Formalization of Refinement Calculus for Reactive Systems

Viorel Preoteasa
Aalto University, Finland

March 17, 2025

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

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 (‹⊥›) **and**
top (‹⊤›) **and**
inf (**infixl** ‹⊓› *70*)
and *sup* (**infixl** ‹⊔› *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* :: $'a \Rightarrow 'a$ (‹□ (-)› [*900*] *900*) **where**
$\Box\ p = (INF\ i\ .\ p\ !!\ i)$

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

definition *next* :: $'a \Rightarrow 'a$ (‹○ (-)› [*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*⟩

end

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

context *temporal*
begin
lemma [*simp*]: $(a \sqcup b)\ !!\ i = (a\ !!\ i) \sqcup (b\ !!\ i)$
⟨*proof*⟩

lemma *always-less* [*simp*]: $\Box\ p \leq p$
⟨*proof*⟩

lemma *always-and*: $\Box\ (p \sqcap q) = (\Box\ p) \sqcap (\Box\ q)$
⟨*proof*⟩

lemma *eventually-or*: $\Diamond\ (p \sqcup q) = (\Diamond\ p) \sqcup (\Diamond\ q)$
⟨*proof*⟩

lemma *neg-until-always*: $-(p\ until\ -p) = \Box\ p$
⟨*proof*⟩

lemma *neg-always-eventually*: $\Box\ p = -\ \Diamond\ (-\ p)$
⟨*proof*⟩

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

lemma *true-until-eventually*: $(\top\ until\ p) = \Diamond\ p$
⟨*proof*⟩
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*⟩
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 ⇒ nat ⇒ 'a (‹-[- ..]› [80, 15] 80)
  assumes [simp]: a[i..][j..] = a[i + j..]
  assumes [simp]: a[0..] = a
  begin
    definition next-trace :: 'a ⇒ 'a  (‹⊙ (-)› [900] 900) where
      ⊙ p = p[Suc 0..]
  end

instantiation fun :: (trace, temporal) temporal
  begin
    definition at-fun-def: (P:: 'a ⇒ 'b) !! i = (λ x . (P (x[i..])) !! i)
    instance ⟨proof⟩
  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
    ⟨proof⟩
  lemma AbsNat-zero [simp]: AbsNat 0 + i = i
    ⟨proof⟩

  subclass comm-monoid-diff
    ⟨proof⟩
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*⟩
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*⟩
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'*))
⟨*proof*⟩

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* ⇝ *y*′, *z* . (*P*::'*a* ⇒ '*b* ⇒ '*c* ⇒ '*d* ⇒ *bool*) *x y y*′ *z* :]
term {: *x*, *y* ⇝ *y*′, *z* . (*P*::'*a* ⇒ '*b* ⇒ '*c* ⇒ '*d* ⇒ *bool*) *x y y*′ *z* :}

lemma *demonic-demonic*: [:*r*:] *o* [:*r*′:] = [:*r OO r*′:]
⟨*proof*⟩

lemma *assert-demonic-comp*: {.*p*.} *o* [:*r*:] *o* {.*p*′.} *o* [:*r*′:] =
{.*x* . *p x* ∧ (∀ *y* . *r x y* ⟶ *p*′ *y*).} *o* [:*r OO r*′:]
⟨*proof*⟩

lemma *demonic-assert-comp*: [:*r*:] *o* {.*p*.} = {.*x*.(∀ *y* . *r x y* ⟶ *p y*).} *o* [:*r*:]
⟨*proof*⟩

lemma *assert-assert-comp*: {.*p*::'*a*::*lattice*.} *o* {.*p*′.} = {.*p* ⊓ *p*′.}
⟨*proof*⟩

definition *inpt r x* = (∃ *y* . *r x y*)

definition *trs r* = {. *inpt r*.} *o* [:*r*:]

lemma *trs* (λ *x y* . *x* = *y*) *q x* = *q x*
⟨*proof*⟩

lemma *assert-demonic-prop*: $\{.p.\}\ o\ [:r:] = \{.p.\}\ o\ [:(\lambda\ x\ y\ .\ p\ x) \sqcap r:]$
⟨*proof*⟩

lemma *relcompp-exists*: $relcompp\ r\ r'\ x\ y = (\exists\ u\ .\ r\ x\ u \wedge r'\ u\ y)$
⟨*proof*⟩

lemma *trs-trs*: $(trs\ r)\ o\ (trs\ r') = trs\ ((\lambda\ s\ t.\ (\forall\ s'\ .\ r\ s\ s' \longrightarrow (inpt\ r'\ s'))) \sqcap (r\ OO\ r'))$ (**is** *?S* = *?T*)
⟨*proof*⟩

lemma *assert-demonic-refinement*: $(\{.p.\}\ o\ [:r:] \leq \{.p'.\}\ o\ [:r':]) = (p \leq p' \wedge (\forall\ x\ .\ p\ x \longrightarrow r'\ x \leq r\ x))$
⟨*proof*⟩

lemma *trs-refinement*: $(trs\ r \leq trs\ r') = ((\forall\ x\ .\ inpt\ r\ x \longrightarrow inpt\ r'\ x) \wedge (\forall\ x\ .\ inpt\ r\ x \longrightarrow r'\ x \leq r\ x))$
⟨*proof*⟩

lemma $trs\ (\lambda\ x\ y\ .\ x \geq 0) \leq trs\ (\lambda\ x\ y\ .\ x = y)$
⟨*proof*⟩

lemma $trs\ (\lambda\ x\ y\ .\ x \geq 0)\ q\ x = (if\ q = \top\ then\ x \geq 0\ else\ False)$
⟨*proof*⟩

lemma $[:r:] \sqcap [:r':] = [:r \sqcup r':]$
⟨*proof*⟩

lemma *spec-demonic-choice*: $(\{.p.\}\ o\ [:r:]) \sqcap (\{.p'.\}\ o\ [:r':]) = (\{.p \sqcap p'.\}\ o\ [:r \sqcup r':])$
⟨*proof*⟩

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*⟩

lemma $p \sqcap p' = \bot \Longrightarrow (\{.p.\}\ o\ [:r:]) \sqcup (\{.p'.\}\ o\ [:r':]) = \{.p \sqcup p'.\}\ o\ [:(\lambda\ x\ y\ .\ p\ x \longrightarrow r\ x\ y) \sqcap ((\lambda\ x\ y\ .\ p'\ x \longrightarrow r'\ x\ y)):]$
⟨*proof*⟩

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*]: $conjunctive\ S \Longrightarrow sconjunctive\ S$
⟨*proof*⟩

lemma [*simp*]: *conjunctive* $\top$
⟨*proof*⟩

lemma *conjuncive-demonic* [*simp*]: *conjunctive* [:*r*:]
⟨*proof*⟩

lemma *sconjunctive-assert* [*simp*]: *sconjunctive* {.*p*.}
⟨*proof*⟩

lemma *sconjunctive-simp*: $x \in Q \Longrightarrow$ *sconjunctive* $S \Longrightarrow S\ (Inf\ Q) = Inf\ (S\ `\ Q)$
⟨*proof*⟩

lemma *sconjunctive-INF-simp*: $x \in X \Longrightarrow$ *sconjunctive* $S \Longrightarrow S\ (Inf\ (Q\ `\ X)) = Inf\ (S\ `\ (Q\ `\ X))$
⟨*proof*⟩

lemma *demonic-comp* [*simp*]: *sconjunctive* $S \Longrightarrow$ *sconjunctive* $S' \Longrightarrow$ *sconjunctive* $(S\ o\ S')$
⟨*proof*⟩

lemma [*simp*]:*conjunctive* $S \Longrightarrow S\ (Inf\ (Q\ `\ X)) = (Inf\ ((S\ o\ Q)\ `\ X))$
⟨*proof*⟩

lemma *conjunctive-simp*: *conjunctive* $S \Longrightarrow\ S\ (Inf\ Q) = Inf\ (S\ `\ Q)$
⟨*proof*⟩

lemma *conjunctive-monotonic*: *sconjunctive* $S \Longrightarrow$ *mono* S
⟨*proof*⟩

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

lemma *grd* [:*r*:] = *inpt r*
⟨*proof*⟩

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

lemma $fail\ (\{.p.\}\ o\ [:r :: 'a \Rightarrow 'b \Rightarrow bool:]) = -p$
⟨*proof*⟩

definition $Fail = \bot$

lemma *mono* $(S::'a::boolean\text{-}algebra \Rightarrow 'b::boolean\text{-}algebra) \Longrightarrow (S = Fail) = (fail\ S = \top)$
⟨*proof*⟩

definition $Skip = id$

lemma [*simp*]: {.⊤::'*a*::*bounded-lattice*.} = *Skip*
⟨*proof*⟩

lemma [*simp*]:*Skip o S* = *S*
⟨*proof*⟩

lemma [*simp*]:*S o Skip* = *S*
⟨*proof*⟩

lemma [*simp*]: *mono S* ⟹ *mono S'* ⟹ *mono* (*S o S'*)
⟨*proof*⟩

lemma [*simp*]: *mono* {.*p*::('*a* ⇒ *bool*).}
⟨*proof*⟩

lemma [*simp*]: *mono* [:*r*::('*a* ⇒ '*b* ⇒ *bool*):]
⟨*proof*⟩

lemma [*simp*]:{. λ *x* . *True* .} = *Skip*
⟨*proof*⟩

lemma [*simp*]: ⊥ *o S* = ⊥
⟨*proof*⟩

lemma [*simp*]: {.⊥::'*a*::*boolean-algebra*.} = ⊥
⟨*proof*⟩

lemma [*simp*]: ⊤ *o S* = ⊤
⟨*proof*⟩

lemma *left-comp*: *T o U* = *T' o U'* ⟹ *S o T o U* = *S o T' o U'*
⟨*proof*⟩

lemma *assert-demonic*: {.*p*.} *o* [:*r*:] = {.*p*.} *o* [:λ *x y* . *p x* ∧ *r x y*:]
⟨*proof*⟩

lemma *trs r* ⊓ *trs r'* = *trs* (λ *x y* . *inpt r x* ∧ *inpt r' x* ∧ (*r x y* ∨ *r' x y*))
⟨*proof*⟩

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**

$(S \parallel S')\ q\ x = (\exists\ (p::'a \Rightarrow bool)\ p'\ .\ p \sqcap p' \leq q \wedge S\ p\ x \wedge S'\ p'\ x)$

lemma *fusion-spec*: $(\{.p.\} \circ [:r:]) \parallel (\{.p'.\} \circ [:r':]) = (\{.p \sqcap p'.\} \circ [:r \sqcap r':])$
⟨*proof*⟩

lemma *fusion-assoc*: $S \parallel (T \parallel U) = (S \parallel T) \parallel U$
⟨*proof*⟩

lemma $P \leq Q \Longrightarrow P' \leq Q' \Longrightarrow P \parallel P' \leq Q \parallel Q'$
⟨*proof*⟩

lemma *conjunctive* $S \Longrightarrow S \parallel \top = \top$
⟨*proof*⟩

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*⟩

lemma *fusion-spec-local-a*: $a \in init \Longrightarrow ([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:r:]) \parallel [:r':]$
$= ([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:u,\ x \rightsquigarrow y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ y:])$
⟨*proof*⟩

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*⟩

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 $ASSERT\text{-}LIVE = \{.\ \Box \Diamond\ (\lambda\ x\ .\ x\ 0).\}$
definition $GUARANTY\text{-}LIVE = [:x \rightsquigarrow y\ .\ \Box \Diamond\ (\lambda\ y\ .\ y\ 0):]$
definition $AE = ASSERT\text{-}LIVE\ o\ HAVOC$
definition $SKIP = [:x \rightsquigarrow y\ .\ x = y:]$

lemma [*simp*]: *SKIP* = *id*
⟨*proof*⟩

definition *REQ-RESP* = [: □(λ *xs ys* . *xs* (*0*::*nat*) ⟶ (◇ (λ *ys* . *ys* (*0*::*nat*))) *ys*) :]
definition *FAIL* = ⊥

lemma *HAVOC o ASSERT-LIVE* = *FAIL*
⟨*proof*⟩

lemma *HAVOC o AE* = *FAIL*
⟨*proof*⟩

lemma *HAVOC o ASSERT-LIVE* = *FAIL*
⟨*proof*⟩

lemma *SKIP o AE* = *AE*
⟨*proof*⟩

lemma (*REQ-RESP o AE*) = *AE*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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* ⇝ *y* . ∃ *u* . *u* ∈ *init* ∧ (□ *lft-rel-st r*) *u x y* :]
⟨*proof*⟩

theorem *SymSystem init p r q x* = *LocalSystem init p r q x*
⟨*proof*⟩

definition *local-init init S* = *Inf* (*S* ‘ *init*)

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 [*simp*]: *fst* ∘ *x* || *y* = *x*
⟨*proof*⟩

lemma [*simp*]: *snd* ∘ *x* || *y* = *y*
⟨*proof*⟩

lemma [*simp*]: *x* ∈ *A* ⟹ *y* ∈ *B* ⟹ (*x* || *y*) ∈ *zip-set A B*
⟨*proof*⟩

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* :]
⟨*proof*⟩

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*⟩

lemma *init-state*: [:$z \rightsquigarrow u$, 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 . $z = x$:] o {.u, x . $u \in init \longrightarrow p\ u\ x$.} o [:u, $x \rightsquigarrow y$. $u \in init \wedge r\ u\ x\ y$:]
⟨*proof*⟩

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*⟩

theorem *SymSystem-comp*: $u' \in init' \Longrightarrow$ *SymSystem init p r o SymSystem init′ p′ r′*
= [:$z \rightsquigarrow u$, 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* ⇝ *y* . (□ *lft-rel-st* (λ(*u*, *v*) (*u′*, *v′*). *r u u′ OO r′ v v′*)) *u x y* :]
(**is** *?p* ⟹ *?S* = *?T*)
⟨*proof*⟩

lemma *always-lft-rel-comp-a*: (□ *lft-rel-st r*) *u OO* (□ *lft-rel-st r′*) *v*
= (□ *lft-rel-st* (λ (*u*, *v*) (*u′*, *v′*) . ((*r u u′*) *OO* (*r′ v v′*)))) (*u* || *v*)
⟨*proof*⟩

theorem *SymSystem-comp-a*: $u' \in init' \Longrightarrow$ *SymSystem init p r o SymSystem init′ p′ r′*
= {.*x* . ∀ *u v* . *u* ∈ *init* ∧ *v* ∈ *init′* ⟶ (*prec-st p r u x* ∧ (∀ *y*. (□ *lft-rel-st r*) *u x y* ⟶ *prec-st p′ r′ v y*)) .}
o [: *x* ⇝ *y* . ∃ *u v* . *u* ∈ *init* ∧ *v* ∈ *init′* ∧ (□ *lft-rel-st* (λ(*u*, *v*) (*u′*, *v′*). *r u u′ OO r′ v v′*)) (*u* || *v*) *x y* :]
(**is** *?p* ⟹ *?S* = *?T*)
⟨*proof*⟩

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* = (⊤:: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *nat* ⇒ *bool*)
definition *localRelS* = (λ (*u*::*nat*, *v*) (*u′*, *v′*::*nat*) (*x*::*nat*) (*y*::*nat*) . *u* = *0* ∧ *u′* = *1* ∧ *v* = *v′*)

definition *initS′* = (⊤::(*nat* ⇒ (*nat* × *nat*)) *set*)
definition *localPrecS′* = (⊥:: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *nat* ⇒ *bool*)
definition *localRelS′* = (λ (*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*: $(\ \lambda x.\ \forall\ (a{::}nat)\ (aa{::}nat)\ (b{::}nat).\ \neg\ (case\ x\ of\ (x{::}nat,\ u{::}nat,\ v{::}nat) \Rightarrow \lambda ab.\ u = 0 \wedge$
$(case\ ab\ of\ (y,\ u',\ v') \Rightarrow u' = Suc\ 0 \wedge v = v'))\ (a,\ aa,\ b))$
$= (\lambda\ (x,\ u,\ v)\ .\ u > 0)$
⟨*proof*⟩

lemma *localSS'-aux-b*: $((\lambda(x,\ u,\ v)\ ab.\ u = 0 \wedge (case\ ab\ of\ (y,\ u',\ v') \Rightarrow u' = Suc\ 0 \wedge v = v'))\ OO\ (\lambda(x,\ u,\ v)\ (y,\ u',\ v').\ u = u'))$
$= (\lambda\ (x,\ u,\ v)\ (y,\ u',\ v')\ .\ u = 0 \wedge u' = 1)$
⟨*proof*⟩

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

lemma [*simp*]: $[:\ \bot{::}('a \Rightarrow 'b => ('c{::}boolean\text{-}algebra))\ :] = \top$
⟨*proof*⟩

definition $symbSS' = SymSystem\ initS\ localPrecSS'\ localRelSS'$

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

lemma *symbSS'*: $symbSS' = \bot$
⟨*proof*⟩

lemma *symbS*: $symbS = \top$
⟨*proof*⟩

lemma $symbS\ o\ symbS' \neq symbSS'$
⟨*proof*⟩

lemma *prec-st-inpt*: $prec\text{-}st\ (inpt\text{-}st\ r)\ r = (\Box\ (lft\text{-}pred\text{-}st\ (inpt\text{-}st\ r)))$
⟨*proof*⟩

lemma *grd* (*SymSystem init p r*) = *Sup* ((− *prec-st p r* ⊔ (□ (*lft-pred-st* (*inpt-st r*)))) ‘ *init*)
⟨*proof*⟩

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

lemma ((*grd* (*local-init init S*))::′*a*⇒*bool*) = *Sup* ((*grd o S*) ‘ *init*)
⟨*proof*⟩

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*) :]
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *inpt-str-comp-b*: *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*
= (□ (*lft-pred-st* (*inpt-st* (*rel-st r r*′)))) *u x*
⟨*proof*⟩

lemma *inpt-str-comp*: (*prec-st* (*inpt-st r*) *r* (*fst* ∘ *u*) *x* ∧ (∀ *y*. (□ *lft-rel-st r*) (*fst*

$\circ$ u) x y $\longrightarrow$ *prec-st* (*inpt-st* r') r' (*snd* $\circ$ u) y))
= ($\Box$ (*lft-pred-st* (*inpt-st* (*rel-st* r r')))) u x
⟨*proof*⟩

lemma *RSysTmp-inpt-comp*: $u' \in init' \Longrightarrow$ *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*⟩

definition *GrdSymSystem init* r = [:$z \rightsquigarrow u$, x . $u \in init \wedge z = x$:] *o trs* (λ (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*⟩

lemma *GrdSymSystem init* r = *SymSystem init* (*inpt-st* r) r
⟨*proof*⟩

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* $\Rightarrow$ *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*: $u\ 0 = 0 \Longrightarrow \forall i < n.\ (x\ i \longrightarrow u\ (Suc\ i) = Suc\ (u\ i)) \wedge (\neg\ x\ i \longrightarrow u\ (Suc\ i) = u\ i) \Longrightarrow (\forall\ i < n\ .\ count\ x\ i = u\ (Suc\ i))$
⟨*proof*⟩

lemma *counter-b-aux*: $u\ 0 = 0 \Longrightarrow \forall n.\ (xa\ n \longrightarrow u\ (Suc\ n) = Suc\ (u\ n)) \wedge (\neg\ xa\ n \longrightarrow u\ (Suc\ n) = u\ n) \wedge xb\ n = u\ (Suc\ n)$
$\Longrightarrow count\ xa\ n = u\ (Suc\ n)$
⟨*proof*⟩

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

lemma *COUNTER* = *Counter-global*
⟨*proof*⟩

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* $(u\ 0)\ (u\ 1)\ (x\ 0)\ (y\ 0)))\ u\ x\ y\ \wedge (\Box\ (\Diamond\ (\lambda\ y\ .\ y\ 0)))\ y\ :]$

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

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

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

lemma *event-ex-aux-a*: $a\ 0 = (0::int) \Longrightarrow \forall n.\ xa\ (Suc\ n) = (\neg\ xa\ n) \Longrightarrow$
$\forall n.\ (xa\ n \wedge a\ (Suc\ n) = a\ n + 1 \vee \neg\ xa\ n \wedge a\ (Suc\ n) = a\ n - 1 \vee a\ (Suc\ n) = 0) \Longrightarrow$
$(a\ n = -1 \longrightarrow xa\ n) \wedge (a\ n = 1 \longrightarrow \neg\ xa\ n) \wedge -1 \leq a\ n \wedge a\ n \leq 1$
⟨*proof*⟩

lemma *event-ex-aux*: $a\ 0 = (0::int) \Longrightarrow \forall n.\ xa\ (Suc\ n) = (\neg\ xa\ n) \Longrightarrow$
$\forall n.\ (xa\ n \wedge a\ (Suc\ n) = a\ n + 1 \vee \neg\ xa\ n \wedge a\ (Suc\ n) = a\ n - 1 \vee a\ (Suc\ n) = 0) \Longrightarrow$
$(\forall\ n\ .\ (a\ n = -1 \longrightarrow xa\ n) \wedge (a\ n = 1 \longrightarrow \neg\ xa\ n) \wedge -1 \leq a\ n \wedge a\ n \leq$

1)
⟨*proof*⟩

lemma {.□ *preca-ex.*} *o LIVE* ≤ *SymSystem* {*u* . *u 0* = *0*} *prec-ex rel-ex*
⟨*proof*⟩
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.