Formalization of Refinement Calculus for Reactive Systems

Viorel Preoteasa
Aalto University, Finland

September 9, 2026

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 productes 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 locic. 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 $\alpha = \Box \diamondsuit (x = 1)$ 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 $\diamondsuit$ 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 instatiations 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 $'a :: temporal$ to $(nat \Rightarrow 'b) \Rightarrow 'a$. A possible extension of $\Box$ could be

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

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

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

which is not correct.

To evercome this problem we introduce as a primitive operation $!! : 'a \Rightarrow nat \Rightarrow 'a$, 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

$$\Box\alpha = \bigwedge_{i:nat} \alpha!!i$$

notation
bot (‹$\bot$›) **and**
top (‹$\top$›) **and**
inf (**infixl** ‹$\sqcap$› *70*)
and *sup* (**infixl** ‹$\sqcup$› *65*)

class *temporal* = *complete-boolean-algebra* +
fixes *at* :: $'a \Rightarrow nat \Rightarrow 'a$ (**infixl** ‹!!› *150*)
assumes [*simp*]: $a\ !!\ i\ !!\ j = a\ !!\ (i + j)$
assumes [*simp*]: $a\ !!\ 0 = a$
assumes [*simp*]: $\top\ !!\ i = \top$
assumes [*simp*]: $-(a\ !!\ i) = (-a)\ !!\ i$
assumes [*simp*]: $(a \sqcap b)\ !!\ i = (a\ !!\ i) \sqcap (b\ !!\ i)$
begin
definition *always* :: $'a \Rightarrow 'a$ (‹$\Box$ (-)› [*900*] *900*) **where**
$\Box\ p = (INF\ i\ .\ p\ !!\ i)$

definition *eventually* :: $'a \Rightarrow 'a$ (‹$\Diamond$ (-)› [*900*] *900*) **where**
$\Diamond\ p = (SUP\ i\ .\ p\ !!\ i)$

definition *next* :: $'a \Rightarrow 'a$ (‹$\bigcirc$ (-)› [*900*] *900*) **where**
$\bigcirc\ p = p\ !!\ (Suc\ 0)$

definition *until* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infix** ‹*until*› *65*) **where**
$(p\ until\ q) = (SUP\ n\ .\ (Inf\ (at\ p\ `\ \{i\ .\ i < n\})) \sqcap (q\ !!\ n))$
end

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

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

simp
        qed
      qed
    also have (*INF* $i{\in}\{i.\ i \leq n\}.\ p\ i) \leq p\ n$ **by** (*rule INF-lower*, *auto*)
    finally show (*INF* $n.\ (SUP\ i \in \{i.\ i < n\}\ .\ -\ p\ i) \sqcup ((p :: nat \Rightarrow {'a})\ n)) \leq p\ n$ **by** *simp*
    qed

end

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

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

  lemma *always-less* [*simp*]: $\Box\ p \leq p$
    proof −
      have $\Box\ p \leq p\ !!\ 0$
        by (*unfold always-def*, *rule INF-lower*, *simp*)
      also have $p\ !!\ 0 = p$ **by** *simp*
      finally show $\Box\ p \leq p$ **by** *simp*
    qed

  lemma *always-and*: $\Box\ (p \sqcap q) = (\Box\ p) \sqcap (\Box\ q)$
    by (*simp add*: *always-def INF-inf-distrib*)

  lemma *eventually-or*: $\Diamond\ (p \sqcup q) = (\Diamond\ p) \sqcup (\Diamond\ q)$
    by (*simp add*: *eventually-def SUP-sup-distrib*)

  lemma *neg-until-always*: $-(p\ until\ -p) = \Box\ p$
    proof (*rule order.antisym*)
      show $-\ (p\ until\ -\ p) \leq \Box\ p$
        by (*simp add*: *until-def always-def uminus-SUP uminus-INF*, *rule INF-greatest*, *cut-tac* $p = \lambda\ n\ .\ p\ !!\ n$ **in** *until-always*, *simp*)
    next
      have $\bigwedge\ n\ .\ \Box\ p \leq p\ !!\ n$
        by (*simp add*: *always-def INF-lower*)
      also have $\bigwedge\ n\ .\ p\ !!\ n \leq (SUP\ x{\in}\{i.\ i < n\}.\ (-\ p)\ !!\ x) \sqcup p\ !!\ n$
        by *simp*
      finally show $\Box\ p \leq -(p\ until\ -p)$
        apply (*simp add*: *until-def uminus-SUP uminus-INF*)
        by (*rule INF-greatest*, *simp*)
    qed

  lemma *neg-always-eventually*: $\Box\ p = -\ \Diamond\ (-\ p)$
    by (*simp add*: *fun-eq-iff always-def eventually-def until-def uminus-SUP*)

  lemma *neg-true-until-always*: $-(\top\ until\ -p) = \Box\ p$

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

lemma *true-until-eventually*: $(\top \ until \ p) = \Diamond \ p$
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*::*bool*) !! $i = p$
instance proof
qed *auto*
end

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

Asumming 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* $\Rightarrow$ $'a$ ($\langle$-[- ..]$\rangle$ [*80*, *15*] *80*)
assumes [*simp*]: $a[i..][j..] = a[i + j..]$
assumes [*simp*]: $a[0..] = a$
begin
definition *next-trace* :: $'a \Rightarrow$ $'a$ ($\langle\odot$ (-)$\rangle$ [*900*] *900*) **where**
$\odot \ p = p[Suc \ 0..]$
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 instatiated 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 instatiations 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 always is available for them as a consequence of the above construction.

```
lemma (□ r) OO (□ r') ≤ (□ (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 (‹⊥›) **and**
top (‹⊤›) **and**
inf (**infixl** ‹⊓› *70*)
and *sup* (**infixl** ‹⊔› *65*)

3.1 Basic predicate transformers

definition
demonic :: (*'a* => *'b*::*lattice*) => *'b* => *'a* ⇒ *bool* (‹[: - :]› [*0*] *1000*) **where**
[:*Q*:] *p s* = (*Q s* ≤ *p*)

definition
assert::*'a*::*semilattice-inf* => *'a* => *'a* (‹{. - .}› [*0*] *1000*) **where**
{.*p*.} *q* ≡ *p* ⊓ *q*

definition
assume::(*'a*::*boolean-algebra*) => *'a* => *'a* (‹[. - .]› [*0*] *1000*) **where**
[.*p*.] *q* ≡ (−*p* ⊔ *q*)

definition
angelic :: (*'a* ⇒ *'b*::{*semilattice-inf*,*order-bot*}) ⇒ *'b* ⇒ *'a* ⇒ *bool* (‹{: - :}› [*0*] *1000*) **where**
{:*Q*:} *p s* = (*Q s* ⊓ *p* ≠ ⊥)

syntax
-assert :: *patterns* => *logic* => *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* => *patterns* => *logic* => *logic* (‹- ⇝ - . -› *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* ⇝ *z* . (*P*::*'a* ⇒ *'b* ⇒ *'c* ⇒ *'d* ⇒ *bool*) *x y y' z* :]

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

lemma *demonic-demonic*: [:r:] o [:r':] = [:r OO r':]
by (*simp add*: *fun-eq-iff le-fun-def demonic-def*, *auto*)

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

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

lemma *assert-assert-comp*: {.$p{::}'a{::}lattice$.} o {.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 = {. *inpt* r.} o [:r:]

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

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

lemma *trs-trs*: (*trs* r) o (*trs* r') = *trs* ((λ s t. ($\forall$ s' . r s s' $\longrightarrow$ (*inpt* r' s'))) $\sqcap$ (r OO r')) (**is** *?S* = *?T*)
proof −
have [*simp*]: (λx. *inpt* r x $\wedge$ ($\forall$ y. r x y $\longrightarrow$ *inpt* r' y)) = *inpt* ((λs t. $\forall$ s'. r s s' $\longrightarrow$ *inpt* r' s') $\sqcap$ r OO r')
by (*auto simp add*: *fun-eq-iff inpt-def relcompp-exists*, *blast*)
have [*simp*]: (λx y. *inpt* r x $\wedge$ ($\forall$ y. r x y $\longrightarrow$ *inpt* r' y)) $\sqcap$ r OO r' = (λs t. $\forall$ s'. r s s' $\longrightarrow$ *inpt* r' s') $\sqcap$ r OO r'
by (*auto simp add*: *fun-eq-iff inpt-def*)
have *?S* = {. *inpt* r .} $\circ$ [: r :] $\circ$ {. *inpt* r' .} $\circ$ [: r' :]
by (*simp add*: *trs-def comp-assoc*[*symmetric*])
also have ... = {. λx. *inpt* r x $\wedge$ ($\forall$ y. r x y $\longrightarrow$ *inpt* r' y) .} $\circ$ [: r OO r' :]
by (*simp add*: *assert-demonic-comp*)
also have ... = {. λx. *inpt* r x $\wedge$ ($\forall$ y. r x y $\longrightarrow$ *inpt* r' y) .} o [:(λ x y . *inpt* r x $\wedge$ ($\forall$ y. r x y $\longrightarrow$ *inpt* r' y)) $\sqcap$ (r OO r'):]
by (*subst assert-demonic-prop*, *simp*)
also have ... = *?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 $trs\ (\lambda\ x\ y\ .\ x \geq 0)\ q\ x = (if\ q = \top\ then\ x \geq 0\ else\ False)$
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 $trs\ r \sqcap trs\ r' = \{.\ inpt\ r \sqcap inpt\ r'\ .\} \circ [:\ r \sqcup r'\ :]$
by (*simp add*: *trs-def spec-demonic-choice*)
also have $... = \{.\ inpt\ r \sqcap inpt\ r'\ .\} \circ [:\ (\lambda\ x\ y\ .\ inpt\ r\ x \wedge inpt\ r'\ x) \sqcap (r \sqcup r')\ :]$
by (*subst assert-demonic-prop*, *simp*)
also have $... = trs\ ((\lambda\ x\ y\ .\ inpt\ r\ x \wedge inpt\ r'\ x) \sqcap (r \sqcup r'))$
by (*simp add*: *trs-def*)
finally show *?thesis* **by** *simp*
qed

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

3.2 Conjunctive predicate transformers

definition $conjunctive\ (S::'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice) = (\forall\ Q\ .\ S\ (Inf\ Q) = Inf\ (S\ `\ Q))$
definition $sconjunctive\ (S::'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice) = (\forall\ Q\ .\ (\exists\ x\ .\ x \in Q) \longrightarrow S\ (Inf\ Q) = Inf\ (S\ `\ Q))$

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

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

lemma *conjuncive-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 $r\ x \leq Inf\ Q$
also from *A* **have** $Inf\ Q \leq y$
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 $r\ x \leq Inf\ Q$
by (*rule Inf-greatest*, *simp add*: *A*)
qed

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

```
      also have ... ≤ x
        by simp
      finally show (INF x∈Q. p ⊓ x) ≤ x
        by (simp only: image-def)
    qed
  qed

  lemma sconjunctive-simp: x ∈ Q ⟹ sconjunctive S ⟹ S (Inf Q) = Inf (S ‘ Q)
    by (auto simp add: sconjunctive-def)

  lemma sconjunctive-INF-simp: x ∈ X ⟹ sconjunctive S ⟹ S (Inf (Q ‘ X)) = Inf (S ‘ (Q ‘ X))
    by (cut-tac x = Q x and Q = Q ‘ X in sconjunctive-simp, auto)

  lemma demonic-comp [simp]: sconjunctive S ⟹ sconjunctive S' ⟹ sconjunctive (S o S')
    proof (subst sconjunctive-def, safe)
      fix X :: 'c set
      fix a :: 'c
      assume [simp]: sconjunctive S
      assume [simp]: sconjunctive S'
      assume [simp]: a ∈ X
      have A: S' (Inf X) = Inf (S' ‘ X)
        by (rule-tac x = a in sconjunctive-simp, auto)
      also have B: S (Inf (S' ‘ X)) = Inf (S ‘ (S' ‘ X))
        by (rule-tac x = S' a in sconjunctive-simp, auto)
     finally show (S o S') (Inf X) = Inf ((S ∘ S') ‘ X) by (simp add: image-comp)
    qed

  lemma [simp]:conjunctive S ⟹ S (Inf (Q ‘ X)) = (Inf ((S o Q) ‘ X))
    by (metis INF-image conjunctive-def)

  lemma conjunctive-simp: conjunctive S ⟹  S (Inf Q) = Inf (S ‘ Q)
    by (metis conjunctive-def)

  lemma conjunctive-monotonic: sconjunctive S ⟹ mono S
    proof (rule monoI)
      fix a b :: 'a
      assume [simp]: a ≤ b
      assume [simp]: sconjunctive S
      have [simp]: a ⊓ b = a
        by (rule antisym, auto)
      have A: S a = S a ⊓ S b
        by (cut-tac S = S and x = a and Q = {a, b} in sconjunctive-simp, auto)
      show S a ≤ S b
        by (subst A, simp)
    qed
```

definition *grd* $S = -\ S\ \bot$

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* $= \bot$

lemma *mono* $(S::'a::boolean\text{-}algebra \Rightarrow 'b::boolean\text{-}algebra) \Longrightarrow (S = Fail) = (fail\ S = \top)$
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* (*opaque-lifting*, *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*]: $\{.\top::'a::bounded\text{-}lattice.\} =$ *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*]: *mono* [:*r*::(*'a* ⇒ *'b* ⇒ *bool*):]
by (*simp add*: *conjunctive-monotonic*)

lemma [*simp*]:{. λ *x* . *True* .} = *Skip*
by (*simp add*: *fun-eq-iff assert-def Skip-def*)

lemma [*simp*]: ⊥ *o* *S* = ⊥
by (*simp add*: *fun-eq-iff*)

lemma [*simp*]: {.⊥::*'a*::*boolean-algebra*.} = ⊥
by (*simp add*: *fun-eq-iff assert-def*)

lemma [*simp*]: ⊤ *o* *S* = ⊤
by (*simp add*: *fun-eq-iff*)

lemma *left-comp*: *T o U* = *T'* *o* *U'* ⟹ *S o T o U* = *S o T'* *o* *U'*
by (*simp add*: *comp-assoc*)

lemma *assert-demonic*: {.*p*.} *o* [:*r*:] = {.*p*.} *o* [:λ *x y* . *p x* ∧ *r x y*:]
by (*auto simp add*: *fun-eq-iff assert-def demonic-def le-fun-def*)

lemma *trs r* ⊓ *trs r'* = *trs* (λ *x y* . *inpt r x* ∧ *inpt r' x* ∧ (*r x y* ∨ *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* ⇒ *bool*) ⇒ (*'b* ⇒ *bool*)) ⇒ ((*'a* ⇒ *bool*) ⇒ (*'b* ⇒ *bool*)) ⇒ ((*'a* ⇒ *bool*) ⇒ (*'b* ⇒ *bool*)) (**infixl** ‹‖› *70*) **where**
(*S* ‖ *S'*) *q x* = (∃ (*p*::*'a*⇒*bool*) *p'* . *p* ⊓ *p'* ≤ *q* ∧ *S p x* ∧ *S' p' x*)

lemma *fusion-spec*: ({.*p*.} ∘ [:*r*:]) ‖ ({.*p'*.} ∘ [:*r'*:]) = ({.*p* ⊓ *p'*.} ∘ [:*r* ⊓ *r'*:])
by (*auto simp add*: *fun-eq-iff fusion-def assert-def demonic-def le-fun-def*)

lemma *fusion-assoc*: *S* ‖ (*T* ‖ *U*) = (*S* ‖ *T*) ‖ *U*
proof (*rule antisym*, *auto simp add*: *fusion-def*)
fix *p p' q s s'* :: *'a* ⇒ *bool*
fix *a*
assume *A*: *p* ⊓ *p'* ≤ *q* **and** *B*: *s* ⊓ *s'* ≤ *p'*
assume *C*: *S p a* **and** *D*: *T s a* **and** *E*: *U s' a*
from *A* **and** *B* **have** *F*: (*p* ⊓ *s*) ⊓ *s'* ≤ *q*
by (*simp add*: *le-fun-def*)
have (∃ *v v'*. *v* ⊓ *v'* ≤ (*p* ⊓ *s*) ∧ *S v a* ∧ *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 $p\ p'\ q\ s\ s' :: 'a \Rightarrow bool$
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*: $a \in init \Longrightarrow ([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:r:]) \parallel (\{.p'.\} \circ [:r':])$
$= [:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.u,x\ .\ p\ (u,\ x) \wedge p'\ x.\} \circ [:u,\ x \rightsquigarrow y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ y:]$ (**is** $?p \Longrightarrow ?S = ?T$)
proof −
assume $?p$
from *this* **have** [*simp*]: $(\lambda x.\ \forall\, a.\ a \in init \longrightarrow p\ (a,\ x) \wedge p'\ x) = (\lambda x.\ \forall\, a.\ a \in init \longrightarrow p\ (a,\ x)) \sqcap p'$
by *auto*
have [*simp*]: $(\lambda x\ (u,\ y).\ u \in init \wedge x = y)\ OO\ (\lambda(u,\ x)\ y.\ r\ (u,\ x)\ y \wedge r'\ x\ y) = (\lambda x\ (u,\ y).\ u \in init \wedge x = y)\ OO\ r \sqcap r'$
by *auto*
have $?S =$
$(\{.\ \lambda x.\ \forall\, a.\ a \in init \longrightarrow p\ (a,\ x)\ .\} \circ [:\ \lambda x\ (u,\ y).\ u \in init \wedge x = y\ :] \circ [:\ r\ :]) \parallel (\{.\ p'\ .\} \circ [:\ r'\ :])$
by (*simp add*: *demonic-assert-comp*)
also have ... = $\{.\ (\lambda x.\ \forall\, a.\ a \in init \longrightarrow p\ (a,\ x)) \sqcap p'\ .\} \circ [:\ (\lambda x\ (u,\ y).\ u \in init \wedge x = y)\ OO\ r \sqcap r'\ :]$
by (*simp add*: *comp-assoc demonic-demonic fusion-spec*)
also have ... = $?T$
by (*simp add*: *demonic-assert-comp comp-assoc demonic-demonic fusion-spec*)
finally show *?thesis* **by** *simp*
qed

lemma *fusion-spec-local-a*: $a \in init \Longrightarrow ([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:r:]) \parallel [:r':]$
$= ([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:u,\ x \rightsquigarrow y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ y:])$
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*:
$a \in init \Longrightarrow (\bigwedge x\ u\ y\ .\ u \in init \Longrightarrow p'\ x \Longrightarrow r\ (u, x)\ y \Longrightarrow r'\ x\ y) \Longrightarrow$
$\{.p'.\}\ o\ (([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:r:]) \parallel [:r':]) \leq [:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:r:]$
proof −
assume *A*: $a \in init$
assume [*simp*]: $(\bigwedge x\ u\ y\ .\ u \in init \Longrightarrow p'\ x \Longrightarrow r\ (u, x)\ y \Longrightarrow r'\ x\ y)$
have $\{.\ x.\ p'\ x \wedge (\forall\, a.\ a \in init \longrightarrow p\ (a, x))\ .\} \circ [:\ (\lambda x\ (u, y).\ u \in init \wedge x = y)\ OO\ (\lambda(u, x)\ y.\ r\ (u, x)\ y \wedge r'\ x\ y)\ :]$
$\leq \{.\ \lambda x.\ \forall\, a.\ a \in init \longrightarrow p\ (a, x)\ .\} \circ [:\ (\lambda x\ (u, y).\ u \in init \wedge x = y)\ OO\ r\ :]$
by (*auto simp add*: *assert-demonic-refinement*)
from *this* **have** $\{.\ x.\ p'\ x \wedge (\forall\, a.\ a \in init \longrightarrow p\ (a, x))\ .\} \circ [:\ (\lambda x\ (u, y).\ u \in init \wedge x = y)\ OO\ (\lambda(u, x)\ y.\ r\ (u, x)\ y \wedge r'\ x\ y)\ :]$
$\leq \{.\ \lambda x.\ \forall\, a.\ a \in init \longrightarrow p\ (a, x)\ .\} \circ [:\ \lambda x\ (u, y).\ u \in init \wedge x = y\ :] \circ [:\ r\ :]$
by (*simp add*: *comp-assoc demonic-demonic*)
from *this* **have** $\{.\ p'\ .\} \circ [:\ \lambda x\ (u, y).\ u \in init \wedge x = y\ :] \circ \{.\ p\ .\} \circ [:\ \lambda(u, x)\ y.\ r\ (u, x)\ y \wedge r'\ x\ y\ :]$
$\leq [:\ x \rightsquigarrow u,\ y.\ u \in init \wedge x = y\ :] \circ \{.\ p\ .\} \circ [:\ r\ :]$
by (*simp add*: *demonic-assert-comp assert-demonic-comp*)
from *this* **have** $\{.\ p'\ .\} \circ ([:\ x \rightsquigarrow (u, y)\ .\ u \in init \wedge x = y\ :] \circ \{.\ p\ .\} \circ [:\ (u, x) \rightsquigarrow y\ .\ r\ (u, x)\ y \wedge r'\ x\ y\ :])$
$\leq [:\ x \rightsquigarrow (u, y)\ .\ u \in init \wedge x = y\ :] \circ \{.\ p\ .\} \circ [:\ r\ :]$
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* = $[:x \rightsquigarrow y\ .\ True:]$
definition *ASSERT-LIVE* = $\{.\ \Box\ \Diamond\ (\lambda\ x\ .\ x\ 0).\}$
definition *GUARANTY-LIVE* = $[:x \rightsquigarrow y\ .\ \Box\ \Diamond\ (\lambda\ y\ .\ y\ 0):]$
definition *AE* = *ASSERT-LIVE o 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* = [: □(λ *xs ys* . *xs* (*0*::*nat*) ⟶ (◇ (λ *ys* . *ys* (*0*::*nat*))) *ys*) :]
definition *FAIL* = ⊥

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* ⇒ *bool*
fix *xa* :: *nat* ⇒ *bool*
fix *xb* :: *nat*
assume ∀ *xb*::*nat* ⇒ *bool* . (∀ *x*. *xa x* ⟶ *Ex* (*xb*[*x* ..])) ⟶ (∀ *x*. ∃ *a*. *xb* (*x* + *a*)) ∧ *All x*
then have (∀ *x*. *xa x* ⟶ *Ex* (*xa*[*x* ..])) ⟶ (∀ *x*. ∃ *a*. *xa* (*x* + *a*)) ∧ *All x*
by *auto*
then show ∃ *x*. *xa* (*xb* + *x*)
by (*auto*, *rule-tac x* = *0* **in** *exI*, *simp*)
next
fix *x* :: *'a* ⇒ *bool*
fix *xa* :: *nat* ⇒ *bool*
fix *xb* :: *'a*
assume ∀ *xb*::*nat* ⇒ *bool* . (∀ *x*. *xa x* ⟶ *Ex* (*xb*[*x* ..])) ⟶ (∀ *x*. ∃ *a*. *xb* (*x* + *a*)) ∧ *All x*
from *this* **show** *x xb*
by (*metis at-trace-def le0*)
next
fix *x* :: *'a* ⇒ *bool* **and** *xa* :: *nat* ⇒ *bool* **and** *xb* :: *nat* ⇒ *bool* **and** *xc* :: *nat*
assume *A*: ∀ *x*. *xa x* ⟶ *Ex* (*xb*[*x* ..])
assume *B*: ∀ *x*. ∃ *xb*. *xa* (*x* + *xb*)
have ∃ *x1*. *xc* ≤ *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 statisfy 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* = (∃ *n u y* . *u* ∈ *init* ∧ (∀ *i* < *n* . *r* (*u i*) (*u* (*Suc i*)) (*x i*) (*y i*)) ∧ (¬ *p* (*u n*) (*u* (*Suc n*)) (*x n*)))
definition *run r u x y* = (∀ *i* . *r* (*u i*) (*u* (*Suc i*)) (*x i*) (*y i*))
definition *LocalSystem init p r q x* = (¬ *fail-sys init p r x* ∧ (∀ *u y* . (*u* ∈ *init* ∧ *run r u x y*) ⟶ *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* = (∃ *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* = −((*lft-pred-st* (*inpt-st r*)) *until* −(*lft-pred-st p*))

lemma *prec-st-simp*: *prec-st p r u x* = (∀ *n* . (∀ *i* < *n* . *inpt-st r* (*u i*) (*u* (*Suc i*)) (*x i*)) ⟶ *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* ⇝ *u*, *x* . *u* ∈ *init* ∧ *z* = *x*:] *o* {.*u*, *x* . *prec-st p r u x*.} *o*
[:*u*, *x* ⇝ *y* . (□ (*lft-rel-st r*)) *u x y* :]

lemma *SymSystem-rel*: *SymSystem init p r* = {. *x* . ∀ *u*. *u* ∈ *init* ⟶ *prec-st p r u x* .} ∘

[: $x \rightsquigarrow y$. $\exists$ u . $u \in init \wedge (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y$:]
proof −
have [*simp*]: $((\lambda z\ (u,\ x).\ u \in init \wedge z = x)\ OO\ (\lambda(x,\ y).\ (\Box\ lft\text{-}rel\text{-}st\ r)\ x\ y)) = (\lambda x\ y.\ \exists u.\ u \in init \wedge (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y)$
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 = Inf* (*S* ‘ *init*)

definition *zip-set A B* = $\{u\ .\ ((fst\ o\ u) \in A) \wedge ((snd\ o\ u) \in B)\}$
definition *nzip*:: $('x \Rightarrow 'a) \Rightarrow ('x \Rightarrow 'b) \Rightarrow 'x \Rightarrow ('a \times 'b)$ (**infixl** ‹||› *65*) **where** $(xs\ ||\ ys)\ i = (xs\ i,\ ys\ i)$

lemma [*simp*]: $fst \circ x\ ||\ y = x$
by (*simp add*: *fun-eq-iff nzip-def*)

lemma [*simp*]: $snd \circ x\ ||\ y = y$
by (*simp add*: *fun-eq-iff nzip-def*)

lemma [*simp*]: $x \in A \Longrightarrow y \in B \Longrightarrow (x\ ||\ y) \in zip\text{-}set\ A\ B$
by (*simp add*: *zip-set-def*)

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

lemma *local-init-comp*: $u' \in init' \Longrightarrow (\forall\ u.\ sconjunctive\ (S\ u)) \Longrightarrow$ (*local-init init S*) o (*local-init init′ S′*)
$= local\text{-}init\ (zip\text{-}set\ init\ init')\ (\lambda\ u\ .\ (S\ (fst\ o\ u))\ o\ (S'\ (snd\ o$

```
u)))
      proof (subst fun-eq-iff, auto)
        fix x :: 'f
        assume A: u' ∈ init'
        assume ∀ u . sconjunctive (S u)
        from this have [simp]: ⋀ u . sconjunctive (S u) by simp
        from A have [simp]: ⋀ y . S y (INF y' ∈ init'. S' y' x) =  (INF y' ∈ init'
. S y (S' y' x))
          by (simp add: sconjunctive-INF-simp image-comp)

        have [simp]: (INF y ∈ init . (INF y' ∈ init' . S y (S' y' x))) ≤ (INF u ∈
zip-set init init'. S (fst ∘ u) (S' (snd ∘ u) x))
          proof (rule INF-greatest, auto simp add: zip-set-def)
            fix u :: 'a ⇒ 'c × 'b
            assume [simp]: fst ∘ u ∈ init
            assume [simp]: snd ∘ u ∈ init'
            have (INF y ∈ init. INF y' ∈ init'. S y (S' y' x)) ≤ (INF y' ∈ init'. S
(fst o u) (S' y' x))
              by (rule INF-lower, simp)
            also have ... ≤ S (fst o u) (S' (snd o u) x)
              by (rule INF-lower, simp)
            finally show (INF y ∈ init. INF y' ∈ init'. S y (S' y' x)) ≤ S (fst o u)
(S' (snd o u) x)
              by simp
          qed
        have [simp]: (INF u ∈ zip-set init init'. S (fst ∘ u) (S' (snd ∘ u) x)) ≤ (INF
y ∈ init . (INF y' ∈ init' . S y (S' y' x)))
          proof (rule INF-greatest, rule INF-greatest)
            fix y :: 'a ⇒ 'c and y' :: 'a ⇒ 'b
            assume [simp]: y ∈ init
            assume [simp]: y' ∈ init'
            have (INF u ∈ zip-set init init'. S (fst ∘ u) (S' (snd ∘ u) x)) ≤ S (fst o
(y || y')) (S' (snd o (y || y')) x)
              by (rule INF-lower, simp)
            also have ... ≤  S y (S' y' x)
              by simp
            finally show (INF u :: 'a ⇒ 'c × 'b ∈ zip-set init init'. S (fst ∘ u) (S'
(snd ∘ u) x)) ≤ S y (S' y' x)
              by simp
          qed
        have local-init init S (local-init init' S' x) = (INF y ∈ init. S y (INF y' ∈
init'. S' y' x))
          by (simp add: local-init-def image-comp)
        also have ... = (INF y ∈ init . (INF y' ∈ init' . S y (S' y' x)))
          by simp
        also have ... = (INF u ∈ zip-set init init'. S (fst ∘ u) ∘ S' (snd ∘ u)) x
          by (rule antisym) (simp-all add: image-comp)
        also have ... = local-init (zip-set init init') (λ u . (S (fst o u)) o (S' (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′*) (λ*u*::′*a* ⇒ ′*c* × ′*b*. *S* (*fst* ∘ *u*) ∘ *S′* (*snd* ∘ *u*)) *x*
by *simp*
qed

lemma *init-state*: [:*z* ⇝ *u*, *x* . *u* ∈ *init* ∧ *z* = *x*:] *o* {.*u*, *x* . *p u x*.} *o* [:*u*, *x* ⇝ *y* . *r u x y* :]
= [:*z* ⇝ *u*, *x* . *z* = *x*:] *o* {.*u*, *x* . *u* ∈ *init* ⟶ *p u x*.} *o* [:*u*, *x* ⇝ *y* . *u* ∈ *init* ∧ *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*: (□ *lft-rel-st r*) (*fst* ∘ *u*) *OO* (□ *lft-rel-st r′*) (*snd* ∘ *u*)
= (□ *lft-rel-st* (λ (*u*, *v*) (*u′*, *v′*) . ((*r u u′*) *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*::*nat* ⇒′*a* **and**
y::*nat* ⇒ ′*b* **and**
v::*nat* ⇒ ′*c* **and**
n:: *nat*
assume ∀ *i* . *r* (*fst* (*u i*)) (*fst* (*u* (*Suc i*))) (*x i*) (*v i*)
and ∀ *i* . *r′* (*snd* (*u i*)) (*snd* (*u* (*Suc i*))) (*v i*) (*y i*)
from *this* **show** (*case u n of* (*u*, *v*) ⇒ λ(*u′*, *v′*). *r u u′ 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*::*nat* ⇒′*a* **and**
z::*nat* ⇒ ′*b*
define *a* **where** *a n* = (*SOME y* . *r* (*fst* (*u n*)) (*fst* (*u* (*Suc n*))) (*x n*) *y* ∧ *r′* (*snd* (*u n*)) (*snd* (*u* (*Suc n*))) *y* (*z n*))
for *n*
assume ∀ *i* . (*case u i of* (*u*, *v*) ⇒ λ(*u′*, *v′*). *r u u′ OO r′ v v′*) (*u* (*Suc i*)) (*x i*) (*z i*)
from *this* **and** *a-def* **have** (∀ *i* :: *nat*. *r* (*fst* (*u i*)) (*fst* (*u* (*Suc i*))) (*x i*) (*a i*)) ∧ (∀ *i* :: *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** ∃ *a* . (∀ *i* . *r* (*fst* (*u i*)) (*fst* (*u* (*Suc i*))) (*x i*) (*a i*)) ∧ (∀ *i* . *r′* (*snd* (*u i*)) (*snd* (*u* (*Suc i*))) (*a i*) (*z i*))
by *blast*
qed

theorem *SymSystem-comp*: *u′* ∈ *init′* ⟹ *SymSystem init p r o SymSystem init′ p′ r′*
= [:*z* ⇝ *u*, *x* . *fst o u* ∈ *init* ∧ *snd o u* ∈ *init′* ∧ *z* = *x*:]
o {. *u*, *x* . *prec-st p r* (*fst* ∘ *u*) *x* ∧ (∀ *y*. (□ *lft-rel-st r*) (*fst* ∘ *u*) *x y* ⟶ *prec-st p′ r′* (*snd* ∘ *u*) *y*) .}

o [: u, $x \rightsquigarrow y$. ($\Box$ *lft-rel-st* ($\lambda(u,\ v)$ (u', v'). r u u' *OO* r' v v')) u x y :]

(**is** *?p* $\Longrightarrow$ *?S* = *?T*)

proof −

assume *A*: *?p*

have *?S* =

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

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

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

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

by (*unfold SymSystem-def*, *simp*)

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

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

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

also from *A* **have** ... = *local-init* (*zip-set init init*$'$)

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

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

also have ... = *local-init* (*zip-set init init*$'$)

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

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

also have ... = *local-init* (*zip-set init init*$'$)

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

[: ($\Box$ *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 ... = *local-init* (*zip-set init init*$'$)

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

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

by *simp*

also have ... = *?T*

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

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

qed

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

= ($\Box$ *lft-rel-st* (λ (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*: $u' \in$ *init*$'$ $\Longrightarrow$ *SymSystem init* p r o *SymSystem init*$'$ p' r'

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

o [: $x \rightsquigarrow y$. $\exists$ u v . $u \in init \wedge v \in init' \wedge (\Box\ lft\text{-}rel\text{-}st\ (\lambda(u,\ v)\ (u',\ v').\ r\ u\ u'\ OO\ r'\ v\ v'))\ (u \parallel v)\ x\ y$:]
(**is** $?p \Longrightarrow ?S = ?T$)
proof −
assume A: $u' \in init'$
from A **have** [*simp*]: $(\lambda x.\ (\forall u.\ u \in init \longrightarrow prec\text{-}st\ p\ r\ u\ x) \wedge (\forall y.\ (\exists u.\ u \in init \wedge (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y) \longrightarrow (\forall u.\ u \in init' \longrightarrow prec\text{-}st\ p'\ r'\ u\ y)))$
$= (\lambda x.\ \forall u\ v.\ u \in init \wedge v \in init' \longrightarrow prec\text{-}st\ p\ r\ u\ x \wedge (\forall y.\ (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y \longrightarrow prec\text{-}st\ p'\ r'\ v\ y))$
by (*auto simp add*: *fun-eq-iff*)
have [*simp*]: $(\lambda x\ y.\ \exists u.\ u \in init \wedge (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y)\ OO\ (\lambda x\ y.\ \exists u.\ u \in init' \wedge (\Box\ lft\text{-}rel\text{-}st\ r')\ u\ x\ y)$
$= (\lambda\ x\ y\ .\ \exists\ u\ v\ .\ u \in init \wedge v \in init' \wedge (((\Box\ lft\text{-}rel\text{-}st\ r)\ u)\ OO\ ((\Box\ lft\text{-}rel\text{-}st\ r')\ v))\ x\ y)$
by (*auto simp add*: *fun-eq-iff*)

from A **have** $?S = \{.x\ .\ \forall u\ .\ u \in init \longrightarrow prec\text{-}st\ p\ r\ u\ x.\} \circ$
[: $x \rightsquigarrow y$. $\exists u{::}nat \Rightarrow 'e.\ u \in init \wedge (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y$:] $\circ$
($\{.x.\ \forall u\ .\ u \in init' \longrightarrow prec\text{-}st\ p'\ r'\ u\ x.\} \circ$ [: $x \rightsquigarrow y$. $\exists u$. $u \in init' \wedge (\Box\ lft\text{-}rel\text{-}st\ r')\ u\ x\ y$:])
by (*simp add*: *SymSystem-rel*)
also have ... $= \{.\ \lambda x\ .\ \forall u\ .\ u \in init \longrightarrow prec\text{-}st\ p\ r\ u\ x\ .\} \circ$ [: $x \rightsquigarrow y$. $\exists u$. $u \in init \wedge (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y$:] $\circ$
$\{.\ x\ .\ \forall u\ .\ u \in init' \longrightarrow prec\text{-}st\ p'\ r'\ u\ x\ .\} \circ$ [: $x \rightsquigarrow y$. $\exists u$. $u \in init' \wedge (\Box\ lft\text{-}rel\text{-}st\ r')\ u\ x\ y$:]
by (*simp add*: *comp-assoc* [*THEN sym*])
also have ... $= ?T$
by (*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 compostion of local transitions of S and S'

definition $initS = \{u\ .\ fst\ (u\ (0{::}nat)) = (0{::}nat)\}$
definition $localPrecS = (\top{::}\ nat \times nat\ \Rightarrow nat \times nat \Rightarrow nat \Rightarrow bool)$
definition $localRelS = (\lambda\ (u{::}nat,\ v)\ (u',\ v'{::}nat)\ (x{::}nat)\ (y{::}nat)\ .\ u = 0 \wedge u' = 1 \wedge v = v')$

definition $initS' = (\top{::}(nat \Rightarrow (nat \times nat))\ set)$
definition $localPrecS' = (\bot{::}\ nat \times nat\ \Rightarrow nat \times nat \Rightarrow nat \Rightarrow bool)$
definition $localRelS' = (\lambda\ (u{::}nat,\ v)\ (u',\ v'{::}nat)\ (x{::}nat)\ (y{::}nat)\ .\ u = u')$

definition $symbS = SymSystem\ initS\ localPrecS\ localRelS$
definition $symbS' = SymSystem\ initS'\ localPrecS'\ localRelS'$

definition $localPrecSS' = (\lambda(u{::}nat,\ v{::}nat)\ (u',\ v')\ (x{::}nat)\ .\ 0 < u)$
definition $localRelSS' = (\lambda\ (u,\ v{::}nat)\ (u'{::}nat,\ v'{::}nat)\ (x{::}nat)\ (z{::}nat)\ .\ (u{::}nat) = 0 \wedge u' = 1)$

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

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

lemma {.x, (u, v) . *localPrecS* (u, v) (a,b) x.} *o* [:x, (u,v) $\rightsquigarrow$ y, (u',v') . *localRelS* (u,v) (u',v') x y:] *o*
{.x, (u, v) . *localPrecS′* (u, v) (c, d) x.} *o* [:x, (u,v) $\rightsquigarrow$ y, (u',v') . *localRelS′* (u,v) (u',v') x y:]
= {.x, (u, v) . *localPrecSS′* (u, v) (e, f) x.} *o* [:x, (u,v) $\rightsquigarrow$ y, (u',v') . *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*]: [: $\bot$::($'a \Rightarrow 'b$ => ($'c$::*boolean-algebra*)) :] = $\top$
by (*simp add*: *fun-eq-iff demonic-def*)

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

lemma *symbSS′-aux*: ($\lambda x.\ \forall u.$ *fst* (u *0*) = *0* $\longrightarrow$
($\forall n.$ ($\forall i{<}n.$ *Ex* ((*case u i of* (u, v) $\Rightarrow \lambda(u',\ v'{::}nat)\ x\ z.\ u = 0 \wedge u' = Suc\ 0$) ($u$ (*Suc i*)) (x i)))
$\longrightarrow$ (*case u n of* (u, v) $\Rightarrow \lambda(u',\ v')\ x.\ 0 < u$) ($u$ (*Suc n*)) (x n))) = $\bot$
apply (*auto simp add*: *fun-eq-iff*)
by (*rule-tac* $x = \lambda\ i\ .\ (i{::}nat,\ i)$ **in** *exI*, *simp*)

lemma *symbSS′*: *symbSS′* = $\bot$
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*]: ($\lambda x.\ \forall u.$ *fst* (u *0*) = *0* $\longrightarrow$ *prec-st* $\top$ (λ (u, v) (u', v') x y . $u = 0 \wedge u' = Suc\ 0 \wedge v = v'$) 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*]: ($\lambda x\ y.\ \exists u.$ *fst* (u *0*) = *0* $\wedge$ ($\Box$ *lft-rel-st* (λ (u, v) (u', v') (x) (y). $u = 0 \wedge u' = Suc\ 0 \wedge v = v'$)) u x y) = $\bot$
proof (*auto simp add*: *fun-eq-iff always-def lft-rel-st-def at-fun-def*)

fix x::$nat \Rightarrow 'a$ **and** $xa :: nat \Rightarrow 'b$ **and** u::$nat \Rightarrow nat \times 'c$
assume A: $\forall\, a\ .\ (case\ u\ a\ of\ (e,\ f) \Rightarrow \lambda(u',\ v')\ x\ y.\ e = 0 \wedge u' = Suc\ 0 \wedge f = v')\ (u\ (Suc\ a))\ (x\ a)\ (xa\ a)$
{fix n:: nat
from A **have** $fst\ (u\ n) = 0 \wedge fst\ (u\ (Suc\ n)) = Suc\ 0$
by ($drule\text{-}tac\ x = n$ **in** $spec$, $case\text{-}tac\ u\ n$, $case\text{-}tac\ u\ (Suc\ n)$, $auto$)
}
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 $\{.\ \lambda x.\ \forall\, u.\ fst\ (u\ 0) = 0 \longrightarrow prec\text{-}st\ \top\ (\lambda(u,\ v)\ (u',\ v')\ x\ y.\ u = 0 \wedge u' = Suc\ 0 \wedge v = v')\ u\ x\ .\} \circ$
$[:\ \lambda\ x\ y\ .\ \exists\, u\ .\ fst\ (u\ 0) = 0 \wedge (\Box\ lft\text{-}rel\text{-}st\ (\lambda(u,\ v)\ (u',\ v')\ x\ y.\ u = 0 \wedge u' = Suc\ 0 \wedge v = v'))\ u\ x\ y\ :] =$
$\top$
by $simp$
qed

lemma $symbS\ o\ symbS' \neq symbSS'$
by ($simp\ add$: $symbSS'\ symbS\ fun\text{-}eq\text{-}iff$)

lemma $prec\text{-}st\text{-}inpt$: $prec\text{-}st\ (inpt\text{-}st\ r)\ r = (\Box\ (lft\text{-}pred\text{-}st\ (inpt\text{-}st\ r)))$
by ($simp\ add$: $prec\text{-}st\text{-}def\ neg\text{-}until\text{-}always$)

lemma $grd\ (SymSystem\ init\ p\ r) = Sup\ ((-\ prec\text{-}st\ p\ r \sqcup (\Box\ (lft\text{-}pred\text{-}st\ (inpt\text{-}st\ r)))) `\ init)$
proof ($unfold\ fun\text{-}eq\text{-}iff$, $auto\ simp\ add$: $grd\text{-}def\ SymSystem\text{-}rel\ demonic\text{-}def\ assert\text{-}def$)
fix $x :: nat \Rightarrow 'a$ **and** $xa :: nat \Rightarrow 'b$ **and** $u :: nat \Rightarrow 'c$
assume $\forall\, xa$::$nat \Rightarrow 'c \in init.\ prec\text{-}st\ p\ r\ xa\ x \wedge \neg\ (\Box\ lft\text{-}pred\text{-}st\ (inpt\text{-}st\ r))\ xa\ x$
and $u \in init$
and $(\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ xa$
then show $False$
by ($auto\ simp\ add$: $always\text{-}def\ lft\text{-}pred\text{-}st\text{-}def\ inpt\text{-}st\text{-}def\ at\text{-}fun\text{-}def\ lft\text{-}rel\text{-}st\text{-}def$)
next
fix $x :: nat \Rightarrow 'a$ **and** $xa :: nat \Rightarrow 'c$
assume B: $xa \in init$
assume $(\lambda y\ .\ \exists\, u\ .\ u \in init \wedge (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y) \leq \bot$
then have A: $\forall\ y\ u\ .\ u \notin init \vee \neg\ (\Box\ lft\text{-}rel\text{-}st\ r)\ u\ x\ y$
by $auto$
let $?y = \lambda\ n\ .\ (SOME\ z\ .\ r\ (xa\ n)\ (xa\ (Suc\ n))\ (x\ n)\ z)$
from B **and** A **have** $\neg\ (\Box\ lft\text{-}rel\text{-}st\ r)\ xa\ x\ ?y$ **by** $simp$
moreover assume $(\Box\ lft\text{-}pred\text{-}st\ (inpt\text{-}st\ r))\ xa\ x$
ultimately show $False$
apply ($simp\ add$: $always\text{-}def\ lft\text{-}pred\text{-}st\text{-}def\ inpt\text{-}st\text{-}def\ at\text{-}fun\text{-}def\ lft\text{-}rel\text{-}st\text{-}def$)

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

definition *guard S* = {.((*grd S*)::′*a*⇒*bool*).} *o S*

lemma ((*grd* (*local-init init S*))::′*a*⇒*bool*) = *Sup* ((*grd o S*) ‘ *init*)
by (*simp add*: *fun-eq-iff local-init-def assert-def grd-def*)

lemma *u* ∈ *init* ⟹ *guard* ([:*z* ⇝ *u*, *x* . *u* ∈ *init* ∧ *z* = *x*:] *o* {.*u*, *x* . *p u x*.} *o* [:*u*, *x* ⇝ *y* . *r u x y* :])
= [:*z* ⇝ *u*, *x* . *u* ∈ *init* ∧ *z* = *x*:] *o* {.*u*, *x* . *u* ∈ *init* ∧ (∃ *a*. *a* ∈ *init* ∧ (*p a x* ⟶ *Ex* (*r a x*))) ∧ *p u x*.} *o* [:*u*, *x* ⇝ *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*: (∀ *n*. (∀ *i*<*n*. *inpt-st* (λ(*u*, *v*) (*u*′, *v*′). *r u u*′ *OO r*′ *v v*′) (*u i*) (*u* (*Suc i*)) (*x i*)) ⟶
inpt-st r (*fst* (*u n*)) (*fst* (*u* (*Suc n*))) (*x n*) ∧ (∀ *y*. *r* (*fst* (*u n*)) (*fst* (*u* (*Suc n*))) (*x n*) *y* ⟶ *inpt-st r*′ (*snd* (*u n*)) (*snd* (*u* (*Suc n*))) *y*)) ⟶
(∀ *i* < *n* . *inpt-st r* ((*fst o u*) *i*) ((*fst o 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*))
(**is** (∀ *n* . *?p n*) ⟶ *?q n*)
proof (*induction n*)
case *0*
show *?case* **by** *auto*
next
case (*Suc n*)
show *?case*
proof *auto*
fix *i*::*nat*
assume *B*: ∀ *n* . *?p n*
then have *A*: *?p n* (**is** *?A* ⟶ *?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 ¬ *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
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*: (∀ *n*. (∀ *i*<*n*. *inpt-st* (λ(*u*, *v*) (*u'*, *v'*). *r u u' OO r' v v'*) (*u i*) (*u* (*Suc i*)) (*x i*)) ⟶
inpt-st r (*fst* (*u n*)) (*fst* (*u* (*Suc n*))) (*x n*) ∧ (∀ *y*. *r* (*fst* (*u n*)) (*fst* (*u* (*Suc n*))) (*x n*) *y* ⟶ *inpt-st r'* (*snd* (*u n*)) (*snd* (*u* (*Suc n*))) *y*)) ⟹
inpt-st r ((*fst o u*) *n*) ((*fst o u*) (*Suc n*)) (*x n*) ∧ (∀ *y*. *r* (*fst* (*u n*)) (*fst* (*u* (*Suc n*))) (*x n*) *y* ⟶ *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'* = (λ (*u*, *v*) (*u'*, *v'*) *x z* . *inpt-st r u u' x* ∧ (∀ *y* . *r u u' x y* ⟶ *inpt-st r' v v' y*) ∧ (*r u u' OO r' v v'*) *x z*)

lemma *inpt-str-comp-a*: (*prec-st* (*inpt-st r*) *r* (*fst* ∘ *u*) *x* ∧ (∀ *y*. (□ *lft-rel-st r*) (*fst* ∘ *u*) *x y* ⟶ *prec-st* (*inpt-st r'*) *r'* (*snd* ∘ *u*) *y*)) =
prec-st (λ *u u' x* . *inpt-st r* (*fst u*) (*fst u'*) *x* ∧ (∀ *y* . *r* (*fst u*) (*fst u'*) *x y* ⟶ (*inpt-st r'* (*snd u*) (*snd u'*) *y*))) (λ(*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 (□ *lft-pred-st* (*inpt-st r*)) (*fst* ∘ *u*) *x*
then show *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*: (□ *lft-pred-st* (*inpt-st r*)) (*fst* ∘ *u*) *x*
assume *B*: *r* (*fst* (*u n*)) (*fst* (*u* (*Suc n*))) (*x n*) *y*
assume *C*: ∀ *i*<*n*. *inpt-st* (λ(*u*::*'a*, *v*::*'d*) (*u'*::*'a*, *v'*::*'d*). *r u u' OO r' v v'*) (*u i*) (*u* (*Suc i*)) (*x i*)
let *?y* = λ *i* . (*if i* = *n then y else* (*SOME y* . *r* ((*fst o u*) *i*) ((*fst o u*) (*Suc i*)) (*x i*) *y*))
assume ∀ *y* . (□ *lft-rel-st r*) (*fst* ∘ *u*) *x y* ⟶ (□ *lft-pred-st* (*inpt-st r'*)) (*snd* ∘ *u*) *y*
then have *D*: (□ *lft-rel-st r*) (*fst* ∘ *u*) *x ?y* ⟶ (□ *lft-pred-st* (*inpt-st r'*)) (*snd* ∘ *u*) *?y*
by *simp*
from *A* **and** *B* **have** *E*: (□ *lft-rel-st r*) (*fst* ∘ *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** (□ *lft-pred-st* (*inpt-st r'*)) (*snd* ∘ *u*) *?y* **by** *simp*

from A **and** E **and** *this* **show** *inpt-st* r' (*snd* (u n)) (*snd* (u (*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 $\forall\ n$. ($\forall i{<}n$. *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)) $\longrightarrow$
inpt-st r (*fst* (u n)) (*fst* (u (*Suc* n))) (x n) $\wedge$ ($\forall y{::}'c$. r (*fst* (u n)) (*fst* (u (*Suc* n))) (x n) y $\longrightarrow$ *inpt-st* r' (*snd* (u n)) (*snd* (u (*Suc* n))) y)
then show ($\square$ *lft-pred-st* (*inpt-st* r)) (*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 $y{::}nat \Rightarrow 'c$
assume $\forall\ n$. ($\forall i{<}n$. *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)) $\longrightarrow$
inpt-st r (*fst* (u n)) (*fst* (u (*Suc* n))) (x n) $\wedge$ ($\forall y{::}'c$. r (*fst* (u n)) (*fst* (u (*Suc* n))) (x n) y $\longrightarrow$ *inpt-st* r' (*snd* (u n)) (*snd* (u (*Suc* n))) y)
moreover assume ($\square$ *lft-rel-st* r) (*fst* $\circ$ u) x y
ultimately show ($\square$ *lft-pred-st* (*inpt-st* r')) (*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*: *prec-st* (λ u u' x . *inpt-st* r (*fst* u) (*fst* u') x $\wedge$
($\forall\ y$. r (*fst* u) (*fst* u') x y $\longrightarrow$ (*inpt-st* r' (*snd* u) (*snd* u') y))) ($\lambda(u,\ v)$ (u', v'). r u u' *OO* r' v v') u x
$=$ ($\square$ (*lft-pred-st* (*inpt-st* (*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 $m{::}nat$
assume A: $\forall n$. ($\forall i{<}n$. *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) (**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 ((*fst* $\circ$ u) m) ((*fst* $\circ$ u) (*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** *inpt-st* ($\lambda(u,\ v)$ (u', v') ($x{::}'c$) z. *inpt-st* r u u' x $\wedge$ ($\forall y$. r u u' x y
$\longrightarrow$ *inpt-st* r' v v' y) $\wedge$ (r u u' *OO* r' v v') x z) (u m) (u (*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 ∀ *m*. *inpt-st* (λ(*u*, *v*) (*u'*, *v'*) (*x*) *z*. *inpt-st r u u' x* ∧ (∀ *y*. *r u u' x y* ⟶ *inpt-st r' v v' y*)
∧ (*r u u' OO r' v v'*) *x z*) (*u m*) (*u* (*Suc m*)) (*x m*) (**is** ∀ *m* . *?p m*)
then have *?p m* **by** *simp*
then show *inpt-st r* (*fst* (*u m*)) (*fst* (*u* (*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 ∀ *m*. *inpt-st* (λ(*u*, *v*) (*u'*, *v'*) (*x*) *z*. *inpt-st r u u' x* ∧ (∀ *y*. *r u u' x y* ⟶ *inpt-st r' v v' y*)
∧ (*r u u' OO r' v v'*) *x z*) (*u m*) (*u* (*Suc m*)) (*x m*) (**is** ∀ *m* . *?p m*)
then have *?p m* **by** *simp*
moreover assume *r* (*fst* (*u m*)) (*fst* (*u* (*Suc m*))) (*x m*) *y*
ultimately show *inpt-st r'* (*snd* (*u m*)) (*snd* (*u* (*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*: (*prec-st* (*inpt-st r*) *r* (*fst* ∘ *u*) *x* ∧ (∀ *y*. (□ *lft-rel-st r*) (*fst* ∘ *u*) *x y* ⟶ *prec-st* (*inpt-st r'*) *r'* (*snd* ∘ *u*) *y*))
= (□ (*lft-pred-st* (*inpt-st* (*rel-st r r'*)))) *u x*
by (*simp add*: *inpt-str-comp-a inpt-str-comp-b*)

lemma *RSysTmp-inpt-comp*: *u'* ∈ *init'* ⟹ *SymSystem init* (*inpt-st r*) *r o SymSystem init'*(*inpt-st r'*) *r'*
= *SymSystem* (*zip-set init init'*) (*inpt-st* (*rel-st r r'*)) (*rel-st r r'*)
proof −
assume *A* : *u'* ∈ *init'*
have [*simp*]: (λ*x y*. (*case x of* (*x*, *xa*) ⇒ (□ *lft-pred-st* (*inpt-st* (*rel-st r r'*))) *x xa*) ∧
(*case x of* (*x*, *xa*) ⇒ (□ *lft-rel-st* (λ(*u*, *v*) (*u'*, *v'*). *r u u' OO r' v v'*)) *x xa*) *y*)
= (λ(*x*, *y*). (□ *lft-rel-st* (*rel-st r r'*)) *x y*) (**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* :: *nat* ⇒ *'e* × *'a* **and** *b* :: *nat* ⇒ *'c* **and** *x* :: *nat* ⇒ *'b* **and** *xa* :: *nat*
assume ∀ *xa*::*nat*. (*case a xa of* (*u*::*'e*, *v*::*'a*) ⇒ λ(*u'*::*'e*, *v'*::*'a*). *r u u' OO r' v v'*) (*a* (*Suc xa*)) (*b xa*) (*x xa*) (**is** ∀ *xa* . *?P xa*)
then have *A*: *?P xa* **by** *simp*
assume ∀ *x* . *Ex* ((*case a x of* (*u*, *v*) ⇒ λ(*u'*, *v'*) (*x*) *z*. *Ex* (*r u u' x*) ∧ (∀ *y*. *r u u' x y* ⟶ *Ex* (*r' v v' y*)) ∧ (*r u u' OO r' v v'*) *x z*) (*a* (*Suc x*)) (*b x*)) (**is** ∀ *xa* . *?Q xa*)
then have *?Q xa* **by** *simp*
from *this* **and** *A* **show** (*case a xa of* (*u*, *v*) ⇒ λ(*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** $SymSystem\ init\ (inpt\text{-}st\ r)\ r\ o\ SymSystem\ init'(inpt\text{-}st\ r')\ r' = [:\ z \rightsquigarrow u,\ x\ .\ fst \circ u \in init \wedge snd \circ u \in init' \wedge z = x\ :] \circ$
$(\{.u,\ x\ .\ prec\text{-}st\ (inpt\text{-}st\ r)\ r\ (fst \circ u)\ x \wedge (\forall y::nat \Rightarrow 'd.\ (\Box\ lft\text{-}rel\text{-}st\ r)\ (fst \circ u)\ x\ y \longrightarrow prec\text{-}st\ (inpt\text{-}st\ r')\ r'\ (snd \circ u)\ y).\} \circ$
$[:\ (\lambda(u,\ x).\ ((\Box\ lft\text{-}rel\text{-}st\ (\lambda(u,\ v)\ (u',\ v').\ r\ u\ u'\ OO\ r'\ v\ v'))\ u\ x))\ :])$

by (*unfold SymSystem-comp*, *simp add*: *comp-assoc*)

also have $... = [:\ z \rightsquigarrow u,\ x\ .\ fst \circ u \in init \wedge snd \circ u \in init' \wedge z = x\ :] \circ (\{.\ x,\ y\ .\ (\Box\ lft\text{-}pred\text{-}st\ (inpt\text{-}st\ (rel\text{-}st\ r\ r'))\ x\ y\ .\} \circ [:\ ?b\ :])$

by (*subst assert-demonic*, *simp add*: *inpt-str-comp*)

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

by (*simp add*: *SymSystem-def prec-st-inpt comp-assoc zip-set-def*)

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

qed

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

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

proof (*auto simp add*: *fun-eq-iff*)

fix $a :: nat \Rightarrow 'a$ **and** $b :: nat \Rightarrow 'b$

assume $inpt\ (\lambda(x,\ y).(\Box\ lft\text{-}rel\text{-}st\ r)\ x\ y)\ (a,\ b)$

then show $(\Box\ lft\text{-}pred\text{-}st\ (inpt\text{-}st\ r))\ a\ b$

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 $(\Box\ lft\text{-}pred\text{-}st\ (inpt\text{-}st\ r))\ a\ b$

then have $(\Box\ lft\text{-}rel\text{-}st\ r)\ a\ b\ ?y$

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 $inpt\ (\lambda(x,\ y).\ (\Box\ lft\text{-}rel\text{-}st\ r)\ x\ y)\ (a,\ b)$

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 boolen 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* ⇒ *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* . (∀ *k* . *count x k* ≤ *n*).} *o* [:*x* ⇝ *y* . *y* = *count x*:]

definition *prec-count M u u′ x* = (*u* ≤ *M*)
definition *rel-count u u′ x y* = ((*x* ⟶ *u′* = *Suc u*) ∧ (¬ *x* ⟶ *u′* = *u*) ∧ *y* = *u′*)

lemma *counter-a-aux*: *u 0* = *0* ⟹ ∀ *i* < *n*. (*x i* ⟶ *u* (*Suc i*) = *Suc* (*u i*)) ∧ (¬ *x i* ⟶ *u* (*Suc i*) = *u i*) ⟹ (∀ *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 ∀ *i*<*Suc n*. (*x i* ⟶ *u* (*Suc i*) = *Suc* (*u i*)) ∧ (¬ *x i* ⟶ *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*: $u\ 0 = 0 \Longrightarrow \forall n.\ (xa\ n \longrightarrow u\ (Suc\ n) = Suc\ (u\ n)) \wedge (\neg\ xa\ n \longrightarrow u\ (Suc\ n) = u\ n) \wedge xb\ n = u\ (Suc\ n)$
$\Longrightarrow count\ xa\ n = u\ (Suc\ n)$
by (*induction n*, *simp-all*)

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

lemma $COUNTER = Counter\text{-}global$
proof −
have A:$(\lambda x\ y\ .\ \exists u{::}nat \Rightarrow nat.\ u\ (0{::}nat) = (0{::}nat) \wedge (\Box\ lft\text{-}rel\text{-}st\ rel\text{-}count)\ u\ x\ y)$
$= (\lambda\ x\ y\ .\ y = count\ x)$
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** $xa{::}nat \Rightarrow nat$
define u **where** $u\ i = (if\ i = 0\ then\ 0\ else\ count\ x\ (i - 1))$ **for** i
assume B: $\forall xb{::}nat.\ xa\ xb = count\ x\ xb$
{fix $xb{::}nat$
from *u-def* **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 $(\lambda xa\ .\ \forall u\ .\ u\ (0{::}nat) = (0{::}nat) \longrightarrow prec\text{-}st\ (prec\text{-}count\ x)\ rel\text{-}count\ u\ xa) =$
$(\lambda xa{::}nat \Rightarrow bool.\ \forall k{::}nat.\ count\ xa\ k \leq x)$
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** $k{::}\ nat$
define *uu* **where** $uu\ i = (if\ i = 0\ then\ 0\ else\ count\ xa\ (i - 1))$ **for** i
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** *uu-def* **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** *uu-def* **show** $count\ xa\ k \leq x$
by (*case-tac xb*, *auto*)
next
fix *xa*:: $nat \Rightarrow bool$ **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 neq0-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 $u\ 0 = 0 \Longrightarrow \forall\, i{<}xaa.\ (xa\ i \longrightarrow u\ (Suc\ i) = Suc\ (u\ i)) \wedge (\neg\ xa\ i \longrightarrow u\ (Suc\ i) = u\ i) \Longrightarrow \forall\, i{<}xaa.\ count\ xa\ i = u\ (Suc\ i)$
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 asumming 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-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-ex u u' x* = $(-1 \leq u \wedge u \leq 3)$

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

lemma *LIVE-fusion*: *LIVE* = (*SymSystem* {u . u 0 = 0} *prec-ex rel-ex*) ‖ [:x $\rightsquigarrow$ y . ($\Box$ ($\Diamond$ (λ y . y 0))) y:]
proof −
define *init* **where** *init* = {u . u (0::*nat*) = (0::*int*)}
then have *A*: (λ *i*::*nat* . 0::*int*) ∈ *init*
by *simp*
then have ([: $x \rightsquigarrow$ (u, y). $u \in$ *init* $\wedge$ $x = y$:] ∘ {.(x, y). *prec-st prec-ex rel-ex x y* .} ∘ [: λ(x, y). ($\Box$ *lft-rel-st rel-ex*) x y :]) ‖
[: λx. $\Box$ $\Diamond$ (λy. y 0) :] =
[: $x \rightsquigarrow$ (u, y). $u \in$ *init* $\wedge$ $x = y$:] ∘ {. (x, y). *prec-st prec-ex rel-ex x y* .} ∘
[: (u, x) $\rightsquigarrow$ y. ($\Box$ *lft-rel-st rel-ex*) u x y $\wedge$ ($\Box$ $\Diamond$ (λy. y 0)) y :]
by (*unfold fusion-spec-local-a*, *auto*)
then show *?thesis*
by (*simp add*: *init-def SymSystem-def*)
(*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*) $\Longrightarrow$ $\forall$ *n*. *xa* (*Suc n*) = ($\neg$ *xa n*) $\Longrightarrow$
$\forall$ *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) $\Longrightarrow$
(a *n* = -1 $\longrightarrow$ *xa n*) $\wedge$ (a *n* = 1 $\longrightarrow$ $\neg$ *xa n*) $\wedge$ $-1 \leq a$ *n* $\wedge$ a *n* ≤ 1
proof (*induction n*)
case 0
show *?case*
by (*metis 0.prems*(1) *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* ≠ − (1::*int*) **from** *this* **and** *Suc* **have** − (1::*int*) ≤ 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* = *this*
{assume a *n* = − (1::*int*) **from** *this* **and** *Suc* **have** − (1::*int*) ≤ a (*Suc n*)
by (*metis add.commute le-minus-one-simps*(4) *monoid-add-class.add.right-neutral*

not-le right-minus zle-add1-eq-le)**}**
note *D* = *this*
from *C* **and** *D* **and** *Suc* **have** *E*: − (*1*::*int*) ≤ *a* (*Suc n*) **by** *auto*
from *Suc* **have** *F*: *a* (*Suc n*) ≤ (*1*::*int*)
by (*metis eq-iff int-one-le-iff-zero-less le-iff-diff-le-0 le-less not-le zle-add1-eq-le*)
from *A B E F* **show** *?case* **by** *auto*
qed

lemma *event-ex-aux*: *a 0* = (*0*::*int*) ⟹ ∀ *n*. *xa* (*Suc n*) = (¬ *xa n*) ⟹
∀ *n*. (*xa n* ∧ *a* (*Suc n*) = *a n* + *1* ∨ ¬ *xa n* ∧ *a* (*Suc n*) = *a n* − *1* ∨ *a* (*Suc n*) = *0*) ⟹
(∀ *n* . (*a n* = −*1* ⟶ *xa n*) ∧ (*a n* = *1* ⟶ ¬ *xa n*) ∧ −*1* ≤ *a n* ∧ *a n* ≤ *1*)
by (*clarify*, *drule event-ex-aux-a*, *auto*)

lemma {.□ *preca-ex.*} *o LIVE* ≤ *SymSystem* {*u* . *u 0* = *0*} *prec-ex rel-ex*
proof (*unfold LIVE-fusion SymSystem-def*, *rule fusion-local-refinement*, *simp-all*)
fix *z*::*nat* ⇒ *bool* **and** *u* :: *nat* ⇒ *int* **and** *x*::*nat* ⇒ *bool*
assume *A*: *u 0* = *0*
assume (□ *preca-ex*) *z*
then have *B*: ∀ *x*::*nat*. *z* (*Suc x*) = (¬ *z x*)
by (*auto simp add*: *preca-ex-def lft-rel-st-def rel-ex-def always-def at-fun-def*)
assume (□ *lft-rel-st rel-ex*) *u z x*
then have *C*: ∀ *xa* . (*z xa* ∧ *u* (*Suc xa*) = *u xa* + *1* ∨ ¬ *z xa* ∧ *u* (*Suc xa*) = *u xa* − *1* ∨ *u* (*Suc xa*) = *0*) ∧ *x xa* = (*u* (*Suc xa*) = *0*)
by (*auto simp add*: *preca-ex-def lft-rel-st-def rel-ex-def always-def at-fun-def*)
have *D*: (∀ *n* . (*u n* = −*1* ⟶ *z n*) ∧ (*u n* = *1* ⟶ ¬ *z n*) ∧ −*1* ≤ *u n* ∧ *u n* ≤ *1*)
by (*cut-tac A B C*, *rule event-ex-aux*, *auto*)
{
fix *a*::*nat*
{assume *u* (*Suc a*) = *0* **from** *this A B C* **have** ∃ *b* . *u* (*Suc* (*a* + *b*)) = *0*
by (*metis monoid-add-class.add.right-neutral*)**}**
note *1* = *this*
{assume *u* (*Suc a*) = −*1* **from** *this A B C D* **have** ∃ *b* . *u* (*Suc* (*a* + *b*)) = *0*
by (*metis add-Suc-right diff-minus-eq-add diff-self monoid-add-class.add.right-neutral*)**}**
note *2* = *this*
{assume *u* (*Suc a*) = *1* **from** *this A B C D* **have** ∃ *b* . *u* (*Suc* (*a* + *b*)) = *0*
by (*metis add-Suc-right diff-self monoid-add-class.add.right-neutral*)**}**
note *3* = *this*
from *1 2 3 A B C D* **have** ∃ *b* . *x* (*a* + *b*)
by (*simp*, *metis diff-0 int-one-le-iff-zero-less le-less not-le zle-diff1-eq*)
}
then show (□ ◇ (λ*y* . *y 0*)) *x*
by (*simp add*: *always-def eventually-def preca-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.