leq u' \wedge T v a \wedge U v' a)$
by (*metis F C D E order-refl*)
qed

lemma $P \leq Q \Longrightarrow P' \leq Q' \Longrightarrow P \parallel P' \leq Q \parallel Q'$
by (*simp add: le-fun-def fusion-def, metis*)

lemma conjunctive $S \Longrightarrow S \parallel \top = \top$
by (*auto simp add: fun-eq-iff fusion-def le-fun-def conjunctive-def*)

lemma fusion-spec-local: $a \in \text{init} \Longrightarrow ([:x \rightsquigarrow u, y . u \in \text{init} \wedge x = y:] \circ \{.p.\})$
 $\circ [r:] \parallel (\{.p'.\} \circ [r':])$
 $= [:x \rightsquigarrow u, y . u \in \text{init} \wedge x = y:] \circ \{.u, x . p(u, x) \wedge p' x.\} \circ [u, x \rightsquigarrow y . r$
 $(u, x) y \wedge r' x y:]$ (**is** $?p \Longrightarrow ?S = ?T$)

proof –
assume $?p$
from *this* **have** [*simp*]: $(\lambda x. \forall a. a \in \text{init} \longrightarrow p(a, x) \wedge p' x) = (\lambda x. \forall a. a \in \text{init} \longrightarrow p(a, x)) \sqcap p'$
by *auto*
have [*simp*]: $(\lambda x (u, y). u \in \text{init} \wedge x = y) \text{ OO } (\lambda(u, x) y. r(u, x) y \wedge r' x y) = (\lambda x (u, y). u \in \text{init} \wedge x = y) \text{ OO } r \sqcap r'$
by *auto*
have $?S =$
 $(\{. \lambda x. \forall a. a \in \text{init} \longrightarrow p(a, x) .\} \circ [:\lambda x (u, y). u \in \text{init} \wedge x = y:] \circ [:$
 $r:] \parallel (\{. p' .\} \circ [r':])$
by (*simp add: demonic-assert-comp*)
also have $\dots = \{. (\lambda x. \forall a. a \in \text{init} \longrightarrow p(a, x)) \sqcap p' .\} \circ [:(\lambda x (u, y). u \in \text{init} \wedge x = y) \text{ OO } r \sqcap r':]$
by (*simp add: comp-assoc demonic-demonic fusion-spec*)
also have $\dots = ?T$
by (*simp add: demonic-assert-comp comp-assoc demonic-demonic fusion-spec*)
finally show $?thesis$ **by** *simp*
qed

lemma fusion-spec-local-a: $a \in \text{init} \Longrightarrow ([:x \rightsquigarrow u, y . u \in \text{init} \wedge x = y:] \circ \{.p.\})$
 $\circ [r:] \parallel [r':]$
 $= ([:x \rightsquigarrow u, y . u \in \text{init} \wedge x = y:] \circ \{.p.\} \circ [u, x \rightsquigarrow y . r(u, x) y \wedge r' x y:])$
by (*cut-tac p' = \top and init = init and p = p and r = r and r' = r' in*)

fusion-spec-local, auto)

lemma *fusion-local-refinement*:

$$a \in \text{init} \implies (\bigwedge x u y . u \in \text{init} \implies p' x \implies r(u, x) y \implies r' x y) \implies \{.p'.\} \circ ([:x \rightsquigarrow u, y . u \in \text{init} \wedge x = y:] \circ \{.p.\} \circ [:r:] \parallel [:r':]) \leq [:x \rightsquigarrow u, y . u \in \text{init} \wedge x = y:] \circ \{.p.\} \circ [:r:]$$

proof –

assume A : $a \in \text{init}$

assume [*simp*]: $(\bigwedge x u y . u \in \text{init} \implies p' x \implies r(u, x) y \implies r' x y)$

have $\{. x . p' x \wedge (\forall a . a \in \text{init} \longrightarrow p(a, x)) .\} \circ [:(\lambda x (u, y) . u \in \text{init} \wedge x = y) \text{ OO } (\lambda(u, x) y . r(u, x) y \wedge r' x y) :]$

$$\leq \{. \lambda x . \forall a . a \in \text{init} \longrightarrow p(a, x) .\} \circ [:(\lambda x (u, y) . u \in \text{init} \wedge x = y) \text{ OO } r :]$$

by (*auto simp add: assert-demonic-refinement*)

from this have $\{. x . p' x \wedge (\forall a . a \in \text{init} \longrightarrow p(a, x)) .\} \circ [:(\lambda x (u, y) . u \in \text{init} \wedge x = y) \text{ OO } (\lambda(u, x) y . r(u, x) y \wedge r' x y) :]$

$$\leq \{. \lambda x . \forall a . a \in \text{init} \longrightarrow p(a, x) .\} \circ [:\lambda x (u, y) . u \in \text{init} \wedge x = y :]$$

$\circ [: r :]$

by (*simp add: comp-assoc demonic-demonic*)

from this have $\{. p' .\} \circ [:\lambda x (u, y) . u \in \text{init} \wedge x = y :] \circ \{. p .\} \circ [:\lambda(u, x) y . r(u, x) y \wedge r' x y :]$

$$\leq [: x \rightsquigarrow u, y . u \in \text{init} \wedge x = y :] \circ \{. p .\} \circ [: r :]$$

by (*simp add: demonic-assert-comp assert-demonic-comp*)

from this have $\{. p' .\} \circ ([: x \rightsquigarrow (u, y) . u \in \text{init} \wedge x = y :] \circ \{. p .\} \circ [: (u, x) \rightsquigarrow y . r(u, x) y \wedge r' x y :])$

$$\leq [: x \rightsquigarrow (u, y) . u \in \text{init} \wedge x = y :] \circ \{. p .\} \circ [: r :]$$

by (*simp add: comp-assoc [THEN sym]*)

from A **and this show** *?thesis*

by (*unfold fusion-spec-local-a, simp*)

qed

end

theory *Reactive*

imports *Temporal Refinement*

begin

4 Reactive Systems

In this section we introduce reactive systems which are modeled as monotonic property transformers where properties are predicates on traces. We start with introducing some examples that uses LTL to specify global behaviour on traces, and later we introduce property transformers based on symbolic transition systems.

definition *HAVOC* = $[: x \rightsquigarrow y . \text{True} :]$

definition *ASSERT-LIVE* = $\{. \square \diamond (\lambda x . x \ 0) .\}$

definition *GUARANTY-LIVE* = $[: x \rightsquigarrow y . \square \diamond (\lambda y . y \ 0) :]$

definition *AE* = *ASSERT-LIVE* \circ *HAVOC*

definition *SKIP* = $[: x \rightsquigarrow y . x = y :]$

lemma [simp]: $SKIP = id$
by (auto simp add: fun-eq-iff SKIP-def demonic-def)

definition REQ-RESP = [$\square(\lambda xs\ ys . xs (0::nat) \longrightarrow (\diamond (\lambda ys . ys (0::nat)))$
 $ys)$:]

definition FAIL = \perp

lemma HAVOC o ASSERT-LIVE = FAIL

by (auto simp add: HAVOC-def AE-def FAIL-def ASSERT-LIVE-def fun-eq-iff
 assert-def demonic-def always-def at-fun-def le-fun-def eventually-def)

lemma HAVOC o AE = FAIL

by (auto simp add: HAVOC-def AE-def FAIL-def ASSERT-LIVE-def fun-eq-iff
 assert-def demonic-def always-def at-fun-def le-fun-def eventually-def)

lemma HAVOC o ASSERT-LIVE = FAIL

by (auto simp add: HAVOC-def AE-def FAIL-def ASSERT-LIVE-def fun-eq-iff
 assert-def demonic-def always-def at-fun-def eventually-def)

lemma SKIP o AE = AE

by simp

lemma (REQ-RESP o AE) = AE

proof (auto simp add: fun-eq-iff HAVOC-def AE-def FAIL-def REQ-RESP-def
 ASSERT-LIVE-def assert-def

demonic-def always-def le-fun-def eventually-def at-fun-def)

fix $x :: 'a \Rightarrow bool$

fix $xa :: nat \Rightarrow bool$

fix $xb :: nat$

assume $\forall xb::nat \Rightarrow bool . (\forall x. xa\ x \longrightarrow Ex\ (xb[x\ ..])) \longrightarrow (\forall x. \exists a. xb\ (x$
 $+ a)) \wedge All\ x$

then have $(\forall x. xa\ x \longrightarrow Ex\ (xa[x\ ..])) \longrightarrow (\forall x. \exists a. xa\ (x + a)) \wedge All\ x$

by auto

then show $\exists x. xa\ (xb + x)$

by (auto, rule-tac $x = 0$ in exI, simp)

next

fix $x :: 'a \Rightarrow bool$

fix $xa :: nat \Rightarrow bool$

fix $xb :: 'a$

assume $\forall xb::nat \Rightarrow bool . (\forall x. xa\ x \longrightarrow Ex\ (xb[x\ ..])) \longrightarrow (\forall x. \exists a. xb\ (x$
 $+ a)) \wedge All\ x$

from this show $x\ xb$

by (metis at-trace-def le0)

next

fix $x :: 'a \Rightarrow bool$ **and** $xa :: nat \Rightarrow bool$ **and** $xb :: nat \Rightarrow bool$ **and** $xc :: nat$

assume $A: \forall x. xa\ x \longrightarrow Ex\ (xb[x\ ..])$

assume $B: \forall x. \exists xb. xa\ (x + xb)$

have $\exists x1. xc \leq AbsNat\ x1$ **by** (metis (full-types) le-add2 plus-Nat-def)

thus $\exists x. xb (xc + x)$ **using** $A B$ **by** (*metis AbsNat-plus add.commute at-trace-def le-Suc-ex trans-le-add2*)

qed

4.1 Symbolic transition systems

In this section we introduce property transformers basend on symbolic transition systems. These are systems with local state. The execution starts in some initial state, and with some input value the system computes a new state and an output value. Then using the current state, and a new input value the system computes a new state, and a new output, and so on. The system may fail if at some point the input and the current state do not satisfy a required predicate.

In the folowing definitions the variables u, x, y stand for the state of the system, the input, and the output respectively. The *init* is the property that the initial state should satisfy. The predicate p is the precondition of the input and the current state, and the relation r gives the next state and the output based on the input and the current state.

definition *fail-sys* $init\ p\ r\ x = (\exists n\ u\ y . u \in init \wedge (\forall i < n . r\ (u\ i)\ (u\ (Suc\ i))\ (x\ i)\ (y\ i)) \wedge (\neg p\ (u\ n)\ (u\ (Suc\ n))\ (x\ n)))$

definition *run* $r\ u\ x\ y = (\forall i . r\ (u\ i)\ (u\ (Suc\ i))\ (x\ i)\ (y\ i))$

definition *LocalSystem* $init\ p\ r\ q\ x = (\neg fail-sys\ init\ p\ r\ x \wedge (\forall u\ y . (u \in init \wedge run\ r\ u\ x\ y) \longrightarrow q\ y))$

lemma *fail* $(LocalSystem\ init\ p\ r) = fail-sys\ init\ p\ r$

by (*simp add: fun-eq-iff LocalSystem-def fail-def fail-sys-def run-def*)

definition *inpt-st* $r\ u\ u'\ x = (\exists y . r\ u\ u'\ x\ y)$

definition *lft-pred-st* $p\ u\ x = p\ (u\ (0::nat))\ (u\ 1)\ (x\ (0::nat))$

definition *lft-pred-loc-st* $p\ u\ x = p\ (u\ (0::nat))\ (x\ (0::nat))$

definition *lft-rel-st* $r\ u\ x\ y = r\ (u\ (0::nat))\ (u\ 1)\ (x\ (0::nat))\ (y\ (0::nat))$

definition *prec-st* $p\ r = \neg((lft-pred-st\ (inpt-st\ r))\ until\ \neg(lft-pred-st\ p))$

lemma *prec-st-simp*: $prec-st\ p\ r\ u\ x = (\forall n . (\forall i < n . inpt-st\ r\ (u\ i)\ (u\ (Suc\ i))\ (x\ i)) \longrightarrow p\ (u\ n)\ (u\ (Suc\ n))\ (x\ n))$

by (*simp add: prec-st-def until-def lft-pred-st-def inpt-st-def at-fun-def, metis*)

definition *SymSystem* $init\ p\ r = [z \rightsquigarrow u, x . u \in init \wedge z = x:] \circ \{u, x . prec-st\ p\ r\ u\ x.\}$

$[u, x \rightsquigarrow y . (\Box (lft-rel-st\ r))\ u\ x\ y:]$

lemma *SymSystem-rel*: $SymSystem\ init\ p\ r = \{x . \forall u. u \in init \longrightarrow prec-st\ p\ r\ u\ x.\} \circ$

$[: x \rightsquigarrow y . \exists u . u \in \text{init} \wedge (\Box \text{lft-rel-st } r) u x y]$
proof –
have $[\text{simp}]$: $((\lambda z (u, x). u \in \text{init} \wedge z = x) OO (\lambda(x, y). (\Box \text{lft-rel-st } r) x y)) = (\lambda x y. \exists u. u \in \text{init} \wedge (\Box \text{lft-rel-st } r) u x y)$
by *auto*
show *?thesis* **by** (*simp add: SymSystem-def demonic-assert-comp comp-assoc demonic-demonic*)
qed

theorem *SymSystem init p r q x = LocalSystem init p r q x*

proof

assume *SymSystem init p r q x*

then show *LocalSystem init p r q x*

apply (*auto simp add: SymSystem-def LocalSystem-def assert-def demonic-def prec-st-simp lft-rel-st-def lft-pred-st-def inpt-st-def always-def le-fun-def fail-sys-def run-def at-fun-def*)

by *metis*

next

assume *LocalSystem init p r q x*

then show *SymSystem init p r q x*

apply (*auto simp add: SymSystem-def LocalSystem-def assert-def demonic-def prec-st-simp lft-rel-st-def lft-pred-st-def inpt-st-def always-def le-fun-def fail-sys-def run-def at-fun-def*)

by *metis*

qed

definition *local-init init S = INFIMUM init S*

definition *zip-set A B = {u . ((fst o u) ∈ A) ∧ ((snd o u) ∈ B)}*

definition *nzip:: ('x ⇒ 'a) ⇒ ('x ⇒ 'b) ⇒ 'x ⇒ ('a × 'b) (infixl || 65) where (xs || ys) i = (xs i, ys i)*

lemma $[\text{simp}]$: $\text{fst} \circ x \parallel y = x$

by (*simp add: fun-eq-iff nzip-def*)

lemma $[\text{simp}]$: $\text{snd} \circ x \parallel y = y$

by (*simp add: fun-eq-iff nzip-def*)

lemma $[\text{simp}]$: $x \in A \implies y \in B \implies (x \parallel y) \in \text{zip-set } A B$

by (*simp add: zip-set-def*)

lemma *local-demonic-init: local-init init (λ u . { . x . p u x. } o [: x ~> y . r u x y :]) =*

[: z ~> u, x . u ∈ init ∧ z = x :] o { . u, x . p u x. } o [: u, x ~> y . r u x y :]

by (*auto simp add: fun-eq-iff demonic-def assert-def local-init-def le-fun-def*)

lemma *local-init-comp: u' ∈ init' ⟹ (∀ u. sconjunctive (S u)) ⟹ (local-init init S) o (local-init init' S')*

= local-init (zip-set init init') (λ u . (S (fst o u)) o (S' (snd o

$u)))$
proof (*subst fun-eq-iff*, *auto*)
fix $x :: 'f$
assume $A: u' \in \text{init}'$
assume $\forall u . \text{sconjunctive } (S\ u)$
from *this* **have** $[\text{simp}]: \bigwedge u . \text{sconjunctive } (S\ u)$ **by** *simp*
from A **have** $[\text{simp}]: \bigwedge y . S\ y\ (\text{INF } y' : \text{init}' . S'\ y'\ x) = (\text{INF } y' : \text{init}' .$
 $S\ y\ (S'\ y'\ x))$
by (*simp add: sconjunctive-INF-simp*)

have $[\text{simp}]: (\text{INF } y : \text{init} . (\text{INF } y' : \text{init}' . S\ y\ (S'\ y'\ x))) \leq (\text{INF } u :$
 $\text{zip-set } \text{init } \text{init}' . S\ (\text{fst } \circ u)\ (S'\ (\text{snd } \circ u)\ x))$
proof (*rule INF-greatest*, *auto simp add: zip-set-def*)
fix $u :: 'a \Rightarrow 'c \times 'b$
assume $[\text{simp}]: \text{fst } \circ u \in \text{init}$
assume $[\text{simp}]: \text{snd } \circ u \in \text{init}'$
have $(\text{INF } y : \text{init} . \text{INF } y' : \text{init}' . S\ y\ (S'\ y'\ x)) \leq (\text{INF } y' : \text{init}' . S\ (\text{fst}$
 $o\ u)\ (S'\ y'\ x))$
by (*rule INF-lower*, *simp*)
also have $\dots \leq S\ (\text{fst } o\ u)\ (S'\ (\text{snd } o\ u)\ x)$
by (*rule INF-lower*, *simp*)
finally show $(\text{INF } y : \text{init} . \text{INF } y' : \text{init}' . S\ y\ (S'\ y'\ x)) \leq S\ (\text{fst } o\ u)$
 $(S'\ (\text{snd } o\ u)\ x)$
by *simp*
qed
have $[\text{simp}]: (\text{INF } u : \text{zip-set } \text{init } \text{init}' . S\ (\text{fst } \circ u)\ (S'\ (\text{snd } \circ u)\ x)) \leq (\text{INF}$
 $y : \text{init} . (\text{INF } y' : \text{init}' . S\ y\ (S'\ y'\ x)))$
proof (*rule INF-greatest*, *rule INF-greatest*)
fix $y :: 'a \Rightarrow 'c$ **and** $y' :: 'a \Rightarrow 'b$
assume $[\text{simp}]: y \in \text{init}$
assume $[\text{simp}]: y' \in \text{init}'$
have $(\text{INF } u : \text{zip-set } \text{init } \text{init}' . S\ (\text{fst } \circ u)\ (S'\ (\text{snd } \circ u)\ x)) \leq S\ (\text{fst } o$
 $(y \parallel y'))\ (S'\ (\text{snd } o\ (y \parallel y'))\ x)$
by (*rule INF-lower*, *simp*)
also have $\dots \leq S\ y\ (S'\ y'\ x)$
by *simp*
finally show $(\text{INF } u :: 'a \Rightarrow 'c \times 'b : \text{zip-set } \text{init } \text{init}' . S\ (\text{fst } \circ u)\ (S'\ (\text{snd}$
 $o\ u)\ x)) \leq S\ y\ (S'\ y'\ x)$
by *simp*
qed
have $\text{local-init } \text{init } S\ (\text{local-init } \text{init}'\ S'\ x) = (\text{INF } y : \text{init} . S\ y\ (\text{INF } y' :$
 $\text{init}' . S'\ y'\ x))$
by (*simp add: local-init-def*)
also have $\dots = (\text{INF } y : \text{init} . (\text{INF } y' : \text{init}' . S\ y\ (S'\ y'\ x)))$
by *simp*
also have $\dots = (\text{INF } u : \text{zip-set } \text{init } \text{init}' . S\ (\text{fst } \circ u) \circ S'\ (\text{snd } \circ u))\ x$
by (*rule antisym*, *auto*)
also have $\dots = \text{local-init } (\text{zip-set } \text{init } \text{init}')\ (\lambda u . (S\ (\text{fst } \circ u)) \circ (S'\ (\text{snd}$
 $o\ u)))\ x$

by (*simp add: local-init-def*)
finally show *local-init init S (local-init init' S' x) = local-init (zip-set init init')* ($\lambda u::'a \Rightarrow 'c \times 'b. S (fst \circ u) \circ S' (snd \circ u)$) x
by simp
qed

lemma *init-state*: $[:z \rightsquigarrow u, x . u \in \text{init} \wedge z = x:] \circ \{.u, x . p \ u \ x.\} \circ [:u, x \rightsquigarrow y . r \ u \ x \ y :]$
 $= [:z \rightsquigarrow u, x . z = x:] \circ \{.u, x . u \in \text{init} \longrightarrow p \ u \ x.\} \circ [:u, x \rightsquigarrow y . u \in \text{init} \wedge r \ u \ x \ y :]$
by (*auto simp add: fun-eq-iff demonic-def assert-def local-init-def le-fun-def*)

lemma *always-lft-rel-comp*: $(\Box \text{lft-rel-st } r) (fst \circ u) \text{ OO } (\Box \text{lft-rel-st } r') (snd \circ u)$

$$= (\Box \text{lft-rel-st } (\lambda (u, v) (u', v') . ((r \ u \ u') \text{ OO } (r' \ v \ v')))) \ u$$

proof (*auto simp add: fun-eq-iff lft-rel-st-def always-def at-fun-def relcompp-exists*)
fix $x::\text{nat} \Rightarrow 'a$ **and**
 $y::\text{nat} \Rightarrow 'b$ **and**
 $v::\text{nat} \Rightarrow 'c$ **and**
 $n::\text{nat}$
assume $\forall i . r (fst (u \ i)) (fst (u (Suc \ i))) (x \ i) (v \ i)$
and $\forall i . r' (snd (u \ i)) (snd (u (Suc \ i))) (v \ i) (y \ i)$
from this show (*case u n of (u, v) $\Rightarrow \lambda(u', v'). r \ u \ u' \text{ OO } r' \ v \ v'$)* ($u (Suc \ n)$) ($x \ n$) ($y \ n$)

by (*metis (mono-tags, lifting) prod.case-eq-if relcompp.relcompI*)

next

fix $x::\text{nat} \Rightarrow 'a$ **and**

$z::\text{nat} \Rightarrow 'b$

def A : $a == (\lambda n . \text{SOME } y . r (fst (u \ n)) (fst (u (Suc \ n))) (x \ n) \ y \ \wedge \ r' (snd (u \ n)) (snd (u (Suc \ n))) \ y \ (z \ n))$
assume $\forall i . (\text{case } u \ i \ \text{of } (u, v) \Rightarrow \lambda(u', v'). r \ u \ u' \text{ OO } r' \ v \ v')$ ($u (Suc \ i)$) ($z \ i$)

from this and A have $(\forall i :: \text{nat} . r (fst (u \ i)) (fst (u (Suc \ i))) (x \ i) (a \ i))$
 $\wedge (\forall i :: \text{nat} . r' (snd (u \ i)) (snd (u (Suc \ i))) (a \ i) (z \ i))$

apply auto

apply (*metis (mono-tags, lifting) pick-middlep prod.collapse split-conv tfl-some*)

by (*metis (mono-tags, lifting) pick-middlep prod.collapse split-conv tfl-some*)

from this show $\exists a . (\forall i . r (fst (u \ i)) (fst (u (Suc \ i))) (x \ i) (a \ i)) \ \wedge \ (\forall i . r' (snd (u \ i)) (snd (u (Suc \ i))) (a \ i) (z \ i))$

by blast

qed

theorem *SymSystem-comp*: $u' \in \text{init}' \Longrightarrow \text{SymSystem } \text{init } p \ r \ o \ \text{SymSystem } \text{init}' \ p' \ r'$

$= [:z \rightsquigarrow u, x . fst \circ u \in \text{init} \wedge snd \circ u \in \text{init}' \wedge z = x:]$
 $\circ \{. u, x . \text{prec-st } p \ r \ (fst \circ u) \ x \ \wedge \ (\forall y . (\Box \text{lft-rel-st } r) (fst \circ u) \ x \ y \longrightarrow \text{prec-st } p' \ r' (snd \circ u) \ y) .\}$
 $\circ [: u, x \rightsquigarrow y . (\Box \text{lft-rel-st } (\lambda(u, v) (u', v') . r \ u \ u' \text{ OO } r' \ v \ v')) \ u$

$x y :$

(is $?p \implies ?S = ?T$)

proof –

assume $A: ?p$

have $?S =$

$[: z \rightsquigarrow (u, x) . u \in \text{init} \wedge z = x :] \circ \{.x, y. \text{prec-st } p \ r \ x \ y.\} \circ$

$[: \text{id } (\lambda(u, x). \text{id } ((\Box \text{lft-rel-st } r) \ u \ x)) :] \circ$

$[: z \rightsquigarrow u, x . u \in \text{init}' \wedge z = x :] \circ \{.x, y. \text{prec-st } p' \ r' \ x \ y.\} \circ$

$[: \text{id } (\lambda(u, x). \text{id } ((\Box \text{lft-rel-st } r') \ u \ x)) :])$

by (*unfold SymSystem-def, simp*)

also have $\dots = \text{local-init } \text{init } (\lambda u::\text{nat} \Rightarrow 'e. \{. \text{id } (\text{prec-st } p \ r \ u) .\} \circ [: \text{id } (\lambda x. \text{id } ((\Box \text{lft-rel-st } r) \ u \ x)) :]) \circ$

$\text{local-init } \text{init}' (\lambda u. \{. \text{id } (\text{prec-st } p' \ r' \ u) .\} \circ [: \text{id } (\lambda x::\text{nat} \Rightarrow 'd. \text{id } ((\Box \text{lft-rel-st } r') \ u \ x)) :])$

by (*unfold local-demonic-init [THEN sym], simp*)

also from A **have** $\dots = \text{local-init } (\text{zip-set } \text{init } \text{init}')$

$(\lambda u. \{. \text{prec-st } p \ r \ (\text{fst } \circ \ u) .\} \circ [: (\Box \text{lft-rel-st } r) \ (\text{fst } \circ \ u) :] \circ (\{. \text{prec-st } p' \ r' \ (\text{snd } \circ \ u) .\} \circ [: (\Box \text{lft-rel-st } r') \ (\text{snd } \circ \ u) :]))$

by (*simp add: local-init-comp*)

also have $\dots = \text{local-init } (\text{zip-set } \text{init } \text{init}')$

$(\lambda u. \{. \text{prec-st } p \ r \ (\text{fst } \circ \ u) .\} \circ [: (\Box \text{lft-rel-st } r) \ (\text{fst } \circ \ u) :] \circ \{. \text{prec-st } p' \ r' \ (\text{snd } \circ \ u) .\} \circ [: (\Box \text{lft-rel-st } r') \ (\text{snd } \circ \ u) :])$

by (*simp add: comp-assoc [THEN sym]*)

also have $\dots = \text{local-init } (\text{zip-set } \text{init } \text{init}')$

$(\lambda u. \{. x . \text{prec-st } p \ r \ (\text{fst } \circ \ u) \ x \wedge (\forall y. (\Box \text{lft-rel-st } r) \ (\text{fst } \circ \ u) \ x \ y \longrightarrow \text{prec-st } p' \ r' \ (\text{snd } \circ \ u) \ y) .\} \circ$

$[: (\Box \text{lft-rel-st } (\lambda(u, v) (u', v'). r \ u \ u' \ OO \ r' \ v \ v')) \ u :])$

by (*simp add: assert-demonic-comp always-lft-rel-comp*)

also have $\dots = \text{local-init } (\text{zip-set } \text{init } \text{init}')$

$(\lambda u. \{. x. \text{prec-st } p \ r \ (\text{fst } \circ \ u) \ x \wedge (\forall y::\text{nat} \Rightarrow 'd. (\Box \text{lft-rel-st } r) \ (\text{fst } \circ \ u) \ x \ y \longrightarrow \text{prec-st } p' \ r' \ (\text{snd } \circ \ u) \ y) .\} \circ$

$[: \text{id } (\lambda x::\text{nat} \Rightarrow 'c. \text{id } ((\Box \text{lft-rel-st } (\lambda(u, v) (u', v'). r \ u \ u' \ OO \ r' \ v \ v')) \ u \ x)) :])$

by *simp*

also have $\dots = ?T$

by (*unfold local-demonic-init, simp add: zip-set-def*)

finally show $?thesis$ **by** *simp*

qed

lemma *always-lft-rel-comp-a*: $(\Box \text{lft-rel-st } r) \ u \ OO (\Box \text{lft-rel-st } r') \ v$

$= (\Box \text{lft-rel-st } (\lambda (u, v) (u', v') . ((r \ u \ u') \ OO (r' \ v \ v')))) (u \ || \ v)$

by (*unfold always-lft-rel-comp [THEN sym], auto*)

theorem *SymSystem-comp-a*: $u' \in \text{init}' \implies \text{SymSystem } \text{init } p \ r \ o \ \text{SymSystem } \text{init}' \ p' \ r'$

$= \{.x . \forall u \ v . u \in \text{init} \wedge v \in \text{init}' \longrightarrow (\text{prec-st } p \ r \ u \ x \wedge (\forall y. (\Box \text{lft-rel-st } r) \ u \ x \ y \longrightarrow \text{prec-st } p' \ r' \ v \ y)) .\}$

$\circ [: x \rightsquigarrow y . \exists u \ v . u \in \text{init} \wedge v \in \text{init}' \wedge (\Box \text{lft-rel-st } (\lambda(u, v)$

$(u', v'). r u u' OO r' v v') (u || v) x y :$
(is ?p \implies ?S = ?T)

proof –

assume $A: u' \in \text{init}'$

from A **have** $[\text{simp}]$: $(\lambda x. (\forall u. u \in \text{init} \longrightarrow \text{prec-st } p r u x) \wedge (\forall y. (\exists u. u \in \text{init} \wedge (\Box \text{lft-rel-st } r) u x y) \longrightarrow (\forall u. u \in \text{init}' \longrightarrow \text{prec-st } p' r' u y)))$
 $= (\lambda x. \forall u v. u \in \text{init} \wedge v \in \text{init}' \longrightarrow \text{prec-st } p r u x \wedge (\forall y. (\Box \text{lft-rel-st } r) u x y \longrightarrow \text{prec-st } p' r' v y))$

by $(\text{auto simp add: fun-eq-iff})$

have $[\text{simp}]$: $(\lambda x y. \exists u. u \in \text{init} \wedge (\Box \text{lft-rel-st } r) u x y) OO (\lambda x y. \exists u. u \in \text{init}' \wedge (\Box \text{lft-rel-st } r') u x y)$
 $= (\lambda x y. \exists u v. u \in \text{init} \wedge v \in \text{init}' \wedge (((\Box \text{lft-rel-st } r) u) OO ((\Box \text{lft-rel-st } r') v)) x y)$

by $(\text{auto simp add: fun-eq-iff})$

from A **have** $?S = \{.x . \forall u . u \in \text{init} \longrightarrow \text{prec-st } p r u x .\} \circ$

$[: x \rightsquigarrow y . \exists u :: \text{nat} \Rightarrow 'e. u \in \text{init} \wedge (\Box \text{lft-rel-st } r) u x y :] \circ$

$(\{.x . \forall u . u \in \text{init}' \longrightarrow \text{prec-st } p' r' u x .\} \circ [: x \rightsquigarrow y . \exists u . u \in \text{init}' \wedge (\Box \text{lft-rel-st } r') u x y :])$

by $(\text{simp add: SymSystem-rel})$

also have $\dots = \{. \lambda x . \forall u . u \in \text{init} \longrightarrow \text{prec-st } p r u x .\} \circ [: x \rightsquigarrow y . \exists u . u \in \text{init} \wedge (\Box \text{lft-rel-st } r) u x y :]$

$\{. x . \forall u . u \in \text{init}' \longrightarrow \text{prec-st } p' r' u x .\} \circ [: x \rightsquigarrow y . \exists u . u \in \text{init}' \wedge (\Box \text{lft-rel-st } r') u x y :]$

by $(\text{simp add: comp-assoc } [THEN \text{sym}])$

also have $\dots = ?T$

by $(\text{simp add: assert-demonic-comp always-lft-rel-comp-a})$

finally show $?thesis$

by simp

qed

We show next that the composition of two SymSystem S and S' is not equal to the SymSystem of the composition of local transitions of S and S'

definition $\text{init}S = \{u . \text{fst } (u (0::\text{nat})) = (0::\text{nat})\}$

definition $\text{localPrec}S = (\top :: \text{nat} \times \text{nat} \Rightarrow \text{nat} \times \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool})$

definition $\text{localRel}S = (\lambda (u :: \text{nat}, v) (u', v' :: \text{nat}) (x :: \text{nat}) (y :: \text{nat}) . u = 0 \wedge u' = 1 \wedge v = v')$

definition $\text{init}S' = (\top :: (\text{nat} \Rightarrow (\text{nat} \times \text{nat})) \text{ set})$

definition $\text{localPrec}S' = (\perp :: \text{nat} \times \text{nat} \Rightarrow \text{nat} \times \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool})$

definition $\text{localRel}S' = (\lambda (u :: \text{nat}, v) (u', v' :: \text{nat}) (x :: \text{nat}) (y :: \text{nat}) . u = u')$

definition $\text{symb}S = \text{SymSystem } \text{init}S \text{ localPrec}S \text{ localRel}S$

definition $\text{symb}S' = \text{SymSystem } \text{init}S' \text{ localPrec}S' \text{ localRel}S'$

definition $\text{localPrec}SS' = (\lambda (u :: \text{nat}, v :: \text{nat}) (u', v') (x :: \text{nat}) . 0 < u)$

definition $\text{localRel}SS' = (\lambda (u, v :: \text{nat}) (u' :: \text{nat}, v' :: \text{nat}) (x :: \text{nat}) (z :: \text{nat}) . (u :: \text{nat}) = 0 \wedge u' = 1)$

lemma *localSS'-aux*: ($\lambda x. \forall (a::nat) (aa::nat) (b::nat). \neg (\text{case } x \text{ of } (x::nat, u::nat, v::nat) \Rightarrow \lambda ab. u = 0 \wedge (\text{case } ab \text{ of } (y, u', v') \Rightarrow u' = \text{Suc } 0 \wedge v = v')) (a, aa, b))$)
= ($\lambda (x, u, v). u > 0$)
by (*auto simp add: fun-eq-iff*)

lemma *localSS'-aux-b*: (($\lambda(x, u, v) ab. u = 0 \wedge (\text{case } ab \text{ of } (y, u', v') \Rightarrow u' = \text{Suc } 0 \wedge v = v')$) *OO* ($\lambda(x, u, v) (y, u', v'). u = u'$))
= ($\lambda (x, u, v) (y, u', v'). u = 0 \wedge u' = 1$)
by (*simp add: fun-eq-iff relcompp-exists*)

lemma { $.x, (u, v). \text{localPrecS } (u, v) (a, b) x.$ } *o* [$:x, (u, v) \rightsquigarrow y, (u', v'). \text{localRelS } (u, v) (u', v') x y.$] *o*
{ $.x, (u, v). \text{localPrecS}' (u, v) (c, d) x.$ } *o* [$:x, (u, v) \rightsquigarrow y, (u', v'). \text{localRelS}' (u, v) (u', v') x y.$]
= { $.x, (u, v). \text{localPrecSS}' (u, v) (e, f) x.$ } *o* [$:x, (u, v) \rightsquigarrow y, (u', v'). \text{localRelSS}' (u, v) (u', v') x y.$]

by (*simp add: assert-demonic-comp localPrecS'-def localPrecS-def localRelS-def localRelS'-def relcompp-exists localPrecSS'-def localRelSS'-def localSS'-aux localSS'-aux-b*)

lemma [*simp*]: [$\perp::('a \Rightarrow 'b \Rightarrow ('c::\text{boolean-algebra}))$] = \top
by (*simp add: fun-eq-iff demonic-def*)

definition *symbSS'* = *SymSystem* *initS* *localPrecSS'* *localRelSS'*

lemma *symbSS'-aux*: ($\lambda x. \forall u. \text{fst } (u \ 0) = 0 \longrightarrow (\forall n. (\forall i < n. \text{Ex } ((\text{case } u \ i \text{ of } (u, v) \Rightarrow \lambda(u', v)::nat) x \ z. u = 0 \wedge u' = \text{Suc } 0) (u (\text{Suc } i)) (x \ i))) \longrightarrow (\text{case } u \ n \text{ of } (u, v) \Rightarrow \lambda(u', v') x. 0 < u) (u (\text{Suc } n)) (x \ n))$) = \perp

apply (*auto simp add: fun-eq-iff*)
by (*rule-tac x = \lambda i . (i::nat, i) in exI, simp*)

lemma *symbSS'*: *symbSS'* = \perp

by (*simp add: symbSS'-def SymSystem-rel initS-def localPrecSS'-def localRelSS'-def prec-st-simp inpt-st-def symbSS'-aux*)

lemma *symbS*: *symbS* = \top

proof (*simp add: symbS-def SymSystem-rel initS-def localPrecS-def localRelS-def*)
have [*simp*]: ($\lambda x. \forall u. \text{fst } (u \ 0) = 0 \longrightarrow \text{prec-st } \top (\lambda (u, v) (u', v') x \ y . u = 0 \wedge u' = \text{Suc } 0 \wedge v = v') u \ x$) = \top

by (*simp-all add: fun-eq-iff prec-st-def always-def lft-rel-st-def at-fun-def lft-pred-st-def inpt-st-def until-def*)

have [*simp*]: ($\lambda x \ y. \exists u. \text{fst } (u \ 0) = 0 \wedge (\Box \text{lft-rel-st } (\lambda (u, v) (u', v') (x) (y)). u = 0 \wedge u' = \text{Suc } 0 \wedge v = v')) u \ x \ y$) = \perp

proof (*auto simp add: fun-eq-iff always-def lft-rel-st-def at-fun-def*)


```

fix x::nat ⇒ 'a and xa :: nat ⇒ 'b and u::nat ⇒ nat × 'c
assume A: ∀ a . (case u a of (e, f) ⇒ λ(u', v') x y. e = 0 ∧ u' = Suc 0 ∧ f
= v') (u (Suc a)) (x a) (xa a)
  {fix n:: nat
    from A have fst (u n) = 0 ∧ fst (u (Suc n)) = Suc 0
    by (drule-tac x = n in spec, case-tac u n, case-tac u (Suc n), auto)
  }
note B = this
then have fst (u (Suc 0)) = 0 by auto
moreover have fst (u (Suc 0)) = Suc 0 using B [of 0] by auto
ultimately show (0) < fst (u (0)) by auto
qed

```

```

show { . λx. ∀ u. fst (u 0) = 0 → prec-st ⊤ (λ(u, v) (u', v') x y. u = 0 ∧ u'
= Suc 0 ∧ v = v') u x . } ∘
  [ : λ x y . ∃ u . fst (u 0) = 0 ∧ (□ lft-rel-st (λ(u, v) (u', v') x y. u = 0
∧ u' = Suc 0 ∧ v = v')) u x y : ] =
  ⊤
by simp
qed

```

```

lemma symbS o symbS' ≠ symbSS'
by (simp add: symbSS' symbS fun-eq-iff)

```

```

lemma prec-st-inpt: prec-st (inpt-st r) r = (□ (lft-pred-st (inpt-st r)))
by (simp add: prec-st-def neg-until-always)

```

```

lemma grd (SymSystem init p r) = SUPREMUM init (-prec-st p r □ (□
(lft-pred-st (inpt-st r))))

```

```

proof (unfold fun-eq-iff, auto simp add: grd-def SymSystem-rel demonic-def
assert-def)

```

```

fix x :: nat ⇒ 'a and xa :: nat ⇒ 'b and u :: nat ⇒ 'c
assume ∀ xa::nat ⇒ 'c ∈ init. prec-st p r xa x ∧ ¬ (□ lft-pred-st (inpt-st r))
xa x

```

```

and u ∈ init
and (□ lft-rel-st r) u x xa
then show False
by (auto simp add: always-def lft-pred-st-def inpt-st-def at-fun-def lft-rel-st-def)

```

next

```

fix x :: nat ⇒ 'a and xa :: nat ⇒ 'c
assume B: xa ∈ init
assume (λy . ∃ u . u ∈ init ∧ (□ lft-rel-st r) u x y) ≤ ⊥
then have A: ∀ y u . u ∉ init ∨ ¬ (□ lft-rel-st r) u x y
by auto
let ?y = λ n . (SOME z . r (xa n) (xa (Suc n)) (x n) z)
from B and A have ¬ (□ lft-rel-st r) xa x ?y by simp
moreover assume (□ lft-pred-st (inpt-st r)) xa x
ultimately show False
apply (simp add: always-def lft-pred-st-def inpt-st-def at-fun-def lft-rel-st-def)

```

by (*metis* (*full-types*) *tfl-some*)
qed

definition *guard* $S = \{.(grd\ S)::'a \Rightarrow bool).\} o\ S$

lemma $((grd\ (local\ init\ init\ S))::'a \Rightarrow bool) = SUPREMUM\ init\ (grd\ o\ S)$
by (*simp* *add: fun-eq-iff local-init-def assert-def grd-def*)

lemma $u \in init \Longrightarrow guard\ ([:z \rightsquigarrow u, x . u \in init \wedge z = x:] o\ \{.u, x . p\ u\ x.\} o\ [:u, x \rightsquigarrow y . r\ u\ x\ y :])$
 $= [:z \rightsquigarrow u, x . u \in init \wedge z = x:] o\ \{.u, x . u \in init \wedge (\exists a. a \in init \wedge (p\ a\ x \longrightarrow Ex\ (r\ a\ x))) \wedge p\ u\ x.\} o\ [:u, x \rightsquigarrow y . ((r\ u\ x\ y)::bool) :]$
by (*auto simp* *add: fun-eq-iff local-init-def guard-def grd-def assert-def demonic-def le-fun-def*)

lemma *inpt-str-comp-aux*: $(\forall n. (\forall i < n. inpt\ st\ (\lambda(u, v) (u', v')). r\ u\ u' OO\ r'\ v\ v') (u\ i) (u\ (Suc\ i)) (x\ i)) \longrightarrow$
 $inpt\ st\ r\ (fst\ (u\ n))\ (fst\ (u\ (Suc\ n)))\ (x\ n) \wedge (\forall y. r\ (fst\ (u\ n))\ (fst\ (u\ (Suc\ n))))\ (x\ n)\ y \longrightarrow inpt\ st\ r'\ (snd\ (u\ n))\ (snd\ (u\ (Suc\ n)))\ y) \longrightarrow$
 $(\forall i < n. inpt\ st\ r\ ((fst\ o\ u)\ i)\ ((fst\ o\ u)\ (Suc\ i))\ (x\ i) \wedge (\forall y. r\ (fst\ (u\ i))\ (fst\ (u\ (Suc\ i))))\ (x\ i)\ y \longrightarrow inpt\ st\ r'\ (snd\ (u\ i))\ (snd\ (u\ (Suc\ i)))\ y)$
(is $(\forall n. ?p\ n) \longrightarrow ?q\ n$ **)**

proof (*induction* n)

case 0

show $?case$ **by** *auto*

next

case $(Suc\ n)$

show $?case$

proof *auto*

fix $i::nat$

assume $B: \forall n. ?p\ n$

then **have** $A: ?p\ n$ **(is** $?A \longrightarrow ?B$ **)**

by *simp*

from *Suc* **and** B **have** $C: ?q\ n$

by *simp*

assume $i < Suc\ n$

then **show** $inpt\ st\ r\ (fst\ (u\ i))\ (fst\ (u\ (Suc\ i)))\ (x\ i)$

proof *cases*

assume $i < n$

then **show** $?thesis$

by (*metis* *Suc.IH* B *comp-apply*)

next

assume $\neg i < n$

from *this* **and** $(i < Suc\ n)$ **have** [*simp*]: $i = n$ **by** *simp*

show $?thesis$

proof *cases*

assume $?A$

from *this* **and** A **have** $D: ?B$ **by** *simp*

```

    from D show ?thesis
      by (metis ⟨i = n⟩)
  next
    assume ¬ ?A
    then obtain j where j: j < n ∧ ¬ inpt-st (λ (u, v) . λ (u', v') . r
u u' OO r' v v') (u j) (u (Suc j)) (x j)
      by auto
    with C have inpt-st r (fst (u j)) (fst (u (Suc j))) (x j) ∧ (∀ y. r (fst
(u j)) (fst (u (Suc j))) (x j) y → inpt-st r' (snd (u j)) (snd (u (Suc j))) y)
      by auto
    with j show ?thesis
      apply (case-tac u j)
      apply (case-tac u (Suc j))
      apply (simp add: inpt-st-def)
      by (metis relcompp.relcompI)
  qed
next
  fix i::nat fix y :: 'e
  assume B: ∀ n . ?p n
  then have A: ?p n (is ?A → ?B)
    by simp
  from Suc and B have C: ∀ i < n. inpt-st r (fst (u i)) (fst (u (Suc i))) (x
i) ∧ (∀ y. r (fst (u i)) (fst (u (Suc i))) (x i) y → inpt-st r' (snd (u i)) (snd (u
(Suc i))) y)
    by simp
  assume E: r (fst (u i)) (fst (u (Suc i))) (x i) y
  assume i < Suc n
  then show inpt-st r' (snd (u i)) (snd (u (Suc i))) y
    proof cases
      assume i < n
      from this and E and C show ?thesis
        by simp
    next
      assume ¬ i < n
      from this and ⟨i < Suc n⟩ have [simp]: i = n by simp
      show ?thesis
        proof (cases ?A)
          case True
          with A have D: ?B by simp
          from D and E show ?thesis
            by (metis ⟨i = n⟩)
        next
          case False
          then obtain j where j: j < n ∧ ¬ inpt-st (λ (u, v) . λ (u', v') . r
u u' OO r' v v') (u j) (u (Suc j)) (x j)
            by auto
          with C have inpt-st r (fst (u j)) (fst (u (Suc j))) (x j) ∧ (∀ y. r (fst
(u j)) (fst (u (Suc j))) (x j) y → inpt-st r' (snd (u j)) (snd (u (Suc j))) y)

```

by *auto*
with *j* **show** *?thesis*
by (*case-tac u j, case-tac u (Suc j), simp add: inpt-st-def, metis*
relcompp.relcompI)
qed
qed
qed
qed

lemma *inpt-str-comp-aux-a*: $(\forall n. (\forall i < n. \text{inpt-st } (\lambda(u, v) (u', v')). r u u' OO r' v v') (u i) (u (Suc i)) (x i)) \longrightarrow$
 $\text{inpt-st } r (fst (u n)) (fst (u (Suc n))) (x n) \wedge (\forall y. r (fst (u n)) (fst (u (Suc n)))) (x n) y \longrightarrow \text{inpt-st } r' (snd (u n)) (snd (u (Suc n))) y) \implies$
 $\text{inpt-st } r ((fst o u) n) ((fst o u) (Suc n)) (x n) \wedge (\forall y. r (fst (u n)) (fst (u (Suc n)))) (x n) y \longrightarrow \text{inpt-st } r' (snd (u n)) (snd (u (Suc n))) y)$
by (*cut-tac n = Suc n and r = r and r' = r' and u = u and x = x in inpt-str-comp-aux, simp*)

definition *rel-st r r'* = $(\lambda (u, v) (u', v') x z . \text{inpt-st } r u u' x \wedge (\forall y . r u u' x y \longrightarrow \text{inpt-st } r' v v' y) \wedge (r u u' OO r' v v') x z)$

lemma *inpt-str-comp-a*: $(\text{prec-st } (\text{inpt-st } r) r (fst o u) x \wedge (\forall y. (\Box \text{lft-rel-st } r) (fst o u) x y \longrightarrow \text{prec-st } (\text{inpt-st } r') r' (snd o u) y)) =$
 $\text{prec-st } (\lambda u u' x . \text{inpt-st } r (fst u) (fst u') x \wedge (\forall y . r (fst u) (fst u') x y \longrightarrow (\text{inpt-st } r' (snd u) (snd u') y))) (\lambda(u, v) (u', v'). r u u' OO r' v v') u x$

proof (*auto simp add: prec-st-inpt prec-st-simp*)
fix *n:: nat*
assume $(\Box \text{lft-pred-st } (\text{inpt-st } r)) (fst o u) x$
then show $\text{inpt-st } r (fst (u n)) (fst (u (Suc n))) (x n)$
by (*simp add: always-def lft-pred-st-def at-fun-def*)
next
fix *n:: nat and y :: 'c*
assume *A*: $(\Box \text{lft-pred-st } (\text{inpt-st } r)) (fst o u) x$
assume *B*: $r (fst (u n)) (fst (u (Suc n))) (x n) y$
assume *C*: $\forall i < n. \text{inpt-st } (\lambda(u::'a, v::'d) (u'::'a, v'::'d). r u u' OO r' v v') (u i) (u (Suc i)) (x i)$
let *?y* = $\lambda i . (\text{if } i = n \text{ then } y \text{ else } (\text{SOME } y . r ((fst o u) i) ((fst o u) (Suc i)) (x i) y))$
assume $\forall y . (\Box \text{lft-rel-st } r) (fst o u) x y \longrightarrow (\Box \text{lft-pred-st } (\text{inpt-st } r')) (snd o u) y$
then have *D*: $(\Box \text{lft-rel-st } r) (fst o u) x ?y \longrightarrow (\Box \text{lft-pred-st } (\text{inpt-st } r')) (snd o u) ?y$
by *simp*
from *A* **and** *B* **have** *E*: $(\Box \text{lft-rel-st } r) (fst o u) x ?y$
apply (*auto simp add: always-def at-fun-def lft-rel-st-def lft-pred-st-def inpt-st-def*)
by (*metis tfl-some*)
from *D* **and** *E* **have** $(\Box \text{lft-pred-st } (\text{inpt-st } r')) (snd o u) ?y$ **by** *simp*

from A **and** E **and this show** $\text{inpt-st } r' (\text{snd } (u \ n)) (\text{snd } (u \ (\text{Suc } n))) \ y$
apply (*simp add: always-def lft-pred-st-def at-fun-def*)
apply (*drule-tac x = n in spec*)
apply (*drule-tac x = n in spec*)
by (*drule-tac x = n in spec, simp*)
next
assume $\forall n . (\forall i < n . \text{inpt-st } (\lambda(u::'a, v::'d) (u'::'a, v'::'d) . r \ u \ u' \ OO \ r' \ v \ v') (u \ i) (u \ (\text{Suc } i)) (x \ i)) \longrightarrow$
 $\text{inpt-st } r (\text{fst } (u \ n)) (\text{fst } (u \ (\text{Suc } n))) (x \ n) \wedge (\forall y::'c . r (\text{fst } (u \ n)) (\text{fst } (u \ (\text{Suc } n))) (x \ n) \ y \longrightarrow \text{inpt-st } r' (\text{snd } (u \ n)) (\text{snd } (u \ (\text{Suc } n))) \ y)$
then show $(\square \text{ lft-pred-st } (\text{inpt-st } r)) (\text{fst } \circ \ u) \ x$
apply (*auto simp add: always-def lft-pred-st-def at-fun-def*)
apply (*drule inpt-str-comp-aux-a*)
by *auto*
next
fix $y::\text{nat} \Rightarrow 'c$
assume $\forall n . (\forall i < n . \text{inpt-st } (\lambda(u::'a, v::'d) (u'::'a, v'::'d) . r \ u \ u' \ OO \ r' \ v \ v') (u \ i) (u \ (\text{Suc } i)) (x \ i)) \longrightarrow$
 $\text{inpt-st } r (\text{fst } (u \ n)) (\text{fst } (u \ (\text{Suc } n))) (x \ n) \wedge (\forall y::'c . r (\text{fst } (u \ n)) (\text{fst } (u \ (\text{Suc } n))) (x \ n) \ y \longrightarrow \text{inpt-st } r' (\text{snd } (u \ n)) (\text{snd } (u \ (\text{Suc } n))) \ y)$
moreover assume $(\square \text{ lft-rel-st } r) (\text{fst } \circ \ u) \ x \ y$
ultimately show $(\square \text{ lft-pred-st } (\text{inpt-st } r')) (\text{snd } \circ \ u) \ y$
apply (*auto simp add: always-def lft-pred-st-def at-fun-def*)
apply (*drule inpt-str-comp-aux-a*)
by (*auto simp add: lft-rel-st-def*)
qed

lemma *inpt-str-comp-b*: $\text{prec-st } (\lambda \ u \ u' \ x . \text{inpt-st } r (\text{fst } u) (\text{fst } u') \ x \wedge$
 $(\forall y . r (\text{fst } u) (\text{fst } u') \ x \ y \longrightarrow (\text{inpt-st } r' (\text{snd } u) (\text{snd } u') \ y))) (\lambda(u, v) (u', v') . r \ u \ u' \ OO \ r' \ v \ v') \ u \ x$
 $= (\square (\text{lft-pred-st } (\text{inpt-st } (\text{rel-st } r \ r')))) \ u \ x$
proof (*auto simp add: prec-st-simp always-def lft-pred-st-def at-fun-def rel-st-def*)
fix $m::\text{nat}$
assume $A: \forall n . (\forall i < n . \text{inpt-st } (\lambda(u, v) (u', v') . r \ u \ u' \ OO \ r' \ v \ v') (u \ i) (u \ (\text{Suc } i)) (x \ i)) \longrightarrow$
 $\text{inpt-st } r (\text{fst } (u \ n)) (\text{fst } (u \ (\text{Suc } n))) (x \ n)$
 $\wedge (\forall y . r (\text{fst } (u \ n)) (\text{fst } (u \ (\text{Suc } n))) (x \ n) \ y \longrightarrow \text{inpt-st } r' (\text{snd } (u \ n)) (\text{snd } (u \ (\text{Suc } n))) \ y) (\text{is } \forall n . ?p \ n \longrightarrow ?q \ n \wedge ?r \ n)$
then have $?q \ m$
by (*drule-tac n = m in inpt-str-comp-aux-a, simp*)
then obtain y **where** $B: r ((\text{fst } \circ \ u) \ m) ((\text{fst } \circ \ u) (\text{Suc } m)) (x \ m) \ y$ **by**
(*auto simp add: inpt-st-def*)
from A **have** $?r \ m$
by (*drule-tac n = m in inpt-str-comp-aux-a, simp*)
from this B **show** $\text{inpt-st } (\lambda(u, v) (u', v') (x::'c) z . \text{inpt-st } r \ u \ u' \ x \wedge (\forall y . r \ u \ u' \ x \ y$
 $\longrightarrow \text{inpt-st } r' \ v \ v' \ y) \wedge (r \ u \ u' \ OO \ r' \ v \ v') \ x \ z) (u \ m) (u \ (\text{Suc } m)) (x \ m)$
apply (*case-tac u m*)
apply (*case-tac u (Suc m)*)

apply (*simp add: inpt-st-def*)
by (*metis relcompp.relcompI*)
next
fix *m::nat*
assume $\forall m. \text{inpt-st } (\lambda(u, v) (u', v') (x) z. \text{inpt-st } r \ u \ u' \ x \wedge (\forall y. r \ u \ u' \ x \ y \longrightarrow \text{inpt-st } r' \ v \ v' \ y))$
 $\wedge (r \ u \ u' \ OO \ r' \ v \ v') \ x \ z) (u \ m) (u \ (\text{Suc } m)) (x \ m) (\text{is } \forall m. ?p \ m)$
then have *?p m* **by** *simp*
then show $\text{inpt-st } r \ (\text{fst } (u \ m)) \ (\text{fst } (u \ (\text{Suc } m))) (x \ m)$
apply (*simp add: inpt-st-def*)
by (*case-tac u m, case-tac u (Suc m), simp*)
next
fix *m::nat* **and** *y :: 'e*
assume $\forall m. \text{inpt-st } (\lambda(u, v) (u', v') (x) z. \text{inpt-st } r \ u \ u' \ x \wedge (\forall y. r \ u \ u' \ x \ y \longrightarrow \text{inpt-st } r' \ v \ v' \ y))$
 $\wedge (r \ u \ u' \ OO \ r' \ v \ v') \ x \ z) (u \ m) (u \ (\text{Suc } m)) (x \ m) (\text{is } \forall m. ?p \ m)$
then have *?p m* **by** *simp*
moreover assume $r \ (\text{fst } (u \ m)) \ (\text{fst } (u \ (\text{Suc } m))) (x \ m) \ y$
ultimately show $\text{inpt-st } r' \ (\text{snd } (u \ m)) \ (\text{snd } (u \ (\text{Suc } m))) y$
apply (*simp add: inpt-st-def*)
by (*case-tac u m, case-tac u (Suc m), simp*)
qed

lemma *inpt-str-comp*: $(\text{prec-st } (\text{inpt-st } r) \ r \ (\text{fst } \circ \ u) \ x \wedge (\forall y. (\Box \ \text{lft-rel-st } r) \ (\text{fst } \circ \ u) \ x \ y \longrightarrow \text{prec-st } (\text{inpt-st } r') \ r' \ (\text{snd } \circ \ u) \ y))$
 $= (\Box \ (\text{lft-pred-st } (\text{inpt-st } (\text{rel-st } r \ r')))) \ u \ x$
by (*simp add: inpt-str-comp-a inpt-str-comp-b*)

lemma *RSysTmp-inpt-comp*: $u' \in \text{init}' \implies \text{SymSystem } \text{init} \ (\text{inpt-st } r) \ r \ o \ \text{SymSystem } \text{init}' \ (\text{inpt-st } r') \ r'$

$= \text{SymSystem } (\text{zip-set } \text{init } \text{init}') \ (\text{inpt-st } (\text{rel-st } r \ r')) \ (\text{rel-st } r \ r')$

proof –

assume $A : u' \in \text{init}'$

have [*simp*]: $(\lambda x \ y. (\text{case } x \ \text{of } (x, xa) \Rightarrow (\Box \ \text{lft-pred-st } (\text{inpt-st } (\text{rel-st } r \ r')))) \ x \ xa) \wedge$

$(\text{case } x \ \text{of } (x, xa) \Rightarrow (\Box \ \text{lft-rel-st } (\lambda(u, v) (u', v'). r \ u \ u' \ OO \ r' \ v \ v')) \ x \ xa)$
 $y) = (\lambda(x, y). (\Box \ \text{lft-rel-st } (\text{rel-st } r \ r')) \ x \ y) (\text{is } ?a = ?b)$

proof (*auto simp add: fun-eq-iff always-def at-fun-def lft-pred-st-def lft-rel-st-def rel-st-def inpt-st-def*)

fix $a :: \text{nat} \Rightarrow 'e \times 'a$ **and** $b :: \text{nat} \Rightarrow 'c$ **and** $x :: \text{nat} \Rightarrow 'b$ **and** $xa :: \text{nat}$

assume $\forall xa::\text{nat}. (\text{case } a \ xa \ \text{of } (u::'e, v::'a) \Rightarrow \lambda(u::'e, v::'a). r \ u \ u' \ OO \ r' \ v \ v') (a \ (\text{Suc } xa)) (b \ xa) (x \ xa) (\text{is } \forall xa. ?P \ xa)$

then have $A : ?P \ xa$ **by** *simp*

assume $\forall x. \text{Ex } ((\text{case } a \ x \ \text{of } (u, v) \Rightarrow \lambda(u', v') (x) z. \text{Ex } (r \ u \ u' \ x) \wedge (\forall y. r \ u \ u' \ x \ y \longrightarrow \text{Ex } (r' \ v \ v' \ y)) \wedge (r \ u \ u' \ OO \ r' \ v \ v') \ x \ z) (a \ (\text{Suc } x)) (b \ x)) (\text{is } \forall xa. ?Q \ xa)$

then have $?Q \ xa$ **by** *simp*

from *this* **and** *A* **show** (*case a xa of* ($u, v \Rightarrow \lambda(u', v')(x) z$). *Ex* ($r u u'$
 $x) \wedge (\forall y. r u u' x y \longrightarrow Ex (r' v v' y)) \wedge (r u u' OO r' v v') x z$) (*a* (*Suc xa*))
(*b xa*) (*x xa*)
by (*case-tac a xa, case-tac a (Suc xa), simp*)
next
fix $a :: nat \Rightarrow 'e \times 'a$ **and** $b :: nat \Rightarrow 'c$ **and** $x :: nat \Rightarrow 'b$ **and** $xa :: nat$
assume $\forall xa . (case a xa of (u::'e, v::'a) \Rightarrow \lambda(u'::'e, v'::'a) (x::'c) z::'b$.
 $Ex (r u u' x) \wedge (\forall y::'d. r u u' x y \longrightarrow Ex (r' v v' y)) \wedge (r u u' OO r' v v') x z$)
(*a (Suc xa)*) (*b xa*) (*x xa*) (**is** $\forall xa . ?Q xa$)
then have *?Q xa* **by** *simp*
then show (*case a xa of* ($u::'e, v::'a) \Rightarrow \lambda(u'::'e, v'::'a). r u u' OO r' v$
 $v')$ (*a (Suc xa)*) (*b xa*) (*x xa*)
by (*case-tac a xa, case-tac a (Suc xa), simp*)
qed

from *A* **have** *SymSystem* *init* (*inpt-st r*) *r* *o* *SymSystem* *init'*(*inpt-st r'*) *r'* =
 $[: z \rightsquigarrow u, x . fst \circ u \in init \wedge snd \circ u \in init' \wedge z = x :] \circ$
 $(\{. u, x . prec-st (inpt-st r) r (fst \circ u) x \wedge (\forall y::nat \Rightarrow 'd. (\Box lft-rel-st r) (fst$
 $\circ u) x y \longrightarrow prec-st (inpt-st r') r' (snd \circ u) y.\} \circ$
 $[: (\lambda(u, x). ((\Box lft-rel-st (\lambda(u, v) (u', v')). r u u' OO r' v v')) u x) :])$
by (*unfold SymSystem-comp, simp add: comp-assoc*)
also have $\dots = [: z \rightsquigarrow u, x . fst \circ u \in init \wedge snd \circ u \in init' \wedge z = x :] \circ (\{.$
 $x, y . (\Box lft-pred-st (inpt-st (rel-st r r')) x y .\} \circ [: ?b :])$
by (*subst assert-demonic, simp add: inpt-str-comp*)
also have $\dots = SymSystem (zip-set init init') (inpt-st (rel-st r r')) (rel-st r$
 $r')$
by (*simp add: SymSystem-def prec-st-inpt comp-assoc zip-set-def*)
finally show *?thesis* **by** *simp*
qed

definition *GrdSymSystem* *init r* = $[: z \rightsquigarrow u, x . u \in init \wedge z = x :] \circ trs (\lambda (u,$
 $x) y . (\Box(lft-rel-st r)) u x y)$

lemma *inpt-always*: *inpt* ($\lambda(x, y). (\Box lft-rel-st r) x y$) = ($\lambda(x, y). (\Box lft-pred-st$
 $(inpt-st r)) x y$)

proof (*auto simp add: fun-eq-iff*)
fix $a :: nat \Rightarrow 'a$ **and** $b :: nat \Rightarrow 'b$
assume *inpt* ($\lambda(x, y). (\Box lft-rel-st r) x y$) (*a, b*)
then show $(\Box lft-pred-st (inpt-st r)) a b$
by (*auto simp add: inpt-def lft-pred-st-def inpt-st-def always-def at-fun-def*
lft-rel-st-def)
next
fix $a :: nat \Rightarrow 'a$ **and** $b :: nat \Rightarrow 'b$
let $?y = \lambda n . (SOME y . r (a n) (a (Suc n)) (b n) y)$
assume $(\Box lft-pred-st (inpt-st r)) a b$
then have $(\Box lft-rel-st r) a b ?y$
apply (*auto simp add: always-def at-fun-def lft-rel-st-def inpt-st-def lft-pred-st-def*)
by (*metis tfl-some*)
then show *inpt* ($\lambda(x, y). (\Box lft-rel-st r) x y$) (*a, b*)

by (auto simp add: inpt-def)
qed

lemma *GrdSymSystem init r = SymSystem init (inpt-st r) r*
by (simp add: GrdSymSystem-def SymSystem-def trs-def prec-st-inpt comp-assoc inpt-always)

4.2 Example: COUNTER

In this section we introduce an example counter that counts how many times the input variable x is true. The input is a sequence of boolean values and the output is a sequence of natural numbers. The output at some moment in time is the number of true values seen so far in the input.

We defined the system counter in two different ways and we show that the two definitions are equivalent. The first definition takes the entire input sequence and it computes the corresponding output sequence. We introduce the second version of the counter as a reactive system based on a symbolic transition system. We use a local variable to record the number of true values seen so far, and initially the local variable is zero. At every step we increase the local variable if the input is true. The output of the system at every step is equal to the local variable.

primrec *count :: bool trace \Rightarrow nat trace where*
count x 0 = (if x 0 then 1 else 0) |
count x (Suc n) = (if x (Suc n) then count x n + 1 else count x n)

definition *Counter-global n = {x . ($\forall k . count x k \leq n$)} o [x \rightsquigarrow y . y = count x:]*

definition *prec-count M u u' x = (u \leq M)*

definition *rel-count u u' x y = ((x \longrightarrow u' = Suc u) \wedge (\neg x \longrightarrow u' = u) \wedge y = u')*

lemma *counter-a-aux: u 0 = 0 \implies $\forall i < n. (x i \longrightarrow u (Suc i) = Suc (u i)) \wedge (\neg x i \longrightarrow u (Suc i) = u i) \implies ($\forall i < n . count x i = u (Suc i)$)$*

proof (induction n)
 case 0
 show ?case by simp
 next
 case (Suc n)
 {fix j::nat
 assume $\forall i < Suc n. (x i \longrightarrow u (Suc i) = Suc (u i)) \wedge (\neg x i \longrightarrow u (Suc i) = u i)$
 and $j < Suc n$
 and $u (0::nat) = (0::nat)$
 from this and Suc have $count x j = u (Suc j)$
 by (case-tac j, auto)
 }

from *Suc* and *this* show *?case*
 by *auto*
 qed

lemma *counter-b-aux*: $u\ 0 = 0 \implies \forall n. (xa\ n \longrightarrow u\ (Suc\ n) = Suc\ (u\ n)) \wedge$
 $(\neg\ xa\ n \longrightarrow u\ (Suc\ n) = u\ n) \wedge xb\ n = u\ (Suc\ n)$
 $\implies count\ xa\ n = u\ (Suc\ n)$
 by (*induction* *n*, *simp-all*)

definition *COUNTER* $M = SymSystem\ \{u . u\ 0 = 0\}$ (*prec-count* *M*) *rel-count*

lemma *COUNTER* = *Counter-global*

proof –

have $A: (\lambda x\ y . \exists u::nat \Rightarrow nat. u\ (0::nat) = (0::nat) \wedge (\Box\ lft\ rel\ st\ rel\ count)$
 $u\ x\ y)$

= $(\lambda x\ y . y = count\ x)$

proof (*simp* *add: fun-eq-iff lft-rel-st-def rel-count-def always-def at-fun-def*,
safe)

fix $x::nat \Rightarrow bool$ and $xa::nat \Rightarrow nat$ and $u::nat \Rightarrow nat$ and $xb::nat$
assume $A: u\ 0 = 0$

assume $B: \forall xb . (x\ xb \longrightarrow u\ (Suc\ xb) = Suc\ (u\ xb)) \wedge (\neg\ x\ xb \longrightarrow u$
 $(Suc\ xb) = u\ xb) \wedge xa\ xb = u\ (Suc\ xb)$

from *A* and *this* have $count\ x\ xb = xa\ xb$

by (*drule-tac counter-b-aux*, *auto*)

then show $xa\ xb = count\ x\ xb$ **by** *simp*

next

fix $x::nat \Rightarrow bool$ and $xa::nat \Rightarrow nat$

def $A: u == \lambda i . if\ i = 0\ then\ 0\ else\ count\ x\ (i - 1)$

assume $B: \forall xb::nat. xa\ xb = count\ x\ xb$

{**fix** $xb::nat$

from *A* and *B* have $u\ 0 = 0 \wedge ((x\ xb \longrightarrow u\ (Suc\ xb) = Suc\ (u\ xb)) \wedge$
 $(\neg\ x\ xb \longrightarrow u\ (Suc\ xb) = u\ xb) \wedge xa\ xb = u\ (Suc\ xb))$

by (*case-tac* *xb*, *auto*)

}

then show $\exists u::nat \Rightarrow nat. u\ 0 = 0 \wedge (\forall xb. (x\ xb \longrightarrow u\ (Suc\ xb) = Suc$
 $(u\ xb)) \wedge (\neg\ x\ xb \longrightarrow u\ (Suc\ xb) = u\ xb) \wedge$
 $xa\ xb = u\ (Suc\ xb))$

by *auto*

qed

{**fix** $x::nat$

have $(\lambda xa . \forall u . u\ (0::nat) = (0::nat) \longrightarrow prec\ st\ (prec\ count\ x)\ rel\ count$
 $u\ xa) =$

$(\lambda xa::nat \Rightarrow bool. \forall k::nat. count\ xa\ k \leq x)$

proof (*simp* *add: fun-eq-iff lft-rel-st-def prec-st-def until-def*

lft-pred-st-def prec-count-def at-fun-def inpt-st-def rel-count-def, *safe*)

fix $xa::nat \Rightarrow bool$ and $k::nat$

def $A: uu == \lambda i . if\ i = 0\ then\ 0\ else\ count\ xa\ (i - 1)$

assume $(\forall u . u\ 0 = 0 \longrightarrow (\forall xb . (\exists x < xb. xa\ x \wedge u\ (Suc\ x) \neq Suc\ (u$
 $x) \vee \neg\ xa\ x \wedge u\ (Suc\ x) \neq u\ x) \vee u\ xb \leq x))$ (**is** $\forall u . ?s\ u$)

```

then have ?s uu (is ?p  $\longrightarrow$   $(\forall xb . (\exists x < xb . ?q\ xb\ x) \vee ?r\ xb)$ )
  by auto
from this and A have  $(\forall xb . (\exists x < xb . ?q\ xb\ x) \vee ?r\ xb)$ 
  by simp
then have  $(\exists x < (Suc\ k) . ?q\ (Suc\ k)\ x) \vee ?r\ (Suc\ k)$ 
  by simp
then obtain xb where  $xb < (Suc\ k) \wedge (?q\ (Suc\ k)\ xb \vee ?r\ (Suc\ k))$ 
  by auto
from this and A show count xa k  $\leq$  x
  by (case-tac xb, auto)
next
fix xaa:: nat  $\Rightarrow$  bool and u::nat  $\Rightarrow$  nat and xaa::nat
assume C:  $\forall k::nat. count\ xa\ k \leq x$ 
assume A:  $u\ (0::nat) = (0::nat)$ 
assume B:  $\neg u\ xaa \leq x$ 
from A and B have D:  $xaa > 0$ 
  by (metis le0 neg0-conv)
from this and B and C have count xa  $(xaa - 1) \neq u\ xaa$ 
  by metis
from this and D have E:  $\exists i < xaa. count\ xa\ i \neq u\ (Suc\ i)$ 
  by (metis One-nat-def Suc-diff-1 diff-Suc-less)
have  $u\ 0 = 0 \implies \forall i < xaa. (xa\ i \longrightarrow u\ (Suc\ i) = Suc\ (u\ i)) \wedge (\neg xa\ i$ 
 $\longrightarrow u\ (Suc\ i) = u\ i) \implies \forall i < xaa. count\ xa\ i = u\ (Suc\ i)$ 
  by (rule counter-a-aux, simp)
from this and A and E show  $(\exists x < xaa. xa\ x \wedge u\ (Suc\ x) \neq Suc\ (u\ x)$ 
 $\vee \neg xa\ x \wedge u\ (Suc\ x) \neq u\ x)$ 
  by auto
qed
}
note B = this
show ?thesis
by (simp add: fun-eq-iff COUNTER-def SymSystem-rel Counter-global-def A
B)

qed

```

4.3 Example: LIVE

The last example of this formalization introduces a system which does some local computation, and ensures some global liveness property. We show that this example is the fusion of a symbolic transition system and a demonic choice which ensures the liveness property of the output sequence. We also show that assuming some liveness property for the input, we can refine the example into an executable system that does not ensure the liveness property of the output on its own, but relies on the liveness of the input.

definition $rel\text{-}ex\ u\ u'\ x\ y = (((x \wedge u' = u + (1::int)) \vee (\neg x \wedge u' = u - 1) \vee u' = 0) \wedge (y = (u' = 0)))$

definition $prec\text{-}ex\ u\ u'\ x = (-1 \leq u \wedge u \leq 3)$

definition $LIVE = [x \rightsquigarrow u, x' . u (0::nat) = 0 \wedge x = x'] \circ \{u, x . prec-st\}$
 $prec-ex\ rel-ex\ u\ x . \}$
 $\circ [u, x \rightsquigarrow y . (\Box(\lambda\ u\ x\ y . rel-ex\ (u\ 0)\ (u\ 1)\ (x\ 0)\ (y\ 0)))\ u\ x\ y \wedge (\Box(\Diamond(\lambda\ y . y\ 0)))\ y :]$

thm *fusion-spec-local-a*

lemma *LIVE-fusion*: $LIVE = (SymSystem\ \{u . u\ 0 = 0\}\ prec-ex\ rel-ex) \parallel [x \rightsquigarrow y . (\Box(\Diamond(\lambda\ y . y\ 0)))\ y :]$

proof –

def $B: init == \{u . u (0::nat) = (0::int)\}$

then have $A: (\lambda\ i::nat . 0::int) \in init$

by *simp*

then have $([x \rightsquigarrow (u, y). u \in init \wedge x = y :] \circ \{(x, y). prec-st\}\ prec-ex\ rel-ex\ x\ y . \} \circ [:\lambda(x, y). (\Box\ lft-rel-st\ rel-ex)\ x\ y :]) \parallel$

$[:\lambda x. \Box\ \Diamond(\lambda y. y\ 0) :] =$

$[x \rightsquigarrow (u, y). u \in init \wedge x = y :] \circ \{(x, y). prec-st\}\ prec-ex\ rel-ex\ x\ y . \} \circ$

$[:(u, x) \rightsquigarrow y. (\Box\ lft-rel-st\ rel-ex)\ u\ x\ y \wedge (\Box\ \Diamond(\lambda y. y\ 0))\ y :]$

by *(unfold fusion-spec-local-a, auto)*

from B **and this show** *?thesis*

apply *(simp add: SymSystem-def)*

by *(auto simp add: LIVE-def lft-rel-st-def always-def at-fun-def)*

qed

definition $preca-ex\ x = (x\ 1 = (\neg x\ 0))$

lemma *monotonic-SymSystem[simp]*: *mono (SymSystem init p r)*

by *(simp add: SymSystem-def)*

lemma *event-ex-aux-a*: $a\ 0 = (0::int) \implies \forall n. xa\ (Suc\ n) = (\neg xa\ n) \implies$

$\forall n. (xa\ n \wedge a\ (Suc\ n) = a\ n + 1 \vee \neg xa\ n \wedge a\ (Suc\ n) = a\ n - 1 \vee a\ (Suc\ n) = 0) \implies$

$(a\ n = -1 \longrightarrow xa\ n) \wedge (a\ n = 1 \longrightarrow \neg xa\ n) \wedge -1 \leq a\ n \wedge a\ n \leq 1$

proof *(induction n)*

case 0

show *?case*

by *(metis 0.prem1 le-minus-one-simps(1) minus-zero zero-le-one zero-neq-neg-one)*

next

case $(Suc\ n)$

{assume $a\ (Suc\ n) = - (1::int)$ **from this and Suc have** $xa\ (Suc\ n)$

by *(metis add commute add-le-same-cancel2 not-one-le-zero zero-neq-neg-one)}*

note $A = this$

{assume $a\ (Suc\ n) = (1::int)$ **and** $xa\ (Suc\ n)$ **from this and Suc have** *False*

by *(metis eq-iff le-iff-diff-le-0 not-one-le-zero)}*

note $B = this$

{assume $a\ n \neq - (1::int)$ **from this and Suc have** $- (1::int) \leq a\ (Suc\ n)$

by *(metis add commute monoid-add-class.add.left-neutral le-less not-le right-minus uminus-add-conv-diff zle-add1-eq-le)}*

note $C = \text{this}$
{assume $a\ n = - (1::\text{int})$ **from** this **and** Suc **have** $- (1::\text{int}) \leq a\ (\text{Suc}\ n)$
by $(\text{metis}\ \text{add.commute}\ \text{le-minus-one-simps}(4)\ \text{monoid-add-class.add.right-neutral}\ \text{not-le}\ \text{right-minus}\ \text{zle-add1-eq-le})$
note $D = \text{this}$
from C **and** D **and** Suc **have** $E: - (1::\text{int}) \leq a\ (\text{Suc}\ n)$ **by** auto
from Suc **have** $F: a\ (\text{Suc}\ n) \leq (1::\text{int})$
by $(\text{metis}\ \text{eq-iff}\ \text{int-one-le-iff-zero-less}\ \text{le-iff-diff-le-0}\ \text{le-less}\ \text{not-le}\ \text{zle-add1-eq-le})$
from $A\ B\ E\ F$ **show** $?case$ **by** auto
qed

lemma $\text{event-ex-aux}: a\ 0 = (0::\text{int}) \implies \forall n. xa\ (\text{Suc}\ n) = (\neg xa\ n) \implies$
 $\forall n. (xa\ n \wedge a\ (\text{Suc}\ n) = a\ n + 1 \vee \neg xa\ n \wedge a\ (\text{Suc}\ n) = a\ n - 1 \vee a$
 $(\text{Suc}\ n) = 0) \implies$
 $(\forall n. (a\ n = -1 \longrightarrow xa\ n) \wedge (a\ n = 1 \longrightarrow \neg xa\ n) \wedge -1 \leq a\ n \wedge a\ n \leq$
 $1)$
by $(\text{clarify},\ \text{drule}\ \text{event-ex-aux-a},\ \text{auto})$

lemma $\{\square\ \text{preca-ex}\} \circ \text{LIVE} \leq \text{SymSystem}\ \{u . u\ 0 = 0\}\ \text{prec-ex}\ \text{rel-ex}$
proof $(\text{unfold}\ \text{LIVE-fusion}\ \text{SymSystem-def},\ \text{rule}\ \text{fusion-local-refinement},\ \text{simp-all})$
fix $z::\text{nat} \Rightarrow \text{bool}$ **and** $u :: \text{nat} \Rightarrow \text{int}$ **and** $x::\text{nat} \Rightarrow \text{bool}$
assume $A: u\ 0 = 0$
assume $(\square\ \text{preca-ex})\ z$
then have $B: \forall x::\text{nat}. z\ (\text{Suc}\ x) = (\neg z\ x)$
by $(\text{auto}\ \text{simp}\ \text{add:}\ \text{preca-ex-def}\ \text{lft-rel-st-def}\ \text{rel-ex-def}\ \text{always-def}\ \text{at-fun-def})$
assume $(\square\ \text{lft-rel-st}\ \text{rel-ex})\ u\ z\ x$
then have $C: \forall xa . (z\ xa \wedge u\ (\text{Suc}\ xa) = u\ xa + 1 \vee \neg z\ xa \wedge u\ (\text{Suc}\ xa)$
 $= u\ xa - 1 \vee u\ (\text{Suc}\ xa) = 0) \wedge x\ xa = (u\ (\text{Suc}\ xa) = 0)$
by $(\text{auto}\ \text{simp}\ \text{add:}\ \text{preca-ex-def}\ \text{lft-rel-st-def}\ \text{rel-ex-def}\ \text{always-def}\ \text{at-fun-def})$
have $D: (\forall n . (u\ n = -1 \longrightarrow z\ n) \wedge (u\ n = 1 \longrightarrow \neg z\ n) \wedge -1 \leq u\ n \wedge$
 $u\ n \leq 1)$
by $(\text{cut-tac}\ A\ B\ C,\ \text{rule}\ \text{event-ex-aux},\ \text{auto})$
{
fix $a::\text{nat}$
{assume $u\ (\text{Suc}\ a) = 0$ **from** $\text{this}\ A\ B\ C$ **have** $\exists b . u\ (\text{Suc}\ (a + b)) = 0$
by $(\text{metis}\ \text{monoid-add-class.add.right-neutral})$
note $1 = \text{this}$
{assume $u\ (\text{Suc}\ a) = -1$ **from** $\text{this}\ A\ B\ C\ D$ **have** $\exists b . u\ (\text{Suc}\ (a + b))$
 $= 0$
by $(\text{metis}\ \text{add-Suc-right}\ \text{diff-minus-eq-add}\ \text{diff-self}\ \text{monoid-add-class.add.right-neutral})$
note $2 = \text{this}$
{assume $u\ (\text{Suc}\ a) = 1$ **from** $\text{this}\ A\ B\ C\ D$ **have** $\exists b . u\ (\text{Suc}\ (a + b))$
 $= 0$
by $(\text{metis}\ \text{add-Suc-right}\ \text{diff-self}\ \text{monoid-add-class.add.right-neutral})$
note $3 = \text{this}$
from $1\ 2\ 3\ A\ B\ C\ D$ **have** $\exists b . x\ (a + b)$
by $(\text{simp},\ \text{metis}\ \text{diff-0}\ \text{int-one-le-iff-zero-less}\ \text{le-less}\ \text{not-le}\ \text{zle-diff1-eq})$
}
then show $(\square\ \diamond\ (\lambda y . y\ 0))\ x$

```
by (simp add: always-def eventually-def prece-ex-def at-fun-def rel-ex-def
lft-rel-st-def)
qed
end
```

References

- [1] R.-J. Back. *On the correctness of refinement in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978.
- [2] R.-J. Back and M. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 128–158. Springer Berlin Heidelberg, 1995.
- [3] R.-J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.
- [4] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. 1985.
- [5] V. Preoteasa and R.-J. Back. Semantics and data refinement of invariant based programs. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. <http://isa-afp.org/entries/DataRefinementIBP.shtml>, May 2010. Formal proof development.
- [6] V. Preoteasa and S. Tripakis. Refinement calculus of reactive systems. Jun 2014. <http://arxiv.org/abs/1406.6035>.
- [7] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A theory of synchronous relational interfaces. *ACM Trans. Program. Lang. Syst.*, 33(4):14:1–14:41, July 2011.