Dynamic Architectures

Diego Marmsoler

October 27, 2022

Abstract

The architecture of a system describes the system's overall organization into components and connections between those components. With the emergence of mobile computing, dynamic architectures have become increasingly important. In such architectures, components may appear or disappear, and connections may change over time. In the following we mechanize a theory of dynamic architectures and verify the soundness of a corresponding calculus. Therefore, we first formalize the notion of configuration traces [5] as a model for dynamic architectures. Then, the behavior of single components is formalized in terms of behavior traces and an operator is introduced and studied to extract the behavior of a single component out of a given configuration trace. Then, behavior trace assertions are introduced as a temporal specification technique to specify behavior of components. Reasoning about component behavior in a dynamic context is formalized in terms of a calculus for dynamic architectures [3]. Finally, the soundness of the calculus is verified by introducing an alternative interpretation for behavior trace assertions over configuration traces and proving the rules of the calculus. Since projection may lead to finite as well as infinite behavior traces, they are formalized in terms of coinductive lists. Thus, our theory is based on Lochbihler's [1] formalization of coinductive lists. The theory may be applied to verify properties for dynamic architectures.

Contents

1 A Theory of Dynamic Architectures

The following theory formalizes configuration traces [4, 5] as a model for dynamic architectures. Since configuration traces may be finite as well as infinite, the theory depends on Lochbihler's theory of co-inductive lists [1].

theory *Configuration-Traces*
  imports *Coinductive.Coinductive-List*
begin

In the following we first provide some preliminary results for natural numbers, extended natural numbers, and lazy lists. Then, we introduce a locale @textdynamic_architectures which introduces basic definitions and corresponding properties for dynamic architectures.

1.1 Natural Numbers

We provide one additional property for natural numbers.

lemma *boundedGreatest*:
  assumes $P\ (i{::}nat)$
    and $\forall\, n' > n.\ \neg\ P\ n'$
  shows $\exists\, i'{\leq}n.\ P\ i' \wedge (\forall\, n'.\ P\ n' \longrightarrow n'{\leq}i')$
⟨*proof*⟩

1.2 Extended Natural Numbers

We provide one simple property for the *strict* order over extended natural numbers.

lemma *enat-min*:
  assumes $m \geq enat\ n'$
    and $enat\ n < m - enat\ n'$
  shows $enat\ n + enat\ n' < m$
  ⟨*proof*⟩

1.3 Lazy Lists

In the following we provide some additional notation and properties for lazy lists.

notation *LNil* $([]_l)$
notation *LCons* (**infixl** $\#_l$ *60*)
notation *lappend* (**infixl** $@_l$ *60*)

lemma *lnth-lappend*[*simp*]:
  assumes *lfinite xs*
    and $\neg$ *lnull ys*
  shows *lnth* (*xs* $@_l$ *ys*) (*the-enat* (*llength xs*)) = *lhd ys*
⟨*proof*⟩

lemma *lfilter-ltake*:
  assumes $\forall\,(n{::}nat){\leq}llength\ xs.\ n{\geq}i \longrightarrow (\neg\ P\ (lnth\ xs\ n))$
  shows *lfilter P xs* = *lfilter P* (*ltake i xs*)
⟨*proof*⟩

lemma *lfilter-lfinite*[*simp*]:
  assumes *lfinite* (*lfilter P t*)
    and $\neg$ *lfinite t*

shows $\exists\, n.\ \forall\, n'{>}n.\ \neg\ P\ (lnth\ t\ n')$
⟨*proof*⟩

1.4 Specifying Dynamic Architectures

In the following we formalize dynamic architectures in terms of configuration traces, i.e., sequences of architecture configurations. Moreover, we introduce definitions for operations to support the specification of configuration traces.

typedecl *cnf*
type-synonym *trace* = *nat* ⇒ *cnf*
consts *arch*:: *trace set*

type-synonym *cta* = *trace* ⇒ *nat* ⇒ *bool*

1.4.1 Implication

definition *imp* :: *cta* ⇒ *cta* ⇒ *cta* (**infixl** $\longrightarrow^c$ *10*)
where $\gamma \longrightarrow^c \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \longrightarrow \gamma'\ t\ n$

declare *imp-def*[*simp*]

lemma *impI*[*intro*!]:
fixes *t n*
assumes $\gamma\ t\ n \Longrightarrow \gamma'\ t\ n$
shows $(\gamma \longrightarrow^c \gamma')\ t\ n$ ⟨*proof*⟩

lemma *impE*[*elim*!]:
fixes *t n*
assumes $(\gamma \longrightarrow^c \gamma')\ t\ n$ **and** $\gamma\ t\ n$ **and** $\gamma'\ t\ n \Longrightarrow \gamma''\ t\ n$
shows $\gamma''\ t\ n$ ⟨*proof*⟩

1.4.2 Disjunction

definition *disj* :: *cta* ⇒ *cta* ⇒ *cta* (**infixl** $\vee^c$ *15*)
where $\gamma \vee^c \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \vee \gamma'\ t\ n$

declare *disj-def*[*simp*]

lemma *disjI1*[*intro*]:
assumes $\gamma\ t\ n$
shows $(\gamma \vee^c \gamma')\ t\ n$ ⟨*proof*⟩

lemma *disjI2*[*intro*]:
assumes $\gamma'\ t\ n$
shows $(\gamma \vee^c \gamma')\ t\ n$ ⟨*proof*⟩

lemma *disjE*[*elim*!]:
assumes $(\gamma \vee^c \gamma')\ t\ n$
and $\gamma\ t\ n \Longrightarrow \gamma''\ t\ n$
and $\gamma'\ t\ n \Longrightarrow \gamma''\ t\ n$
shows $\gamma''\ t\ n$ ⟨*proof*⟩

1.4.3 Conjunction

definition *conj* :: *cta* ⇒ *cta* ⇒ *cta* (**infixl** $\wedge^c$ *20*)

where $\gamma \wedge^c \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \wedge \gamma'\ t\ n$

declare *conj-def*[*simp*]

lemma *conjI*[*intro*!]:
fixes n
assumes $\gamma\ t\ n$ **and** $\gamma'\ t\ n$
shows $(\gamma \wedge^c \gamma')\ t\ n$ ⟨*proof*⟩

lemma *conjE*[*elim*!]:
fixes n
assumes $(\gamma \wedge^c \gamma')\ t\ n$ **and** $\gamma\ t\ n \Longrightarrow \gamma'\ t\ n \Longrightarrow \gamma''\ t\ n$
shows $\gamma''\ t\ n$ ⟨*proof*⟩

1.4.4 Negation

definition *neg* :: *cta* ⇒ *cta* ($\neg^c$ - [*19*] *19*)
where $\neg^c\ \gamma \equiv \lambda\ t\ n.\ \neg\ \gamma\ t\ n$

declare *neg-def*[*simp*]

lemma *negI*[*intro*!]:
assumes $\gamma\ t\ n \Longrightarrow$ *False*
shows $(\neg^c\ \gamma)\ t\ n$ ⟨*proof*⟩

lemma *negE*[*elim*!]:
assumes $(\neg^c\ \gamma)\ t\ n$
and $\gamma\ t\ n$
shows $\gamma'\ t\ n$ ⟨*proof*⟩

1.4.5 Quantifiers

definition *all* :: ('*a* ⇒ *cta*)
⇒ *cta* (**binder** $\forall_c$ *10*)
where *all* $P \equiv \lambda t\ n.\ (\forall y.\ (P\ y\ t\ n))$

declare *all-def*[*simp*]

lemma *allI*[*intro*!]:
assumes $\bigwedge x.\ \gamma\ x\ t\ n$
shows $(\forall_c x.\ \gamma\ x)\ t\ n$ ⟨*proof*⟩

lemma *allE*[*elim*!]:
fixes n
assumes $(\forall_c x.\ \gamma\ x)\ t\ n$ **and** $\gamma\ x\ t\ n \Longrightarrow \gamma'\ t\ n$
shows $\gamma'\ t\ n$ ⟨*proof*⟩

definition *ex* :: ('*a* ⇒ *cta*)
⇒ *cta* (**binder** $\exists_c$ *10*)
where *ex* $P \equiv \lambda t\ n.\ (\exists y.\ (P\ y\ t\ n))$

declare *ex-def*[*simp*]

lemma *exI*[*intro*!]:
assumes $\gamma\ x\ t\ n$
shows $(\exists_c x.\ \gamma\ x)\ t\ n$ ⟨*proof*⟩

lemma *exE*[*elim*!]:
assumes $(\exists_c x.\ \gamma\ x)\ t\ n$ **and** $\bigwedge x.\ \gamma\ x\ t\ n \Longrightarrow \gamma'\ t\ n$
shows $\gamma'\ t\ n$ ⟨*proof*⟩

1.4.6 Atomic Assertions

First we provide rules for basic behavior assertions.

definition *ca* :: (*cnf* ⇒ *bool*) ⇒ *cta*
where $ca\ \varphi \equiv \lambda\ t\ n.\ \varphi\ (t\ n)$

lemma *caI*[*intro*]:
fixes *n*
assumes $\varphi\ (t\ n)$
shows $(ca\ \varphi)\ t\ n$ ⟨*proof*⟩

lemma *caE*[*elim*]:
fixes *n*
assumes $(ca\ \varphi)\ t\ n$
shows $\varphi\ (t\ n)$ ⟨*proof*⟩

1.4.7 Next Operator

definition *nxt* :: *cta* ⇒ *cta* ($\bigcirc_c$(-) *24*)
where $\bigcirc_c(\gamma) \equiv \lambda(t{::}(nat \Rightarrow cnf))\ n.\ \gamma\ t\ (Suc\ n)$

1.4.8 Eventually Operator

definition *evt* :: *cta* ⇒ *cta* ($\Diamond_c$(-) *23*)
where $\Diamond_c(\gamma) \equiv \lambda(t{::}(nat \Rightarrow cnf))\ n.\ \exists\, n'{\geq}n.\ \gamma\ t\ n'$

1.4.9 Globally Operator

definition *glob* :: *cta* ⇒ *cta* ($\Box_c$(-) *22*)
where $\Box_c(\gamma) \equiv \lambda(t{::}(nat \Rightarrow cnf))\ n.\ \forall\, n'{\geq}n.\ \gamma\ t\ n'$

lemma *globI*[*intro*!]:
fixes n'
assumes $\forall\, n{\geq}n'.\ \gamma\ t\ n$
shows $(\Box_c(\gamma))\ t\ n'$ ⟨*proof*⟩

lemma *globE*[*elim*!]:
fixes $n\ n'$
assumes $(\Box_c(\gamma))\ t\ n$ **and** $n'{\geq}n$
shows $\gamma\ t\ n'$ ⟨*proof*⟩

1.4.10 Until Operator

definition *until* :: *cta* ⇒ *cta* ⇒ *cta* (**infixl** $\mathfrak{U}_c$ *21*)
where $\gamma'\ \mathfrak{U}_c\ \gamma \equiv \lambda(t{::}(nat \Rightarrow cnf))\ n.\ \exists\, n''{\geq}n.\ \gamma\ t\ n'' \wedge (\forall\, n'{\geq}n.\ n' < n'' \longrightarrow \gamma'\ t\ n')$

lemma *untilI*[*intro*]:
fixes *n*
assumes $\exists\, n''{\geq}n.\ \gamma\ t\ n'' \wedge (\forall\, n'{\geq}n.\ n'{<}n'' \longrightarrow \gamma'\ t\ n')$
shows $(\gamma'\ \mathfrak{U}_c\ \gamma)\ t\ n$ ⟨*proof*⟩

lemma *untilE*[*elim*]:
fixes n
assumes $(\gamma'\ \mathfrak{U}_c\ \gamma)\ t\ n$
shows $\exists\, n''{\geq}n.\ \gamma\ t\ n'' \wedge (\forall\, n'{\geq}n.\ n'{<}n'' \longrightarrow \gamma'\ t\ n')$ ⟨*proof*⟩

1.4.11 Weak Until

definition *wuntil* :: *cta* ⇒ *cta* ⇒ *cta* (**infixl** $\mathfrak{W}_c$ *20*)
where $\gamma'\ \mathfrak{W}_c\ \gamma \equiv \gamma'\ \mathfrak{U}_c\ \gamma \vee^c \Box_c(\gamma')$

lemma *wUntilI*[*intro*]:
fixes n
assumes $(\exists\, n''{\geq}n.\ \gamma\ t\ n'' \wedge (\forall\, n'{\geq}n.\ n'{<}n'' \longrightarrow \gamma'\ t\ n')) \vee (\forall\, n'{\geq}n.\ \gamma'\ t\ n')$
shows $(\gamma'\ \mathfrak{W}_c\ \gamma)\ t\ n$ ⟨*proof*⟩

lemma *wUntilE*[*elim*]:
fixes $n\ n'$
assumes $(\gamma'\ \mathfrak{W}_c\ \gamma)\ t\ n$
shows $(\exists\, n''{\geq}n.\ \gamma\ t\ n'' \wedge (\forall\, n'{\geq}n.\ n'{<}n'' \longrightarrow \gamma'\ t\ n')) \vee (\forall\, n'{\geq}n.\ \gamma'\ t\ n')$
⟨*proof*⟩

lemma *wUntil-Glob*:
assumes $(\gamma'\ \mathfrak{W}_c\ \gamma)\ t\ n$
and $(\Box_c(\gamma' \longrightarrow^c \gamma''))\ t\ n$
shows $(\gamma''\ \mathfrak{W}_c\ \gamma)\ t\ n$
⟨*proof*⟩

1.5 Dynamic Components

To support the specification of patterns over dynamic architectures we provide a locale for dynamic components. It takes the following type parameters:

- id: a type for component identifiers
- cmp: a type for components
- cnf: a type for architecture configurations

locale *dynamic-component* =
fixes *tCMP* :: *'id* ⇒ *cnf* ⇒ *'cmp* (σ_(-) [*0*,*110*]*60*)
and *active* :: *'id* ⇒ *cnf* ⇒ *bool* (‖-‖_ [*0*,*110*]*60*)
begin

The locale requires two parameters:

- *tCMP* is an operator to obtain a component with a certain identifier from an architecture configuration.
- *active* is a predicate to assert whether a certain component is activated within an architecture configuration.

The locale provides some general properties about its parameters and introduces six important operators over configuration traces:

- An operator to extract the behavior of a certain component out of a given configuration trace.

- An operator to obtain the number of activations of a certain component within a given configuration trace.
- An operator to obtain the least point in time (before a certain point in time) from which on a certain component is not activated anymore.
- An operator to obtain the latest point in time where a certain component was activated.
- Two operators to map time-points between configuration traces and behavior traces.

Moreover, the locale provides several properties about the operators and their relationships.

lemma *nact-active*:
fixes t::$nat \Rightarrow cnf$
and n::nat
and n''
and id
assumes $\|id\|_{t\ n}$
and $n'' \geq n$
and $\neg\ (\exists\, n' \geq n.\ n' < n'' \wedge \|id\|_{t\ n'})$
shows $n{=}n''$
⟨*proof*⟩

lemma *nact-exists*:
fixes t::$nat \Rightarrow cnf$
assumes $\exists\, i \geq n.\ \|c\|_{t\ i}$
shows $\exists\, i \geq n.\ \|c\|_{t\ i} \wedge (\nexists k.\ n \leq k \wedge k < i \wedge \|c\|_{t\ k})$
⟨*proof*⟩

lemma *lActive-least*:
assumes $\exists\, i \geq n.\ i < llength\ t \wedge \|c\|_{lnth\ t\ i}$
shows $\exists\, i \geq n.\ (i < llength\ t \wedge \|c\|_{lnth\ t\ i} \wedge (\nexists k.\ n \leq k \wedge k < i \wedge k < llength\ t \wedge \|c\|_{lnth\ t\ k}))$
⟨*proof*⟩

1.6 Projection

In the following we introduce an operator which extracts the behavior of a certain component out of a given configuration trace.

definition *proj*:: $'id \Rightarrow (cnf\ llist) \Rightarrow ('cmp\ llist)$ $(\pi_{\text{-}}(\text{-})\ [0,110]60)$
where $proj\ c = lmap\ (\lambda cnf.\ (\sigma_c(cnf))) \circ (lfilter\ (active\ c))$

lemma *proj-lnil* [*simp*,*intro*]:
$\pi_c([]_l) = []_l$ ⟨*proof*⟩

lemma *proj-lnull* [*simp*]:
$\pi_c(t) = []_l \longleftrightarrow (\forall\, k \in lset\ t.\ \neg\ \|c\|_k)$
⟨*proof*⟩

lemma *proj-LCons* [*simp*]:
$\pi_i(x\ \#_l\ xs) = (if\ \|i\|_x\ then\ (\sigma_i(x))\ \#_l\ (\pi_i(xs))\ else\ \pi_i(xs))$
⟨*proof*⟩

lemma *proj-llength*[*simp*]:
$llength\ (\pi_c(t)) \leq llength\ t$
⟨*proof*⟩

lemma *proj-ltake*:
assumes $\forall (n'::nat) \leq llength\ t.\ n' \geq n \longrightarrow (\neg \|c\|_{lnth\ t\ n'})$
shows $\pi_c(t) = \pi_c(ltake\ n\ t)$ ⟨*proof*⟩

lemma *proj-finite-bound*:
assumes *lfinite* $(\pi_c(inf\text{-}llist\ t))$
shows $\exists n.\ \forall n' > n.\ \neg \|c\|_{t\ n'}$
⟨*proof*⟩

1.6.1 Monotonicity and Continuity

lemma *proj-mcont*:
shows *mcont lSup lprefix lSup lprefix* (*proj c*)
⟨*proof*⟩

lemma *proj-mcont2mcont*:
assumes *mcont lub ord lSup lprefix f*
shows *mcont lub ord lSup lprefix* $(\lambda x.\ \pi_c(f\ x))$
⟨*proof*⟩

lemma *proj-mono-prefix*[*simp*]:
assumes *lprefix t t'*
shows *lprefix* $(\pi_c(t))$ $(\pi_c(t'))$
⟨*proof*⟩

1.6.2 Finiteness

lemma *proj-finite*[*simp*]:
assumes *lfinite t*
shows *lfinite* $(\pi_c(t))$
⟨*proof*⟩

lemma *proj-finite2*:
assumes $\forall (n'::nat) \leq llength\ t.\ n' \geq n \longrightarrow (\neg \|c\|_{lnth\ t\ n'})$
shows *lfinite* $(\pi_c(t))$ ⟨*proof*⟩

lemma *proj-append-lfinite*[*simp*]:
fixes *t t'*
assumes *lfinite t*
shows $\pi_c(t\ @_l\ t') = (\pi_c(t))\ @_l\ (\pi_c(t'))$ (**is** *?lhs=?rhs*)
⟨*proof*⟩

lemma *proj-one*:
assumes $\exists i.\ i < llength\ t \wedge \|c\|_{lnth\ t\ i}$
shows $llength\ (\pi_c(t)) \geq 1$
⟨*proof*⟩

1.6.3 Projection not Active

lemma *proj-not-active*[*simp*]:
assumes $enat\ n < llength\ t$
and $\neg \|c\|_{lnth\ t\ n}$
shows $\pi_c(ltake\ (Suc\ n)\ t) = \pi_c(ltake\ n\ t)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *proj-not-active-same*:

assumes *enat* $n \leq (n'::enat)$
and $\neg$ *lfinite* $t \vee n'-1 <$ *llength* t
and $\nexists k.\ k \geq n \wedge k < n' \wedge k <$ *llength* $t \wedge \|c\|_{lnth\ t\ k}$
shows $\pi_c(ltake\ n'\ t) = \pi_c(ltake\ n\ t)$
⟨*proof*⟩

1.6.4 Projection Active

lemma *proj-active*[*simp*]:
assumes *enat* $i <$ *llength* $t\ \|c\|_{lnth\ t\ i}$
shows $\pi_c(ltake\ (Suc\ i)\ t) = (\pi_c(ltake\ i\ t))\ @_l\ ((\sigma_c(lnth\ t\ i))\ \#_l\ []_l)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *proj-active-append*:
assumes *a1*: $(n::nat) \leq i$
and *a2*: *enat* $i <$ $(n'::enat)$
and *a3*: $\neg$ *lfinite* $t \vee n'-1 <$ *llength* t
and *a4*: $\|c\|_{lnth\ t\ i}$
and $\forall i'.\ (n \leq i' \wedge$ *enat* $i' < n' \wedge i' <$ *llength* $t \wedge \|c\|_{lnth\ t\ i'}) \longrightarrow (i' = i)$
shows $\pi_c(ltake\ n'\ t) = (\pi_c(ltake\ n\ t))\ @_l\ ((\sigma_c(lnth\ t\ i))\ \#_l\ []_l)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

1.6.5 Same and not Same

lemma *proj-same-not-active*:
assumes $n \leq n'$
and *enat* $(n'-1) <$ *llength* t
and $\pi_c(ltake\ n'\ t) = \pi_c(ltake\ n\ t)$
shows $\nexists k.\ k \geq n \wedge k < n' \wedge \|c\|_{lnth\ t\ k}$
⟨*proof*⟩

lemma *proj-not-same-active*:
assumes *enat* $n \leq (n'::enat)$
and $(\neg$ *lfinite* $t) \vee n'-1 <$ *llength* t
and $\neg(\pi_c(ltake\ n'\ t) = \pi_c(ltake\ n\ t))$
shows $\exists k.\ k \geq n \wedge k < n' \wedge$ *enat* $k <$ *llength* $t \wedge \|c\|_{lnth\ t\ k}$
⟨*proof*⟩

1.7 Activations

We also introduce an operator to obtain the number of activations of a certain component within a given configuration trace.

definition *nAct* :: *'id* ⇒ *enat* ⇒ (*cnf llist*) ⇒ *enat* (⟨- #_ -⟩) **where**
$\langle c\ \#_n\ t\rangle \equiv$ *llength* $(\pi_c(ltake\ n\ t))$

lemma *nAct-0*[*simp*]:
$\langle c\ \#_0\ t\rangle = 0$ ⟨*proof*⟩

lemma *nAct-NIL*[*simp*]:
$\langle c\ \#_n\ []_l\rangle = 0$ ⟨*proof*⟩

lemma *nAct-Null*:
assumes *llength* $t \geq n$
and $\langle c\ \#_n\ t\rangle = 0$
shows $\forall i < n.\ \neg\ \|c\|_{lnth\ t\ i}$

⟨*proof*⟩

lemma *nAct-ge-one*[*simp*]:
assumes *llength t* ≥ *n*
and *i* < *n*
and ‖*c*‖$_{lnth\ t\ i}$
shows ⟨*c* #$_n$ *t*⟩ ≥ *enat 1*
⟨*proof*⟩

lemma *nAct-finite*[*simp*]:
assumes *n* ≠ ∞
shows ∃ *n'*. ⟨*c* #$_n$ *t*⟩ = *enat n'*
⟨*proof*⟩

lemma *nAct-enat-the-nat*[*simp*]:
assumes *n* ≠ ∞
shows *enat* (*the-enat* (⟨*c* #$_n$ *t*⟩)) = ⟨*c* #$_n$ *t*⟩
⟨*proof*⟩

1.7.1 Monotonicity and Continuity

lemma *nAct-mcont*:
shows *mcont lSup lprefix Sup* (≤) (*nAct c n*)
⟨*proof*⟩

lemma *nAct-mono*:
assumes *n* ≤ *n'*
shows ⟨*c* #$_n$ *t*⟩ ≤ ⟨*c* #$_{n'}$ *t*⟩
⟨*proof*⟩

lemma *nAct-strict-mono-back*:
assumes ⟨*c* #$_n$ *t*⟩ < ⟨*c* #$_{n'}$ *t*⟩
shows *n* < *n'*
⟨*proof*⟩

1.7.2 Not Active

lemma *nAct-not-active*[*simp*]:
fixes *n*::*nat*
and *n'*::*nat*
and *t*::(*cnf llist*)
and *c*::*'id*
assumes *enat i* < *llength t*
and ¬ ‖*c*‖$_{lnth\ t\ i}$
shows ⟨*c* #$_{Suc\ i}$ *t*⟩ = ⟨*c* #$_i$ *t*⟩
⟨*proof*⟩

lemma *nAct-not-active-same*:
assumes *enat n* ≤ (*n'*::*enat*)
and *n'*−*1* < *llength t*
and ∄ *k*. *enat k*≥*n* ∧ *k*<*n'* ∧ ‖*c*‖$_{lnth\ t\ k}$
shows ⟨*c* #$_{n'}$ *t*⟩ = ⟨*c* #$_n$ *t*⟩
⟨*proof*⟩

1.7.3 Active

lemma *nAct-active*[*simp*]:
fixes n::*nat*
and n'::*nat*
and t::(*cnf llist*)
and c::'*id*
assumes *enat* i < *llength* t
and $\|c\|_{lnth\ t\ i}$
shows $\langle c\ \#_{Suc\ i}\ t\rangle$ = *eSuc* $(\langle c\ \#_i\ t\rangle)$
⟨*proof*⟩

lemma *nAct-active-suc*:
fixes n::*nat*
and n'::*enat*
and t::(*cnf llist*)
and c::'*id*
assumes ¬ *lfinite* t ∨ $n'-1$ < *llength* t
and $n \leq i$
and *enat* i < n'
and $\|c\|_{lnth\ t\ i}$
and $\forall i'.\ (n \leq i' \wedge$ *enat* $i' < n' \wedge i'$ < *llength* $t \wedge \|c\|_{lnth\ t\ i'}) \longrightarrow (i' = i)$
shows $\langle c\ \#_{n'}\ t\rangle$ = *eSuc* $(\langle c\ \#_n\ t\rangle)$
⟨*proof*⟩

lemma *nAct-less*:
assumes *enat* k < *llength* t
and $n \leq k$
and k < (n'::*enat*)
and $\|c\|_{lnth\ t\ k}$
shows $\langle c\ \#_n\ t\rangle < \langle c\ \#_{n'}\ t\rangle$
⟨*proof*⟩

lemma *nAct-less-active*:
assumes $n' - 1$ < *llength* t
and $\langle c\ \#_{enat\ n}\ t\rangle < \langle c\ \#_{n'}\ t\rangle$
shows $\exists\, i{\geq}n.\ i{<}n' \wedge \|c\|_{lnth\ t\ i}$
⟨*proof*⟩

1.7.4 Same and Not Same

lemma *nAct-same-not-active*:
assumes $\langle c\ \#_{n'}$ *inf-llist* $t\rangle = \langle c\ \#_n$ *inf-llist* $t\rangle$
shows $\forall\, k{\geq}n.\ k{<}n' \longrightarrow \neg\ \|c\|_{t\ k}$
⟨*proof*⟩

lemma *nAct-not-same-active*:
assumes $\langle c\ \#_{enat\ n}\ t\rangle < \langle c\ \#_{n'}\ t\rangle$
and ¬ *lfinite* t ∨ $n' - 1$ < *llength* t
shows $\exists\,(i{::}nat){\geq}n.$ *enat* $i{<}$ $n' \wedge i{<}$*llength* $t \wedge \|c\|_{lnth\ t\ i}$
⟨*proof*⟩

lemma *nAct-less-llength-active*:
assumes x < *llength* $(\pi_c(t))$
and *enat* $x = \langle c\ \#_{enat\ n'}\ t\rangle$
shows $\exists\,(i{::}nat){\geq}n'.\ i{<}$*llength* $t \wedge \|c\|_{lnth\ t\ i}$

⟨*proof*⟩

lemma *nAct-exists*:
assumes $x < llength\ (\pi_c(t))$
shows $\exists\,(n'::nat).\ enat\ x = \langle c\ \#_{n'}\ t\rangle$
⟨*proof*⟩

1.8 Projection and Activation

In the following we provide some properties about the relationship between the projection and activations operator.

lemma *nAct-le-proj*:
$\langle c\ \#_n\ t\rangle \leq llength\ (\pi_c(t))$
⟨*proof*⟩

lemma *proj-nAct*:
assumes $(enat\ n < llength\ t)$
shows $\pi_c(ltake\ n\ t) = ltake\ (\langle c\ \#_n\ t\rangle)\ (\pi_c(t))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *proj-active-nth*:
assumes $enat\ (Suc\ i) < llength\ t\ \|c\|_{lnth\ t\ i}$
shows $lnth\ (\pi_c(t))\ (the\text{-}enat\ (\langle c\ \#_i\ t\rangle)) = \sigma_c(lnth\ t\ i)$
⟨*proof*⟩

lemma *nAct-eq-proj*:
assumes $\neg(\exists\, i{\geq}n.\ \|c\|_{lnth\ t\ i})$
shows $\langle c\ \#_n\ t\rangle = llength\ (\pi_c(t))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *nAct-llength-proj*:
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
shows $llength\ (\pi_c(inf\text{-}llist\ t)) \geq eSuc\ (\langle c\ \#_n\ inf\text{-}llist\ t\rangle)$
⟨*proof*⟩

1.9 Least not Active

In the following, we introduce an operator to obtain the least point in time before a certain point in time where a component was deactivated.

definition *lNAct* :: *'id* ⇒ (*nat* ⇒ *cnf*) ⇒ *nat* ⇒ *nat* (⟨- ⇐ -⟩-)
where $\langle c \Leftarrow t\rangle_n \equiv (LEAST\ n'.\ n{=}n' \vee (n'{<}n \wedge (\nexists\, k.\ k{\geq}n' \wedge k{<}n \wedge \|c\|_{t\ k})))$

lemma *lNact0*[*simp*]:
$\langle c \Leftarrow t\rangle_0 = 0$
⟨*proof*⟩

lemma *lNact-least*:
assumes $n{=}n' \vee n'{<}n \wedge (\nexists\, k.\ k{\geq}n' \wedge k{<}n \wedge \|c\|_{t\ k})$
shows $\langle c \Leftarrow t\rangle_n \leq n'$
⟨*proof*⟩

lemma *lNAct-ex*: $\langle c \Leftarrow t\rangle_n{=}n \vee \langle c \Leftarrow t\rangle_n{<}n \wedge (\nexists\, k.\ k{\geq}\langle c \Leftarrow t\rangle_n \wedge k{<}n \wedge \|c\|_{t\ k})$
⟨*proof*⟩

lemma *lNact-notActive*:
fixes $c\ t\ n\ k$
assumes $k \geq \langle c \Leftarrow t \rangle_n$
and $k < n$
shows $\neg \|c\|_{t\ k}$
⟨*proof*⟩

lemma *lNactGe*:
fixes $c\ t\ n\ n'$
assumes $n' \geq \langle c \Leftarrow t \rangle_n$
and $\|c\|_{t\ n'}$
shows $n' \geq n$
⟨*proof*⟩

lemma *lNactLe*[*simp*]:
fixes $n\ n'$
shows $\langle c \Leftarrow t \rangle_n \leq n$
⟨*proof*⟩

lemma *lNactLe-nact*:
fixes $n\ n'$
assumes $n'=n \vee (n'<n \wedge (\nexists k.\ k \geq n' \wedge k<n \wedge \|c\|_{t\ k}))$
shows $\langle c \Leftarrow t \rangle_n \leq n'$
⟨*proof*⟩

lemma *lNact-active*:
fixes *cid t n*
assumes $\forall k<n.\ \|cid\|_{t\ k}$
shows $\langle cid \Leftarrow t \rangle_n = n$
⟨*proof*⟩

lemma *nAct-mono-back*:
fixes $c\ t$ **and** n **and** n'
assumes $\langle c\ \#_{n'}\ inf\text{-}llist\ t \rangle \geq \langle c\ \#_n\ inf\text{-}llist\ t \rangle$
shows $n' \geq \langle c \Leftarrow t \rangle_n$
⟨*proof*⟩

lemma *nAct-mono-lNact*:
assumes $\langle c \Leftarrow t \rangle_n \leq n'$
shows $\langle c\ \#_n\ inf\text{-}llist\ t \rangle \leq \langle c\ \#_{n'}\ inf\text{-}llist\ t \rangle$
⟨*proof*⟩

1.10 Next Active

In the following, we introduce an operator to obtain the next point in time when a component is activated.

definition *nxtAct* :: *'id* ⇒ (*nat* ⇒ *cnf*) ⇒ *nat* ⇒ *nat* (⟨- → -⟩-)
where $\langle c \rightarrow t \rangle_n \equiv (THE\ n'.\ n' \geq n \wedge \|c\|_{t\ n'} \wedge (\nexists k.\ k \geq n \wedge k<n' \wedge \|c\|_{t\ k}))$

lemma *nxtActI*:
fixes *n*::*nat*
and *t*::*nat* ⇒ *cnf*
and *c*::*'id*
assumes $\exists i \geq n.\ \|c\|_{t\ i}$
shows $\langle c \rightarrow t \rangle_n \geq n \wedge \|c\|_{t\ \langle c \rightarrow t \rangle_n} \wedge (\nexists k.\ k \geq n \wedge k<\langle c \rightarrow t \rangle_n \wedge \|c\|_{t\ k})$

⟨*proof*⟩

lemma *nxtActLe*:
fixes $n\ n'$
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
shows $n \leq \langle c \to t\rangle_n$
⟨*proof*⟩

lemma *nxtAct-eq*:
assumes $n'{\geq}n$
and $\|c\|_{t\ n'}$
and $\forall\, n''{\geq}n.\ n''{<}n' \longrightarrow \neg\ \|c\|_{t\ n''}$
shows $n' = \langle c \to t\rangle_n$
⟨*proof*⟩

lemma *nxtAct-active*:
fixes $i{::}nat$
and $t{::}nat \Rightarrow cnf$
and $c{::}'id$
assumes $\|c\|_{t\ i}$
shows $\langle c \to t\rangle_i = i$ ⟨*proof*⟩

lemma *nxtActive-no-active*:
assumes $\exists!i.\ i{\geq}n \wedge \|c\|_{t\ i}$
shows $\neg\ (\exists\, i'{\geq}Suc\ \langle c \to t\rangle_n.\ \|c\|_{t\ i'})$
⟨*proof*⟩

lemma *nxt-geq-lNact*[*simp*]:
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
shows $\langle c \to t\rangle_n{\geq}\langle c \Leftarrow t\rangle_n$
⟨*proof*⟩

lemma *active-geq-nxtAct*:
assumes $\|c\|_{t\ i}$
and *the-enat* $(\langle c\ \#_i\ inf\text{-}llist\ t\rangle){\geq}$*the-enat* $(\langle c\ \#_n\ inf\text{-}llist\ t\rangle)$
shows $i{\geq}\langle c \to t\rangle_n$
⟨*proof*⟩

lemma *nAct-same*:
assumes $\langle c \Leftarrow t\rangle_n \leq n'$ **and** $n' \leq \langle c \to t\rangle_n$
shows *the-enat* $(\langle c\ \#_{enat\ n'}\ inf\text{-}llist\ t\rangle) =$ *the-enat* $(\langle c\ \#_{enat\ n}\ inf\text{-}llist\ t\rangle)$
⟨*proof*⟩

lemma *nAct-mono-nxtAct*:
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
and $\langle c \to t\rangle_n \leq n'$
shows $\langle c\ \#_n\ inf\text{-}llist\ t\rangle \leq \langle c\ \#_{n'}\ inf\text{-}llist\ t\rangle$
⟨*proof*⟩

1.11 Latest Activation

In the following, we introduce an operator to obtain the latest point in time when a component is activated.

abbreviation *latestAct-cond*:: $'id \Rightarrow trace \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where *latestAct-cond* $c\ t\ n\ n' \equiv n'{<}n \wedge \|c\|_{t\ n'}$

definition *latestAct*:: *'id* ⇒ *trace* ⇒ *nat* ⇒ *nat* (⟨- ← -⟩_)
where *latestAct c t n* = (*GREATEST n'*. *latestAct-cond c t n n'*)

lemma *latestActEx*:
assumes ∃ *n'*<*n*. ‖*nid*‖$_{t\ n'}$
shows ∃ *n'*. *latestAct-cond nid t n n'* ∧ (∀ *n''*. *latestAct-cond nid t n n''* ⟶ *n''* ≤ *n'*)
⟨*proof*⟩

lemma *latestAct-prop*:
assumes ∃ *n'*<*n*. ‖*nid*‖$_{t\ n'}$
shows ‖*nid*‖$_{t\ (latestAct\ nid\ t\ n)}$ **and** *latestAct nid t n*<*n*
⟨*proof*⟩

lemma *latestAct-less*:
assumes *latestAct-cond nid t n n'*
shows *n'* ≤ ⟨*nid* ← *t*⟩*n*
⟨*proof*⟩

lemma *latestActNxt*:
assumes ∃ *n'*<*n*. ‖*nid*‖$_{t\ n'}$
shows ⟨*nid* → *t*⟩$_{\langle nid \leftarrow t\rangle n}$=⟨*nid* ← *t*⟩*n*
⟨*proof*⟩

lemma *latestActNxtAct*:
assumes ∃ *n'*≥*n*. ‖*tid*‖$_{t\ n'}$
and ∃ *n'*<*n*. ‖*tid*‖$_{t\ n'}$
shows ⟨*tid* → *t*⟩*n* > ⟨*tid* ← *t*⟩*n*
⟨*proof*⟩

lemma *latestActless*:
assumes ∃ *n'*≥n_s. *n'*<*n* ∧ ‖*nid*‖$_{t\ n'}$
shows ⟨*nid* ← *t*⟩*n*≥n_s
⟨*proof*⟩

lemma *latestActEq*:
fixes *nid*::*'id*
assumes ‖*nid*‖$_{t\ n'}$ **and** ¬(∃ *n''*>*n'*. *n''*<*n* ∧ ‖*nid*‖$_{t\ n'}$) **and** *n'*<*n*
shows ⟨*nid* ← *t*⟩*n* = *n'*
⟨*proof*⟩

1.12 Last Activation

In the following we introduce an operator to obtain the latest point in time where a certain component was activated within a certain configuration trace.

definition *lActive* :: *'id* ⇒ (*nat* ⇒ *cnf*) ⇒ *nat* (⟨- ∧ -⟩)
where ⟨*c* ∧ *t*⟩ ≡ (*GREATEST i*. ‖*c*‖$_{t\ i}$)

lemma *lActive-active*:
assumes ‖*c*‖$_{t\ i}$
and ∀ *n'* > *n*. ¬ (‖*c*‖$_{t\ n'}$)
shows ‖*c*‖$_{t\ (\langle c \wedge t\rangle)}$
⟨*proof*⟩

lemma *lActive-less*:
assumes $\|c\|_{t\ i}$
and $\forall\, n' > n.\ \neg\ (\|c\|_{t\ n'})$
shows $\langle c \wedge t\rangle \leq n$
⟨*proof*⟩

lemma *lActive-greatest*:
assumes $\|c\|_{t\ i}$
and $\forall\, n' > n.\ \neg\ (\|c\|_{t\ n'})$
shows $i \leq \langle c \wedge t\rangle$
⟨*proof*⟩

lemma *lActive-greater-active*:
assumes $n > \langle c \wedge t\rangle$
and $\forall\, n'' > n'.\ \neg\ \|c\|_{t\ n''}$
shows $\neg\ \|c\|_{t\ n}$
⟨*proof*⟩

lemma *lActive-greater-active-all*:
assumes $\forall\, n'' > n'.\ \neg\ \|c\|_{t\ n''}$
shows $\neg(\exists\, n > \langle c \wedge t\rangle.\ \|c\|_{t\ n})$
⟨*proof*⟩

lemma *lActive-equality*:
assumes $\|c\|_{t\ i}$
and $(\bigwedge x.\ \|c\|_{t\ x} \Longrightarrow x \leq i)$
shows $\langle c \wedge t\rangle = i$ ⟨*proof*⟩

lemma *nxtActive-lactive*:
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
and $\neg\ (\exists\, i{>}\langle c \rightarrow t\rangle_n.\ \|c\|_{t\ i})$
shows $\langle c \rightarrow t\rangle_n{=}\langle c \wedge t\rangle$
⟨*proof*⟩

1.13 Mapping Time Points

In the following we introduce two operators to map time-points between configuration traces and behavior traces.

1.13.1 Configuration Trace to Behavior Trace

First we provide an operator which maps a point in time of a configuration trace to the corresponding point in time of a behavior trace.

definition *cnf2bhv* :: *'id* ⇒ (*nat* ⇒ *cnf*) ⇒ *nat* ⇒ *nat* (-↓-(-) [*150*,*150*,*150*] *110*)
where $c{\downarrow}_t(n) \equiv$ *the-enat*(*llength* (π_c(*inf-llist t*))) − *1* + (n − $\langle c \wedge t\rangle$)

lemma *cnf2bhv-mono*:
assumes $n'{\geq}n$
shows $c{\downarrow}_t(n') \geq c{\downarrow}_t(n)$
⟨*proof*⟩

lemma *cnf2bhv-mono-strict*:
assumes $n{\geq}\langle c \wedge t\rangle$ **and** $n'{>}n$
shows $c{\downarrow}_t(n') > c{\downarrow}_t(n)$

⟨*proof*⟩

Note that the functions are nat, that means that also in the case the difference is negative they will return a 0!

lemma *cnf2bhv-ge-llength*[*simp*]:
assumes $n \geq \langle c \wedge t \rangle$
shows $_c\downarrow_t(n) \geq$ *the-enat*(*llength* (π_c(*inf-llist t*))) − *1*
⟨*proof*⟩

lemma *cnf2bhv-greater-llength*[*simp*]:
assumes $n > \langle c \wedge t \rangle$
shows $_c\downarrow_t(n) >$ *the-enat*(*llength* (π_c(*inf-llist t*))) − *1*
⟨*proof*⟩

lemma *cnf2bhv-suc*[*simp*]:
assumes $n \geq \langle c \wedge t \rangle$
shows $_c\downarrow_t(Suc\ n) = Suc\ (_c\downarrow_t(n))$
⟨*proof*⟩

lemma *cnf2bhv-lActive*[*simp*]:
shows $_c\downarrow_t(\langle c \wedge t \rangle) =$ *the-enat*(*llength* (π_c(*inf-llist t*))) − *1*
⟨*proof*⟩

lemma *cnf2bhv-lnth-lappend*:
assumes *act*: $\exists\, i.\ \|c\|_{t\ i}$
and *nAct*: $\nexists\, i.\ i \geq n \wedge \|c\|_{t\ i}$
shows *lnth* ((*inf-llist t*) $@_l$ (*inf-llist t'*)) ($_c\downarrow_t(n)$) = *lnth* (*inf-llist t'*) (*n* − ⟨*c* ∧ *t*⟩ − *1*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *nAct-cnf2proj-Suc-dist*:
assumes $\exists\, i \geq n.\ \|c\|_{t\ i}$
and $\neg(\exists\, i > \langle c \rightarrow t \rangle_n.\ \|c\|_{t\ i})$
shows *Suc* (*the-enat* ⟨*c* $\#_{enat\ n}$ *inf-llist t*⟩)=$_c\downarrow_t$(*Suc* ⟨*c* → *t*⟩$_n$)
⟨*proof*⟩

1.13.2 Behavior Trace to Configuration Trace

Next we define an operator to map a point in time of a behavior trace back to a corresponding point in time for a configuration trace.

definition *bhv2cnf* :: *'id* ⇒ (*nat* ⇒ *cnf*) ⇒ *nat* ⇒ *nat* (-↑-(-) [*150*,*150*,*150*] *110*)
where $_c\uparrow_t(n) \equiv \langle c \wedge t \rangle + (n - (\textit{the-enat}(\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1))$

lemma *bhv2cnf-mono*:
assumes $n' \geq n$
shows $_c\uparrow_t(n') \geq\ _c\uparrow_t(n)$
⟨*proof*⟩

lemma *bhv2cnf-mono-strict*:
assumes $n' > n$
and $n \geq$ *the-enat* (*llength* (π_c(*inf-llist t*))) − *1*
shows $_c\uparrow_t(n') >\ _c\uparrow_t(n)$
⟨*proof*⟩

Note that the functions are nat, that means that also in the case the difference is negative they

will return a 0!

lemma *bhv2cnf-ge-lActive*[*simp*]:
shows $_c\uparrow_t(n) \geq \langle c \wedge t \rangle$
⟨*proof*⟩

lemma *bhv2cnf-greater-lActive*[*simp*]:
assumes n>*the-enat*(*llength* (π_c(*inf-llist t*))) − *1*
shows $_c\uparrow_t(n) > \langle c \wedge t \rangle$
⟨*proof*⟩

lemma *bhv2cnf-lActive*[*simp*]:
assumes $\exists\, i.\ \|c\|_t\ i$
and *lfinite* (π_c(*inf-llist t*))
shows $_c\uparrow_t$(*the-enat*(*llength* (π_c(*inf-llist t*)))) = *Suc* (⟨$c \wedge t$⟩)
⟨*proof*⟩

1.13.3 Relating the Mappings

In the following we provide some properties about the relationship between the two mapping operators.

lemma *bhv2cnf-cnf2bhv*:
assumes $n \geq \langle c \wedge t \rangle$
shows $_c\uparrow_t(_c\downarrow_t(n)) = n$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *cnf2bhv-bhv2cnf*:
assumes $n \geq$ *the-enat* (*llength* (π_c(*inf-llist t*))) − *1*
shows $_c\downarrow_t(_c\uparrow_t(n)) = n$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *p2c-mono-c2p*:
assumes $n \geq \langle c \wedge t \rangle$
and $n' \geq {}_c\downarrow_t(n)$
shows $_c\uparrow_t(n') \geq n$
⟨*proof*⟩

lemma *p2c-mono-c2p-strict*:
assumes $n \geq \langle c \wedge t \rangle$
and $n <{}_c\uparrow_t(n')$
shows $_c\downarrow_t(n) < n'$
⟨*proof*⟩

lemma *c2p-mono-p2c*:
assumes $n \geq$ *the-enat* (*llength* (π_c(*inf-llist t*))) − *1*
and $n' \geq {}_c\uparrow_t(n)$
shows $_c\downarrow_t(n') \geq n$
⟨*proof*⟩

lemma *c2p-mono-p2c-strict*:
assumes $n \geq$ *the-enat* (*llength* (π_c(*inf-llist t*))) − *1*
and $n <{}_c\downarrow_t(n')$
shows $_c\uparrow_t(n) < n'$
⟨*proof*⟩

end

end

2 A Calculus for Dynamic Architectures

The following theory formalizes our calculus for dynamic architectures [2, 3] and verifies its soundness. The calculus allows to reason about temporal-logic specifications of component behavior in a dynamic setting. The theory is based on our theory of configuration traces and introduces the notion of behavior trace assertion to specify component behavior in a dynamic setting.

theory *Dynamic-Architecture-Calculus*
imports *Configuration-Traces*
begin

2.1 Extended Natural Numbers

We first provide one additional property for extended natural numbers.

lemma *the-enat-mono*[*simp*]:
assumes $m \neq \infty$
and $n \leq m$
shows *the-enat* $n \leq$ *the-enat* m
⟨*proof*⟩

2.2 Lazy Lists

Finally, we provide an additional property for lazy lists.

lemma *llength-geq-enat-lfiniteD*: *llength xs* $\leq$ *enat n* $\Longrightarrow$ *lfinite xs*
⟨*proof*⟩

context *dynamic-component*
begin

2.3 Dynamic Evaluation of Temporal Operators

In the following we introduce a function to evaluate a behavior trace assertion over a given configuration trace.

type-synonym *'c bta* = (*nat* ⇒ *'c*) ⇒ *nat* ⇒ *bool*

definition *eval*:: *'id* ⇒ (*nat* ⇒ *cnf*) ⇒ (*nat* ⇒ *'cmp*) ⇒ *nat*
⇒ *'cmp bta* ⇒ *bool*
where *eval cid t t' n* $\gamma \equiv$
$(\exists\, i{\geq}n.\ \|cid\|_{t\ i}) \wedge \gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ (the\text{-}enat(\langle cid\ \#_n\ inf\text{-}llist\ t\rangle)) \vee$
$(\exists\, i.\ \|cid\|_{t\ i}) \wedge (\nexists\, i'.\ i'{\geq}n \wedge \|cid\|_{t\ i'}) \wedge \gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ ({}_{cid}{\downarrow_t}(n)) \vee$
$(\nexists\, i.\ \|cid\|_{t\ i}) \wedge \gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ n$

eval takes a component identifier *cid*, a configuration trace t, a behavior trace t', and point in time n and evaluates behavior trace assertion γ as follows:

- If component *cid* is again activated in the future, γ is evaluated at the next point in time where *cid* is active in t.

- If component *cid* is not again activated in the future but it is activated at least once in t, then γ is evaluated at the point in time given by ${}_{cid}{\downarrow_t} n$.
- If component *cid* is never active in t, then γ is evaluated at time point n.

The following proposition evaluates definition *eval* by showing that a behavior trace assertion γ holds over configuration trace t and continuation t' whenever it holds for the concatenation of the corresponding projection with t'.

proposition *eval-corr*:
eval cid t t′ 0 γ $\longleftrightarrow$ *γ* (*lnth* ((*π*$_{cid}$(*inf-llist t*)) @$_l$ (*inf-llist t′*))) *0*
⟨*proof*⟩

2.3.1 Simplification Rules

lemma *validCI-act*[*simp*]:
assumes ∃ *i*≥*n*. ‖*cid*‖$_{t\ i}$
and *γ* (*lnth* ((*π*$_{cid}$(*inf-llist t*)) @$_l$ (*inf-llist t′*))) (*the-enat*(⟨*cid* #$_n$ *inf-llist t*⟩))
shows *eval cid t t′ n γ*
⟨*proof*⟩

lemma *validCI-cont*[*simp*]:
assumes ∃ *i*. ‖*cid*‖$_{t\ i}$
and ∄ *i′*. *i′*≥*n* ∧ ‖*cid*‖$_{t\ i'}$
and *γ* (*lnth* ((*π*$_{cid}$(*inf-llist t*)) @$_l$ (*inf-llist t′*))) (${}_{cid}{\downarrow_t}(n)$)
shows *eval cid t t′ n γ*
⟨*proof*⟩

lemma *validCI-not-act*[*simp*]:
assumes ∄ *i*. ‖*cid*‖$_{t\ i}$
and *γ* (*lnth* ((*π*$_{cid}$(*inf-llist t*)) @$_l$ (*inf-llist t′*))) *n*
shows *eval cid t t′ n γ*
⟨*proof*⟩

lemma *validCE-act*[*simp*]:
assumes ∃ *i*≥*n*. ‖*cid*‖$_{t\ i}$
and *eval cid t t′ n γ*
shows *γ* (*lnth* ((*π*$_{cid}$(*inf-llist t*)) @$_l$ (*inf-llist t′*))) (*the-enat*(⟨*cid* #$_n$ *inf-llist t*⟩))
⟨*proof*⟩

lemma *validCE-cont*[*simp*]:
assumes ∃ *i*. ‖*cid*‖$_{t\ i}$
and ∄ *i′*. *i′*≥*n* ∧ ‖*cid*‖$_{t\ i'}$
and *eval cid t t′ n γ*
shows *γ* (*lnth* ((*π*$_{cid}$(*inf-llist t*)) @$_l$ (*inf-llist t′*))) (${}_{cid}{\downarrow_t}(n)$)
⟨*proof*⟩

lemma *validCE-not-act*[*simp*]:
assumes ∄ *i*. ‖*cid*‖$_{t\ i}$
and *eval cid t t′ n γ*
shows *γ* (*lnth* ((*π*$_{cid}$(*inf-llist t*)) @$_l$ (*inf-llist t′*))) *n*
⟨*proof*⟩

2.3.2 No Activations

proposition *validity1*:

assumes $n \leq n'$
and $\exists\, i \geq n'.\ \|c\|_{t\ i}$
and $\forall\, k \geq n.\ k < n' \longrightarrow \neg\ \|c\|_{t\ k}$
shows *eval c t t' n γ* $\Longrightarrow$ *eval c t t' n' γ*
⟨*proof*⟩

proposition *validity2*:
assumes $n \leq n'$
and $\exists\, i \geq n'.\ \|c\|_{t\ i}$
and $\forall\, k \geq n.\ k < n' \longrightarrow \neg\ \|c\|_{t\ k}$
shows *eval c t t' n' γ* $\Longrightarrow$ *eval c t t' n γ*
⟨*proof*⟩

2.4 Specification Operators

In the following we introduce some basic operators for behavior trace assertions.

2.4.1 Predicates

Every predicate can be transformed to a behavior trace assertion.

definition *pred* :: *bool* ⇒ (*'cmp bta*)
where *pred P* ≡ λ *t n*. *P*

lemma *predI*[*intro*]:
fixes *cid t t' n P*
assumes *P*
shows *eval cid t t' n* (*pred P*)
⟨*proof*⟩

lemma *predE*[*elim*]:
fixes *cid t t' n P*
assumes *eval cid t t' n* (*pred P*)
shows *P*
⟨*proof*⟩

2.4.2 True and False

abbreviation *true* :: *'cmp bta*
where *true* ≡ λ*t n*. *HOL.True*

abbreviation *false* :: *'cmp bta*
where *false* ≡ λ*t n*. *HOL.False*

2.4.3 Implication

definition *imp* :: (*'cmp bta*) ⇒ (*'cmp bta*) ⇒ (*'cmp bta*) (**infixl** $\longrightarrow^b$ *10*)
where $\gamma \longrightarrow^b \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \longrightarrow \gamma'\ t\ n$

lemma *impI*[*intro*!]:
assumes *eval cid t t' n γ* $\longrightarrow$ *eval cid t t' n γ'*
shows *eval cid t t' n* ($\gamma \longrightarrow^b \gamma'$)
⟨*proof*⟩

lemma *impE*[*elim*!]:
assumes *eval cid t t' n* ($\gamma \longrightarrow^b \gamma'$)

shows *eval cid t t′ n γ* ⟶ *eval cid t t′ n γ′*
⟨*proof*⟩

2.4.4 Disjunction

definition *disj* :: (*′cmp bta*) ⇒ (*′cmp bta*) ⇒ (*′cmp bta*) (**infixl** ∨^{b} *15*)
where *γ* ∨^{b} *γ′* ≡ *λ t n. γ t n* ∨ *γ′ t n*

lemma *disjI*[*intro*!]:
assumes *eval cid t t′ n γ* ∨ *eval cid t t′ n γ′*
shows *eval cid t t′ n* (*γ* ∨^{b} *γ′*)
⟨*proof*⟩

lemma *disjE*[*elim*!]:
assumes *eval cid t t′ n* (*γ* ∨^{b} *γ′*)
shows *eval cid t t′ n γ* ∨ *eval cid t t′ n γ′*
⟨*proof*⟩

2.4.5 Conjunction

definition *conj* :: (*′cmp bta*) ⇒ (*′cmp bta*) ⇒ (*′cmp bta*) (**infixl** ∧^{b} *20*)
where *γ* ∧^{b} *γ′* ≡ *λ t n. γ t n* ∧ *γ′ t n*

lemma *conjI*[*intro*!]:
assumes *eval cid t t′ n γ* ∧ *eval cid t t′ n γ′*
shows *eval cid t t′ n* (*γ* ∧^{b} *γ′*)
⟨*proof*⟩

lemma *conjE*[*elim*!]:
assumes *eval cid t t′ n* (*γ* ∧^{b} *γ′*)
shows *eval cid t t′ n γ* ∧ *eval cid t t′ n γ′*
⟨*proof*⟩

2.4.6 Negation

definition *neg* :: (*′cmp bta*) ⇒ (*′cmp bta*) (¬^{b} - [*19*] *19*)
where ¬^{b} *γ* ≡ *λ t n.* ¬ *γ t n*

lemma *negI*[*intro*!]:
assumes ¬ *eval cid t t′ n γ*
shows *eval cid t t′ n* (¬^{b} *γ*)
⟨*proof*⟩

lemma *negE*[*elim*!]:
assumes *eval cid t t′ n* (¬^{b} *γ*)
shows ¬ *eval cid t t′ n γ*
⟨*proof*⟩

2.4.7 Quantifiers

definition *all* :: (*′a* ⇒ (*′cmp bta*))
⇒ (*′cmp bta*) (**binder** ∀$_b$ *10*)
where *all P* ≡ *λt n.* (∀ *y.* (*P y t n*))

lemma *allI*[*intro*!]:
assumes ∀ *p. eval cid t t′ n* (*γ p*)

shows *eval cid t t′ n* (*all* (λ*p*. *γ p*))
⟨*proof*⟩

lemma *allE*[*elim*!]:
assumes *eval cid t t′ n* (*all* (λ*p*. *γ p*))
shows ∀ *p*. *eval cid t t′ n* (*γ p*)
⟨*proof*⟩

definition *ex* :: (′*a* ⇒ (′*cmp bta*))
⇒ (′*cmp bta*) (**binder** ∃$_b$ *10*)
where *ex P* ≡ λ*t n*. (∃ *y*. (*P y t n*))

lemma *exI*[*intro*!]:
assumes ∃ *p*. *eval cid t t′ n* (*γ p*)
shows *eval cid t t′ n* (∃$_b$*p*. *γ p*)
⟨*proof*⟩

lemma *exE*[*elim*!]:
assumes *eval cid t t′ n* (∃$_b$*p*. *γ p*)
shows ∃ *p*. *eval cid t t′ n* (*γ p*)
⟨*proof*⟩

2.4.8 Behavior Assertions

First we provide rules for basic behavior assertions.

definition *ba* :: (′*cmp* ⇒ *bool*) ⇒ (′*cmp bta*) ([-]$_b$)
where *ba φ* ≡ λ *t n*. *φ* (*t n*)

lemma *baIA*[*intro*]:
fixes *c*::′*id*
and *t*::*nat* ⇒ *cnf*
and *t′*::*nat* ⇒ ′*cmp*
and *n*::*nat*
assumes ∃ *i*≥*n*. ‖*c*‖$_{t\ i}$
and *φ* (*σ*$_c$(*t* ⟨*c* → *t*⟩$_n$))
shows *eval c t t′ n* (*ba φ*)
⟨*proof*⟩

lemma *baIN1*[*intro*]:
fixes *c*::′*id*
and *t*::*nat* ⇒ *cnf*
and *t′*::*nat* ⇒ ′*cmp*
and *n*::*nat*
assumes *act*: ∃ *i*. ‖*c*‖$_{t\ i}$
and *nAct*: ∄ *i*. *i*≥*n* ∧ ‖*c*‖$_{t\ i}$
and *al*: *φ* (*t′* (*n* − ⟨*c* ∧ *t*⟩ − *1*))
shows *eval c t t′ n* (*ba φ*)
⟨*proof*⟩

lemma *baIN2*[*intro*]:
fixes *c*::′*id*
and *t*::*nat* ⇒ *cnf*
and *t′*::*nat* ⇒ ′*cmp*
and *n*::*nat*
assumes *nAct*: ∄ *i*. ‖*c*‖$_{t\ i}$

and *al*: $\varphi\ (t'\ n)$
shows *eval c t t' n* (*ba* φ)
⟨*proof*⟩

lemma *baIANow*[*intro*]:
fixes *t n c* φ
assumes $\varphi\ (\sigma_c(t\ n))$
and $\|c\|_{t\ n}$
shows *eval c t t' n* (*ba* φ)
⟨*proof*⟩

lemma *baEA*[*elim*]:
fixes *c*::*'id*
and *t*::*nat* ⇒ *cnf*
and *t'*::*nat* ⇒ *'cmp*
and *n*::*nat*
and *i*::*nat*
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
and *eval c t t' n* (*ba* φ)
shows $\varphi\ (\sigma_c(t\ \langle c \rightarrow t\rangle_n))$
⟨*proof*⟩

lemma *baEN1*[*elim*]:
fixes *c*::*'id*
and *t*::*nat* ⇒ *cnf*
and *t'*::*nat* ⇒ *'cmp*
and *n*::*nat*
assumes *act*: $\exists\, i.\ \|c\|_{t\ i}$
and *nAct*: $\nexists\, i.\ i{\geq}n \wedge \|c\|_{t\ i}$
and *al*: *eval c t t' n* (*ba* φ)
shows $\varphi\ (t'\ (n - \langle c \wedge t\rangle - 1))$
⟨*proof*⟩

lemma *baEN2*[*elim*]:
fixes *c*::*'id*
and *t*::*nat* ⇒ *cnf*
and *t'*::*nat* ⇒ *'cmp*
and *n*::*nat*
assumes *nAct*: $\nexists\, i.\ \|c\|_{t\ i}$
and *al*: *eval c t t' n* (*ba* φ)
shows $\varphi\ (t'\ n)$
⟨*proof*⟩

lemma *baEANow*[*elim*]:
fixes *t n c* φ
assumes *eval c t t' n* (*ba* φ)
and $\|c\|_{t\ n}$
shows $\varphi\ (\sigma_c(t\ n))$
⟨*proof*⟩

2.4.9 Next Operator

definition *nxt* :: (*'cmp bta*) ⇒ (*'cmp bta*) ($\bigcirc_b$(-) *24*)
where $\bigcirc_b(\gamma) \equiv \lambda\ t\ n.\ \gamma\ t\ (Suc\ n)$

lemma *nxtIA*[*intro*]:

fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
and $[\![\exists\, i{>}\langle c \rightarrow t\rangle_n.\ \|c\|_{t\ i}]\!] \Longrightarrow \exists\, n' \geq n.\ (\exists!i.\ n{\leq}i \wedge i{<}n' \wedge \|c\|_{t\ i}) \wedge eval\ c\ t\ t'\ n'\ \gamma$
and $[\![\neg(\exists\, i{>}\langle c \rightarrow t\rangle_n.\ \|c\|_{t\ i})]\!] \Longrightarrow eval\ c\ t\ t'\ (Suc\ \langle c \rightarrow t\rangle_n)\ \gamma$
shows $eval\ c\ t\ t'\ n\ (\bigcirc_b(\gamma))$
⟨*proof*⟩

lemma *nxtIN*[*intro*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
assumes $\neg(\exists\, i{\geq}n.\ \|c\|_{t\ i})$
and $eval\ c\ t\ t'\ (Suc\ n)\ \gamma$
shows $eval\ c\ t\ t'\ n\ (\bigcirc_b(\gamma))$
⟨*proof*⟩

lemma *nxtEA1*[*elim*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
assumes $\exists\, i{>}\langle c \rightarrow t\rangle_n.\ \|c\|_{t\ i}$
and $eval\ c\ t\ t'\ n\ (\bigcirc_b(\gamma))$
and $n'{\geq}n$
and $\exists!i.\ i{\geq}n \wedge i{<}n' \wedge \|c\|_{t\ i}$
shows $eval\ c\ t\ t'\ n'\ \gamma$
⟨*proof*⟩

lemma *nxtEA2*[*elim*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
and i
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$ **and** $\neg(\exists\, i{>}\langle c \rightarrow t\rangle_n.\ \|c\|_{t\ i})$
and $eval\ c\ t\ t'\ n\ (\bigcirc_b(\gamma))$
shows $eval\ c\ t\ t'\ (Suc\ \langle c \rightarrow t\rangle_n)\ \gamma$
⟨*proof*⟩

lemma *nxtEN*[*elim*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
assumes $\neg(\exists\, i{\geq}n.\ \|c\|_{t\ i})$
and $eval\ c\ t\ t'\ n\ (\bigcirc_b(\gamma))$
shows $eval\ c\ t\ t'\ (Suc\ n)\ \gamma$
⟨*proof*⟩

2.4.10 Eventually Operator

definition *evt* :: $('cmp\ bta) \Rightarrow ('cmp\ bta)$ $(\Diamond_b(\text{-})\ 23)$

where $\Diamond_b(\gamma) \equiv \lambda\ t\ n.\ \exists\, n' \geq n.\ \gamma\ t\ n'$

lemma *evtIA*[*intro*]:
fixes c::*'id*
and t::*nat* $\Rightarrow$ *cnf*
and t'::*nat* $\Rightarrow$ *'cmp*
and n::*nat*
and n'::*nat*
assumes $\exists\, i \geq n.\ \|c\|_{t\ i}$
and $n' \geq \langle c \Leftarrow t \rangle_n$
and $[\![\exists\, i \geq n'.\ \|c\|_{t\ i}]\!] \Longrightarrow \exists\, n'' \geq \langle c \Leftarrow t \rangle_{n'}.\ n'' \leq \langle c \rightarrow t \rangle_{n'} \wedge \mathit{eval}\ c\ t\ t'\ n''\ \gamma$
and $[\![\neg(\exists\, i \geq n'.\ \|c\|_{t\ i})]\!] \Longrightarrow \mathit{eval}\ c\ t\ t'\ n'\ \gamma$
shows $\mathit{eval}\ c\ t\ t'\ n\ (\Diamond_b(\gamma))$
⟨*proof*⟩

lemma *evtIN*[*intro*]:
fixes c::*'id*
and t::*nat* $\Rightarrow$ *cnf*
and t'::*nat* $\Rightarrow$ *'cmp*
and n::*nat*
and n'::*nat*
assumes $\neg(\exists\, i \geq n.\ \|c\|_{t\ i})$
and $n' \geq n$
and $\mathit{eval}\ c\ t\ t'\ n'\ \gamma$
shows $\mathit{eval}\ c\ t\ t'\ n\ (\Diamond_b(\gamma))$
⟨*proof*⟩

lemma *evtEA*[*elim*]:
fixes c::*'id*
and t::*nat* $\Rightarrow$ *cnf*
and t'::*nat* $\Rightarrow$ *'cmp*
and n::*nat*
assumes $\exists\, i \geq n.\ \|c\|_{t\ i}$
and $\mathit{eval}\ c\ t\ t'\ n\ (\Diamond_b(\gamma))$
shows $\exists\, n' \geq \langle c \rightarrow t \rangle_n.$
$(\exists\, i \geq n'.\ \|c\|_{t\ i} \wedge (\forall\, n'' \geq \langle c \Leftarrow t \rangle_{n'}.\ n'' \leq \langle c \rightarrow t \rangle_{n'} \longrightarrow \mathit{eval}\ c\ t\ t'\ n''\ \gamma)) \vee$
$(\neg(\exists\, i \geq n'.\ \|c\|_{t\ i}) \wedge \mathit{eval}\ c\ t\ t'\ n'\ \gamma)$
⟨*proof*⟩

lemma *evtEN*[*elim*]:
fixes c::*'id*
and t::*nat* $\Rightarrow$ *cnf*
and t'::*nat* $\Rightarrow$ *'cmp*
and n::*nat*
and n'::*nat*
assumes $\neg(\exists\, i \geq n.\ \|c\|_{t\ i})$
and $\mathit{eval}\ c\ t\ t'\ n\ (\Diamond_b(\gamma))$
shows $\exists\, n' \geq n.\ \mathit{eval}\ c\ t\ t'\ n'\ \gamma$
⟨*proof*⟩

2.4.11 Globally Operator

definition *glob* :: (*'cmp bta*) $\Rightarrow$ (*'cmp bta*) ($\Box_b$(-) *22*)
where $\Box_b(\gamma) \equiv \lambda\ t\ n.\ \forall\, n' \geq n.\ \gamma\ t\ n'$

lemma *globIA*[*intro*]:

fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
and $\bigwedge n'.\ [\![\exists\, i{\geq}n'.\ \|c\|_{t\ i};\ n'{\geq}\langle c \rightarrow t\rangle_n]\!] \Longrightarrow \exists\, n''{\geq}\langle c \Leftarrow t\rangle_{n'}.\ n''{\leq}\langle c \rightarrow t\rangle_{n'} \wedge eval\ c\ t\ t'\ n''\ \gamma$
and $\bigwedge n'.\ [\![\neg(\exists\, i{\geq}n'.\ \|c\|_{t\ i});\ n'{\geq}\langle c \rightarrow t\rangle_n]\!] \Longrightarrow eval\ c\ t\ t'\ n'\ \gamma$
shows $eval\ c\ t\ t'\ n\ (\Box_b(\gamma))$
⟨*proof*⟩

lemma *globIN*[*intro*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
assumes $\neg(\exists\, i{\geq}n.\ \|c\|_{t\ i})$
and $\bigwedge n'.\ n'{\geq}n \Longrightarrow eval\ c\ t\ t'\ n'\ \gamma$
shows $eval\ c\ t\ t'\ n\ (\Box_b(\gamma))$
⟨*proof*⟩

lemma *globEA*[*elim*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
and n'::nat
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
and $eval\ c\ t\ t'\ n\ (\Box_b(\gamma))$
and $n'{\geq}\langle c \Leftarrow t\rangle_n$
shows $eval\ c\ t\ t'\ n'\ \gamma$
⟨*proof*⟩

lemma *globEANow*:
fixes $c\ t\ t'\ n\ i\ \gamma$
assumes $n \leq i$
and $\|c\|_{t\ i}$
and $eval\ c\ t\ t'\ n\ (\Box_b\gamma)$
shows $eval\ c\ t\ t'\ i\ \gamma$
⟨*proof*⟩

lemma *globEN*[*elim*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
and n'::nat
assumes $\neg(\exists\, i{\geq}n.\ \|c\|_{t\ i})$
and $eval\ c\ t\ t'\ n\ (\Box_b(\gamma))$
and $n'{\geq}n$
shows $eval\ c\ t\ t'\ n'\ \gamma$
⟨*proof*⟩

2.4.12 Until Operator

definition *until* :: $('cmp\ bta) \Rightarrow ('cmp\ bta) \Rightarrow ('cmp\ bta)$ (**infixl** $\mathfrak{U}_b$ *21*)
where $\gamma'\ \mathfrak{U}_b\ \gamma \equiv \lambda\ t\ n.\ \exists\, n''{\geq}n.\ \gamma\ t\ n'' \wedge (\forall\, n'{\geq}n.\ n' < n'' \longrightarrow \gamma'\ t\ n')$

lemma *untilIA*[*intro*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
and n'::nat
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
and $n'{\geq}\langle c \Leftarrow t\rangle_n$
and $[\![\exists\, i{\geq}n'.\ \|c\|_{t\ i}]\!] \Longrightarrow \exists\, n''{\geq}\langle c \Leftarrow t\rangle_{n'}.\ n''{\leq}\ \langle c \rightarrow t\rangle_{n'} \wedge eval\ c\ t\ t'\ n''\ \gamma \wedge$
$(\forall\, n'''{\geq}\langle c \rightarrow t\rangle_n.\ n'''{<}\ \langle c \Leftarrow t\rangle_{n''}$
$\longrightarrow (\exists\, n''''{\geq}\langle c \Leftarrow t\rangle_{n'''}.\ n''''{\leq}\ \langle c \rightarrow t\rangle_{n'''} \wedge eval\ c\ t\ t'\ n''''\ \gamma'))$
and $[\![\neg(\exists\, i{\geq}n'.\ \|c\|_{t\ i})]\!] \Longrightarrow eval\ c\ t\ t'\ n'\ \gamma \wedge$
$(\forall\, n''{\geq}\langle c \rightarrow t\rangle_n.\ n''{<}\ n'$
$\longrightarrow ((\exists\, i{\geq}n''.\ \|c\|_{t\ i}) \wedge (\exists\, n'''{\geq}\langle c \Leftarrow t\rangle_{n''}.\ n'''{\leq}\ \langle c \rightarrow t\rangle_{n''} \wedge eval\ c\ t\ t'\ n'''\ \gamma')) \vee$
$(\neg(\exists\, i{\geq}n''.\ \|c\|_{t\ i}) \wedge eval\ c\ t\ t'\ n''\ \gamma'))$
shows $eval\ c\ t\ t'\ n\ (\gamma'\ \mathfrak{U}_b\ \gamma)$
⟨*proof*⟩

lemma *untilIN*[*intro*]:
fixes c::$'id$
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and n::nat
and n'::nat
assumes $\neg(\exists\, i{\geq}n.\ \|c\|_{t\ i})$
and $n'{\geq}n$
and $eval\ c\ t\ t'\ n'\ \gamma$
and $a1$: $\bigwedge n''.\ [\![n{\leq}n'';\ n''{<}n']\!] \Longrightarrow eval\ c\ t\ t'\ n''\ \gamma'$
shows $eval\ c\ t\ t'\ n\ (\gamma'\ \mathfrak{U}_b\ \gamma)$
⟨*proof*⟩

lemma *untilEA*[*elim*]:
fixes n::nat
and n'::nat
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and c::$'id$
assumes $\exists\, i{\geq}n.\ \|c\|_{t\ i}$
and $eval\ c\ t\ t'\ n\ (\gamma'\ \mathfrak{U}_b\ \gamma)$
shows $\exists\, n'{\geq}\langle c \rightarrow t\rangle_n.$
$((\exists\, i{\geq}n'.\ \|c\|_{t\ i}) \wedge (\forall\, n''{\geq}\ \langle c \Leftarrow t\rangle_{n'}.\ n''{\leq}\langle c \rightarrow t\rangle_{n'} \longrightarrow eval\ c\ t\ t'\ n''\ \gamma)$
$\wedge\ (\forall\, n''{\geq}\langle c \Leftarrow t\rangle_n.\ n''{<}\ \langle c \Leftarrow t\rangle_{n'} \longrightarrow eval\ c\ t\ t'\ n''\ \gamma') \vee$
$(\neg(\exists\, i{\geq}n'.\ \|c\|_{t\ i})) \wedge eval\ c\ t\ t'\ n'\ \gamma \wedge (\forall\, n''{\geq}\langle c \Leftarrow t\rangle_n.\ n''{<}\ n' \longrightarrow eval\ c\ t\ t'\ n''\ \gamma'))$
⟨*proof*⟩

lemma *untilEN*[*elim*]:
fixes n::nat
and n'::nat
and t::$nat \Rightarrow cnf$
and t'::$nat \Rightarrow 'cmp$
and c::$'id$
assumes $\nexists\, i.\ i{\geq}n \wedge \|c\|_{t\ i}$
and $eval\ c\ t\ t'\ n\ (\gamma'\ \mathfrak{U}_b\ \gamma)$
shows $\exists\, n'{\geq}n.\ eval\ c\ t\ t'\ n'\ \gamma \wedge$

$(\forall\, n''{\geq}n.\ n'' < n' \longrightarrow eval\ c\ t\ t'\ n''\ \gamma')$
⟨*proof*⟩

2.4.13 Weak Until

definition *wuntil* :: (*'cmp bta*) ⇒ (*'cmp bta*) ⇒ (*'cmp bta*) (**infixl** $\mathfrak{W}_b$ *20*)
where $\gamma'\ \mathfrak{W}_b\ \gamma \equiv \gamma'\ \mathfrak{U}_b\ \gamma \vee^b \Box_b(\gamma')$

end

end

References

[1] A. Lochbihler. Coinduction. *The Archive of Formal Proof s. https://isa-afp.org/entries/Coinductive.shtml*, 2010.

[2] D. Marmsoler. On the semantics of temporal specifications of component-behavior for dynamic architectures. In *Eleventh International Symposium on Theoretical Aspects of Software Engineering*. Springer, 2017.

[3] D. Marmsoler. Towards a calculus for dynamic architectures. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 2017.

[4] D. Marmsoler and M. Gleirscher. On activation, connection, and behavior in dynamic architectures. *Scientific Annals of Computer Science*, 26(2):187–248, 2016.

[5] D. Marmsoler and M. Gleirscher. Specifying properties of dynamic architectures using configuration traces. In *International Colloquium on Theoretical Aspects of Computing*, pages 235–254. Springer, 2016.